Parallel Programming for Science and Engineering

Using MPI, OpenMP, and the PETSc library

Victor Eijkhout

2nd edition 2020, formatted February 25, 2021

Public draft - open for comments

This book is open source under the CC-BY 4.0 license.

Download location for the latest pdf is: https://web.corral.tacc.utexas.edu/CompEdu/pdf/pcse/

The term 'parallel computing' means different things depending on the application area. In this book we focus on parallel computing – and more specifically parallel *programming*; we will not discuss a lot of theory – in the context of scientific computing.

Two of the most common software systems for parallel programming in scientific computing are MPI and OpenMP. They target different types of parallelism, and use very different constructs. Thus, by covering both of them in one book we can offer a treatment of parallelism that spans a large range of possible applications.

Finally, we also discuss the PETSc (Portable Toolkit for Scientific Computing) library, which offers an abstraction level higher than MPI or OpenMP, geared specifically towards parallel linear algebra, and very specifically the sort of linear algebra computations arising from Partial Differential Equation modeling.

The main languages in scientific computing are C/C++ and Fortran. We will discuss both MPI and OpenMP with many examples in these two languages. (A Java interface to MPI is rumored to exist but we lend this no credit.) For MPI and the PETSc library we will also discuss the Python interfaces.

Contents

I MPI *13*
1 **Getting started with MPI** *14*
1.1 *Distributed memory and message passing* *14*
1.2 *History* *15*
1.3 *Basic model* *15*
1.4 *Making and running an MPI program* *16*
1.5 *Language bindings* *17*
1.6 *Review* *21*
1.7 *Sources used in this chapter* *22*
2 **MPI topic: Functional parallelism** *23*
2.1 *The SPMD model* *23*
2.2 *Starting and running MPI processes* *25*
2.3 *Processor identification* *29*
2.4 *Functional parallelism* *32*
2.5 *Distributed computing and distributed data* *34*
2.6 *Review questions* *35*
2.7 *Sources used in this chapter* *36*
3 **MPI topic: Collectives** *38*
3.1 *Working with global information* *38*
3.2 *Reduction* *41*
3.3 *Rooted collectives: broadcast, reduce* *45*
3.4 *Scan operations* *52*
3.5 *Rooted collectives: gather and scatter* *58*
3.6 *All-to-all* *62*
3.7 *Reduce-scatter* *65*
3.8 *Barrier* *69*
3.9 *Variable-size-input collectives* *69*
3.10 *MPI Operators* *74*
3.11 *Nonblocking collectives* *78*
3.12 *Performance of collectives* *85*
3.13 *Collectives and synchronization* *85*
3.14 *Performance considerations* *88*
3.15 *Review questions* *91*

3.16 *Sources used in this chapter* 93
4 **MPI topic: Point-to-point** 117
4.1 *Blocking point-to-point operations* 117
4.2 *Nonblocking point-to-point operations* 133
4.3 *More about point-to-point communication* 146
4.4 *Review questions* 153
4.5 *Sources used in this chapter* 157
5 **MPI topic: Communication modes** 177
5.1 *Persistent communication requests* 177
5.2 *Partitioned communication* 179
5.3 *Synchronous and asynchronous communication* 180
5.4 *Local and nonlocal operations* 181
5.5 *Buffered communication* 182
5.6 *Sources used in this chapter* 186
6 **MPI topic: Data types** 191
6.1 *Data type handling* 191
6.2 *Elementary data types* 192
6.3 *Derived datatypes* 198
6.4 *Type maps and type matching* 218
6.5 *Type extent* 218
6.6 *More about data* 223
6.7 *Review questions* 228
6.8 *Sources used in this chapter* 229
7 **MPI topic: Communicators** 260
7.1 *Basic communicators* 260
7.2 *Duplicating communicators* 261
7.3 *Sub-communicators* 265
7.4 *Splitting a communicator* 266
7.5 *Communicators and groups* 269
7.6 *Inter-communicators* 271
7.7 *Review questions* 276
7.8 *Sources used in this chapter* 277
8 **MPI topic: Process management** 285
8.1 *Process spawning* 285
8.2 *Socket-style communications* 288
8.3 *Sessions* 291
8.4 *Functionality available outside init/finalize* 292
8.5 *Sources used in this chapter* 293
9 **MPI topic: One-sided communication** 299
9.1 *Windows* 300
9.2 *Active target synchronization: epochs* 304
9.3 *Put, get, accumulate* 305
9.4 *Passive target synchronization* 317

9.5 *More about window memory* 322
9.6 *Assertions* 325
9.7 *Implementation* 326
9.8 *Review questions* 328
9.9 *Sources used in this chapter* 329
10 **MPI topic: File I/O** 345
10.1 *File handling* 346
10.2 *File reading and writing* 347
10.3 *Consistency* 353
10.4 *Constants* 354
10.5 *Review questions* 355
10.6 *Sources used in this chapter* 356
11 **MPI topic: Topologies** 357
11.1 *Cartesian grid topology* 357
11.2 *Distributed graph topology* 359
11.3 *Sources used in this chapter* 365
12 **MPI topic: Shared memory** 367
12.1 *Recognizing shared memory* 367
12.2 *Shared memory for windows* 368
12.3 *Sources used in this chapter* 372
13 **MPI topic: Tools interface** 378
13.1 *Control variables* 378
13.2 *Performance variables* 379
13.3 *Categories of variables* 381
13.4 *Sources used in this chapter* 382
14 **MPI leftover topics** 383
14.1 *Contextual information, attributes, etc.* 383
14.2 *Error handling* 389
14.3 *Fortran issues* 393
14.4 *Fault tolerance* 394
14.5 *Performance, tools, and profiling* 394
14.6 *Determinism* 399
14.7 *Subtleties with processor synchronization* 399
14.8 *Shell interaction* 400
14.9 *The origin of one-sided communication in ShMem* 401
14.10 *Leftover topics* 402
14.11 *Literature* 404
14.12 *Sources used in this chapter* 405

II OpenMP 409
15 **Getting started with OpenMP** 410
15.1 *The OpenMP model* 410

15.2 *Compiling and running an OpenMP program* 413
15.3 *Your first OpenMP program* 414
15.4 *Thread data* 416
15.5 *Creating parallelism* 418
15.6 *Sources used in this chapter* 420
16 **OpenMP topic: Parallel regions** 421
16.1 *Nested parallelism* 423
16.2 *Cancel parallel construct* 425
16.3 *Sources used in this chapter* 426
17 **OpenMP topic: Loop parallelism** 428
17.1 *Loop parallelism* 428
17.2 *Loop schedules* 430
17.3 *Reductions* 433
17.4 *Collapsing nested loops* 433
17.5 *Ordered iterations* 434
17.6 `nowait` 435
17.7 *While loops* 435
17.8 *Sources used in this chapter* 437
18 **OpenMP topic: Work sharing** 438
18.1 *Sections* 438
18.2 *Single/master* 439
18.3 *Fortran array syntax parallelization* 440
18.4 *Sources used in this chapter* 441
19 **OpenMP topic: Controlling thread data** 442
19.1 *Shared data* 442
19.2 *Private data* 442
19.3 *Data in dynamic scope* 443
19.4 *Temporary variables in a loop* 444
19.5 *Default* 444
19.6 *Array data* 445
19.7 *First and last private* 446
19.8 *Persistent data through* `threadprivate` 446
19.9 *Sources used in this chapter* 449
20 **OpenMP topic: Reductions** 451
20.1 *Built-in reduction operators* 453
20.2 *Initial value for reductions* 453
20.3 *User-defined reductions* 454
20.4 *Reductions and floating-point math* 455
20.5 *Sources used in this chapter* 456
21 **OpenMP topic: Synchronization** 458
21.1 *Barrier* 458
21.2 *Mutual exclusion* 459
21.3 *Locks* 460

21.4 *Example: Fibonacci computation* 462
21.5 *Sources used in this chapter* 465
22 **OpenMP topic: Tasks** 466
22.1 *Task data* 467
22.2 *Task synchronization* 468
22.3 *Task dependencies* 470
22.4 *More* 471
22.5 *Examples* 472
22.6 *Sources used in this chapter* 474
23 **OpenMP topic: Affinity** 477
23.1 *OpenMP thread affinity control* 477
23.2 *First-touch* 481
23.3 *Affinity control outside OpenMP* 481
23.4 *Sources used in this chapter* 483
24 **OpenMP topic: Memory model** 484
24.1 *Thread synchronization* 484
24.2 *Data races* 485
24.3 *Relaxed memory model* 486
24.4 *Sources used in this chapter* 487
25 **OpenMP topic: SIMD processing** 488
25.1 *Sources used in this chapter* 492
26 **OpenMP remaining topics** 493
26.1 *Runtime functions and internal control variables* 493
26.2 *Timing* 495
26.3 *Thread safety* 495
26.4 *Performance and tuning* 496
26.5 *Accelerators* 497
26.6 *Sources used in this chapter* 498
27 **OpenMP Review** 499
27.1 *Concepts review* 499
27.2 *Review questions* 500
27.3 *Sources used in this chapter* 508

III PETSc 509
28 **PETSc basics** 510
28.1 *What is PETSc and why?* 510
28.2 *Basics of running a PETSc program* 512
28.3 *PETSc installation* 516
28.4 *Sources used in this chapter* 518
29 **PETSc objects** 519
29.1 *Distributed objects* 519
29.2 *Scalars* 520

29.3 *Vec: Vectors* 522
29.4 *Mat: Matrices* 531
29.5 *Index sets and Vector Scatters* 541
29.6 *AO: Application Orderings* 543
29.7 *Partitionings* 543
29.8 *Sources used in this chapter* 544
30 **Grid support** 549
30.1 *Grid definition* 549
30.2 *Constructing a vector on a grid* 552
30.3 *Constructing a matrix on a grid* 553
30.4 *Vectors of a distributed array* 554
30.5 *Matrices of a distributed array* 554
30.6 *Sources used in this chapter* 556
31 **Finite Elements support** 561
31.1 *Sources used in this chapter* 563
32 **PETSc solvers** 564
32.1 *KSP: linear system solvers* 564
32.2 *Direct solvers* 574
32.3 *Control through command line options* 574
32.4 *Sources used in this chapter* 575
33 **PETSC nonlinear solvers** 576
33.1 *Nonlinear systems* 576
33.2 *Time-stepping* 578
33.3 *Sources used in this chapter* 579
34 **PETSc tools** 580
34.1 *Error checking and debugging* 580
34.2 *Program output* 582
34.3 *Commandline options* 586
34.4 *Timing and profiling* 588
34.5 *Memory management* 588
34.6 *Sources used in this chapter* 590
35 **PETSc topics** 594
35.1 *Communicators* 594
35.2 *Sources used in this chapter* 595

IV Other programming models 597
36 **Co-array Fortran** 598
36.1 *History and design* 598
36.2 *Compiling and running* 598
36.3 *Basics* 598
36.4 *Sources used in this chapter* 602
37 **Sycl, OneAPI, DPC++** 603

37.1 *Logistics* 603
37.2 *Platforms and devices* 604
37.3 *Queues* 604
37.4 *Kernels* 605
37.5 *Parallel operations* 606
37.6 *Memory access* 610
37.7 *Parallel output* 613
37.8 *DPCPP extensions* 613
37.9 *Intel devcloud notes* 613
37.10 *Examples* 614
37.11 *Sources used in this chapter* 615

V The Rest 623
38 **Exploring computer architecture** 624
38.1 *Tools for discovery* 624
38.2 *Sources used in this chapter* 625
39 **Process and thread affinity** 626
39.1 *What does the hardware look like?* 627
39.2 *Affinity control* 629
39.3 *Sources used in this chapter* 630
40 **Hybrid computing** 631
40.1 *Discussion* 632
40.2 *Hybrid MPI-plus-threads execution* 633
40.3 *Sources used in this chapter* 636
41 **Random number generation** 637
41.1 *Sources used in this chapter* 638
42 **Parallel I/O** 639
42.1 *Sources used in this chapter* 640
43 **Support libraries** 641
43.1 *SimGrid* 641
43.2 *Other* 641
43.3 *Sources used in this chapter* 642

VI Tutorials 643
43.4 *Debugging* 645
43.5 *Tracing and profiling with TAU* 653
43.6 *SimGrid* 658
43.7 *Batch systems* 661

VII Class projects 671
44 **A Style Guide to Project Submissions** 672

45 **Warmup Exercises** *675*
46 **Mandelbrot set** *679*
47 **Data parallel grids** *685*
48 **N-body problems** *688*

VIII Didactics *689*
49 **Teaching guide** *690*
49.1 *Sources used in this chapter* *691*
50 **Teaching from mental modesl** *692*
50.1 *Sources used in this chapter* *707*

IX Bibliography, index, and list of acronyms *709*
51 **Bibliography** *710*
52 **List of acronyms** *712*
53 **General Index** *713*
54 **Index of MPI commands and keywords** *723*
54.1 *From the standard document* *731*
54.2 *MPL commands and topics* *736*
54.3 *Python* *739*
55 **Index of OpenMP commands** *741*
56 **Index of PETSc commands** *743*
57 **Index of SYCL commands** *747*

PART I

MPI

Chapter 1

Getting started with MPI

In this chapter you will learn the use of the main tool for distributed memory programming: the Message Passing Interface (MPI) library. The MPI library has about 250 routines, many of which you may never need. Since this is a textbook, not a reference manual, we will focus on the important concepts and give the important routines for each concept. What you learn here should be enough for most common purposes. You are advised to keep a reference document handy, in case there is a specialized routine, or to look up subtleties about the routines you use.

1.1 Distributed memory and message passing

In its simplest form, a distributed memory machine is a collection of single computers hooked up with network cables. In fact, this has a name: a *Beowulf cluster*. As you recognize from that setup, each processor can run an independent program, and has its own memory without direct access to other processors' memory. MPI is the magic that makes multiple instantiations of the same executable run so that they know about each other and can exchange data through the network.

One of the reasons that MPI is so successful as a tool for high performance on clusters is that it is very explicit: the programmer controls many details of the data motion between the processors. Consequently, a capable programmer can write very efficient code with MPI. Unfortunately, that programmer will have to spell things out in considerable detail. For this reason, people sometimes call MPI 'the assembly language of parallel programming'. If that sounds scary, be assured that things are not that bad. You can get started fairly quickly with MPI, using just the basics, and coming to the more sophisticated tools only when necessary.

Another reason that MPI was a big hit with programmers is that it does not ask you to learn a new language: it is a library that can be interface to C/C++ or Fortran; there are even bindings to Python. A related point is that it is easy to install: there are free implementations that you can download and install on any computer that has a Unix-like operating system, even if that is not a parallel machine.

1.2 History

Before the MPI standard was developed in 1993-4, there were many libraries for distributed memory computing, often proprietary to a vendor platform. MPI standardized the inter-process communication mechanisms. Other features, such as process management in *PVM*, or parallel I/O were omitted. Later versions of the standard have included many of these features.

Since MPI was designed by a large number of academic and commercial participants, it quickly became a standard. A few packages from the pre-MPI era, such as *Charmpp* [16], are still in use since they support mechanisms that do not exist in MPI.

1.3 Basic model

Here we sketch the two most common scenarios for using MPI. In the first, the user is working on an interactive machine, which has network access to a number of hosts, typically a network of workstations; see figure 1.1. The user types the command `mpiexec`[1] and supplies

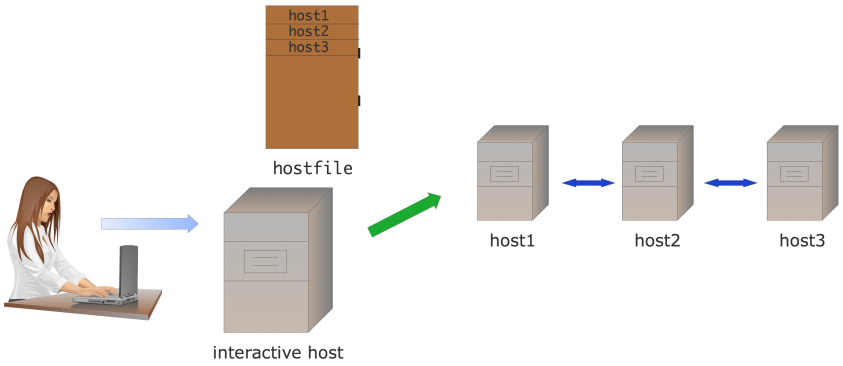


Figure 1.1: Interactive MPI setup

- The number of hosts involved,
- their names, possibly in a hostfile,
- and other parameters, such as whether to include the interactive host; followed by
- the name of the program and its parameters.

The `mpirun` program then makes an `ssh` connection to each of the hosts, giving them sufficient information that they can find each other. All the output of the processors is piped through the `mpirun` program, and appears on the interactive console.

In the second scenario (figure 1.2) the user prepares a *batch job* script with commands, and these will be run when the *batch scheduler* gives a number of hosts to the job. Now the batch script contains the `mpirun` command, and the hostfile is dynamically generated when the job starts. Since the job now runs at a time when the user may not be logged in, any screen output goes into an output file.

1. A command variant is `mpirun`; your local cluster may have a different mechanism.

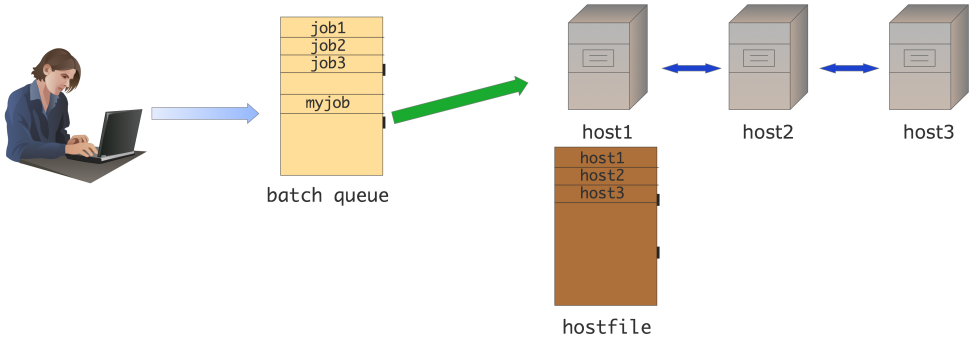


Figure 1.2: Batch MPI setup

You see that in both scenarios the parallel program is started by the `mpirun` command using an Single Program Multiple Data (SPMD) mode of execution: all hosts execute the same program. It is possible for different hosts to execute different programs, but we will not consider that in this book.

There can be options and environment variables that are specific to some MPI installations, or to the network.

- *mpich* and its derivatives such as *Intel MPI* or *Cray MPI* have *mpiexec options*: https://www.mpich.org/static/docs/v3.1/www1/mpiexec.html

1.4 Making and running an MPI program

MPI is a library, called from programs in ordinary programming languages such as C/C++ or Fortran. To compile such a program you use your regular compiler:

```
gcc -c my_mpi_prog.c -I/path/to/mpi.h
gcc -o my_mpi_prog my_mpi_prog.o -L/path/to/mpi -lmpich
```

However, MPI libraries may have different names between different architectures, making it hard to have a portable makefile. Therefore, MPI typically has shell scripts around your compiler call:

```
mpicc -c my_mpi_prog.c
mpicc -o my_mpi_prog my_mpi_prog.o
```

MPI programs can be run on many different architectures. Obviously it is your ambition (or at least your dream) to run your code on a cluster with a hundred thousand processors and a fast network. But maybe you only have a small cluster with plain *ethernet*. Or maybe you're sitting in a plane, with just your laptop. An MPI program can be run in all these circumstances – within the limits of your available memory of course.

The way this works is that you do not start your executable directly, but you use a program, typically called `mpirun` or something similar, which makes a connection to all available processors and starts a run of your executable there. So if you have a thousand nodes in your cluster, `mpirun` can start your program once on each, and if you only have your laptop it can start a few instances there. In the latter case you will of course not get great performance, but at least you can test your code for correctness.

Python note. Load the TACC-provided python:

```
module load python
```

and run it as:

```
ibrun python-mpi yourprogram.py
```

1.5 Language bindings

1.5.1 C

The MPI library is written in C. Thus, its bindings are the most natural for that language.

1.5.2 C++, including MPL

C++ bindings were defined in the standard at one point, but they were declared deprecated, and have been officially removed in the *MPI 3*MPI-3 Thus, MPI can be used from C++ by including

```
#include <mpi.h>
```

and using the C API.

The following material is for the (unreleased) MPI-4 standard only

The MPI-4 standard supports integer arguments larger than 32 bits, through the `MPI_Count` datatype. All MPI routines are now polymorphic between using `int` and `MPI_Count`. This requires including `mpi.hpp`:

```
#include <mpi.hpp>
```

rather than `mpi.h`.

End of MPI-4 material

The *boost* library has its own version of MPI, but it seems not to be under further development. A recent effort at idiomatic C++ support is *Message Passing Layer (MPL)* https://github.com/rabauke/mpl. This book has an index of MPL notes and commands: section 54.2.

MPL note 1. MPL is a C++ header-only library. Notes on MPI usage from MPL will be indicated like this.

End of MPL note

1.5.3 Fortran

Fortran note. Fortran-specific notes will be indicated with a note like this.

Traditionally, *Fortran bindings* for MPI look very much like the C ones, except that each routine has a final *error return* parameter. You will find that a lot of MPI code in Fortran conforms to this.

However, in the *MPI 3* standard it is recommended that an MPI implementation providing a Fortran interface provide a module named `mpi_f08` that can be used in a Fortran program. This incorporates the following improvements:

- This defines MPI routines to have an optional final parameter for the error.

- There are some visible implications of using the `mpi_f08` module, mostly related to the fact that some of the 'MPI datatypes' such as `MPI_Comm`, which were declared as `Integer` previously, are now a Fortran `Type`. See the following sections for details: Communicator 7.1, Datatype 6.3.1.1, Info 14.1.1, Op 3.10.2, Request 4.2.1, Status 4.3.2, Window 9.1.
- The `mpi_f08` module solves a problem with previous *Fortran90 bindings*: Fortran90 is a strongly typed language, so it is not possible to pass argument by reference to their address, as C/C++ do with the `void*` type for send and receive buffers. This was solved by having separate routines for each datatype, and providing an `Interface` block in the MPI module. If you manage to request a version that does not exist, the compiler will display a message like
  `There is no matching specific subroutine for this generic subroutine call [MPI_Send]`
  For details see http://mpi-forum.org/docs/mpi-3.1/mpi31-report/node409.htm.

1.5.4 Python

Python note. Python-specific notes will be indicated with a note like this.

The `mpi4py` package [5] of *python bindings* is not defined by the MPI standards committee. Instead, it is the work of an individual, *Lisandro Dalcin*.

In a way, the Python interface is the most elegant. It uses Object-Oriented (OO) techniques such as methods on objects, and many default arguments.

Notable about the Python bindings is that many communication routines exist in two variants:

- a version that can send native Python objects. These routines have lowercase names such as `bcast`; and
- a version that sends *numpy* objects; these routines have names such as `Bcast`. Their syntax can be slightly different.

The first version looks more 'pythonic', is easier to write, and can do things like sending python objects, but it is also decidedly less efficient since data is packed and unpacked with `pickle`. As a common sense guideline, use the `numpy` interface in the performance-critical parts of your code, and the native interface only for complicated actions in a setup phase.

Codes with `mpi4py` can be interfaced to other languages through Swig or conversion routines.

Data in `numpy` can be specified as a simple object, or `[data, (count,displ), datatype]`.

1.5.5 How to read routine prototypes

Throughout the MPI part of this book we will give the reference syntax of the routines. This typically comprises:

- The semantics: routine name and list of parameters and what they mean.
- C synxtax: the routine definition as it appears in the `mpi.h` file.
- Fortran syntax: routine definition with parameters, giving in/out specification.
- Python syntax: routine name, indicating to what class it applies, and parameter, indicating which ones are optional.

These 'routine prototypes' look like code but they are not! Here is how you translate them.

1.5.5.1 C

The typically C routine specification in MPI looks like:

```
int MPI_Comm_size(MPI_Comm comm,int *nprocs)
```

This means that

- The routine returns an `int` parameter. Strictly speaking you should test against `MPI_SUCCESS` (for all error codes, see section 14.2.1):

  ```
  MPI_Comm comm = MPI_COMM_WORLD;
  int nprocs;
  int errorcode;
  errorcode = MPI_Comm_size( MPI_COMM_WORLD,&nprocs);
  if (errorcode!=MPI_SUCCESS) {
    printf("Routine MPI_Comm_size failed! code=%d\n",
           errorcode);
    return 1;
  }
  ```

  However, the error codes are hardly ever useful, and there is not much your program can do to recover from an error. Most people call the routine as

  ```
  MPI_Comm_size( /* parameter ... */ );
  ```

  For more on error handling, see section 14.2.
- The first argument is of type `MPI_Comm`. This is not a C built-in datatype, but it behaves like one. There are many of these `MPI_something` datatypes in MPI. So you can write:

  ```
  MPI_Comm my_comm =
      MPI_COMM_WORLD; // using a predefined value
  MPI_Comm_size( comm, /* remaining parameters */ );
  ```

- Finally, there is a 'star' parameter. This means that the routine wants an address, rather than a value. You would typically write:

  ```
  MPI_Comm my_comm = MPI_COMM_WORLD; // using a predefined value
  int nprocs;
  MPI_Comm_size( comm, &nprocs );
  ```

  Seeing a 'star' parameter usually means either: the routine has an array argument, or: the routine internally sets the value of a variable. The latter is the case here.

1.5.5.2 Fortran

The Fortran specification looks like:

```
MPI_Comm_size(comm, size, ierror)
Type(MPI_Comm), Intent(In) :: comm
Integer, Intent(Out) :: size
Integer, Optional, Intent(Out) :: ierror
```

or for the pre-2008 legacy mode:

```
MPI_Comm_size(comm, size, ierror)
INTEGER, INTENT(IN) :: comm
INTEGER, INTENT(OUT) :: size
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

The syntax of using this routine is close to this specification: you write

```
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
! legacy: Integer :: comm = MPI_COMM_WORLD
Integer :: comm = MPI_COMM_WORLD
Integer :: size,ierr
CALL MPI_Comm_size( comm, size ) ! without the optional ierr
```

- Most Fortran routines have the same parameters as the corresponding C routine, except that they all have the error code as final parameter, instead of as a function result. As with C, you can ignore the value of that parameter. Just don't forget it.
- The types of the parameters are given in the specification.
- Where C routines have `MPI_Comm` and `MPI_Request` and such parameters, Fortran has `INTEGER` parameters, or sometimes arrays of integers.

1.5.5.3 *Python*

The Python interface to MPI uses classes and objects. Thus, a specification like:

```
MPI.Comm.Send(self, buf, int dest, int tag=0)
```

should be parsed as follows.

- First of all, you need the `MPI` class:

  ```
  from mpi4py import MPI
  ```

- Next, you need a `Comm` object. Often you will use the predefined communicator

  ```
  comm = MPI.COMM_WORLD
  ```

- The keyword `self` indicates that the actual routine `Send` is a method of the `Comm` object, so you call:

  ```
  comm.Send( .... )
  ```

- Parameters that are listed by themselves, such as `buf`, as positional. Parameters that are listed with a type, such as `int dest` are keyword parameters. Keyword parameters that have a value specified, such as `int tag=0` are optional, with the default value indicated. Thus, the typical call for this routine is:

  ```
  comm.Send(sendbuf,dest=other)
  ```

  specifying the send buffer as positional parameter, the destination as keyword parameter, and using the default value for the optional tag.

Some python routines are 'class methods', and their specification lacks the `self` keyword. For instance:

```
MPI.Request.Waitall(type cls, requests, statuses=None)
```

would be used as

```
MPI.Request.Waitall(requests)
```

1.6 Review

Review 1.1. What determines the parallelism of an MPI job?

1. The size of the cluster you run on.
2. The number of cores per cluster node.
3. The parameters of the MPI starter (`mpiexec`, `ibrun`,...)

Review 1.2. T/F: the number of cores of your laptop is the limit of how many MPI proceses you can start up.

Review 1.3. Do the following languages have an object-oriented interface to MPI? In what sense?

1. C
2. C++
3. Fortran2008
4. Python

1.7 Sources used in this chapter

1.7.1 Listing of code header

Chapter 2

MPI topic: Functional parallelism

2.1 The SPMD model

MPI programs conform largely to the Single Program Multiple Data (SPMD) model, where each processor runs the same executable. This running executable we call a *process*.

When MPI was first written, 20 years ago, it was clear what a processor was: it was what was in a computer on someone's desk, or in a rack. If this computer was part of a networked cluster, you called it a *node*. So if you ran an MPI program, each node would have one MPI process; figure 2.1. You could of course run

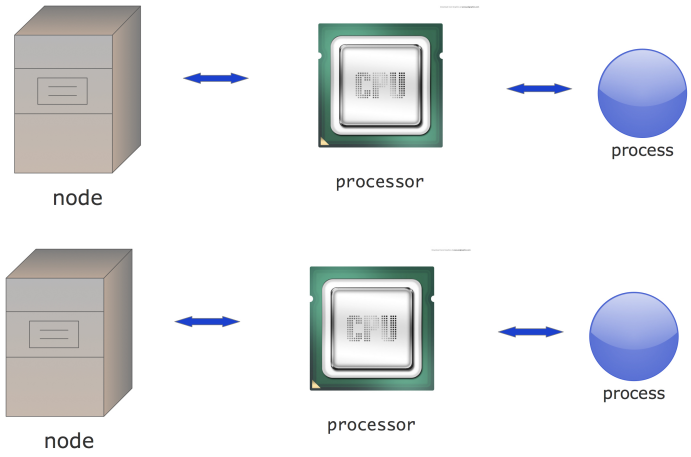


Figure 2.1: Cluster structure as of the mid 1990s

more than one process, using the *time slicing* of the Operating System (OS), but that would give you no extra performance.

These days the situation is more complicated. You can still talk about a node in a cluster, but now a node can contain more than one processor chip (sometimes called a *socket*), and each processor chip probably has multiple *cores*. Figure 2.2 shows how you could explore this using a mix of MPI between the nodes, and a shared memory programming system on the nodes.

However, since each core can act like an independent processor, you can also have multiple MPI processes per node. To MPI, the cores look like the old completely separate processors. This is the 'pure MPI' model

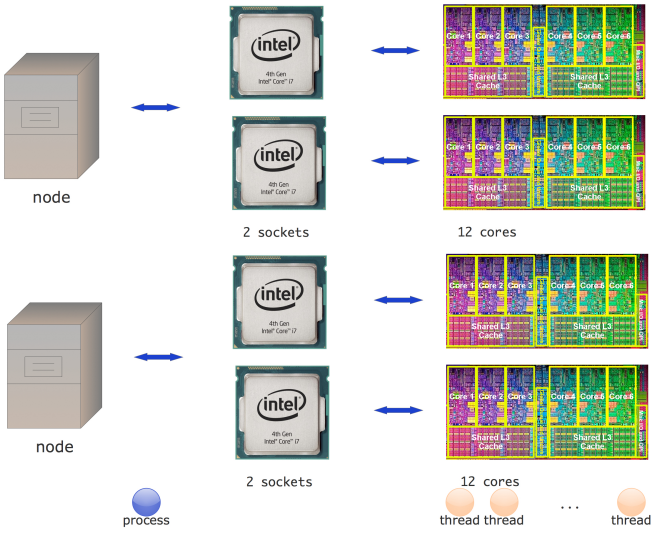


Figure 2.2: Hybrid cluster structure

of figure 2.3, which we will use in most of this part of the book. (Hybrid computing will be discussed in chapter 40.)

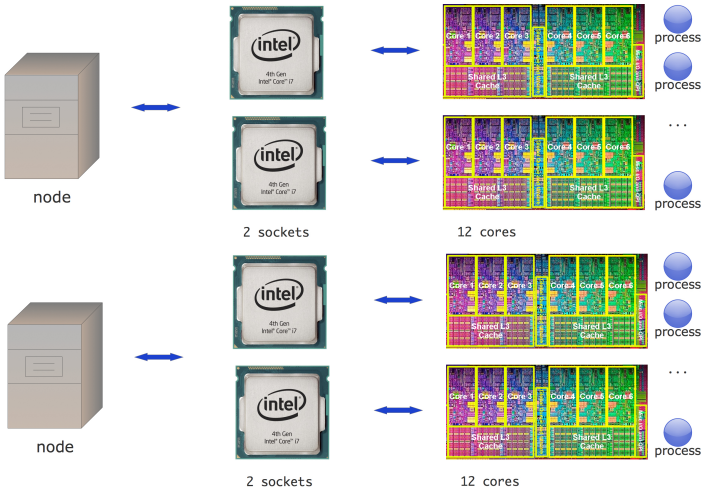


Figure 2.3: MPI-only cluster structure

This is somewhat confusing: the old processors needed MPI programming, because they were physically separated. The cores on a modern processor, on the other hand, share the same memory, and even some caches. In its basic mode MPI seems to ignore all of this: each core receives an MPI process and the programmer writes the same send/receive call no matter where the other process is located. In fact, you can't immediately see whether two cores are on the same node or different nodes. Of course, on the implementation level MPI uses a different communication mechanism depending on whether cores are

on the same socket or on different nodes, so you don't have to worry about lack of efficiency.

Remark 1 *In some rare cases you may want to run in an Multiple Program Multiple Data (MPMD) mode, rather than SPMD. This can be achieved either on the OS level (see section 14.8.4), using options of the* mpiexec *mechanism, or you can use MPI's built-in process management; chapter 8. Like I said, this concerns only rare cases.*

2.2 Starting and running MPI processes

The SPMD model may be initially confusing. Even though there is only a single source, compiled into a single executable, the parallel run comprises a number of independently started MPI processes (see section 1.3 for the mechanism).

The following exercises are designed to give you an intuition for this one-source-many-processes setup. In the first exercise you will see that the mechanism for starting MPI programs starts up independent copies. There is nothing in the source that says 'and now you become parallel'.

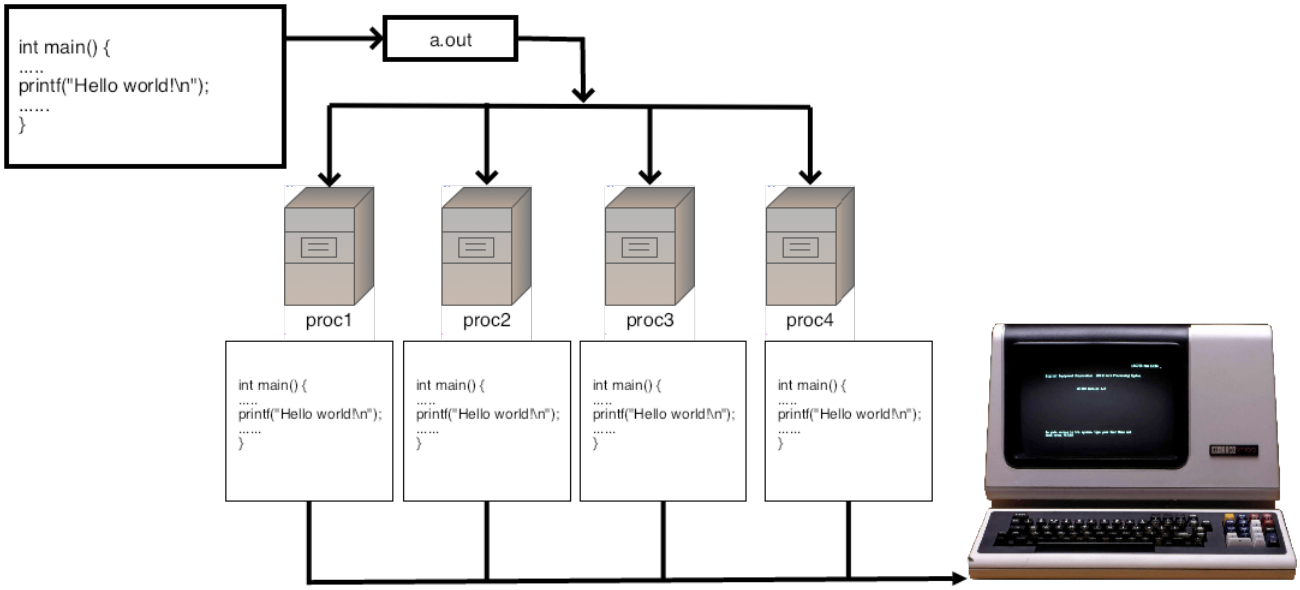


Figure 2.4: Running a hello world program in parallel

The following exercise demonstrates this point.

Exercise 2.1. Write a 'hello world' program, without any MPI in it, and run it in parallel with `mpiexec` or your local equivalent. Explain the output.
(There is a skeleton for this exercise under the name `hello`.)

This exercise is illustrated in figure 2.4.

Figure 2.1 `MPI_Init`

Name	Param name	C type	F type	inout
`mpi_init (`				
	`argc`	`int*`		`inout`
	`argv`	`char***`		`inout`
	`length: argc`			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Init(`		
`) -> None`		

2.2.1 Headers

If you use MPI commands in a program file, be sure to include the proper header file, *mpi.h* or *mpif.h*.

```
#include "mpi.h" // for C
#include "mpif.h" ! for Fortran
```

For *Fortran90*, many MPI installations also have an MPI module, so you can write

```
use mpi     ! pre 3.0
use mpi_f08 ! 3.0 standard
```

The internals of these files can be different between MPI installations, so you can not compile one file against one `mpi.h` file and another file, even with the same compiler on the same machine, against a different MPI.

Python note. It's easiest to

```
from mpi4py import MPI
```

MPL note 2. To compile MPL programs, add a line

```
#include <mpl/mpl.hpp>
```

to your file.

End of MPL note

2.2.2 Initialization / finalization

Every (useful) MPI program has to start with *MPI initialization* through a call to `MPI_Init` (figure 2.1), and have `MPI_Finalize` (figure 2.2) to finish the use of MPI in your program. The init call is different between the various languages.

In C, you can pass `argc` and `argv`, the arguments of a C language main program:

```
int main(int argc,char **argv) {
    ....
    return 0;
}
```

Figure 2.2 `MPI_Finalize`

Name	Param name	C type	F type	inout
`mpi_finalize (`				
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Finalize(`		
`) -> None`		

(It is allowed to pass `NULL` for these arguments.)

Fortran (before 2008) lacks this commandline argument handling, so `MPI_Init` lacks those arguments.

Python note. There are no initialize and finalize calls: the `import` statement performs the initialization.

MPL note 3. There is no initialization or finalize call.

> *MPL implementation note:* Initialization is done at the first `mpl::environment` method call, such as `comm_world`.
>
> *End of MPL note*

This may look a bit like declaring 'this is the parallel part of a program', but that's not true: again, the whole code is executed multiple times in parallel.

Exercise 2.2. Add the commands `MPI_Init` and `MPI_Finalize` to your code. Put three different print statements in your code: one before the init, one between init and finalize, and one after the finalize. Again explain the output.
Run your program on a large scale, using a batch job. Where does the output go? Experiment with
`MY_MPIRUN_OPTIONS="-prepend-rank" ibrun yourprogram`

Remark 2 *For hybrid MPI-plus-threads programming there is also a call `MPI_Init_thread`. For that, see section 40.2.*

2.2.2.1 Aborting an MPI run

Apart from `MPI_Finalize`, which signals a successful conclusion of the MPI run, an abnormal end to a run can be forced by `MPI_Abort` (figure 2.3). This aborts execution on all processes associated with the communicator, but many implementations simply abort all processes. The `value` parameter is returned to the environment.

2.2.2.2 Testing the initialized/finalized status

The commandline arguments `argc` and `argv` are only guaranteed to be passed to process zero, so the best way to pass commandline information is by a broadcast (section 3.3.3).

There are a few commands, such as `MPI_Get_processor_name`, that are allowed before `MPI_Init`.

Figure 2.3 `MPI_Abort`

Name	Param name	C type	F type	inout
`mpi_abort (`				
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator of tasks to abort			
	`errorcode`	`int`	`INTEGER`	`in`
	error code to return to invoking environment			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
void mpl::communicator::abort ( int ) const
```

Python:

Name	Param name	param type
`Comm.Abort(`		
	`self`	
	`errorcode`	`int = 0`
`) -> NoReturn`		

Figure 2.4 `MPI_Initialized`

Name	Param name	C type	F type	inout
`mpi_initialized (`				
	`flag`	`int*`	`LOGICAL`	`out`
	Flag is true if `MPI_INIT` *has been called and false otherwise*			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

If MPI is used in a library, MPI can have already been initialized in a main program. For this reason, one can test where `MPI_Init` has been called with `MPI_Initialized` (figure 2.4).

You can test whether `MPI_Finalize` has been called with `MPI_Finalized` (figure 2.5).

2.2.2.3 *Information about the run*

Once MPI has been initialized, the `MPI_INFO_ENV` object contains a number of key/value pairs describing run-specific information; see section 14.1.1.1.

2.2.2.4 *Commandline arguments*

The `MPI_Init` routines takes a reference to `argc` and `argv` for the following reason: the `MPI_Init` calls filters out the arguments to *mpirun* or *mpiexec*, thereby lowering the value of `argc` and elimitating some of the `argv` arguments.

On the other hand, the commandline arguments that are meant for `mpiexec` wind up in the `MPI_INFO_ENV` object as a set of key/value pairs; see section 14.1.1.

Figure 2.5 `MPI_Finalized`

Name	Param name	C type	F type	inout
`mpi_finalized (`				
	`flag` *true if MPI was finalized*	`int*`	`LOGICAL`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Figure 2.6 `MPI_Get_processor_name`

Name	Param name	C type	F type	inout
`mpi_get_processor_name (`				

	`name`
	`length: MPI_MAX_PROCESSOR_NAME`
	A unique specifier for the actual (as opposed to virtual) node.
		`char*`	`CHARACTER`	`out`

	`resultlen` *Length (in printable characters) of the result returned in name*	`int*`	`INTEGER`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Get_processor_name(`		
`) -> str`		

2.3 Processor identification

Since all processes in an MPI job are instantiations of the same executable, you'd think that they all execute the exact same instructions, which would not be terribly useful. You will now learn how to distinguish processes from each other, so that together they can start doing useful work.

2.3.1 Processor name

In the following exercise you will print out the hostname of each MPI process with `MPI_Get_processor_name` (figure 2.6) as a first way of distinguishing between processes.

Exercise 2.3. Now use the command `MPI_Get_processor_name` in between the init and finalize statement, and print out on what processor your process runs. Confirm that you are able to run a program that uses two different nodes.
The character buffer needs to be allocated by you, it is not created by MPI, with size at least `MPI_MAX_PROCESSOR_NAME`.

The character storage is provided by the user: the character array must be at least `MPI_MAX_PROCESSOR_NAME` characters long. The actual length of the name is returned in the `resultlen` parameter.

MPL note 4. The `processor_name` call is an environment method returning a `std::string`:

```
std::string mpl::environment::processor_name ();
```

End of MPL note

2.3.2 Communicators

First we need to introduce the concept of *communicator*, which is an abstract description of a group of processes. For now you only need to know about the existence of the `MPI_Comm` data type, and that there is a pre-defined communicator `MPI_COMM_WORLD` which describes all the processes involved in your parallel run.

In the procedural languages C, a *communicator* is a *variable* that is passed to most routines:

```
#include <mpi.h>
MPI_Comm comm = MPI_COMM_WORLD;
MPI_Send( /* stuff */ comm );
```

Fortran note. In Fortran, pre-2008 a communicator was an *opaque handle*, stored in an `Integer`. With *Fortran 2008*, communicators are derived types:

```
use mpi_f098
Type(MPI_Comm} :: comm = MPI_COMM_WORLD
call MPI_Send( ... comm )
```

Python note. In object-oriented languages, a communicator is an *object*, and rather than passing it to routines, the routines are often methods of the communicator object:

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
comm.Send( buffer, target )
```

MPL note 5. The naive way of declaring a communicator would be:

```
// commrank.cxx
mpl::communicator comm_world =
  mpl::environment::comm_world();
```

For the full source of this example, see section 2.7.2

calling the predefined environment method `comm_world`.

However, if the variable will always correspond to the world communicator, it is better to make it `const` and declare it to be a reference:

```
const mpl::communicator &comm_world =
  mpl::environment::comm_world();
```

For the full source of this example, see section 2.7.2

End of MPL note

MPL note 6. The communicator class has its copy operator deleted; however, copy initialization exists:

```
// commcompare.cxx
const mpl::communicator &comm =
  mpl::environment::comm_world();
cout << "same: " << boolalpha << (comm==comm) << endl;

mpl::communicator copy =
```

Figure 2.7 `MPI_Comm_size`

Name	Param name	C type	F type	inout
`mpi_comm_size (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`size` *number of processes in the group of comm*	`int*`	`INTEGER`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
int mpl::communicator::size ( ) const
```

Python:

```
MPI.Comm.Get_size(self)
```

```
  mpl::environment::comm_world();
cout << "copy: " << boolalpha << (comm==copy) << endl;

mpl::communicator init = comm;
cout << "init: " << boolalpha << (init==comm) << endl;
```

For the full source of this example, see section 2.7.3

(This outputs true/false/false respectively.)

MPL implementation note: The copy initializer performs an `MPI_Comm_dup`.

End of MPL note

MPL note 7. Pass communicators by reference to avoid communicator duplication:

```
// commpass.cxx
// BAD! this does a MPI_Comm_dup.
void comm_val( const mpl::communicator comm );

// correct!
void comm_ref( const mpl::communicator &comm );
```

End of MPL note

You will learn much more about communicators in chapter 7.

2.3.3 Process and communicator properties: rank and size

To distinguish between processes in a communicator, MPI provides two calls

1. `MPI_Comm_size` (figure 2.7) reports how many processes there are in all; and
2. `MPI_Comm_rank` (figure 2.8) states what the number of the process is that calls this routine.

If every process executes the `MPI_Comm_size` call, they all get the same result, namely the total number of processes in your run. On the other hand, if every process executes `MPI_Comm_rank`, they all get a different result, namely their own unique number, an integer in the range from zero to the number of processes minus 1. See figure 2.5. In other words, each process can find out 'I am process 5 out of a total of 20'.

Figure 2.8 `MPI_Comm_rank`

Name	Param name	C type	F type	inout
`mpi_comm_rank (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`rank` *rank of the calling process in group of comm*	`int*`	`INTEGER`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
int mpl::communicator::rank ( ) const
```

Python:

```
MPI.Comm.Get_rank(self)
```

Exercise 2.4. Write a program where each process prints out a message reporting its number, and how many processes there are:

```
Hello from process 2 out of 5!
```

Write a second version of this program, where each process opens a unique file and writes to it. *On some clusters this may not be advisable if you have large numbers of processors, since it can overload the file system.*

(There is a skeleton for this exercise under the name `commrank`.)

Exercise 2.5. Write a program where only the process with number zero reports on how many processes there are in total.

In object-oriented approaches to MPI, that is, `mpi4py` and *MPL*, the `MPI_Comm_rank` and `MPI_Comm_size` routines are methods of the communicator class:

Python note. Rank and size are methods of the communicator object. Note that their names are slightly different from the MPI standard names.

```
procid = comm.Get_rank()
nprocs = comm.Get_size()
```

MPL note 8. The rank of a process (by `mpl::communicator::rank`) and the size of a communicator (by `mpl::communicator::size`) are both methods of the `communicator` class:

```
const mpl::communicator &comm_world =
    mpl::environment::comm_world();
int procid = comm_world.rank();
int nprocs = comm_world.size();
```

End of MPL note

2.4 Functional parallelism

Now that processes can distinguish themselves from each other, they can decide to engage in different activities. In an extreme case you could have a code that looks like

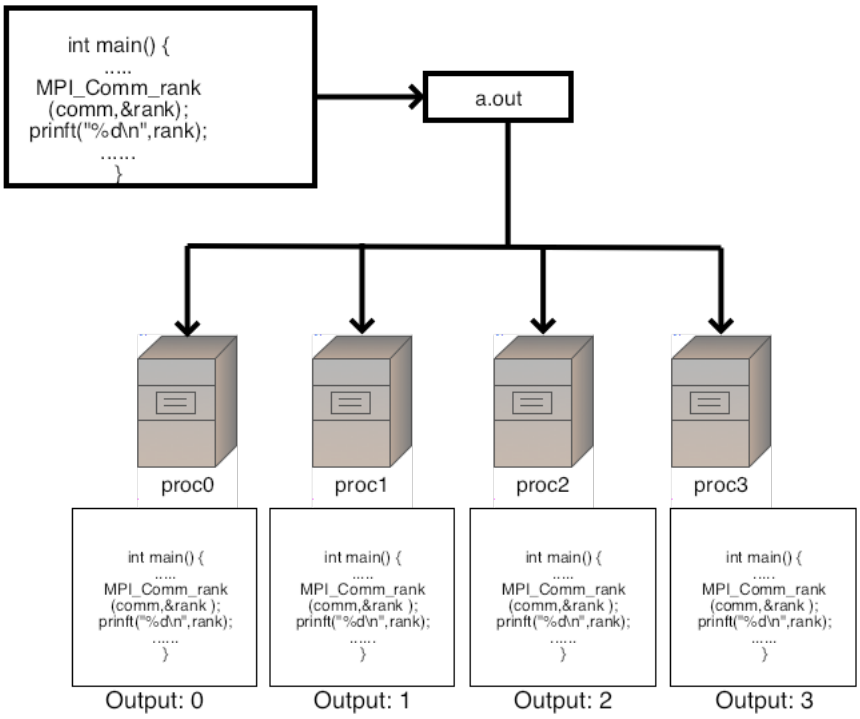


Figure 2.5: Parallel program that prints process rank

```
// climate simulation:
if (procid==0)
  earth_model();
else if (procid=1)
  sea_model();
else
  air_model();
```

Practice is a little more complicated than this. But we will start exploring this notion of processes deciding on their activity based on their process number.

Being able to tell processes apart is already enough to write some applications, without knowing any other MPI. We will look at a simple parallel search algorithm: based on its rank, a processor can find its section of a search space. For instance, in *Monte Carlo codes* a large number of random samples is generated and some computation performed on each. (This particular example requires each MPI process to run an independent random number generator, which is not entirely trivial.)

Exercise 2.6. Is the number $N = 2,000,000,111$ prime? Let each process test a disjoint set of integers, and print out any factor they find. You don't have to test all integers $< N$: any factor is at most $\sqrt{N} \approx 45,200$.
(Hint: `i%0` probably gives a runtime error.)
Can you find more than one solution?
(There is a skeleton for this exercise under the name `prime`.)

Remark 3 *Normally, we expect parallel algorithms to be faster than sequential. Now consider the above exercise. Suppose the number we are testing is divisible by some small prime number, but every process has a large block of numbers to test. In that case the sequential algorithm would have been faster than the parallel one. Food for thought.*

As another example, in *Boolean satisfiability* problems a number of points in a search space needs to be evaluated. Knowing a process's rank is enough to let it generate its own portion of the search space. The computation of the *Mandelbrot set* can also be considered as a case of functional parallelism. However, the image that is constructed is data that needs to be kept on one processor, which breaks the symmetry of the parallel run.

Of course, at the end of a functionally parallel run you need to summarize the results, for instance printing out some total. The mechanisms for that you will learn next.

2.5 Distributed computing and distributed data

One reason for using MPI is that sometimes you need to work on a single object, say a vector or a matrix, with a data size larger than can fit in the memory of a single processor. With distributed memory, each processor then gets a part of the whole data structure and only works on that.

So let's say we have a large array, and we want to distribute the data over the processors. That means that, with `p` processes and `n` elements per processor, we have a total of $n \cdot p$ elements.

```
int n;
double data[n];
```

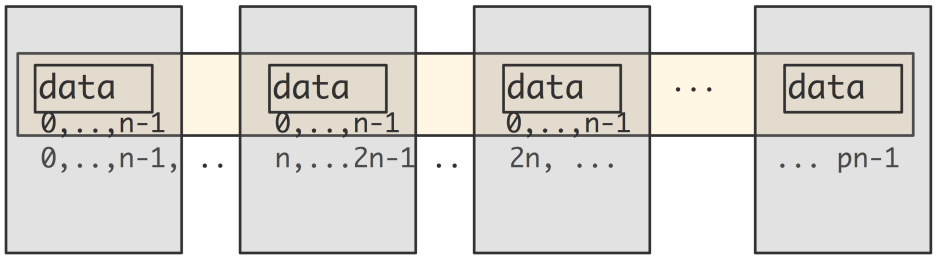


Figure 2.6: Local parts of a distributed array

We sometimes say that `data` is the local part of a *distributed array* with a total size of $n \cdot p$ elements. However, this array only exists conceptually: each processor has an array with lowest index zero, and you have to translate that yourself to an index in the global array. In other words, you have to write your code in such a way that it acts like you're working with a large array that is distributed over the processors, while actually manipulating only the local arrays on the processors.

Your typical code then looks like

```
int myfirst = .....;
for (int ilocal=0; ilocal<nlocal; ilocal++) {
   int iglobal = myfirst+ilocal;
   array[ilocal] = f(iglobal);
}
```

Exercise 2.7. Allocate on each process an array:

```
int my_ints[10];
```

and fill it so that process 0 has the integers $0 \cdots 9$, process 1 has $10 \cdots 19$, et cetera. Can you find a way to print the output in a nonmessy way?

If the array size is not perfectly divisible by the number of processors, we have to come up with a division that is uneven, but not too much. You could for instance, write

```
int Nglobal, // is something large
    Nlocal = Nglobal/ntids,
    excess = Nglobal%ntids;
if (mytid==ntids-1)
  Nlocal += excess;
```

Exercise 2.8. Argue that this strategy is not optimal. Can you come up with a better distribution? Load balancing is further discussed in HPSC-2.10.

2.6 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

Exercise 2.9. True or false: `mpicc` is a compiler.

Exercise 2.10. T/F?

1. In C, the result of `MPI_Comm_rank` is a number from zero to number-of-processes-minus-one, inclusive.
2. In Fortran, the result of `MPI_Comm_rank` is a number from one to number-of-processes, inclusive.

Exercise 2.11. What is the function of a hostfile?

2.7 Sources used in this chapter

2.7.1 Listing of code header

2.7.2 Listing of code examples/mpi/mpl/commrank.cxx

```
#include <cstdlib>
#include <iostream>
#include <mpl/mpl.hpp>

int main() {
#if 1
  mpl::communicator comm_world =
    mpl::environment::comm_world();
#else
  const mpl::communicator &comm_world =
    mpl::environment::comm_world();
#endif
  std::cout << "Hello world! I am running on \""
        << mpl::environment::processor_name()
        << "\". My rank is "
       << comm_world.rank()
       << " out of "
        << comm_world.size() << " processes.\n" << std::endl;
  return EXIT_SUCCESS;
}
```

2.7.3 Listing of code examples/mpi/mpl/commcompare.cxx

```
#include <cstdlib>
#include <iostream>
#include <iomanip>
using namespace std;

#include <mpl/mpl.hpp>

int main() {

  const mpl::communicator &comm =
    mpl::environment::comm_world();
  cout << "same: " << boolalpha << (comm==comm) << endl;

  mpl::communicator copy =
    mpl::environment::comm_world();
  cout << "copy: " << boolalpha << (comm==copy) << endl;

  mpl::communicator init = comm;
  cout << "init: " << boolalpha << (init==comm) << endl;

  // WRONG: copy = comm;
```

```
  // error: overload resolution selected deleted operator '='

  {
    mpl::communicator init;
    // WRONG: init = mpl::environment::comm_world();
    // error: overload resolution selected deleted operator '='
  }

  auto eq = comm.compare(copy);
  cout << static_cast<int>(eq) << endl;

  return EXIT_SUCCESS;
}
```

Chapter 3

MPI topic: Collectives

A certain class of MPI routines are called 'collective', or more correctly: 'collective on a communicator'. This means that if process one in that communicator calls that routine, they all need to call that routine. In this chapter we will discuss collective routines that are about combining the data on all processes in that communicator, but there are also operations such as opening a shared file that are collective, which will be discussed in a later chapter.

3.1 Working with global information

If all processes have individual data, for instance the result of a local computation, you may want to bring that information together, for instance to find the maximal computed value or the sum of all values. Conversely, sometimes one processor has information that needs to be shared with all. For this sort of operation, MPI has *collectives*.

There are various cases, illustrated in figure 3.1, which you can (sort of) motivate by considering some classroom activities:

- The teacher tells the class when the exam will be. This is a *broadcast*: the same item of information goes to everyone.
- After the exam, the teacher performs a *gather* operation to collect the invidivual exams.
- On the other hand, when the teacher computes the average grade, each student has an individual number, but these are now combined to compute a single number. This is a *reduction*.
- Now the teacher has a list of grades and gives each student their grade. This is a *scatter* operation, where one process has multiple data items, and gives a different one to all the other processes.

This story is a little different from what happens with MPI processes, because these are more symmetric; the process doing the reducing and broadcasting is no different from the others. Any process can function as the *root process* in such a collective.

Exercise 3.1. How would you realize the following scenarios with MPI collectives?

1. Let each process compute a random number. You want to print the maximum of these numbers to your screen.

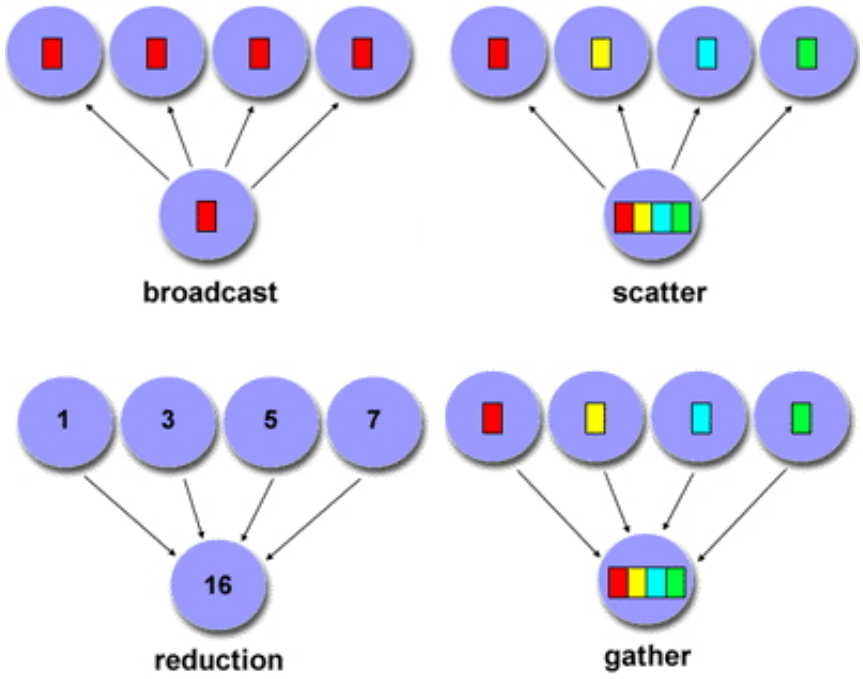


Figure 3.1: The four most common collectives

2. Each process computes a random number again. Now you want to scale these numbers by their maximum.
3. Let each process compute a random number. You want to print on what processor the maximum value is computed.

Think about time and space complexity of your suggestions.

3.1.1 Practical use of collectives

Collectives are quite common in scientific applications. For instance, if one process reads data from disc or the commandline, it can use a broadcast or a gather to get the information to other processes. Likewise, at the end of a program run, a gather or reduction can be used to collect summary information about the program run.

However, a more common scenario is that the result of a collective is needed on all processes.

Consider the computation of the *standard deviation*:

$$\sigma = \sqrt{\frac{1}{N-1}\sum_{i}^{N}(x_i - \mu)^2} \qquad \text{where} \qquad \mu = \frac{\sum_i^N x_i}{N}$$

and assume that every process stores just one x_i value.

1. The calculation of the average μ is a reduction, since all the distributed values need to be added.

2. Now every process needs to compute $x_i - \mu$ for its value x_i, so μ is needed everywhere. You can compute this by doing a reduction followed by a broadcast, but it is better to use a so-called *allreduce* operation, which does the reduction and leaves the result on all processors.
3. The calculation of $\sum_i (x_i - \mu)$ is another sum of distributed data, so we need another reduction operation. Depending on whether each process needs to know σ, we can again use an allreduce.

3.1.2 Synchronization

Collectives are operations that involve all processes in a communicator. A collective is a single call, and it blocks on all processors, meaning that a process calling a collective cannot proceed until the other processes have similarly called the collective.

That does not mean that all processors exit the call at the same time: because of implementational details and network latency they need not be synchronized in their execution. However, semantically we can say that a process can not finish a collective until every other process has at least started the collective.

In addition to these collective operations, there are operations that are said to be 'collective on their communicator', but which do not involve data movement. Collective then means that all processors must call this routine; not to do so is an error that will manifest itself in 'hanging' code. One such example is `MPI_File_open`.

3.1.3 Collectives in MPI

We will now explain the MPI collectives in the following order.

Allreduce We use the allreduce as an introduction to the concepts behind collectives; section 3.2. As explained above, this routines serves many practical scenarios.

Broadcast and reduce We then introduce the concept of a root in the reduce (section 3.3.1) and broadcast (section 3.3.3) collectives.

- Sometimes you want a reduction with partial results, where each processor computes the sum (or other operation) on the values of lower-numbered processors. For this, you use a *scan* collective (section 3.4).

Gather and scatter The gather/scatter collectives deal with more than a single data item; section 3.5.

There are more collectives or variants on the above.

- If every processor needs to broadcast to every other, you use an *all-to-all* operation (section 3.6).
- The reduce-scatter is a lesser known combination of collectives; section 3.7.
- A barrier is an operation that makes all processes wait until every process has reached the barrier (section 3.8).
- If you want to gather or scatter information, but the contribution of each processor is of a different size, there are 'variable' collectives; they have a `v` in the name (section 3.9).

Finally, there are some advanced topics in collectives.

- User-defined reduction operators; section 3.10.2.
- Nonblocking collectives; section 3.11.
- We briefly discuss performance aspects of collectives in section 3.12.
- We discuss synchronization aspects in section 3.13.

Figure 3.1 `MPI_Allreduce`

Name	Param name	C type	F type	inout
`mpi_allreduce (`				
	`sendbuf` *starting address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`recvbuf` *starting address of receive buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`count` *number of elements in send buffer*	`int`	`INTEGER`	`in`
	`datatype` *data type of elements of send buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`op` *operation*	`MPI_Op`	`TYPE(MPI_Op)`	`in`
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T , typename F >
void mpl::communicator::allreduce
   ( F,const T &, T & ) const;
   ( F,const T *, T *,
     const contiguous_layout< T > & ) const;
   ( F,T & ) const;
   ( F,T *, const contiguous_layout< T > & ) const;
F : reduction function
T : type
```

Python:

Name	Param name	param type
`Comm.allreduce(`		
	`self`	
	`sendobj`	`Any`
	`op`	`Union[Op, Callable[[Any, Any], Any]] = Ellipsis`
`) -> Any`		

3.2 Reduction

3.2.1 Reduce to all

Above we saw a couple of scenarios where a quantity is reduced, with all proceses getting the result. The MPI call for this is `MPI_Allreduce` (figure 3.1).

Example: we give each process a random number, and sum these numbers together. The result should be approximately $1/2$ times the number of processes.

```
// allreduce.c
float myrandom,sumrandom;
myrandom = (float) rand()/(float)RAND_MAX;
// add the random variables together
MPI_Allreduce(&myrandom,&sumrandom,
                1,MPI_FLOAT,MPI_SUM,comm);
// the result should be approx nprocs/2:
```

```
if (procno==nprocs-1)
  printf("Result %6.9f compared to .5\n",sumrandom/nprocs);
```

For the full source of this example, see section 3.16.2

Exercise 3.2. Let each process compute a random number, and compute the sum of these numbers using the `MPI_Allreduce` routine.

$$\xi = \sum_i x_i$$

Each process then scales its value by this sum.

$$x_i' \leftarrow x_i/\xi$$

Compute the sum of the scaled numbers

$$\xi' = \sum_i x_i'$$

and check that it is 1.
(There is a skeleton for this exercise under the name `randommax`.)

Exercise 3.3. Implement a (very simple-minded) Fourier transform: if f is a function on the interval $[0,1]$, then the n-th Fourier coefficient is

$$f_n \hat{=} \int_0^1 f(t)e^{-2\pi x}\,dx$$

which we approximate by

$$f_n \hat{=} \sum_{i=0}^{N-1} f(ih)e^{-in\pi/N}$$

- Make one distributed array for the e^{-inh} coefficients,
- make one distributed array for the $f(ih)$ values
- calculate a couple of coefficients

Exercise 3.4. In the previous exercise you worked with a distributed array, computing a local quantity and combining that into a global quantity. Why is it not a good idea to gather the whole distributed array on a single processor, and do all the computation locally?

MPL note 9. The usual reduction operators are given as templated operators:

```
float
  xrank = static_cast<float>( comm_world.rank() ),
  xreduce;
// separate recv buffer
comm_world.allreduce(mpl::plus<float>(), xrank,xreduce);
// in place
comm_world.allreduce(mpl::plus<float>(), xrank);
```

For the full source of this example, see section 3.16.3

Note the parentheses after the operator. Also note that the operator comes first, not last.

MPL implementation note: The reduction operator has to be compatible with `T(T,T)>`

End of MPL note

For more about operators, see section 3.10.

3.2.2 Inner product as allreduce

One of the more common applications of the reduction operation is the *inner product* computation. Typically, you have two vectors x, y that have the same distribution, that is, where all processes store equal parts of x and y. The computation is then

```
local_inprod = 0;
for (i=0; i<localsize; i++)
  local_inprod += x[i]*y[i];
MPI_Allreduce( &local_inprod, &global_inprod, 1,MPI_DOUBLE ... )
```

Exercise 3.5. The *Gram-Schmidt* method is a simple way to orthogonalize two vectors:

$$u \leftarrow u - (u^t v)/(u^t u)$$

Implement this, and check that the result is indeed orthogonal.
Suggestion: fill v with the values $\sin 2nh\pi$ where $n = 2\pi/N$, and u with $\sin 2nh\pi + \sin 4nh\pi$. What does u become after orthogonalization?

3.2.3 Reduction operations

Several `MPI_Op` values are pre-defined. For the list, see section 3.10.1.

For use in reductions and scans it is possible to define your own operator.

```
MPI_Op_create( MPI_User_function *func, int commute, MPI_Op *op);
```

For more details, see section 3.10.2.

3.2.4 Data buffers

Collectives are the first example you see of MPI routines that involve transfer of user data. Here, and in every other case, you see that the data description involves:

- A buffer. This can be a scalar or an array.
- A datatype. This describes whether the buffer contains integers, single/double floating point numbers, or more complicated types, to be discussed later.
- A count. This states how many of the elements of the given datatype are in the send buffer, or will be received into the receive buffer.

These three together describe what MPI needs to send through the network.

In the various languages such a buffer/count/datatype triplet is specified in different ways.

First of all, in *C* the *buffer* is always an *opaque handle*, that is, a `void*` parameter to which you supply an address. This means that an MPI call can take two forms.

For scalars we need to use the ampersand operator to take the address:

```
float x,y;
MPI_Allreduce( &x,&y,1,MPI_FLOAT, ... );
```

But for arrays we use the fact that arrays and addresses are more-or-less equivalent in:

```
float xx[2],yy[2];
MPI_Allreduce( xx,yy,2,MPI_FLOAT, ... );
```

You could *cast* the buffers and write:

```
MPI_Allreduce( (void*)&x,(void*)&y,1,MPI_FLOAT, ... );
MPI_Allreduce( (void*)xx,(void*)yy,2,MPI_FLOAT, ... );
```

but that is not necessary. The compiler will not complain if you leave out the cast.

C++ note. Buffer treatment Treatment of scalars in C++ is the same as in C. However, for arrays you have the choice between C-style arrays, and `std::vector` or `std::array`. For the latter there are two ways of dealing with buffers:

```
vector<float> xx(25);
MPI_Send( xx.data(),25,MPI_FLOAT, .... );
MPI_Send( &xx[0],25,MPI_FLOAT, .... );
```

Fortran note. In *Fortran* parameters are always passed by reference, so the *buffer* is treated the same way:

```
Real*4 :: x
Real*4,dimension(2) :: xx
call MPI_Allreduce( x,1,MPI_REAL4, ... )
call MPI_Allreduce( xx,2,MPI_REAL4, ... )
```

In discussing OO languages, we first note that the official C++ Application Programmer Interface (API) has been removed from the standard.

Specification of the buffer/count/datatype triplet is not needed explicitly in OO languages.

Python note. Most MPI routines in Python have both a native and a numpy variant. The native variant looks the most 'pythonic', and is very flexible, but is usually demonstrably inefficient.

```
## allreduce.py
random_number = random.randint(1,random_bound)
# native mode send
max_random = comm.allreduce(random_number,op=MPI.MAX)
```

For the full source of this example, see section 3.16.4

In the numpy variant, all *buffers* are *numpy* objects, which carry information about their type and size. For scalar reductions this means we have to create an array for the receive buffer, even though only one element is used.

```
myrandom = np.empty(1,dtype=np.int)
myrandom[0] = random_number
allrandom = np.empty(nprocs,dtype=np.int)
# numpy mode send
comm.Allreduce(myrandom,allrandom[:1],op=MPI.MAX)
```

For the full source of this example, see section 3.16.4

MPL note 10. Buffer type handling is done through polymorphism and templating: no explicit indiation of types.

Scalars are handled as such:

```
float x,y;
comm.bcast( 0,x ); // note: root first
comm.allreduce( mpl::plus<float>,x,y );
```

where the reduction function needs to be compatible with the type of the buffer.

End of MPL note

MPL note 11. If your buffer is a `std::vector` you need to take the `.data()` component of it:

```
vector<float> xx(2),yy(2);
comm.allreduce( mpl::plus<float>(),
    xx.data(), yy.data(), mpl::contiguous_layout<float>(2) );
```

The `contiguous_layout` is a 'derived type'; this will be discussed in more detail elsewhere (see note 39 and later). For now, interpret it as a way of indicating the count/type part of a buffer specification.

End of MPL note

MPL note 12. Many MPL routines have a way of specifying the buffer(s) through a `begin` and `end` iterator.

```
// sendrange.cxx
vector<double> v(15);
  comm_world.send(v.begin(), v.end(), 1);  // send to rank 1
  comm_world.recv(v.begin(), v.end(), 0);  // receive from rank 0
```

For the full source of this example, see section 6.8.14

End of MPL note

3.3 Rooted collectives: broadcast, reduce

In some scenarios there is a certain process that has a privileged status.

- One process can generate or read in the initial data for a program run. This then needs to be communicated to all other processes.
- At the end of a program run, often one process needs to output some summary information.

This process is called the *root* process, and we will now consider routines that have a root.

3.3.1 Reduce to a root

In the broadcast operation a single data item was communicated to all processes. A reduction operation with `MPI_Reduce` (figure 3.2) goes the other way: each process has a data item, and these are all brought together into a single item.

Here are the essential elements of a reduction operation:

```
MPI_Reduce( senddata, recvdata..., operator,
    root, comm );
```

- There is the original data, and the data resulting from the reduction. It is a design decision of MPI that it will not by default overwrite the original data. The send data and receive data are of the same size and type: if every processor has one real number, the reduced result is again one real number.
- It is possible to indicate explicitly that a single buffer is used, and thereby the original data overwritten; see section 3.3.2 for this 'in place' mode.
- There is a reduction operator. Popular choices are `MPI_SUM`, `MPI_PROD` and `MPI_MAX`, but complicated operators such as finding the location of the maximum value exist. (For the full list, see section 3.10.1.) You can also define your own operators; section 3.10.2.
- There is a root process that receives the result of the reduction. Since the nonroot processes do not receive the reduced data, they can actually leave the receive buffer undefined.

```
// reduce.c
float myrandom = (float) rand()/(float)RAND_MAX,
  result;
int target_proc = nprocs-1;
// add all the random variables together
MPI_Reduce(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,
           target_proc,comm);
// the result should be approx nprocs/2:
if (procno==target_proc)
  printf("Result %6.3f compared to nprocs/2=%5.2f\n",
         result,nprocs/2.);
```

For the full source of this example, see section 3.16.6

Exercise 3.6. Write a program where each process computes a random number, and process 0 finds and prints the maximum generated value. Let each process print its value, just to check the correctness of your program.

Collective operations can also take an array argument, instead of just a scalar. In that case, the operation is applied pointwise to each location in the array.

Exercise 3.7. Create on each process an array of length 2 integers, and put the values 1, 2 in it on each process. Do a sum reduction on that array. Can you predict what the result should be? Code it. Was your prediction right?

Figure 3.2 `MPI_Reduce`

Name	Param name	C type	F type	inout
`mpi_reduce (`				
	`sendbuf` *address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`recvbuf` *address of receive buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`count` *number of elements in send buffer*	`int`	`INTEGER`	`in`
	`datatype` *data type of elements of send buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`op` *reduce operation*	`MPI_Op`	`TYPE(MPI_Op)`	`in`
	`root` *rank of root process*	`int`	`INTEGER`	`in`
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
void mpl::communicator::reduce
    // root, in place
    ( F  f,int  root_rank,T &  sendrecvdata ) const
    ( F  f,int  root_rank,T *  sendrecvdata,const contiguous_layout< T > &  l ) const
    // non-root
    ( F  f,int  root_rank,const T &  senddata ) const
    ( F  f,int  root_rank,
            const T *  sendrecvdata,const contiguous_layout< T > &  l ) const
    // general
    ( F  f,int  root_rank,const T &  senddata,T &  recvdata ) const
    ( F  f,int  root_rank,
            const T *  senddata,T *  recvdata,const contiguous_layout< T > &  l ) const
```

Python:

Name	Param name	param type
`Comm.reduce(`		
	`self`	
	`sendobj`	`Any`
	`op`	`Union[Op, Callable[[Any, Any], Any]] = Ellipsis`
	`root`	`int = 0`
`) -> Optional[Any]`		

3.3.2 Reduce in place

By default MPI will not overwrite the original data with the reduction result, but you can tell it to do so using the MPI_IN_PLACE specifier:

```
// allreduceinplace.c
for (int irand=0; irand<nrandoms; irand++)
        myrandoms[irand] = (float) rand()/(float)RAND_MAX;
// add all the random variables together
MPI_Allreduce(MPI_IN_PLACE,myrandoms,
                      nrandoms,MPI_FLOAT,MPI_SUM,comm);
```

For the full source of this example, see section 3.16.7

Now every process only has a receive buffer, so this has the advantage of saving half the memory. Each process puts its input values in the receive buffer, and these are overwritten by the reduced result.

The above example used MPI_IN_PLACE in MPI_Allreduce; in MPI_Reduce it's little tricky. The reasoning is a follows:

- In MPI_Reduce every process has a buffer to contribute, but only the root needs a receive buffer. Therefore, MPI_IN_PLACE takes the place of the receive buffer on any processor except for the root ...
- ... while the root, which needs a receive buffer, has MPI_IN_PLACE takes the place of the send buffer. In order to contribute its value, the root needs to put this in the receive buffer.

Here is one way you could write the in-place version of MPI_Reduce:

```
if (procno==root)
  MPI_Reduce(MPI_IN_PLACE,myrandoms,
                    nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
else
  MPI_Reduce(myrandoms,MPI_IN_PLACE,
                    nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
```

For the full source of this example, see section 3.16.7

However, as a point of style, having different versions of a a collective in different branches of a condition is infelicitous. The following may be preferable:

```
float *sendbuf,*recvbuf;
if (procno==root) {
  sendbuf = MPI_IN_PLACE; recvbuf = myrandoms;
} else {
  sendbuf = myrandoms; recvbuf = MPI_IN_PLACE;
}
MPI_Reduce(sendbuf,recvbuf,
          nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
```

For the full source of this example, see section 3.16.7

In Fortran you can not do these address calculations. You can use the solution with a conditional:

```
// reduceinplace.F90
call random_number(mynumber)
target_proc = ntids-1;
```

```
! add all the random variables together
if (mytid.eq.target_proc) then
   result = mytid
   call MPI_Reduce(MPI_IN_PLACE,result,1,MPI_REAL,MPI_SUM,&
        target_proc,comm,err)
else
   mynumber = mytid
   call MPI_Reduce(mynumber,result,1,MPI_REAL,MPI_SUM,&
        target_proc,comm,err)
end if
```

For the full source of this example, see section 3.16.8

but you can also solve this with pointers:

```
// reduceinplaceptr.F90
in_place_val = MPI_IN_PLACE
if (mytid.eq.target_proc) then
   result_ptr => result
   mynumber_ptr => in_place_val
   result_ptr = mytid
else
   mynumber_ptr => mynumber
   result_ptr = in_place_val
   mynumber_ptr = mytid
end if
call MPI_Reduce(mynumber_ptr,result_ptr,1,MPI_REAL,MPI_SUM,&
     target_proc,comm,err)
```

For the full source of this example, see section 3.16.9

Python note. The value `MPI.IN_PLACE` can be used for the send buffer:

```
## allreduceinplace.py
myrandom = np.empty(1,dtype=np.int)
myrandom[0] = random_number

comm.Allreduce(MPI.IN_PLACE,myrandom,op=MPI.MAX)
```

For the full source of this example, see section 3.16.10

MPL note 13. The in-place variant is activated by specifying only one instead of two buffer arguments.

```
float
  xrank = static_cast<float>( comm_world.rank() ),
  xreduce;
// separate recv buffer
comm_world.allreduce(mpl::plus<float>(), xrank,xreduce);
// in place
comm_world.allreduce(mpl::plus<float>(), xrank);
```

For the full source of this example, see section 3.16.3

Reducing a buffer requires specification of a `contiguous_layout`:

```
// collectbuffer.cxx
float
```

```
  xrank = static_cast<float>( comm_world.rank() );
vector<float> rank2p2p1{ 2*xrank,2*xrank+1 },reduce2p2p1{0,0};
mpl::contiguous_layout<float> two_floats(rank2p2p1.size());
comm_world.allreduce(mpl::plus<float>(), rank2p2p1.data(),reduce2p2p1,two_floats);
if ( iprint )
  cout << "Got: " << reduce2p2p1.at(0) << ","
       << reduce2p2p1.at(1) << endl;
```

For the full source of this example, see section 3.16.11

End of MPL note

MPL note 14. There is a separate variant for non-root usage of rooted collectives:

```
// scangather.cxx
if (procno==0) {
  comm_world.reduce
    ( mpl::plus<int>(),0,my_number_of_elements,total_number_of_elements );
} else {
  comm_world.reduce
    ( mpl::plus<int>(),0,my_number_of_elements );
}
```

End of MPL note

3.3.3 Broadcast

A broadcast models the scenario where one process, the 'root' process, owns some data, and it communicates it to all other processes.

The broadcast routine `MPI_Bcast` (figure 3.3) has the following structure:

```
MPI_Bcast( data..., root , comm);
```

Here:

- There is only one buffer, the send buffer. Before the call, the root process has data in this buffer; the other processes allocate a same sized buffer, but for them the contents are irrelevant.
- The root is the process that is sending its data. Typically, it will be the root of a broadcast tree.

Example: in general we can not assume that all processes get the commandline arguments, so we broadcast them from process 0.

```
// init.c
if (procno==0) {
  if ( argc==1 || // the program is called without parameter
       ( argc>1 && !strcmp(argv[1],"-h") ) // user asked for help
       ) {
    printf("\nUsage: init [0-9]+\n");
    MPI_Abort(comm,1);
  }
  input_argument = atoi(argv[1]);
}
MPI_Bcast(&input_argument,1,MPI_INT,0,comm);
```

Figure 3.3 `MPI_Bcast`

Name	Param name	C type	F type	inout
`mpi_bcast (`				
	`buffer` *starting address of buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`inout`
	`count` *number of entries in buffer*	`int`	`INTEGER`	`in`
	`datatype` *data type of buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`root` *rank of broadcast root*	`int`	`INTEGER`	`in`
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T >
void mpl::communicator::bcast
   ( int  root, T &  data ) const
   ( int  root, T *  data, const layout< T > &  l ) const
```

Python:

Name	Param name	param type
`Comm.bcast(`		
	`self`	
	`obj`	`Any`
	`root`	`int = 0`
`) -> Any`		

For the full source of this example, see section 3.16.12

Python note. In python it is both possible to send objects, and to send more C-like buffers. The two possibilities correspond (see section 1.5.4) to different routine names; the buffers have to be created as numpy objects.

We illustrate both the native and numpy variants. In the native variant the result is given as a function return; in the numpy variant the send buffer is reused.

```
## bcast.py
# first native
if procid==root:
    buffer = [ 5.0 ] * dsize
else:
    buffer = [ 0.0 ] * dsize
buffer = comm.bcast(obj=buffer,root=root)
if not reduce( lambda x,y:x and y,
               [ buffer[i]==5.0 for i in  range(len(buffer)) ] ):
    print( "Something wrong on proc %d: native buffer <<%s>>" \
           % (procid,str(buffer)) )

# then with NumPy
buffer = np.arange(dsize, dtype=np.float64)
```

```
if procid==root:
    for i in range(dsize):
        buffer[i] = 5.0
comm.Bcast( buffer,root=root )
if not all( buffer==5.0 ):
    print( "Something wrong on proc %d: numpy buffer <<%s>>" \
           % (procid,str(buffer)) )
else:
    if procid==root:
        print("Success.")
```

For the full source of this example, see section 3.16.13

MPL note 15. The broadcast call comes in two variants, with scalar argument and general layout:

```
 template<typename T >
void mpl::communicator::bcast
   ( int root_rank, T &data ) const;
void mpl::communicator::bcast
   ( int root_rank, T *data, const layout< T > &l ) const;
```

Note that the root argument comes first.

End of MPL note

For the following exercise, study figure 3.2.

Exercise 3.8. The *Gauss-Jordan algorithm* for solving a linear system with a matrix A (or computing its inverse) runs as follows:

for pivot $k = 1, \ldots, n$
  let the vector of scalings $\ell_i^{(k)} = A_{ik}/A_{kk}$
  for row $r \neq k$
    for column $c = 1, \ldots, n$
      $A_{rc} \leftarrow A_{rc} - \ell_r^{(k)} A_{kc}$

where we ignore the update of the righthand side, or the formation of the inverse. Let a matrix be distributed with each process storing one column. Implement the Gauss-Jordan algorithm as a series of broadcasts: in iteration k process k computes and broadcasts the scaling vector $\{\ell_i^{(k)}\}_i$. Replicate the right-hand side on all processors.
(There is a skeleton for this exercise under the name `jordan`.)

Exercise 3.9. Add partial pivoting to your implementation of Gauss-Jordan elimination. Change your implementation to let each processor store multiple columns, but still do one broadcast per column. Is there a way to have only one broadcast per processor?

3.4 Scan operations

The `MPI_Scan` operation also performs a reduction, but it keeps the partial results. That is, if processor i contains a number x_i, and $\oplus$ is an operator, then the scan operation leaves $x_0 \oplus \cdots \oplus x_i$ on processor i. This

Initial:

matrix			sol	rhs	action
2	2	13	1	17	
4	5	32	1	41	
-2	-3	-16	1	-21	

Step 1:

matrix			sol	rhs	action
2	2	13	1	17	take this row
4	5	32	1	41	
-2	-3	-16	1	-21	

Step 2:

matrix			sol	rhs	action
2	2	13	1	17	take this row
↓	↓	↓			
4	5	32	1	41	minus ×2
-2	-3	-16	1	-21	

Step 3:

matrix			sol	rhs	action
2	2	13	1	17	take this row
0	1	6	1	7	
-2	-3	-16	1	-21	

Step 4:

matrix			sol	rhs	action
2	2	13	1	17	take this row
▯	▯	▯			
0	1	6	1	7	
-2	-3	-16	1	-21	plus ×1

Step 5:

matrix			sol	rhs	action
2	2	13	1	17	take this row
0	1	6	1	7	
0	-1	-3	1	-4	

Step 6:

matrix			sol	rhs	action
2	2	13	1	17	first column done
0	1	6	1	7	
0	-1	-3	1	-4	

Step 7:

matrix			sol	rhs	action
2	2	13	1	17	
0	1	6	1	7	take this row
0	-1	-3	1	-4	

Step 8:

matrix			sol	rhs	action
2	**2**	13	1	17	minus ×2
↑	↑	↑			
0	**1**	6	1	7	take this row
0	-1	-3	1	-4	

Step 9:

matrix			sol	rhs	action
2	0	1	1	3	
0	1	6	1	7	take this row
0	-1	-3	1	-4	

Step 10:

matrix			sol	rhs	action
2	0	1	1	3	
0	**1**	6	1	7	take this row
↓	↓	↓			
0	**-1**	-3	1	-4	plus ×1

Step 11:

matrix			sol	rhs	action
2	0	1	1	3	
0	1	6	1	7	take this row
0	0	3	1	3	

Step 12:

matrix			sol	rhs	action
2	0	1	1	3	
0	1	6	1	7	second column done
0	0	3	1	3	

Step 13:

matrix			sol	rhs	action
2	0	1	1	3	
0	1	6	1	7	
0	0	3	1	3	take this row

Step 14:

matrix			sol	rhs	action
2	0	**1**	1	3	minus ×1/3
0	1	6	1	7	
▯	▯	▯			
0	0	**3**	1	3	take this row

Step 15:

matrix			sol	rhs	action
2	0	0	1	2	
0	1	6	1	7	
0	0	3	1	3	take this row

Step 16:

matrix			sol	rhs	action
2	0	0	1	2	
0	1	**6**	1	7	minus ×2
↑	↑	↑			
0	0	**3**	1	3	take this row

Step 17:

matrix			sol	rhs	action
2	0	0	1	2	
0	1	0	1	1	
0	0	3	1	3	take this row

Step 18:

matrix			sol	rhs	action
2	0	0	1	2	
0	1	0	1	1	
0	0	3	1	3	third column done

Finished:

matrix			sol	rhs	action
2	0	0	1	2	
0	1	0	1	1	
0	0	3	1	3	

Figure 3.2: Gauss-Jordan elimination example

Figure 3.4 `MPI_Scan`

Name	Param name	C type	F type	inout
`mpi_scan (`				
	`sendbuf` *starting address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`recvbuf` *starting address of receive buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`count` *number of elements in input buffer*	`int`	`INTEGER`	`in`
	`datatype` *data type of elements of input buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`op` *operation*	`MPI_Op`	`TYPE(MPI_Op)`	`in`
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T , typename F >
void mpl::communicator::scan
   ( F,const T &, T & ) const;
   ( F,const T *, T *,
     const contiguous_layout< T > & ) const;
   ( F,T & ) const;
   ( F,T *, const contiguous_layout< T > & ) const;
F : reduction function
T : type
```

Python:

Name	Param name	param type
`Intracomm.scan(`		
	`self`	
	`sendobj`	`Any`
	`op`	`Union[Op, Callable[[Any, Any], Any]] = Ellipsis`
`) -> Any`		

type of operation is often called a *prefix operation*; see HPSC-19.

The `MPI_Scan` (figure 3.4) routine is an *inclusive scan* operation, meaning that it includes the data on the process itself; `MPI_Exscan` (figure 3.6) is exclusive, and does not include the data on the calling process.

process :	0	1	2	$\cdots$	$p-1$
data :	x_0	x_1	x_2	$\cdots$	x_{p-1}
inclusive :	x_0	$x_0 \oplus x_1$	$x_0 \oplus x_1 \oplus x_2$	$\cdots$	$\oplus_{i=0}^{p-1} x_i$
exclusive :	unchanged	x_0	$x_0 \oplus x_1$	$\cdots$	$\oplus_{i=0}^{p-2} x_i$

```
// scan.c
// add all the random variables together
MPI_Scan(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,comm);
// the result should be approaching nprocs/2:
```

Figure 3.6 `MPI_Exscan`

Name	Param name	C type	F type	inout
`mpi_exscan (`				
	`sendbuf` *starting address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`recvbuf` *starting address of receive buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`count` *number of elements in input buffer*	`int`	`INTEGER`	`in`
	`datatype` *data type of elements of input buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`op` *operation*	`MPI_Op`	`TYPE(MPI_Op)`	`in`
	`comm` *intra-communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T , typename F >
void mpl::communicator::exscan
   ( F,const T &, T & ) const;
   ( F,const T *, T *,
     const contiguous_layout< T > & ) const;
   ( F,T & ) const;
   ( F,T *, const contiguous_layout< T > & ) const;
F : reduction function
T : type
```

Python:

Name	Param name	param type
`Intracomm.exscan(`		
	`self`	
	`sendobj`	`Any`
	`op`	`Union[Op, Callable[[Any, Any], Any]] = Ellipsis`
`) -> Any`		

```
if (procno==nprocs-1)
  printf("Result %6.3f compared to nprocs/2=%5.2f\n",
         result,nprocs/2.);
```

For the full source of this example, see section 3.16.14

In python native mode the result is a function return value, with numpy the result is passed as the second parameter.

```
## scan.py
mycontrib = 10+random.randint(1,nprocs)
myfirst = 0
mypartial = comm.scan(mycontrib)
sbuf = np.empty(1,dtype=np.int)
rbuf = np.empty(1,dtype=np.int)
sbuf[0] = mycontrib
comm.Scan(sbuf,rbuf)
```

Figure 3.6 `MPI_Exscan`

Name	Param name	C type	F type	inout
`mpi_exscan (`				
	`sendbuf` *starting address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`recvbuf` *starting address of receive buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`count` *number of elements in input buffer*	`int`	`INTEGER`	`in`
	`datatype` *data type of elements of input buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`op` *operation*	`MPI_Op`	`TYPE(MPI_Op)`	`in`
	`comm` *intra-communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T , typename F >
void mpl::communicator::exscan
   ( F,const T &, T & ) const;
   ( F,const T *, T *,
     const contiguous_layout< T > & ) const;
   ( F,T & ) const;
   ( F,T *, const contiguous_layout< T > & ) const;
F : reduction function
T : type
```

Python:

Name	Param name	param type
`Intracomm.exscan(`		
	`self`	
	`sendobj`	`Any`
	`op`	`Union[Op, Callable[[Any, Any], Any]] = Ellipsis`
`) -> Any`		

For the full source of this example, see section 3.16.15

You can use any of the given reduction operators, (for the list, see section 3.10.1), or a user-defined one. In the latter case, the `MPI_Op` operations do not return an error code.

MPL note 16. As in the C/F interfaces, MPL interfaces to the scan routines have the same calling sequences as the 'Allreduce' routine.

End of MPL note

3.4.1 Exclusive scan

Often, the more useful variant is the *exclusive scan* `MPI_Exscan` (figure 3.6) with the same prototype.

The result of the exclusive scan is undefined on processor 0 (`None` in python), and on processor 1 it is a copy of the send value of processor 1. In particular, the `MPI_Op` need not be called on these two processors.

Exercise 3.10. The exclusive definition, which computes $x_0 \oplus x_{i-1}$ on processor i, can easily be derived from the inclusive operation for operations such as `MPI_SUM` or `MPI_PROD`. Are there operators where that is not the case?

3.4.2 Use of scan operations

The `MPI_Scan` operation is often useful with indexing data. Suppose that every processor p has a local vector where the number of elements n_p is dynamically determined. In order to translate the local numbering $0 \ldots n_p - 1$ to a global numbering one does a scan with the number of local elements as input. The output is then the global number of the first local variable.

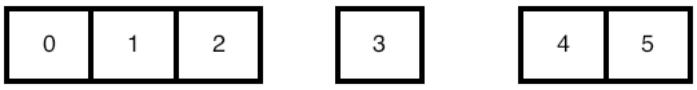


Figure 3.3: Local arrays that together form a consecutive range

Exercise 3.11.

- Let each process compute a random value n_{local}, and allocate an array of that length. Define
$$N = \sum n_{\text{local}}$$
- Fill the array with consecutive integers, so that all local arrays, laid end-to-end, contain the numbers $0 \cdots N - 1$. (See figure 3.3.)

(There is a skeleton for this exercise under the name `scangather`.)

Exercise 3.12. Did you use `MPI_Scan` or `MPI_Exscan` for the previous exercise? How would you describe the result of the other scan operation, given the same input?

It is possible to do a *segmented scan*. Let x_i be a series of numbers that we want to sum to X_i as follows. Let y_i be a series of booleans such that

$$\begin{cases} X_i = x_i & \text{if } y_i = 0 \\ X_i = X_{i-1} + x_i & \text{if } y_i = 1 \end{cases}$$

(This is the basis for the implementation of the *sparse matrix vector product* as prefix operation; see HPSC-19.2.) This means that X_i sums the segments between locations where $y_i = 0$ and the first subsequent place where $y_i = 1$. To implement this, you need a user-defined operator

$$\begin{pmatrix} X \\ x \\ y \end{pmatrix} = \begin{pmatrix} X_1 \\ x_1 \\ y_1 \end{pmatrix} \bigoplus \begin{pmatrix} X_2 \\ x_2 \\ y_2 \end{pmatrix} : \begin{cases} X = x_1 + x_2 & \text{if } y_2 == 1 \\ X = x_2 & \text{if } y_2 == 0 \end{cases}$$

This operator is not communitative, and it needs to be declared as such with `MPI_Op_create`; see section 3.10.2

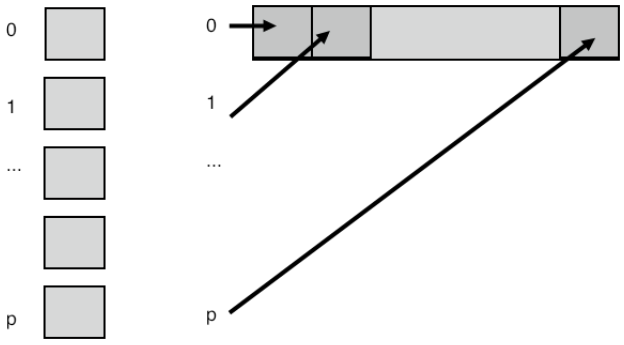


Figure 3.4: Gather collects all data onto a root

3.5 Rooted collectives: gather and scatter

In the `MPI_Scatter` operation, the root spreads information to all other processes. The difference with a broadcast is that it involves individual information from/to every process. Thus, the gather operation typically has an array of items, one coming from each sending process, and scatter has an array, with an

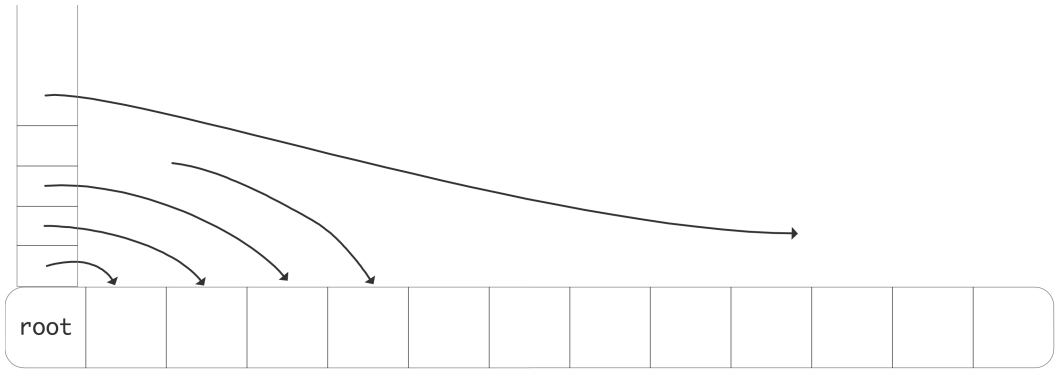


Figure 3.5: A scatter operation

individual item for each receiving process; see figure 3.5.

These gather and scatter collectives have a different parameter list from the broadcast/reduce. The broadcast/reduce involves the same amount of data on each process, so it was enough to have a single datatype-/size specification; for one buffer in the broadcast, and for both buffers in the reduce call. In the gather/scatter calls you have

- a large buffer on the root, with a datatype and size specification, and
- a smaller buffer on each process, with its own type and size specification.

In the gather and scatter calls, each processor has n elements of individual data. There is also a root processor that has an array of length np, where p is the number of processors. The gather call collects all this data from the processors to the root; the scatter call assumes that the information is initially on the root and it is spread to the individual processors.

Here is a small example:

```
// gather.c
// we assume that each process has a value "localsize"
// the root process collects these values

  if (procno==root)
    localsizes = (int*) malloc( nprocs*sizeof(int) );

// everyone contributes their info
  MPI_Gather(&localsize,1,MPI_INT,
             localsizes,1,MPI_INT,root,comm);
```

For the full source of this example, see section 3.16.16

This will also be the basis of a more elaborate example in section 3.9.

Exercise 3.13. Let each process compute a random number. You want to print the maximum value and on what processor it is computed. What collective(s) do you use? Write a short program.

The `MPI_Scatter` operation is in some sense the inverse of the gather: the root process has an array of length np where p is the number of processors and n the number of elements each processor will receive.

```
int MPI_Scatter
  (void* sendbuf, int sendcount, MPI_Datatype sendtype,
   void* recvbuf, int recvcount, MPI_Datatype recvtype,
   int root, MPI_Comm comm)
```

Two things to note about these routines:

- The prototype for `MPI_Gather` (figure 3.7) has two 'count' parameters, one for the length of the individual send buffers, and one for the receive buffer. However, confusingly, the second parameter (which is only relevant on the root) does not indicate the total amount of information coming in, but rather the size of *each* contribution. Thus, the two count parameters will usually be the same (at least on the root); they can differ if you use different `MPI_Datatype` values for the sending and receiving processors.
- While every process has a sendbuffer in the gather, and a receive buffer in the scatter call, only the root process needs the long array in which to gather, or from which to scatter. However, because in SPMD mode all processes need to use the same routine, a parameter for this long array is always present. Nonroot processes can use a *null pointer* here.
- More elegantly, the `MPI_IN_PLACE` option can be used for buffers that are not applicable, such as the receive buffer on a sending process. See section 3.3.2.

MPL note 17. Gathering (by `communicator::gather`) or scattering (by `communicator::scatter`) a single scalar takes a scalar argument and a raw array:

```
vector<float> v;
float x;
comm_world.scatter(0, v.data(), x);
```

For the full source of this example, see section 3.16.11

If more than a single scalar is gathered, or scattered into, it becomes necessary to specify a layout:

Figure 3.7 `MPI_Gather`

Name	Param name	C type	F type	inout
`mpi_gather (`				
	`sendbuf`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	starting address of send buffer			
	`sendcount`	`int`	`INTEGER`	`in`
	number of elements in send buffer			
	`sendtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of send buffer elements			
	`recvbuf`	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	address of receive buffer			
	`recvcount`	`int`	`INTEGER`	`in`
	number of elements for any single receive			
	`recvtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of recv buffer elements			
	`root`	`int`	`INTEGER`	`in`
	rank of receiving process			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
void mpl::communicator::gather
   ( int  root_rank, const T & senddata ) const
   ( int  root_rank, const T & senddata, T *  recvdata ) const
   ( int  root_rank, const T * senddata, const layout< T > &  sendl ) const
   ( int  root_rank, const T * senddata, const layout< T > &  sendl,
                           T * recvdata, const layout< T > &  recvl ) const
```

Python:

Name	Param name	param type
`Comm.gather(`		
	`self`	
	`sendobj`	`Any`
	`root`	`int = 0`
`) -> Optional[List[Any]]`		

```
vector<float> vrecv(2),vsend(2*nprocs);
mpl::contiguous_layout<float> twonums(2);
comm_world.scatter
  (0, vsend.data(),twonums, vrecv.data(),twonums );
```

For the full source of this example, see section 3.16.11

End of MPL note

MPL note 18. Logically speaking, on every nonroot process, the gather call only has a send buffer. MPL supports this by having two variants that only specify the send data.

```
if (procno==0) {
  vector<int> size_buffer(nprocs);
  comm_world.gather
    (
     0,my_number_of_elements,size_buffer.data()
```

```
    );
} else {
  comm_world.gather
    ( 0,my_number_of_elements );
}
```

End of MPL note

3.5.1 Examples

In some applications, each process computes a row or column of a matrix, but for some calculation (such as the determinant) it is more efficient to have the whole matrix on one process. You should of course only do this if this matrix is essentially smaller than the full problem, such as an interface system or the last coarsening level in multigrid.

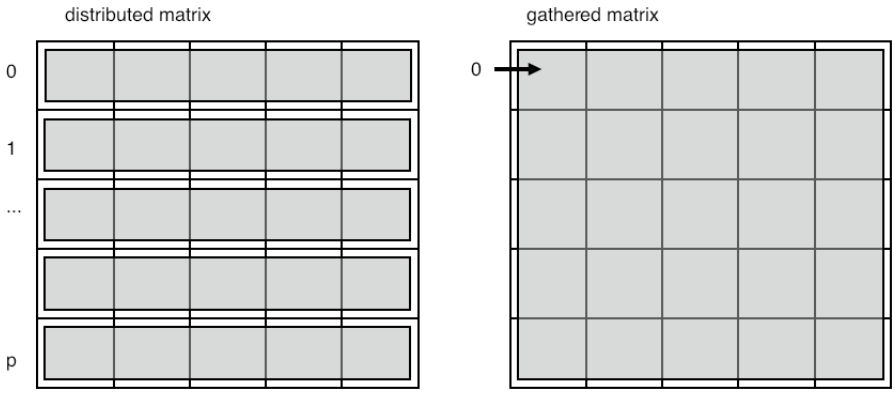


Figure 3.6: Gather a distributed matrix onto one process

Figure 3.6 pictures this. Note that conceptually we are gathering a two-dimensional object, but the buffer is of course one-dimensional. You will later see how this can be done more elegantly with the 'subarray' datatype; section 6.3.4.

Another thing you can do with a distributed matrix is to transpose it.

```
// itransposeblock.c
for (int iproc=0; iproc<nprocs; iproc++) {
  MPI_Scatter( regular,1,MPI_DOUBLE,
               &(transpose[iproc]),1,MPI_DOUBLE,
               iproc,comm);
}
```

For the full source of this example, see section 3.16.17

In this example, each process scatters its column. This needs to be done only once, yet the scatter happens in a loop. The trick here is that a process only originates the scatter when it is the root, which happens only once. Why do we need a loop? That is because each element of a process' row originates from a different scatter operation.

Exercise 3.14. Can you rewrite this code so that it uses a gather rather than a scatter? Does that change anything essential about structure of the code?

Exercise 3.15. Take the code from exercise 3.11 and extend it to gather all local buffers onto rank zero. Since the local arrays are of differing lengths, this requires `MPI_Gatherv`. How do you construct the lengths and displacements arrays?
(There is a skeleton for this exercise under the name `scangather`.)

3.5.2 Allgather

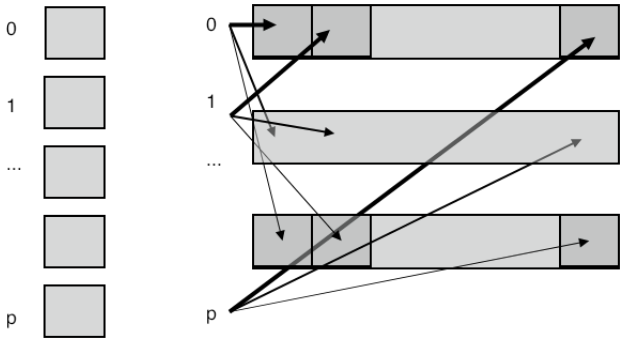


Figure 3.7: All gather collects all data onto every process

The `MPI_Allgather` (figure 3.8) routine does the same gather onto every process: each process winds up with the totality of all data; figure 3.7.

This routine can be used in the simplest implementation of the *dense matrix-vector product* to give each processor the full input; see HPSC-6.2.2.

Some cases look like an all-gather but can be implemented more efficiently. Suppose you have two distributed vectors, and you want to create a new vector that contains those elements of the one that do not appear in the other. You could implement this by gathering the second vector on each processor, but this may be prohibitive in memory usage.

Exercise 3.16. Can you think of another algorithm for taking the set difference of two distributed vectors. Hint: look up *bucket brigade*; section **??**lgorithm'. What is the time and space complexity of this algorithm? Can you think of other advantages beside a reduction in workspace?

3.6 All-to-all

The all-to-all operation `MPI_Alltoall` (figure 3.9) can be seen as a collection of simultaneous broadcasts or simultaneous gathers. The parameter specification is much like an allgather, with a separate send and receive buffer, and no root specified. As with the gather call, the receive count corresponds to an individual receive, not the total amount.

Unlike the gather call, the send buffer now obeys the same principle: with a send count of 1, the buffer has a length of the number of processes.

Figure 3.8 `MPI_Allgather`

Name	Param name	C type	F type	inout
`mpi_allgather (`				
	`sendbuf` *starting address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`sendcount` *number of elements in send buffer*	`int`	`INTEGER`	`in`
	`sendtype` *data type of send buffer elements*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`recvbuf` *address of receive buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`recvcount` *number of elements received from any process*	`int`	`INTEGER`	`in`
	`recvtype` *data type of receive buffer elements*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.allgather(`		
	`self`	
	`sendobj`	`Any`
`) -> List[Any]`		

Figure 3.9 `MPI_Alltoall`

Name	Param name	C type	F type	inout
`mpi_alltoall (`				
	`sendbuf` *starting address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`sendcount` *number of elements sent to each process*	`int`	`INTEGER`	`in`
	`sendtype` *data type of send buffer elements*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`recvbuf` *address of receive buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`recvcount` *number of elements received from any process*	`int`	`INTEGER`	`in`
	`recvtype` *data type of receive buffer elements*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.alltoall(`		
	`self`	
	`sendobj`	`Sequence[Any]`
`) -> List[Any]`		

3.6.1 All-to-all as data transpose

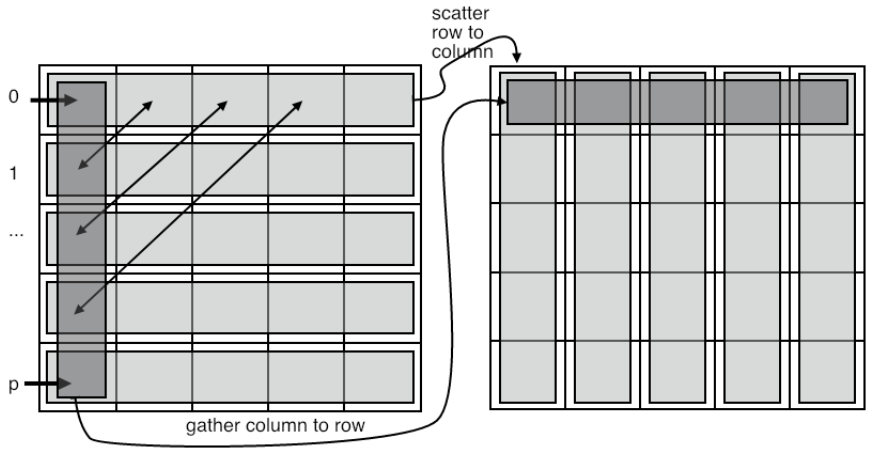


Figure 3.8: All-to-all transposes data

The all-to-all operation can be considered as a data transpose. For instance, assume that each process knows how much data to send to every other process. If you draw a connectivity matrix of size $P \times P$, denoting who-sends-to-who, then the send information can be put in rows:

$$\forall_i : C[i,j] > 0 \quad \text{if process } i \text{ sends to process } j.$$

Conversely, the columns then denote the receive information:

$$\forall_j : C[i,j] > 0 \quad \text{if process } j \text{ receives from process } i.$$

The typical application for such data transposition is in the Fast Fourier Transform (FFT) algorithm, where it can take tens of percents of the running time on large clusters.

We will consider another application of data transposition, namely *radix sort*, but we will do that in a couple of steps. First of all:

Exercise 3.17. In the initial stage of a *radix sort*, each process considers how many elements to send to every other process. Use `MPI_Alltoall` to derive from this how many elements they will receive from every other process.

3.6.2 All-to-all-v

The major part of the *radix sort* algorithm consist of every process sending some of its elements to each of the other processes. The routine `MPI_Alltoallv` (figure 3.10) is used for this pattern:

- Every process scatters its data to all others,
- but the amount of data is different per process.

Exercise 3.18. The actual data shuffle of a *radix sort* can be done with `MPI_Alltoallv`. Finish the code of exercise 3.17.

Figure 3.10 `MPI_Alltoallv`

Name	Param name	C type	F type	inout
`mpi_alltoallv (`				
	`sendbuf`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	starting address of send buffer			
	`sendcounts`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	non-negative integer array (of length group size) specifying the number of elements to send to each rank			
	`sdispls`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	integer array (of length group size). Entry j specifies the displacement (relative to sendbuf) from which to take the outgoing data destined for			
	`sendtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of send buffer elements			
	`recvbuf`	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	address of receive buffer			
	`recvcounts`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	non-negative integer array (of length group size) specifying the number of elements that can be received from each rank			
	`rdispls`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i			
	`recvtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of receive buffer elements			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Alltoallv(`		
	`self`	
	`sendbuf`	`Union[BufSpecV, InPlace]`
	`recvbuf`	`BufSpecV`
`) -> None`		

3.7 Reduce-scatter

There are several MPI collectives that are functionally equivalent to a combination of others. You have already seen `MPI_Allreduce` which is equivalent to a reduction followed by a broadcast. Often such combinations can be more efficient than using the individual calls; see HPSC-6.1.

Here is another example: `MPI_Reduce_scatter` is equivalent to a reduction on an array of data (meaning a pointwise reduction on each array location) followed by a scatter of this array to the individual processes.

We will discuss this routine, or rather its variant `MPI_Reduce_scatter_block` (figure 3.11), using an important example: the *sparse matrix-vector product* (see HPSC-6.5.1 for background information). Each process contains one or more matrix rows, so by looking at indices the process can decide what other processes it needs to receive data from, that is, each process knows how many messages it will receive, and from which processes. The problem is for a process to find out what other processes it needs to send data to.

Let's set up the data:

```
// reducescatter.c
```

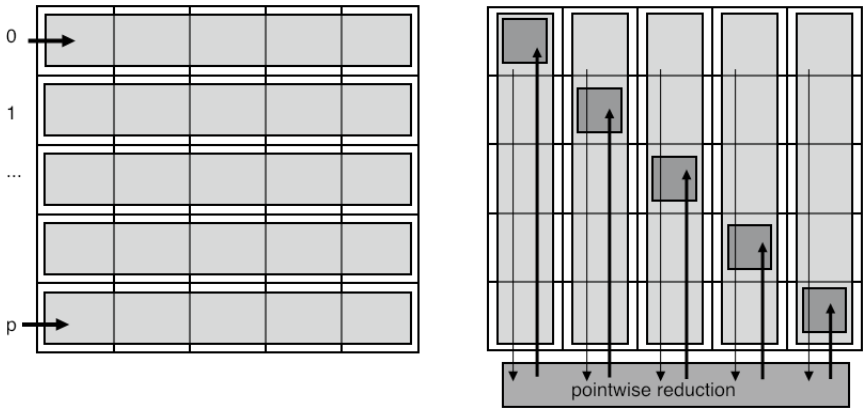


Figure 3.9: Reduce scatter

```
int
// data that we know:
  *i_recv_from_proc = (int*) malloc(nprocs*sizeof(int)),
  *procs_to_recv_from, nprocs_to_recv_from=0,
// data we are going to determin:
  *procs_to_send_to,nprocs_to_send_to;
```

For the full source of this example, see section 3.16.18

Each process creates an array of ones and zeros, describing who it needs data from. Ideally, we only need the array *procs_to_recv_from* but initially we need the (possibly much larger) array *i_recv_from_proc*.

Next, the MPI_Reduce_scatter_block call then computes, on each process, how many messages it needs to send.

```
MPI_Reduce_scatter_block
  (i_recv_from_proc,&nprocs_to_send_to,1,MPI_INT,
  MPI_SUM,comm);
```

For the full source of this example, see section 3.16.18

We do not yet have the information to which processes to send. For that, each process sends a zero-size message to each of its senders. Conversely, it then does a receive to with MPI_ANY_SOURCE to discover who is requesting data from it. The crucial point to the MPI_Reduce_scatter_block call is that, without it, a process would not know how many of these zero-size messages to expect.

```
/*
 * Send a zero-size msg to everyone that you receive from,
 * just to let them know that they need to send to you.
 */
MPI_Request send_requests[nprocs_to_recv_from];
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++) {
  int proc=procs_to_recv_from[iproc];
  double send_buffer=0.;
  MPI_Isend(&send_buffer,0,MPI_DOUBLE, /*to:*/ proc,0,comm,
            &(send_requests[iproc]));
```

Figure 3.11 `MPI_Reduce_scatter`

Name	Param name	C type	F type	inout
`mpi_reduce_scatter (`				
	`sendbuf`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	starting address of send buffer			
	`recvbuf`	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	starting address of receive buffer			
	`recvcounts`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	non-negative integer array (of length group size) specifying the number of elements of the result distributed to each process.			
	`datatype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of elements of send and receive buffers			
	`op`	`MPI_Op`	`TYPE(MPI_Op)`	`in`
	operation			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Reduce_scatter(`		
	`self`	
	`sendbuf`	`Union[BufSpec, InPlace]`
	`recvbuf`	`BufSpec`
	`recvcounts`	`Optional[Sequence[int]] = None`
	`op`	`Op = Ellipsis`
`) -> None`		

```
}

/*
 * Do as many receives as you know are coming in;
 * use wildcards since you don't know where they are coming from.
 * The source is a process you need to send to.
 */
procs_to_send_to = (int*)malloc( nprocs_to_send_to * sizeof(int) );
for (int iproc=0; iproc<nprocs_to_send_to; iproc++) {
  double recv_buffer;
  MPI_Status status;
  MPI_Recv(&recv_buffer,0,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,
           &status);
  procs_to_send_to[iproc] = status.MPI_SOURCE;
}
MPI_Waitall(nprocs_to_recv_from,send_requests,MPI_STATUSES_IGNORE);
```

For the full source of this example, see section 3.16.18

The `MPI_Reduce_scatter` (figure 3.11) call is more general: instead of indicating the mere presence of a message between two processes, by having individual receive counts one can, for instance, indicate the size of the messages.

We can look at reduce-scatter as a limited form of the all-to-all data transposition discussed above (sec-

tion 3.6.1). Suppose that the matrix C contains only 0/1, indicating whether or not a messages is send, rather than the actual amounts. If a receiving process only needs to know how many messages to receive, rather than where they come from, it is enough to know the column sum, rather than the full column (see figure 3.9).

Another application of the reduce-scatter mechanism is in the dense matrix-vector product, if a two-dimensional data distribution is used.

3.7.1 Examples

An important application of this is establishing an irregular communication pattern. Assume that each process knows which other processes it wants to communicate with; the problem is to let the other processes know about this. The solution is to use `MPI_Reduce_scatter` to find out how many processes want to communicate with you

```
MPI_Reduce_scatter_block
  (i_recv_from_proc,&nprocs_to_send_to,1,MPI_INT,
  MPI_SUM,comm);
```

For the full source of this example, see section 3.16.18

and then wait for precisely that many messages with a source value of `MPI_ANY_SOURCE`.

```
/*
 * Send a zero-size msg to everyone that you receive from,
 * just to let them know that they need to send to you.
 */
MPI_Request send_requests[nprocs_to_recv_from];
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++) {
  int proc=procs_to_recv_from[iproc];
  double send_buffer=0.;
  MPI_Isend(&send_buffer,0,MPI_DOUBLE, /*to:*/ proc,0,comm,
              &(send_requests[iproc]));
}

/*
 * Do as many receives as you know are coming in;
 * use wildcards since you don't know where they are coming from.
 * The source is a process you need to send to.
 */
procs_to_send_to = (int*)malloc( nprocs_to_send_to * sizeof(int) );
for (int iproc=0; iproc<nprocs_to_send_to; iproc++) {
  double recv_buffer;
  MPI_Status status;
  MPI_Recv(&recv_buffer,0,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,
            &status);
  procs_to_send_to[iproc] = status.MPI_SOURCE;
}
MPI_Waitall(nprocs_to_recv_from,send_requests,MPI_STATUSES_IGNORE);
```

For the full source of this example, see section 3.16.18

Use of `MPI_Reduce_scatter` to implement the two-dimensional matrix-vector product. Set up separate row and column communicators with `MPI_Comm_split`, use `MPI_Reduce_scatter` to combine local products.

Figure 3.12 `MPI_Barrier`

Name	Param name	C type	F type	inout
`mpi_barrier (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.barrier(`		
	`self`	
`) -> None`		

```
MPI_Allgather(&my_x,1,MPI_DOUBLE,
          local_x,1,MPI_DOUBLE,environ.col_comm);
MPI_Reduce_scatter(local_y,&my_y,&ione,MPI_DOUBLE,
          MPI_SUM,environ.row_comm);
```

3.8 Barrier

A barrier call, `MPI_Barrier` (figure 3.12) is a routine that blocks all processes until they have all reached the barrier call. Thus it achieves time synchronization of the processes.

This call's simplicity is contrasted with its usefulness, which is very limited. It is almost never necessary to synchronize processes through a barrier: for most purposes it does not matter if processors are out of sync. Conversely, collectives (except the new nonblocking ones; section 3.11) introduce a barrier of sorts themselves.

3.9 Variable-size-input collectives

In the gather and scatter call above each processor received or sent an identical number of items. In many cases this is appropriate, but sometimes each processor wants or contributes an individual number of items.

Let's take the gather calls as an example. Assume that each processor does a local computation that produces a number of data elements, and this number is different for each processor (or at least not the same for all). In the regular `MPI_Gather` call the root processor had a buffer of size nP, where n is the number of elements produced on each processor, and P the number of processors. The contribution from processor p would go into locations $pn, \ldots, (p+1)n-1$.

For the variable case, we first need to compute the total required buffer size. This can be done through a simple `MPI_Reduce` with `MPI_SUM` as reduction operator: the buffer size is $\sum_p n_p$ where n_p is the number of elements on processor p. But you can also postpone this calculation for a minute.

The next question is where the contributions of the processor will go into this buffer. For the contribution from processor p that is $\sum_{q<p} n_p, \ldots \sum_{q\leq p} n_p - 1$. To compute this, the root processor needs to have all the n_p numbers, and it can collect them with an `MPI_Gather` call.

We now have all the ingredients. All the processors specify a send buffer just as with `MPI_Gather`. However, the receive buffer specification on the root is more complicated. It now consists of:

`outbuffer, array-of-outcounts, array-of-displacements, outtype`

and you have just seen how to construct that information.

For example, in an `MPI_Gatherv` (figure 3.13) call each process has an individual number of items to contribute. To gather this, the root process needs to find these individual amounts with an `MPI_Gather` call, and locally construct the offsets array. Note how the offsets array has size `ntids+1`: the final offset value is automatically the total size of all incoming data. See the example below.

There are various calls where processors can have buffers of differing sizes.

- In `MPI_Scatterv` (figure 3.14) the root process has a different amount of data for each recipient.
- In `MPI_Gatherv`, conversely, each process contributes a different sized send buffer to the received result; `MPI_Allgatherv` (figure 3.15) does the same, but leaves its result on all processes; `MPI_Alltoallv` does a different variable-sized gather on each process.

3.9.1 Example of Gatherv

We use `MPI_Gatherv` to do an irregular gather onto a root. We first need an `MPI_Gather` to determine offsets.

```
// gatherv.c
// we assume that each process has an array "localdata"
// of size "localsize"

// the root process decides how much data will be coming:
// allocate arrays to contain size and offset information
  if (procno==root) {
    localsizes = (int*) malloc( nprocs*sizeof(int) );
    offsets = (int*) malloc( nprocs*sizeof(int) );
  }
// everyone contributes their local size info
  MPI_Gather(&localsize,1,MPI_INT,
             localsizes,1,MPI_INT,root,comm);
// the root constructs the offsets array
  if (procno==root) {
    int total_data = 0;
    for (int i=0; i<nprocs; i++) {
      offsets[i] = total_data;
      total_data += localsizes[i];
    }
    alldata = (int*) malloc( total_data*sizeof(int) );
  }
// everyone contributes their data
  MPI_Gatherv(localdata,localsize,MPI_INT,
              alldata,localsizes,offsets,MPI_INT,root,comm);
```

For the full source of this example, see section 3.16.19

Figure 3.13 `MPI_Gatherv`

Name	Param name	C type	F type	inout
`mpi_gatherv (`				
	`sendbuf`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	starting address of send buffer			
	`sendcount`	`int`	`INTEGER`	`in`
	number of elements in send buffer			
	`sendtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of send buffer elements			
	`recvbuf`	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	address of receive buffer			
	`recvcounts`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	non-negative integer array (of length group size) containing the number of elements that are received from each process			
	`displs`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	integer array (of length group size). Entry i specifies the displacement relative to recvbuf at which to place the incoming data from process i			
	`recvtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of recv buffer elements			
	`root`	`int`	`INTEGER`	`in`
	rank of receiving process			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T>
void gatherv
   (int root_rank, const T *senddata, const layout<T> &sendl,
    T *recvdata, const layouts<T> &recvls, const displacements &recvdispls) const
   (int root_rank, const T *senddata, const layout<T> &sendl,
    T *recvdata, const layouts<T> &recvls) const
   (int root_rank, const T *senddata, const layout<T> &sendl ) const
```

Python:

Name	Param name	param type
`Comm.Gatherv(`		
	`self`	
	`sendbuf`	`Union[BufSpec, InPlace]`
	`recvbuf`	`Optional[BufSpecV]`
	`root`	`int = 0`
`) -> None`		

Figure 3.14 `MPI_Scatterv`

Name	Param name	C type	F type	inout
`mpi_scatterv (`				
	`sendbuf`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	address of send buffer			
	`sendcounts`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	non-negative integer array (of length group size) specifying the number of elements to send to each rank			
	`displs`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	integer array (of length group size). Entry i specifies the displacement (relative to sendbuf) from which to take the outgoing data to process i			
	`sendtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of send buffer elements			
	`recvbuf`	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	address of receive buffer			
	`recvcount`	`int`	`INTEGER`	`in`
	number of elements in receive buffer			
	`recvtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of receive buffer elements			
	`root`	`int`	`INTEGER`	`in`
	rank of sending process			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Scatterv(`		
	`self`	
	`sendbuf`	`Optional[BufSpecV]`
	`recvbuf`	`Union[BufSpec, InPlace]`
	`root`	`int = 0`
`) -> None`		

```
## gatherv.py
# implicitly using root=0
globalsize = comm.reduce(localsize)
if procid==0:
    print("Global size=%d" % globalsize)
collecteddata = np.empty(globalsize,dtype=np.int)
counts = comm.gather(localsize)
comm.Gatherv(localdata, [collecteddata, counts])
```

For the full source of this example, see section 3.16.20

MPL note 19. No specification of displacement is neded. Instead, a `layouts` object, essentially a `std::vector` of `layout`s, is used.

```
mpl::layouts<int> receive_layout;
for ( int iproc=0,loc=0; iproc<nprocs; iproc++ ) {
  auto siz = size_buffer.at(iproc);
  receive_layout.push_back
        ( mpl::indexed_layout<int>( {{ siz,loc }} ) );
  loc += siz;
```

Figure 3.15 `MPI_Allgatherv`

Name	Param name	C type	F type	inout
`mpi_allgatherv (`				
	`sendbuf`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	starting address of send buffer			
	`sendcount`	`int`	`INTEGER`	`in`
	number of elements in send buffer			
	`sendtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of send buffer elements			
	`recvbuf`	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	address of receive buffer			
	`recvcounts`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	non-negative integer array (of length group size) containing the number of elements that are received from each process			
	`displs`	`const int[]`	`INTEGER`	`in`
	`length: *`			
	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i			
	`recvtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of receive buffer elements			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Allgatherv(`		
	`self`	
	`sendbuf`	`Union[BufSpec, InPlace]`
	`recvbuf`	`BufSpecV`
`) -> None`		

```
}
```

End of MPL note

3.9.2 Example of Allgatherv

Prior to the actual gatherv call, we need to construct the count and displacement arrays. The easiest way is to use a reduction.

```
// allgatherv.c
MPI_Allgather
  ( &my_count,1,MPI_INT,
    recv_counts,1,MPI_INT, comm );
int accumulate = 0;
for (int i=0; i<nprocs; i++) {
  recv_displs[i] = accumulate; accumulate += recv_counts[i]; }
int *global_array = (int*) malloc(accumulate*sizeof(int));
MPI_Allgatherv
  ( my_array,procno+1,MPI_INT,
    global_array,recv_counts,recv_displs,MPI_INT, comm );
```

For the full source of this example, see section 3.16.21

In python the receive buffer has to contain the counts and displacements arrays.

```
## allgatherv.py
mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)
```

For the full source of this example, see section 6.8.3

```
my_count = np.empty(1,dtype=np.int)
my_count[0] = mycount
comm.Allgather( my_count,recv_counts )

accumulate = 0
for p in range(nprocs):
    recv_displs[p] = accumulate; accumulate += recv_counts[p]
global_array = np.empty(accumulate,dtype=np.float64)
comm.Allgatherv( my_array, [global_array,recv_counts,recv_displs,MPI.DOUBLE] )
```

For the full source of this example, see section 6.8.3

3.9.3 Variable all-to-all

The variable all-to-all routine `MPI_Alltoallv` is discussed in section 3.6.2.

3.10 MPI Operators

MPI *operators*, that is, objects of type `MPI_Op`, are used in reduction operators. Most common operators, such as sum or maximum, have been built into the MPI library; see section 3.10.1. It is also possible to define new operators; see section 3.10.2.

3.10.1 Pre-defined operators

The following is the list of *pre-defined operators* `MPI_Op` values.

MPI type	meaning	applies to
MPI_MAX	maximum	integer, floating point
MPI_MIN	minimum	
MPI_SUM	sum	integer, floating point, complex, multilanguage types
MPI_REPLACE	overwrite	
MPI_NO_OP	no change	
MPI_PROD	product	
MPI_LAND	logical and	C integer, logical
MPI_LOR	logical or	
MPI_LXOR	logical xor	
MPI_BAND	bitwise and	integer, byte, multilanguage types
MPI_BOR	bitwise or	
MPI_BXOR	bitwise xor	
MPI_MAXLOC	max value and location	MPI_DOUBLE_INT and such
MPI_MINLOC	min value and location	

3.10.1.1 *Minloc and maxloc*

The MPI_MAXLOC and MPI_MINLOC operations yield both the maximum and the rank on which it occurs. Their result is a `struct` of the data over which the reduction happens, and an int.

In C, the types to use in the reduction call are: MPI_FLOAT_INT, MPI_LONG_INT, MPI_DOUBLE_INT, MPI_SHORT_INT, MPI_2INT, MPI_LONG_DOUBLE_INT. Likewise, the input needs to consist of such structures: the input should be an array of such struct types, where the `int` is the rank of the number.

The Fortran interface to MPI was designed to use pre-Fortran90 features, so it is not using Fortran derived types (*Type* keyword). Instead, all integer indices are stored in whatever the type is that is being reduced. The available result types are then MPI_2REAL, MPI_2DOUBLE_PRECISION, MPI_2INTEGER.

Likewise, the input needs to be arrays of such type. Consider this example:

```
Real*8,dimension(2,N) :: input,output
call MPI_Reduce( input,output, N, MPI_2DOUBLE_PRECISION, &
                 MPI_MAXLOC, root, comm )
```

3.10.2 User-defined operators

In addition to predefined operators, MPI has the possibility of *user-defined operators* to use in a reduction or scan operation.

The routine for this is MPI_Op_create (figure 3.16), which takes a user function and turns it into an object of type MPI_Op, which can then be used in any reduction:

```
MPI_Op rwz;
MPI_Op_create(reduce_without_zero,1,&rwz);
MPI_Allreduce(data+procno,&positive_minimum,1,MPI_INT,rwz,comm);
```

For the full source of this example, see section 3.16.23

Python note. In python, *Op.Create* is a class method for the *MPI* class.

Figure 3.16 `MPI_Op_create`

Name	Param name	C type	F type	inout
`mpi_op_create (`				
	`user_fn` *user defined function*		`PROCEDURE`	`in`
	`commute` *true if commutative; false otherwise.*	`int`	`LOGICAL`	`in`
	`op` *operation*	`MPI_Op*`	`TYPE(MPI_Op)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Op.Create(`		
	`cls`	
	`function`	`Callable[[Buffer, Buffer, Datatype], None]`
	`commute`	`bool = False`
`) -> Op`		

```
rwz = MPI.Op.Create(reduceWithoutZero)
positive_minimum = np.zeros(1,dtype=np.intc)
comm.Allreduce(data[procid],positive_minimum,rwz);
```

For the full source of this example, see section 3.16.24

The user function needs to have the following prototype:

```
typedef void MPI_User_function
    ( void *invec, void *inoutvec, int *len,
      MPI_Datatype *datatype);
```

```
FUNCTION USER_FUNCTION( INVEC(*), INOUTVEC(*), LEN, TYPE)
<type> INVEC(LEN), INOUTVEC(LEN)
INTEGER LEN, TYPE
```

For example, here is an operator for finding the smallest nonzero number in an array of nonnegative integers:

```
// reductpositive.c
void reduce_without_zero(void *in,void *inout,int *len,MPI_Datatype *type) {
// r is the already reduced value, n is the new value
  int n = *(int*)in, r = *(int*)inout;
  int m;
  if (n==0) { // new value is zero: keep r
    m = r;
  } else if (r==0) {
    m = n;
  } else if (n<r) { // new value is less but not zero: use n
    m = n;
  } else { // new value is more: use r
    m = r;
  };
```

```
  *(int*)inout = m;
}
```

For the full source of this example, see section 3.16.23

Python note. The python equivalent of such a function receives bare buffers as arguments. Therefore, it is best to turn them first into NumPy arrays using `np.frombuffer`:

```
## reductpositive.py
def reduceWithoutZero(in_buf, inout_buf, datatype):
    typecode = MPI._typecode(datatype)
    assert typecode is not None ## check MPI datatype is built-in
    dtype = np.dtype(typecode)

    in_array = np.frombuffer(in_buf, dtype)
    inout_array = np.frombuffer(inout_buf, dtype)

    n = in_array[0]; r = inout_array[0]
    if n==0:
        m = r
    elif r==0:
        m = n
    elif n<r:
        m = n
    else:
        m = r
    inout_array[0] = m
```

For the full source of this example, see section 3.16.24

The `assert` statement accounts for the fact that this mapping of MPI datatype to NumPy dtype only works for built-in MPI datatypes.

MPL note 20. A user-defined operator can be a templated class with an `operator()`. Example:

```
// reduceuser.cxx
template<typename T>
class lcm {
public:
  T operator()(T a, T b) {
    T zero=T();
    T t((a/gcd(a, b))*b);
    if (t<zero)
      return -t;
    return t;
  }
```

End of MPL note

MPL note 21. You can also do the reduction by lambda:

```
comm_world.reduce
  ( [] (int i,int j) -> int
       { return i+j; },
        0,data );
```

Figure 3.17 `MPI_Op_commutative`

Name	Param name	C type	F type	inout
`mpi_op_commutative (`				
	`op` *operation*	`MPI_Op`	`TYPE(MPI_Op)`	`in`
	`commute` *true if op is commutative, false otherwise*	`int*`	`LOGICAL`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

End of MPL note

The function has an array length argument `len`, to allow for pointwise reduction on a whole array at once. The `inoutvec` array contains partially reduced results, and is typically overwritten by the function.

There are some restrictions on the user function:

- It may not call MPI functions, except for `MPI_Abort`.
- It must be associative; it can be optionally commutative, which fact is passed to the `MPI_Op_create` call.

Exercise 3.19. Write the reduction function to implement the *one-norm* of a vector:

$$\|x\|_1 \equiv \sum_i |x_i|.$$

(There is a skeleton for this exercise under the name `onenorm`.)

The operator can be destroyed with a corresponding `MPI_Op_free`.

```
int MPI_Op_free(MPI_Op *op)
```

This sets the operator to `MPI_OP_NULL`. This is not necessary in OO languages, where the destructor takes care of it.

You can query the commutativity of an operator with `MPI_Op_commutative` (figure 3.17).

3.10.3 Local reduction

The application of an `MPI_Op` can be performed with the routine `MPI_Reduce_local` (figure 3.18). Using this routine and some send/receive scheme you can build your own global reductions. Note that this routine does not take a communicator because it is purely local.

3.11 Nonblocking collectives

Above you have seen how the 'Isend' and 'Irecv' routines can overlap communication with computation. This is not possible with the collectives you have seen so far: they act like blocking sends or receives. However, there are also *nonblocking collectives*, introduced in MPI-3.

Figure 3.18 `MPI_Reduce_local`

Name	Param name	C type	F type	inout
`mpi_reduce_local (`				
	`inbuf` *input buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`inoutbuf` *combined input and output buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`inout`
	`count` *number of elements in inbuf and inoutbuf buffers*	`int`	`INTEGER`	`in`
	`datatype` *data type of elements of inbuf and inoutbuf buffers*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`op` *operation*	`MPI_Op`	`TYPE(MPI_Op)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Such operations can be used to increase efficiency. For instance, computing

$$y \leftarrow Ax + (x^t x)y$$

involves a matrix-vector product, which is dominated by computation in the *sparse matrix* case, and an inner product which is typically dominated by the communication cost. You would code this as

```
MPI_Iallreduce( .... x ..., &request);
// compute the matrix vector product
MPI_Wait(request);
// do the addition
```

This can also be used for 3D FFT operations [13]. Occasionally, a nonblocking collective can be used for nonobvious purposes, such as the `MPI_Ibarrier` in [14].

These have roughly the same calling sequence as their blocking counterparts, except that they output an `MPI_Request`. You can then use an `MPI_Wait` call to make sure the collective has completed.

Nonblocking collectives offer a number of performance advantages:

- Do two reductions (on the same communicator) with different operators simultaneously:

  $$\alpha \leftarrow x^t y$$
  $$\beta \leftarrow \|z\|_\infty$$

  which translates to:

  ```
  MPI_Allreduce( &local_xy,  &global_xy, 1,MPI_DOUBLE,MPI_SUM,comm);
  MPI_Allreduce( &local_xinf,&global_xin,1,MPI_DOUBLE,MPI_MAX,comm);
  ```

- do collectives on overlapping communicators simultaneously;
- overlap a nonblocking collective with a blocking one.

Exercise 3.20. Revisit exercise 7.1. Let only the first row and first column have certain data, which they broadcast through columns and rows respectively. Each process is now involved in two simultaneous collectives. Implement this with nonblocking broadcasts, and time the difference between a blocking and a nonblocking solution. (There is a skeleton for this exercise under the name `procgridnonblock`.)

Figure 3.19 `MPI_Iallreduce`

Name	Param name	C type	F type	inout
`mpi_iallreduce (`				
	`sendbuf` *starting address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`recvbuf` *starting address of receive buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`count` *number of elements in send buffer*	`int`	`INTEGER`	`in`
	`datatype` *data type of elements of send buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`op` *operation*	`MPI_Op`	`TYPE(MPI_Op)`	`in`
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`request` *communication request*	`MPI_Request*`	`TYPE(MPI_Request)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Iallreduce(`		
	`self`	
	`sendbuf`	`Union[BufSpec, InPlace]`
	`recvbuf`	`BufSpec`
	`op`	`Op = Ellipsis`
`) -> Request`		

Remark 4 *Blocking and nonblocking don't match: either all processes call the nonblocking or all call the blocking one. Thus the following code is incorrect:*

```
if (rank==root)
  MPI_Reduce( &x /* ... */ root,comm );
else
  MPI_Ireduce( &x /* ... */ );
```

This is unlike the point-to-point behavior of nonblocking calls: you can catch a message with `MPI_Irecv` that was sent with `MPI_Send`.

Remark 5 *Unlike sends and received, collectives have no identifying tag. With blocking collectives that does not lead to ambiguity problems. With nonblocking collectives it means that all processes need to issue them in identical order.*

List of nonblocking collectives:

- `MPI_Iallreduce` (figure 3.19)
- `MPI_Igather`,`MPI_Igatherv`, `MPI_Iallgather` (figure 3.20),`MPI_Iallgatherv`,
- `MPI_Iscatter`, `MPI_Iscatterv`,
- `MPI_Ireduce`, `MPI_Iallreduce`, `MPI_Ireduce_scatter`, `MPI_Ireduce_scatter_block`.

Figure 3.20 `MPI_Iallgather`

Name	Param name	C type	F type	inout
`mpi_iallgather (`				
	`sendbuf`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	starting address of send buffer			
	`sendcount`	`int`	`INTEGER`	`in`
	number of elements in send buffer			
	`sendtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of send buffer elements			
	`recvbuf`	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	address of receive buffer			
	`recvcount`	`int`	`INTEGER`	`in`
	number of elements received from any process			
	`recvtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of receive buffer elements			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator			
	`request`	`MPI_Request*`	`TYPE(MPI_Request)`	`out`
	communication request			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Iallgather(`		
	`self`	
	`sendbuf`	`Union[BufSpec, InPlace]`
	`recvbuf`	`BufSpecB`
`) -> Request`		

- `MPI_Ialltoall`,`MPI_Ialltoallv`, `MPI_Ialltoallw`,
- `MPI_Ibarrier`,
- `MPI_Ibcast`,
- `MPI_Iexscan`, `MPI_Iscan`,

MPL note 22. Nonblocking collectives have the same argument list as the corresponding blocking variant, except that instead of a `void` result, they return an `irequest`. (See 27)

```
// ireducescalar.cxx
float x{1.},sum;
auto reduce_request =
  comm_world.ireduce(mpl::plus<float>(), 0, x, sum);
reduce_request.wait();
if (comm_world.rank()==0) {
  std::cout << "sum = " << sum << '\n';
}
```

End of MPL note

3.11.1 Examples

3.11.1.1 Array transpose

To illustrate the overlapping of multiple nonblocking collectives, consider transposing a data matrix. Initially, each process has one row of the matrix; after transposition each process has a column. Since each row needs to be distributed to all processes, algorithmically this corresponds to a series of scatter calls, one originating from each process.

```
// itransposeblock.c
for (int iproc=0; iproc<nprocs; iproc++) {
  MPI_Scatter( regular,1,MPI_DOUBLE,
                &(transpose[iproc]),1,MPI_DOUBLE,
                iproc,comm);
}
```

For the full source of this example, see section 3.16.17

Introducing the nonblocking `MPI_Iscatter` call, this becomes:

```
MPI_Request scatter_requests[nprocs];
for (int iproc=0; iproc<nprocs; iproc++) {
  MPI_Iscatter( regular,1,MPI_DOUBLE,
                 &(transpose[iproc]),1,MPI_DOUBLE,
                 iproc,comm,scatter_requests+iproc);
}
MPI_Waitall(nprocs,scatter_requests,MPI_STATUSES_IGNORE);
```

For the full source of this example, see section 3.16.17

Exercise 3.21. Can you implement the same algorithm with `MPI_Igather`?

3.11.1.2 Stencils

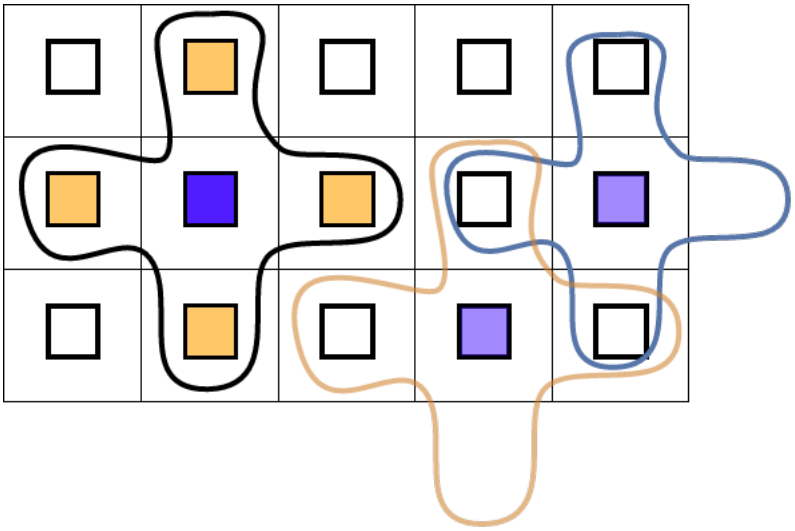


Figure 3.10: Illustration of five-point stencil gather

Figure 3.21 `MPI_Ibarrier`

Name	Param name	C type	F type	inout
`mpi_ibarrier (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`request` *communication request*	`MPI_Request*`	`TYPE(MPI_Request)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Ibarrier(`		
	`self`	
`) -> Request`		

The ever-popular *five-point stencil* evaluation does not look like a collective operation, and indeed, it is usually evaluated with (nonblocking) send/recv operations. However, if we create a subcommunicator on each subdomain that contains precisely that domain and its neighbors, (see figure 3.10) we can formulate the communication pattern as a gather on each of these. With ordinary collectives this can not be formulated in a *deadlock*-free manner, but nonblocking collectives make this feasible.

We will see an even more elegant formulation of this operation in section 11.2.

3.11.2 Nonblocking barrier

Probably the most surprising nonblocking collective is the *nonblocking barrier* `MPI_Ibarrier` (figure 3.21). The way to understand this is to think of a barrier not in terms of temporal synchronization, but state agreement: reaching a barrier is a sign that a process has attained a certain state, and leaving a barrier means that all processes are in the same state. The ordinary barrier is then a blocking wait for agreement, while with a nonblocking barrier:

- Posting the barrier means that a process has reached a certain state; and
- the request being fullfilled means that all processes have reached the barrier.

One scenario would be *local refinement*, where some processes decide to refine their subdomain, which fact they need to communicate to their neighbors. The problem here is that most processes are not among these neighbors, so they should not post a receive of any type. Instead, any refining process sends to its neighbors, and every process posts a barrier.

```
// ibarrierprobe.c
if (i_do_send) {
  /*
   * Pick a random process to send to,
   * not yourself.
   */
  int receiver = rand()%nprocs;
  MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);
}
/*
```

```
 * Everyone posts the non-block barrier
 * and gets a request to test/wait for
 */
MPI_Request barrier_request;
MPI_Ibarrier(comm,&barrier_request);
```

For the full source of this example, see section 3.16.25

Now every process alternately probes for messages and tests for completion of the barrier. Probing is done through the nonblocking `MPI_Iprobe` call, while testing completion of the barrier is done through `MPI_Test`.

```
for ( ; ; step++) {
  int barrier_done_flag=0;
  MPI_Test(&barrier_request,&barrier_done_flag,
           MPI_STATUS_IGNORE);
//stop if you're done!
// if you're not done with the barrier:
    int flag; MPI_Status status;
    MPI_Iprobe
      ( MPI_ANY_SOURCE,MPI_ANY_TAG,
        comm, &flag, &status );
    if (flag) {
// absorb message!
```

For the full source of this example, see section 3.16.25

We can use a nonblocking barrier to good effect, utilizing the idle time that would result from a blocking barrier. In the following code fragment processes test for completion of the barrier, and failing to detect such completion, perform some local work.

```
// findbarrier.c
MPI_Request final_barrier;
MPI_Ibarrier(comm,&final_barrier);

int global_finish=mysleep;
do {
  int all_done_flag=0;
  MPI_Test(&final_barrier,&all_done_flag,MPI_STATUS_IGNORE);
  if (all_done_flag) {
    break;
  } else {
    int flag; MPI_Status status;
// force progress
    MPI_Iprobe
      ( MPI_ANY_SOURCE,MPI_ANY_TAG,
        comm, &flag, MPI_STATUS_IGNORE );
    printf("[%d] going to work for another second\n",procid);
    sleep(1);
    global_finish++;
  }
} while (1);
```

For the full source of this example, see section 3.16.26

3.12 Performance of collectives

It is easy to visualize a broadcast as in figure 3.11: see figure 3.11. the root sends all of its data directly to

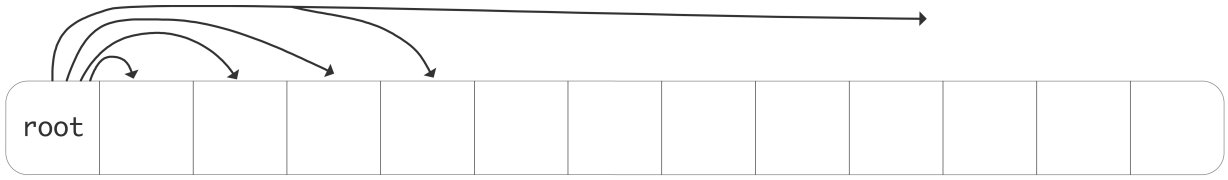


Figure 3.11: A simple broadcast

every other process. While this describes the semantics of the operation, in practice the implementation works quite differently.

The time that a message takes can simply be modeled as

$$\alpha + \beta n,$$

where α is the *latency*, a one time delay from establishing the communication between two processes, and β is the time-per-byte, or the inverse of the *bandwidth*, and n the number of bytes sent.

Under the assumption that a processor can only send one message at a time, the broadcast in figure 3.11 would take a time proportional to the number of processors.

Exercise 3.22. What is the total time required for a broadcast involving p processes? Give α and β terms separately.

One way to ameliorate that is to structure the broadcast in a tree-like fashion. This is depicted in figure 3.12.

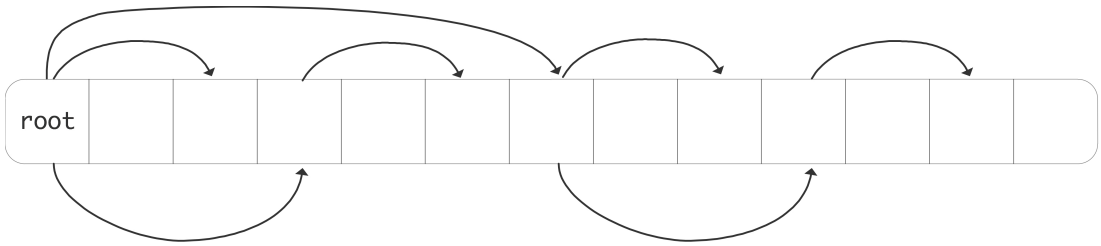


Figure 3.12: A tree-based broadcast

Exercise 3.23. How does the communication time now depend on the number of processors, again α and β terms separately.
What would be a lower bound on the α, β terms?

The theory of the complexity of collectives is described in more detail in HPSC-6.1; see also [3].

3.13 Collectives and synchronization

Collectives, other than a barrier, have a synchronizing effect between processors. For instance, in

```
MPI_Bcast( ....data... root);
MPI_Send(....);
```

the send operations on all processors will occur after the root executes the broadcast. Conversely, in a

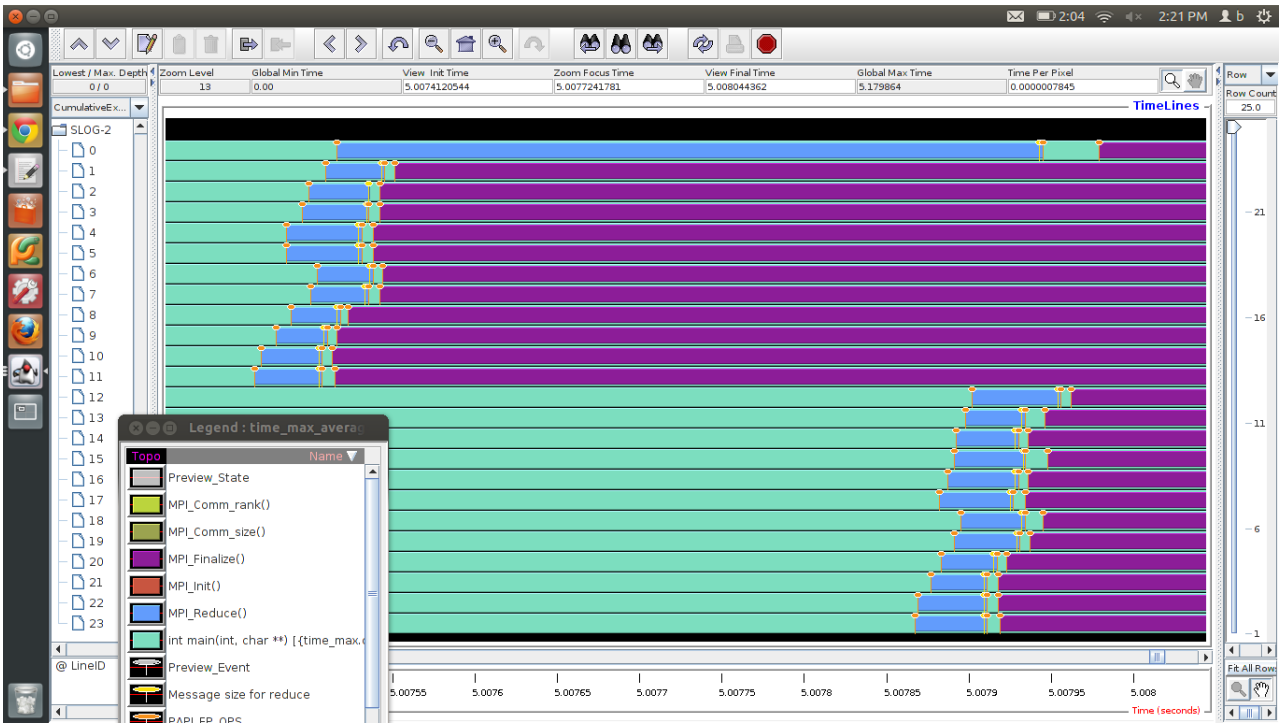


Figure 3.13: Trace of a reduction operation between two dual-socket 12-core nodes

reduce operation the root may have to wait for other processors. This is illustrated in figure 3.13, which gives a TAU trace of a reduction operation on two nodes, with two six-core sockets (processors) each. We see that[1]:

- In each socket, the reduction is a linear accumulation;
- on each node, cores zero and six then combine their result;
- after which the final accumulation is done through the network.

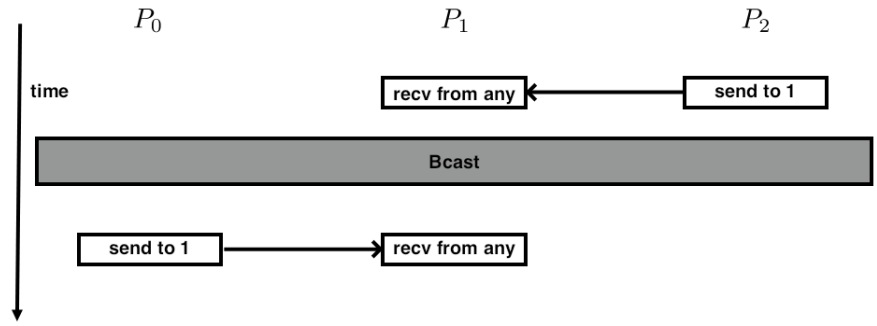
We also see that the two nodes are not perfectly in sync, which is normal for MPI applications. As a result, core 0 on the first node will sit idle until it receives the partial result from core 12, which is on the second node.

While collectives synchronize in a loose sense, it is not possible to make any statements about events before and after the collectives between processors:

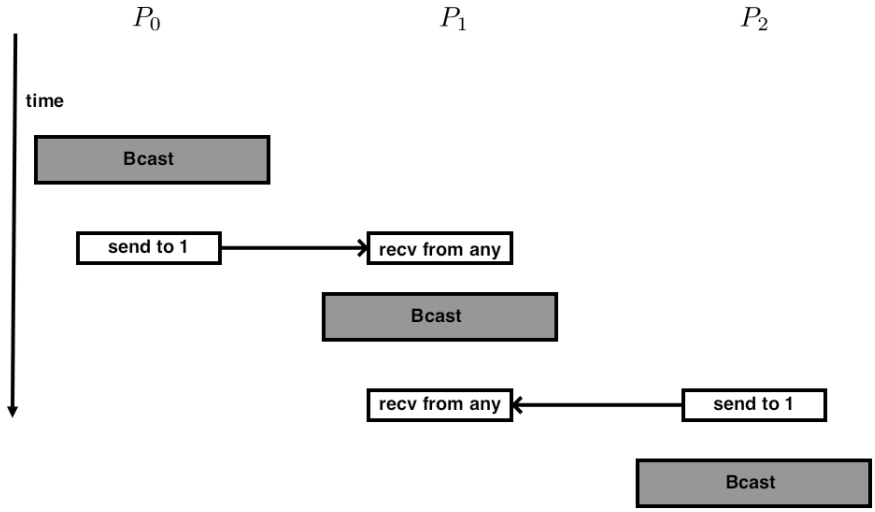
```
...event 1...
MPI_Bcast(....);
...event 2....
```

1. This uses mvapich version 1.6; in version 1.9 the implementation of an on-node reduction has changed to simulate shared memory.

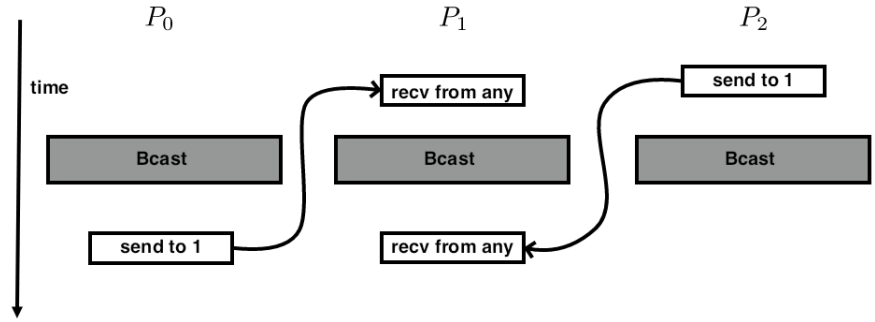
The most logical execution is:



However, this ordering is allowed too:



Which looks from a distance like:



In other words, one of the messages seems to go 'back in time'.

Figure 3.14: Possible temporal orderings of send and collective calls

Consider a specific scenario:

```
switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Send(buf2, count, type, 1, tag, comm);
        break;
    case 1:
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status);
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Recv(buf2, count, type, MPI_ANY_SOURCE, tag, comm, &status);
        break;
    case 2:
        MPI_Send(buf2, count, type, 1, tag, comm);
        MPI_Bcast(buf1, count, type, 0, comm);
        break;
}
```

Note the `MPI_ANY_SOURCE` parameter in the receive calls on processor 1. One obvious execution of this would be:

1. The send from 2 is caught by processor 1;
2. Everyone executes the broadcast;
3. The send from 0 is caught by processor 1.

However, it is equally possible to have this execution:

1. Processor 0 starts its broadcast, then executes the send;
2. Processor 1's receive catches the data from 0, then it executes its part of the broadcast;
3. Processor 1 catches the data sent by 2, and finally processor 2 does its part of the broadcast.

This is illustrated in figure 3.14.

3.14 Performance considerations

In this section we will consider how collectives can be implemented in multiple ways, and the performance implications of such decisions. You can test the algorithms described here using *SimGrid* (section 43.6).

3.14.1 Scalability

We are motivated to write parallel software from two considerations. First of all, if we have a certain problem to solve which normally takes time T, then we hope that with p processors it will take time T/p. If this is true, we call our parallelization scheme *scalable in time*. In practice, we often accept small extra terms: as you will see below, parallelization often adds a term $\log_2 p$ to the running time.

Exercise 3.24. Discuss scalability of the following algorithms:

- You have an array of floating point numbers. You need to compute the sine of each
- You a two-dimensional array, denoting the interval $[-2,2]^2$. You want to make a picture of the *Mandelbrot set*, so you need to compute the color of each point.

- The primality test of exercise 2.6.

There is also the notion that a parallel algorithm can be *scalable in space*: more processors gives you more memory so that you can run a larger problem.

Exercise 3.25. Discuss space scalability in the context of modern processor design.

3.14.2 Complexity and scalability of collectives

3.14.2.1 Broadcast

Naive broadcast Write a broadcast operation where the root does an `MPI_Send` to each other process.

What is the expected performance of this in terms of α, β?

Run some tests and confirm.

Simple ring Let the root only send to the next process, and that one send to its neighbour. This scheme is known as a *bucket brigade*; see also section 4.1.3.

What is the expected performance of this in terms of α, β?

Run some tests and confirm.

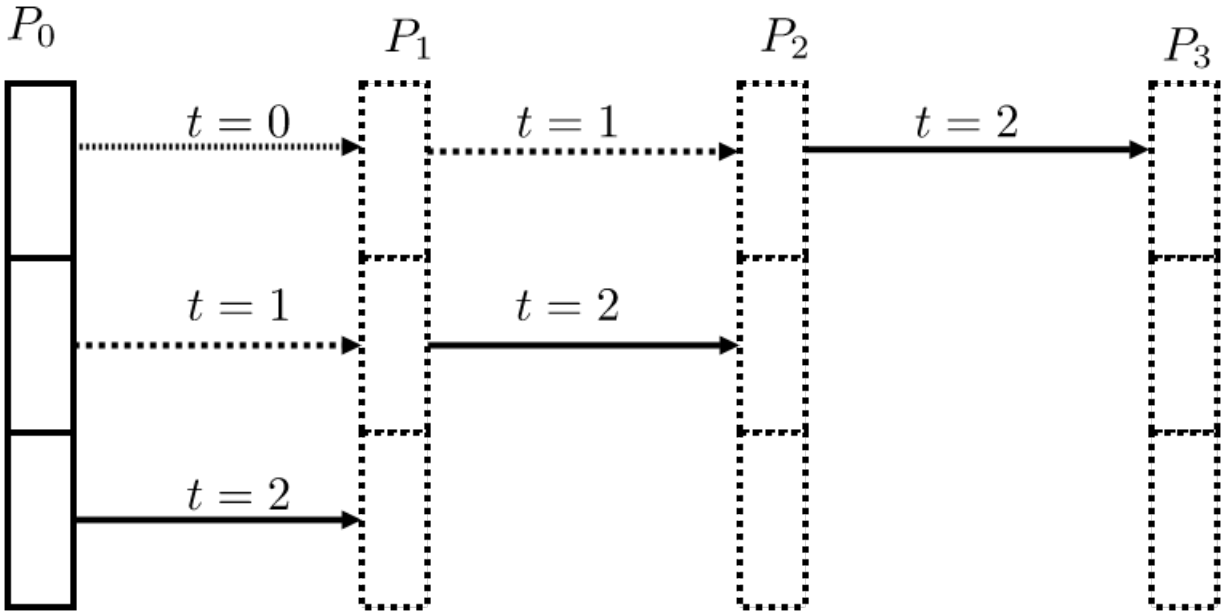


Figure 3.15: A pipelined bucket brigade

Pipelined ring In a ring broadcast, each process needs to receive the whole message before it can pass it on. We can increase the efficiency by breaking up the message and sending it in multiple parts.

(See figure 3.15.) This will be advantageous for messages that are long enough that the bandwidth cost dominates the latency.

Assume a send buffer of length more than 1. Divide the send buffer into a number of chunks. The root sends the chunks successively to the next process, and each process sends on whatever chunks it receives.

What is the expected performance of this in terms of α, β? Why is this better than the simple ring?

Run some tests and confirm.

Recursive doubling Collectives such as broadcast can be *implemented* through *recursive doubling*, where the root sends to another process, then the root and the other process send to two more, those four send to four more, et cetera. However, in an actual physical architecture this scheme can be realized in multiple ways that have drastically different performance.

First consider the implementation where process 0 is the root, and it starts by sending to process 1; then they send to 2 and 3; these four send to 4–7, et cetera. If the architecture is a linear array of procesors, this will lead to *contention*: multiple messages wanting to go through the same wire. (This is also related to the concept of *bisecection bandwidth*.)

In the following analyses we will assume *wormhole routing*: a message sets up a path through the network, reserving the necessary wires, and performing a send in time independent of the distance through the network. That is, the send time for any message can be modeled as

$$T(n) = \alpha + \beta n$$

regardless source and destination, as long as the necessary connections are available.

Exercise 3.26. Analyze the running time of a recursive doubling broad cast as just described, with wormhole routing.
Implement this broadcast in terms of blocking MPI send and receive calls. If you have SimGrid available, run tests with a number of parameters.

The alternative, that avoids contention, is to let each doubling stage divide the network into separate halves. That is, process 0 sends to $P/2$, after which these two repeat the algorithm in the two halves of the network, sending to $P/4$ and $3P/4$ respectively.

Exercise 3.27. Analyze this variant of recursive doubling. Code it and measure runtimes on SimGrid.

Exercise 3.28. Revisit exercise 3.26 and replace the blocking calls by nonblocking `MPI_Isend` / `MPI_Irecv` calls.
Make sure to test that the data is correctly propagated.

MPI implementations often have multiple algorithms, which they dynamicaly switch between. Sometimes you can determine the choice yourself through environment variables.

TACC note. For *Intel MPI*, see https://software.intel.com/en-us/mpi-developer-reference-linux-i-mpi-adjust-family-environment-variables.

3.15 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

Review 3.29. How would you realize the following scenarios with MPI collectives?

- Let each process compute a random number. You want to print the maximum of these numbers to your screen.
- Each process computes a random number again. Now you want to scale these numbers by their maximum.
- Let each process compute a random number. You want to print on what processor the maximum value is computed.

Review 3.30. MPI collectives can be sorted in at least the following categories

1. rooted vs rootless
2. using uniform buffer lengths vs variable length buffers
3. blocking vs nonblocking.

Give examples of each type.

Review 3.31. True or false: there are collective routines that do not communicate user data. If true, give an example.

Review 3.32. True or false: an `MPI_Scatter` call puts the same data on each process.

Review 3.33. True or false: using the option `MPI_IN_PLACE` you only need space for a send buffer in `MPI_Reduce`.

Review 3.34. True or false: using the option `MPI_IN_PLACE` you only need space for a send buffer in `MPI_Gather`.

Review 3.35. Given a distributed array, with every processor storing

```
double x[N]; // N can vary per processor
```

give the approximate MPI-based code that computes the maximum value in the array, and leaves the result on every processor.

Review 3.36.

```
double data[Nglobal];
int myfirst = /* something */, mylast = /* something */;
for (int i=myfirst; i<mylast; i++) {
  if (i>0 && i<N-1) {
    process_point( data,i,Nglobal );
  }
}
void process_point( double *data,int i,int N ) {
    data[i-1] = g(i-1); data[i] = g(i); data[i+1] = g(i+1);
    data[i] = f(data[i-1],data[i],data[i+1]);
}
```

Is this scalable in time? Is this scalable in space?

Review 3.37.

```
double data[Nlocal+2]; // include left and right neighbor
int myfirst = /* something */, mylast = myfirst+Nlocal;
```

```
for (int i=0; i<Nlocal; i++) {
  if (i>0 && i<N-1) {
    process_point( data,i,Nlocal );
}
void process_point( double *data,int i0,int n ) {
  int i = i0+1;
  data[i-1] = g(i-1); data[i] = g(i); data[i+1] = g(i+1);
  data[i] = f(data[i-1],data[i],data[i+1]);
}
```

Is this scalable in time? Is this scalable in space?

Review 3.38. With data as in the previous question, given the code for normalizing the array, that is, scaling each element so that $\|x\|_2 = 1$.

Review 3.39. Just like `MPI_Allreduce` is equivalent to `MPI_Reduce` following by `MPI_Bcast`, `MPI_Reduce_scatter` is equivalent to at least one of the following combinations. Select those that are equivalent, and discuss differences in time or space complexity:

1. `MPI_Reduce` followed by `MPI_Scatter`;
2. `MPI_Gather` followed by `MPI_Scatter`;
3. `MPI_Allreduce` followed by `MPI_Scatter`;
4. `MPI_Allreduce` followed by a local operation (which?);
5. `MPI_Allgather` followed by a local operation (which?).

Review 3.40. Think of at least two algorithms for doing a broadcast? Compare them with regards to asymptotic behavior.

3.16 Sources used in this chapter

3.16.1 Listing of code header

3.16.2 Listing of code examples/mpi/c/allreduce.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  float myrandom,sumrandom;
  myrandom = (float) rand()/(float)RAND_MAX;
  // add the random variables together
  MPI_Allreduce(&myrandom,&sumrandom,
   1,MPI_FLOAT,MPI_SUM,comm);
  // the result should be approx nprocs/2:
  if (procno==nprocs-1)
    printf("Result %6.9f compared to .5\n",sumrandom/nprocs);

  MPI_Finalize();
  return 0;
}
```

3.16.3 Listing of code examples/mpi/mpl/collectscalar.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
#include <vector>
#include <mpl/mpl.hpp>

int main() {

  // MPI Comm world
  const mpl::communicator &comm_world=mpl::environment::comm_world();

  // vector of consecutive floats
  std::vector<float> v;
  if (comm_world.rank()==0)
    for (int i=0; i<comm_world.size(); ++i)
      v.push_back(i);

  // if you scatter, everyone gets a number equal to their rank.
  // rank 0 scatters data to all processes
  float x;
  comm_world.scatter(0, v.data(), x);
```

```
std::cout << "rank " << comm_world.rank() << " got " << x << '\n';

// wait until all processes have reached this point
comm_world.barrier();

// multiply that number, giving twice your rank
x*=2;

// rank 0 gathers data from all processes
comm_world.gather(0, x, v.data());
if (comm_world.rank()==0) {
  std::cout << "got";
  for (int i=0; i<comm_world.size(); ++i)
    std::cout << " " << i << ":" << v[i];
  std::cout << std::endl;
}

// wait until all processes have reached this point
comm_world.barrier();

// calculate global sum and pass result to rank 0
if (comm_world.rank()==0) {
  float sum;
  comm_world.reduce(mpl::plus<float>(), 0, x, sum);
  std::cout << "sum = " << sum << '\n';
} else
  comm_world.reduce(mpl::plus<float>(), 0, x);

// wait until all processes have reached this point
comm_world.barrier();

// calculate global sum and pass result to all
{
  float
    xrank = static_cast<float>( comm_world.rank() ),
    xreduce;
  // separate recv buffer
  comm_world.allreduce(mpl::plus<float>(), xrank,xreduce);
  // in place
  comm_world.allreduce(mpl::plus<float>(), xrank);
  if ( comm_world.rank()==comm_world.size()-1 )
    std::cout << "Allreduce got: separate=" << xreduce
              << ", inplace=" << xrank << std::endl;
}

// calculate global sum and pass result to root
{
  int root = 1;
  float
    xrank = static_cast<float>( comm_world.rank() ),
    xreduce;
  // separate receive buffer
  comm_world.reduce(mpl::plus<float>(), root, xrank,xreduce);
```

```
    // in place
    comm_world.reduce(mpl::plus<float>(), root, xrank);
    if ( comm_world.rank()==root )
      std::cout << "Allreduce got: separate=" << xreduce
<< ", inplace=" << xrank << std::endl;
  }

  return EXIT_SUCCESS;
}
```

3.16.4 Listing of code examples/mpi/p/allreduce.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

random.seed(procid)
random_bound = nprocs*nprocs
random_number = random.randint(1,random_bound)
#print("[%d] random=%d" % (procid,random_number))

# native mode send
max_random = comm.allreduce(random_number,op=MPI.MAX)

if procid==0:
    print("Python native:\n  max=%d" % max_random)

myrandom = np.empty(1,dtype=np.int)
myrandom[0] = random_number
allrandom = np.empty(nprocs,dtype=np.int)
# numpy mode send
comm.Allreduce(myrandom,allrandom[:1],op=MPI.MAX)

if procid==0:
    print("Python numpy:\n  max=%d" % allrandom[0])

sumrandom = np.zeros(1,dtype=np.int)
sumrandom[0] = myrandom[0]
#### WRONG polymorphic use does not work
#comm.Allreduce(sumrandom[:1])
comm.Allreduce(MPI.IN_PLACE,sumrandom[:1],op=MPI.SUM)

if procid==0:
    print( "Sum of randoms: %d, compare %d" % (sumrandom[0],nprocs*random_bound/2) )
```

3.16.5 Listing of code examples/mpi/mpl/sendrange.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  if (comm_world.size()<2)
    return EXIT_FAILURE;

  vector<double> v(15);

  if (comm_world.rank()==0) {

    // initialize
    for ( auto &x : v ) x = 1.41;

    /*
     * Send and report
     */
    comm_world.send(v.begin(), v.end(), 1);  // send to rank 1

  } else if (comm_world.rank()==1) {

    /*
     * Receive data and report
     */
    comm_world.recv(v.begin(), v.end(), 0);  // receive from rank 0

    cout << "Got:";
    for ( auto x : v )
      cout << " " << x;
    cout << endl;
  }
  return EXIT_SUCCESS;
}
```

3.16.6 Listing of code examples/mpi/c/reduce.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"
```

```
int main(int argc,char **argv) {

#include "globalinit.c"

  float myrandom = (float) rand()/(float)RAND_MAX,
    result;
  int target_proc = nprocs-1;
  // add all the random variables together
  MPI_Reduce(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,
             target_proc,comm);
  // the result should be approx nprocs/2:
  if (procno==target_proc)
    printf("Result %6.3f compared to nprocs/2=%5.2f\n",
           result,nprocs/2.);

  MPI_Finalize();
  return 0;
}
```

3.16.7 Listing of code examples/mpi/c/allreduceinplace.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  int nrandoms = 500000;
  float *myrandoms;
  myrandoms = (float*) malloc(nrandoms*sizeof(float));
  for (int iter=1; iter<=3; iter++) {
    /*
     * We show three different ways of doing the same reduction;
     * this illustrates syntax more than semantics
     */
    if (iter==1) {
      for (int irand=0; irand<nrandoms; irand++)
myrandoms[irand] = (float) rand()/(float)RAND_MAX;
      // add all the random variables together
      MPI_Allreduce(MPI_IN_PLACE,myrandoms,
    nrandoms,MPI_FLOAT,MPI_SUM,comm);
    } else if (iter==2) {
      for (int irand=0; irand<nrandoms; irand++)
myrandoms[irand] = (float) rand()/(float)RAND_MAX;
      int root=nprocs-1;
      if (procno==root)
        MPI_Reduce(MPI_IN_PLACE,myrandoms,
   nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
      else
        MPI_Reduce(myrandoms,MPI_IN_PLACE,
```

```
   nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
    } else if (iter==2) {
      for (int irand=0; irand<nrandoms; irand++)
myrandoms[irand] = (float) rand()/(float)RAND_MAX;
      int root=nprocs-1;
      float *sendbuf,*recvbuf;
      if (procno==root) {
        sendbuf = MPI_IN_PLACE; recvbuf = myrandoms;
      } else {
        sendbuf = myrandoms; recvbuf = MPI_IN_PLACE;
      }
      MPI_Reduce(sendbuf,recvbuf,
          nrandoms,MPI_FLOAT,MPI_SUM,root,comm);
    }
    // the result should be approx nprocs/2:
    if (procno==nprocs-1) {
      float sum=0.;
      for (int i=0; i<nrandoms; i++) sum += myrandoms[i];
      sum /= nrandoms*nprocs;
      printf("Result %6.9f compared to .5\n",sum);
    }
  }
  free(myrandoms);

  MPI_Finalize();
  return 0;
}
```

3.16.8 Listing of code examples/mpi/f/reduceinplace.F90

```
Program ReduceInPlace

  use mpi

  real :: mynumber,result
  integer :: target_proc

#include "globalinit.F90"

  call random_number(mynumber)
  target_proc = ntids-1;
  ! add all the random variables together
  if (mytid.eq.target_proc) then
     result = mytid
     call MPI_Reduce(MPI_IN_PLACE,result,1,MPI_REAL,MPI_SUM,&
          target_proc,comm,err)
  else
     mynumber = mytid
     call MPI_Reduce(mynumber,result,1,MPI_REAL,MPI_SUM,&
          target_proc,comm,err)
  end if
  ! the result should be ntids*(ntids-1)/2:
  if (mytid.eq.target_proc) then
```

```
    write(*,'("Result ",f5.2," compared to n(n-1)/2=",f5.2)') &
         result,ntids*(ntids-1)/2.
 end if

  call MPI_Finalize(err)

end Program ReduceInPlace
```

3.16.9 Listing of code examples/mpi/f08/reduceinplaceptr.F90

```
Program ReduceInPlace

  use mpi_f08

  real,target :: mynumber,result,in_place_val
  real,pointer :: mynumber_ptr,result_ptr
  integer :: target_proc

#include "globalinit.F90"

  call random_number(mynumber)
  target_proc = ntids-1;
  in_place_val = MPI_IN_PLACE
  if (mytid.eq.target_proc) then
     ! set pointers
     result_ptr => result
     mynumber_ptr => in_place_val
     ! target sets value in receive buffer
     result_ptr = mytid
  else
     ! set pointers
     mynumber_ptr => mynumber
     result_ptr => in_place_val
     ! non-targets set value in send buffer
     mynumber_ptr = mytid
  end if
  call MPI_Reduce(mynumber_ptr,result_ptr,1,MPI_REAL,MPI_SUM,&
       target_proc,comm,err)
  ! the result should be ntids*(ntids-1)/2:
  if (mytid.eq.target_proc) then
    write(*,'("Result ",f7.4," compared to n(n-1)/2=",f7.4)') &
         result,ntids*(ntids-1)/2.
 end if

  call MPI_Finalize(err)

end Program ReduceInPlace
```

3.16.10 Listing of code examples/mpi/p/allreduceinplace.py

```
import numpy as np
import random
```

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

random_number = random.randint(1,nprocs*nprocs)
print("[%d] random=%d" % (procid,random_number))

myrandom = np.empty(1,dtype=np.int)
myrandom[0] = random_number

comm.Allreduce(MPI.IN_PLACE,myrandom,op=MPI.MAX)

if procid==0:
    print("Python numpy:\n  max=%d" % myrandom[0])
```

3.16.11 Listing of code examples/mpi/mpl/collectbuffer.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <mpl/mpl.hpp>

int main() {

  // MPI Comm world
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  int nprocs = comm_world.size(), procno = comm_world.rank();
  int iprint = procno==nprocs-1;

  /*
   * Reduce a 2 int buffer
   */
  if (iprint) cout << "Reducing 2p, 2p+1" << endl;

  float
    xrank = static_cast<float>( comm_world.rank() );
  vector<float> rank2p2p1{ 2*xrank,2*xrank+1 },reduce2p2p1{0,0};
  mpl::contiguous_layout<float> two_floats(rank2p2p1.size());
  comm_world.allreduce(mpl::plus<float>(), rank2p2p1.data(),reduce2p2p1,two_floats);
  if ( iprint )
    cout << "Got: " << reduce2p2p1.at(0) << ","
```

```
<< reduce2p2p1.at(1) << endl;

 /*
  * Scatter one number to each proc
  */
 if (iprint) cout << "Scattering 0--p" << endl;

 vector<float> v;

 if (comm_world.rank()==0)
   for (int i=0; i<comm_world.size(); ++i)
     v.push_back(i);

 // if you scatter, everyone gets a number equal to their rank.
 // rank 0 scatters data to all processes
 float x;
 comm_world.scatter(0, v.data(), x);

 if (iprint)
   cout << "rank " << procno << " got " << x << '\n';

 /*
  * Scatter two numbers to each proc
  */

 if (iprint) cout << "Scatter 0--2p" << endl;

 vector<float> vrecv(2),vsend(2*nprocs);

 if (comm_world.rank()==0)
   for (int i=0; i<2*nprocs; ++i)
     vsend.at(i) = i;

 // rank 0 scatters data to all processes
 // if you scatter, everyone gets 2p,2p+1
 mpl::contiguous_layout<float> twonums(2);
 comm_world.scatter
   (0, vsend.data(),twonums, vrecv.data(),twonums );

 if (iprint)
   cout << "rank " << procno << " got "
        << vrecv[0] << "," << vrecv[1] << '\n';

 return 0;
 // multiply that number, giving twice your rank
 x*=2;

 // rank 0 gathers data from all processes
 comm_world.gather(0, x, v.data());
 if (comm_world.rank()==0) {
   cout << "got";
   for (int i=0; i<comm_world.size(); ++i)
     cout << " " << i << ":" << v[i];
```

```
    cout << endl;
  }

  // wait until all processes have reached this point
  comm_world.barrier();

  // calculate global sum and pass result to rank 0
  if (comm_world.rank()==0) {
    float sum;
    comm_world.reduce(mpl::plus<float>(), 0, x, sum);
    cout << "sum = " << sum << '\n';
  } else
    comm_world.reduce(mpl::plus<float>(), 0, x);

  // wait until all processes have reached this point
  comm_world.barrier();

  return EXIT_SUCCESS;
}
```

3.16.12 Listing of code examples/mpi/c/init.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
  int input_argument;

#include "globalinit.c"

  if (procno==0) {
    if ( argc==1 || // the program is called without parameter
         ( argc>1 && !strcmp(argv[1],"-h") ) // user asked for help
         ) {
      printf("\nUsage: init [0-9]+\n");
      MPI_Abort(comm,1);
    }
    input_argument = atoi(argv[1]);
  }
  MPI_Bcast(&input_argument,1,MPI_INT,0,comm);
  printf("Processor %d gets %d\n",procno,input_argument);

  MPI_Finalize();
  return 0;
}
```

3.16.13 Listing of code examples/mpi/p/bcast.py

```
from mpi4py import MPI
```

```
import numpy as np

from functools import reduce
import sys

comm = MPI.COMM_WORLD

procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

root = 1
dsize = 10

# first native
if procid==root:
    buffer = [ 5.0 ] * dsize
else:
    buffer = [ 0.0 ] * dsize
buffer = comm.bcast(obj=buffer,root=root)
if not reduce( lambda x,y:x and y,
               [ buffer[i]==5.0 for i in  range(len(buffer)) ] ):
    print( "Something wrong on proc %d: native buffer <<%s>>" \
           % (procid,str(buffer)) )

# then with NumPy
buffer = np.arange(dsize, dtype=np.float64)
if procid==root:
    for i in range(dsize):
        buffer[i] = 5.0
comm.Bcast( buffer,root=root )
if not all( buffer==5.0 ):
    print( "Something wrong on proc %d: numpy buffer <<%s>>" \
           % (procid,str(buffer)) )
else:
    if procid==root:
        print("Success.")
```

3.16.14 Listing of code examples/mpi/c/scan.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  float myrandom = (float) rand()/(float)RAND_MAX,
    result;
```

```
  // add all the random variables together
  MPI_Scan(&myrandom,&result,1,MPI_FLOAT,MPI_SUM,comm);
  // the result should be approaching nprocs/2:
  if (procno==nprocs-1)
    printf("Result %6.3f compared to nprocs/2=%5.2f\n",
           result,nprocs/2.);

  MPI_Finalize();
  return 0;
}
```

3.16.15 Listing of code examples/mpi/p/scan.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

mycontrib = 10+random.randint(1,nprocs)
myfirst = 0
mypartial = comm.scan(mycontrib)
print("[%d] local: %d, partial: %d" % (procid,mycontrib,mypartial))

sbuf = np.empty(1,dtype=np.int)
rbuf = np.empty(1,dtype=np.int)
sbuf[0] = mycontrib
comm.Scan(sbuf,rbuf)

print("[%d] numpy local: %d, partial: %d" % (procid,mycontrib,rbuf[0]))
```

3.16.16 Listing of code examples/mpi/c/gather.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"
  int localsize =  10+10*( (float) rand()/(float)RAND_MAX - .5),
    root = nprocs-1;

  int *localsizes=NULL;
  // create local data
```

```
  int *localdata = (int*) malloc( localsize*sizeof(int) );
  for (int i=0; i<localsize; i++)
    localdata[i] = procno+1;
  // we assume that each process has a value "localsize"
  // the root process collects these values

  if (procno==root)
    localsizes = (int*) malloc( nprocs*sizeof(int) );

  // everyone contributes their info
  MPI_Gather(&localsize,1,MPI_INT,
             localsizes,1,MPI_INT,root,comm);
  if (procno==root) {
    printf("Local sizes: ");
    for (int i=0; i<nprocs; i++)
      printf("%d, ",localsizes[i]);
    printf("\n");
  }

  MPI_Finalize();
  return 0;
}
```

3.16.17 Listing of code examples/mpi/c/itransposeblock.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {

#include "globalinit.c"

  /*
   * Allocate matrix and transpose:
   * - one column per rank for regular
   * - one row per rank for transpose
   */
  double *regular,*transpose;
  regular = (double*) malloc( nprocs*sizeof(double) );
  transpose = (double*) malloc( nprocs*sizeof(double) );
  // each process has columnns m*nprocs -- m*(nprocs+1)
  for (int ip=0; ip<nprocs; ip++)
    regular[ip] = procno*nprocs + ip;

  /*
   * Each proc does a scatter
   */
#if 0
  // reference code:
  for (int iproc=0; iproc<nprocs; iproc++) {
    MPI_Scatter( regular,1,MPI_DOUBLE,
```

```
 &(transpose[iproc]),1,MPI_DOUBLE,
 iproc,comm);
  }
#else
  MPI_Request scatter_requests[nprocs];
  for (int iproc=0; iproc<nprocs; iproc++) {
    MPI_Iscatter( regular,1,MPI_DOUBLE,
  &(transpose[iproc]),1,MPI_DOUBLE,
  iproc,comm,scatter_requests+iproc);
  }
  MPI_Waitall(nprocs,scatter_requests,MPI_STATUSES_IGNORE);
#endif

  /*
   * Check the result
   */
  printf("[%d] :",procno);
  for (int ip=0; ip<nprocs; ip++)
    printf(" %5.2f",transpose[ip]);
  printf("\n");

  MPI_Finalize();

  return 0;
}
```

3.16.18 Listing of code examples/mpi/c/reducescatter.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  /*
   * Set up an array of which processes you will receive from
   */
  int
    // data that we know:
    *i_recv_from_proc = (int*) malloc(nprocs*sizeof(int)),
    *procs_to_recv_from, nprocs_to_recv_from=0,
    // data we are going to determin:
    *procs_to_send_to,nprocs_to_send_to;

  /*
   * Initialize
   */
  for (int i=0; i<nprocs; i++) {
    i_recv_from_proc[i] = 0;
  }
```

```
/*
 * Generate array of "yes/no I recv from proc p",
 * and condensed array of procs I receive from.
 */
nprocs_to_recv_from = 0;
for (int iproc=0; iproc<nprocs; iproc++)
  // pick random procs to receive from, not yourself.
  if ( (float) rand()/(float)RAND_MAX < 2./nprocs && iproc!=procno ) {
    i_recv_from_proc[iproc] = 1;
    nprocs_to_recv_from++;
  }
procs_to_recv_from = (int*) malloc(nprocs_to_recv_from*sizeof(int));
int count_procs_to_recv_from = 0;
for (int iproc=0; iproc<nprocs; iproc++)
  if ( i_recv_from_proc[iproc] )
    procs_to_recv_from[count_procs_to_recv_from++] = iproc;
ASSERT( count_procs_to_recv_from==nprocs_to_recv_from );

/*
 */
printf("[%d] receiving from:",procno);
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++)
  printf(" %3d",procs_to_recv_from[iproc]);
printf(".\n");

/*
 * Now find how many procs will send to you
 */
MPI_Reduce_scatter_block
  (i_recv_from_proc,&nprocs_to_send_to,1,MPI_INT,
  MPI_SUM,comm);

/*
 * Send a zero-size msg to everyone that you receive from,
 * just to let them know that they need to send to you.
 */
MPI_Request send_requests[nprocs_to_recv_from];
for (int iproc=0; iproc<nprocs_to_recv_from; iproc++) {
  int proc=procs_to_recv_from[iproc];
  double send_buffer=0.;
  MPI_Isend(&send_buffer,0,MPI_DOUBLE, /*to:*/ proc,0,comm,
    &(send_requests[iproc]));
}

/*
 * Do as many receives as you know are coming in;
 * use wildcards since you don't know where they are coming from.
 * The source is a process you need to send to.
 */
procs_to_send_to = (int*)malloc( nprocs_to_send_to * sizeof(int) );
for (int iproc=0; iproc<nprocs_to_send_to; iproc++) {
  double recv_buffer;
```

```
    MPI_Status status;
    MPI_Recv(&recv_buffer,0,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,
     &status);
    procs_to_send_to[iproc] = status.MPI_SOURCE;
  }
  MPI_Waitall(nprocs_to_recv_from,send_requests,MPI_STATUSES_IGNORE);

  printf("[%d] sending to:",procno);
  for (int iproc=0; iproc<nprocs_to_send_to; iproc++)
    printf(" %3d",procs_to_send_to[iproc]);
  printf(".\n");

  MPI_Finalize();
  return 0;
}
```

3.16.19 Listing of code examples/mpi/c/gatherv.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"
  int localsize =  10+10*( (float) rand()/(float)RAND_MAX - .5),
    root = nprocs-1;

  int *localsizes=NULL,*offsets=NULL,*localdata=NULL,*alldata=NULL;
  // create local data
  localdata = (int*) malloc( localsize*sizeof(int) );
  for (int i=0; i<localsize; i++)
    localdata[i] = procno+1;
  // we assume that each process has an array "localdata"
  // of size "localsize"

  // the root process decides how much data will be coming:
  // allocate arrays to contain size and offset information
  if (procno==root) {
    localsizes = (int*) malloc( nprocs*sizeof(int) );
    offsets = (int*) malloc( nprocs*sizeof(int) );
  }
  // everyone contributes their local size info
  MPI_Gather(&localsize,1,MPI_INT,
     localsizes,1,MPI_INT,root,comm);

  if (procno==root) {
    printf("Local sizes: ");
    for (int i=0; i<nprocs; i++)
      printf("%d, ",localsizes[i]);
    printf("\n");
  }
```

```
  // the root constructs the offsets array
  if (procno==root) {
    int total_data = 0;
    for (int i=0; i<nprocs; i++) {
      offsets[i] = total_data;
      total_data += localsizes[i];
    }
    alldata = (int*) malloc( total_data*sizeof(int) );
  }
  // everyone contributes their data
  MPI_Gatherv(localdata,localsize,MPI_INT,
      alldata,localsizes,offsets,MPI_INT,root,comm);

  if (procno==root) {
    int p=0;
    printf("Collected:\n");
    for (int i=0; i<nprocs; i++) {
      int j;
      printf("  %d:",i);
      for (j=0; j<localsizes[i]-1; j++)
printf("%d,",alldata[p++]);
      j=localsizes[i]-1;
      printf("%d;\n",alldata[p++]);
    }
  }

  MPI_Finalize();
  return 0;
}
```

3.16.20 Listing of code examples/mpi/p/gatherv.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

localsize = random.randint(2,10)
print("[%d] local size=%d" % (procid,localsize))
localdata = np.empty(localsize,dtype=np.int)
for i in range(localsize):
    localdata[i] = procid

# implicitly using root=0
globalsize = comm.reduce(localsize)
if procid==0:
```

```
    print("Global size=%d" % globalsize)
collecteddata = np.empty(globalsize,dtype=np.int)
counts = comm.gather(localsize)
comm.Gatherv(localdata, [collecteddata, counts])
if procid==0:
    print("Collected",str(collecteddata))
```

3.16.21 Listing of code examples/mpi/c/allgatherv.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  int my_count = procno+1;
  int *my_array = (int*) malloc(my_count*sizeof(int));
  for (int i=0; i<my_count; i++)
    my_array[i] = procno;
  int *recv_counts = (int*) malloc(nprocs*sizeof(int));
  int *recv_displs = (int*) malloc(nprocs*sizeof(int));

  MPI_Allgather
    ( &my_count,1,MPI_INT,
      recv_counts,1,MPI_INT, comm );
  int accumulate = 0;
  for (int i=0; i<nprocs; i++) {
    recv_displs[i] = accumulate; accumulate += recv_counts[i]; }
  int *global_array = (int*) malloc(accumulate*sizeof(int));
  MPI_Allgatherv
    ( my_array,procno+1,MPI_INT,
      global_array,recv_counts,recv_displs,MPI_INT, comm );

  if (procno==0) {
    for (int p=0; p<nprocs; p++)
      if (recv_counts[p]!=p+1)
printf("count[%d] should be %d, not %d\n",
       p,p+1,recv_counts[p]);
    int c = 0;
    for (int p=0; p<nprocs; p++)
      for (int q=0; q<=p; q++)
if (global_array[c++]!=p)
  printf("p=%d, q=%d should be %d, not %d\n",
 p,q,p,global_array[c-1]);
  }

  MPI_Finalize();
  return 0;
}
```

3.16.22 Listing of code examples/mpi/p/allgatherv.py

```
import numpy as np
import random # random.randint(1,N), random.random()
import sys
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)

for i in range(mycount):
    my_array[i] = procid
recv_counts = np.empty(nprocs,dtype=np.int)
recv_displs = np.empty(nprocs,dtype=np.int)

my_count = np.empty(1,dtype=np.int)
my_count[0] = mycount
comm.Allgather( my_count,recv_counts )

accumulate = 0
for p in range(nprocs):
    recv_displs[p] = accumulate; accumulate += recv_counts[p]
global_array = np.empty(accumulate,dtype=np.float64)
comm.Allgatherv( my_array, [global_array,recv_counts,recv_displs,MPI.DOUBLE] )

# other syntax:
# comm.Allgatherv( [my_array,mycount,0,MPI.DOUBLE], [global_array,recv_counts,recv_displs,MPI.DOUBLE] )

if procid==0:
    #print(procid,global_array)
    for p in range(nprocs):
        if recv_counts[p]!=p+1:
            print( "recv count[%d] should be %d, not %d" \
                % (p,p+1,recv_counts[p]) )
    c = 0
    for p in range(nprocs):
        for q in range(p+1):
            if global_array[c]!=p:
                print( "p=%d, q=%d should be %d, not %d" \
                    % (p,q,p,global_array[c]) )
            c += 1
    print "finished"
```

3.16.23 Listing of code examples/mpi/c/reductpositive.c

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <string.h>
#include "mpi.h"

//int reduce_without_zero(int r,int n);
void reduce_without_zero(void *in,void *inout,int *len,MPI_Datatype *type) {
  // r is the already reduced value, n is the new value
  int n = *(int*)in, r = *(int*)inout;
  int m;
  if (n==0) { // new value is zero: keep r
    m = r;
  } else if (r==0) {
    m = n;
  } else if (n<r) { // new value is less but not zero: use n
    m = n;
  } else { // new value is more: use r
    m = r;
  };
#ifdef DEBUG
  printf("combine %d %d : %d\n",r,n,m);
#endif
  // return the new value
  *(int*)inout = m;
}

int main(int argc,char **argv) {

#include "globalinit.c"

  int m,mreduct=2000000000,ndata = 10, data[10] = {2,3,0,5,0,1,8,12,4,0},
    positive_minimum;
  if (nprocs>ndata) {
    printf("Too many procs for this example: at most %d\n",ndata);
    return 1;
  }

  for (int i=0; i<nprocs; i++)
    if (data[i]<mreduct && data[i]>0)
      mreduct = data[i];

  MPI_Op rwz;
  MPI_Op_create(reduce_without_zero,1,&rwz);
  MPI_Allreduce(data+procno,&positive_minimum,1,MPI_INT,rwz,comm);

  // check that the distributed result is the same as sequential
  if (mreduct!=positive_minimum)
    printf("[%d] Result %d should be %d\n",
           procno,positive_minimum,mreduct);
  else if (procno==0)
    printf("User-defined reduction successful: %d\n",positive_minimum);

  MPI_Finalize();
  return 0;
```

```
}
```

3.16.24 Listing of code examples/mpi/p/reductpositive.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

def reduceWithoutZero(in_buf, inout_buf, datatype):
    typecode = MPI._typecode(datatype)
    assert typecode is not None ## check MPI datatype is built-in
    dtype = np.dtype(typecode)

    in_array = np.frombuffer(in_buf, dtype)
    inout_array = np.frombuffer(inout_buf, dtype)

    n = in_array[0]; r = inout_array[0]
    if n==0:
        m = r
    elif r==0:
        m = n
    elif n<r:
        m = n
    else:
        m = r
    inout_array[0] = m

ndata = 10
data = np.zeros(10,dtype=np.intc)
data[:] = [2,3,0,5,0,1,8,12,4,0]

if nprocs>ndata:
    print("Too many procs for this example: at most %d\n" %ndata)
    sys.exit(1)

#
# compute reduction by hand
#
mreduct=2000000000
for i in range(nprocs):
    if data[i]<mreduct and data[i]>0:
        mreduct = data[i]

rwz = MPI.Op.Create(reduceWithoutZero)
positive_minimum = np.zeros(1,dtype=np.intc)
comm.Allreduce(data[procid],positive_minimum,rwz);
```

```
#
# check that the distributed result is the same as sequential
#
if mreduct!=positive_minimum:
    print("[%d] Result %d should be %d\n" % \
          procid,positive_minimum,mreduct)
elif procid==0:
    print("User-defined reduction successful: %d\n" % positive_minimum)
```

3.16.25 Listing of code examples/mpi/c/ibarrierprobe.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {

#include "globalinit.c"

  /*
   * Pick one random process
   * that will do a send
   */
  int sender,receiver;
  if (procno==0)
    sender = rand()%nprocs;
  MPI_Bcast(&sender,1,MPI_INT,0,comm);
  int i_do_send = sender==procno;

  float data=1.;
  MPI_Request send_request;
  if (i_do_send) {
    /*
     * Pick a random process to send to,
     * not yourself.
     */
    int receiver = rand()%nprocs;
    while (receiver==procno) receiver = rand()%nprocs;
    printf("[%d] random send performed to %d\n",procno,receiver);
    //MPI_Isend(&data,1,MPI_FLOAT,receiver,0,comm,&send_request);
    MPI_Ssend(&data,1,MPI_FLOAT,receiver,0,comm);
  }
  /*
   * Everyone posts the non-block barrier
   * and gets a request to test/wait for
   */
  MPI_Request barrier_request;
  MPI_Ibarrier(comm,&barrier_request);

  int step=0;
```

```
  /*
   * Now everyone repeatedly tests the barrier
   * and probes for incoming message.
   * If the barrier completes, there are no
   * incoming message.
   */
  MPI_Barrier(comm);
  double tstart = MPI_Wtime();

  for ( ; ; step++) {
    int barrier_done_flag=0;
    MPI_Test(&barrier_request,&barrier_done_flag,
             MPI_STATUS_IGNORE);
    //stop if you're done!
    if (barrier_done_flag) {
      break;
    } else {
      // if you're not done with the barrier:
      int flag; MPI_Status status;
      MPI_Iprobe
        ( MPI_ANY_SOURCE,MPI_ANY_TAG,
          comm, &flag, &status );
      if (flag) {
        // absorb message!
int sender = status.MPI_SOURCE;
MPI_Recv(&data,1,MPI_FLOAT,sender,0,comm,MPI_STATUS_IGNORE);
printf("[%d] random receive from %d\n",procno,sender);
      }
    }
  }

  MPI_Barrier(comm);
  double duration = MPI_Wtime()-tstart;
  if (procno==0) printf("Probe loop: %e\n",duration);

  printf("[%d] concluded after %d steps\n",procno,step);
  MPI_Wait(&barrier_request,MPI_STATUS_IGNORE);
  /* if (i_do_send) */
  /*   MPI_Wait(&send_request,MPI_STATUS_IGNORE); */

  MPI_Finalize();
  return 0;
}
```

3.16.26 Listing of code examples/mpi/c/findbarrier.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {
```

```
  MPI_Comm comm;
  int nprocs,procid;

  MPI_Init(&argc,&argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_rank(comm,&procid);

  int mysleep;
  srand(procid*time(NULL));
  mysleep = nprocs * (rand()/(double)RAND_MAX);
  printf("[%d] working for %d seconds\n",procid,mysleep);
  sleep(mysleep);

  printf("[%d] finished, now posting barrier\n",procid);
  MPI_Request final_barrier;
  MPI_Ibarrier(comm,&final_barrier);

  int global_finish=mysleep;
  do {
    int all_done_flag=0;
    MPI_Test(&final_barrier,&all_done_flag,MPI_STATUS_IGNORE);
    if (all_done_flag) {
      break;
    } else {
      int flag; MPI_Status status;
      // force progress
      MPI_Iprobe
        ( MPI_ANY_SOURCE,MPI_ANY_TAG,
          comm, &flag, MPI_STATUS_IGNORE );
      printf("[%d] going to work for another second\n",procid);
      sleep(1);
      global_finish++;
    }
  } while (1);

  MPI_Wait(&final_barrier,MPI_STATUS_IGNORE);
  printf("[%d] concluded %d work, total time %d\n",
 procid,mysleep,global_finish);

  MPI_Finalize();
  return 0;
}
```

Chapter 4

MPI topic: Point-to-point

4.1 Blocking point-to-point operations

Suppose you have an array of numbers $x_i : i = 0, \ldots, N$ and you want to compute

$$y_i = (x_{i-1} + x_i + x_{i+1})/3 : i = 1, \ldots, N-1.$$

As before (see figure 2.6), we give each processor a subset of the x_is and y_is. Let's define i_p as the first index of y that is computed by processor p. (What is the last index computed by processor p? How many indices are computed on that processor?)

We often talk about the *owner computes* model of parallel computing: each processor 'owns' certain data items, and it computes their value.

Now let's investigate how processor p goes about computing y_i for the i-values it owns. Let's assume that process p also stores the values x_i for these same indices. Now, for many values i it can evalute the computation

$$y_i = (x_{i-1} + x_i + x_{i+1})/3$$

locally (figure 4.1).

However, there is a problem with computing y in the first index i_p on processor p:

$$y_{i_p} = (x_{i_p-1} + x_{i_p} + x_{i_p+1})/3$$

The point to the left, x_{i_p-1}, is not stored on process p (it is stored on $p-1$), so it is not immediately available for use by process p. (figure 4.2). There is a similar story with the last index that p tries to compute: that involves a value that is only present on $p+1$.

You see that there is a need for processor-to-processor, or technically *point-to-point*, information exchange. MPI realizes this through matched send and receive calls:

- One process does a send to a specific other process;
- the other process does a specific receive from that source.

We will now discuss the send and receive routines in detail.

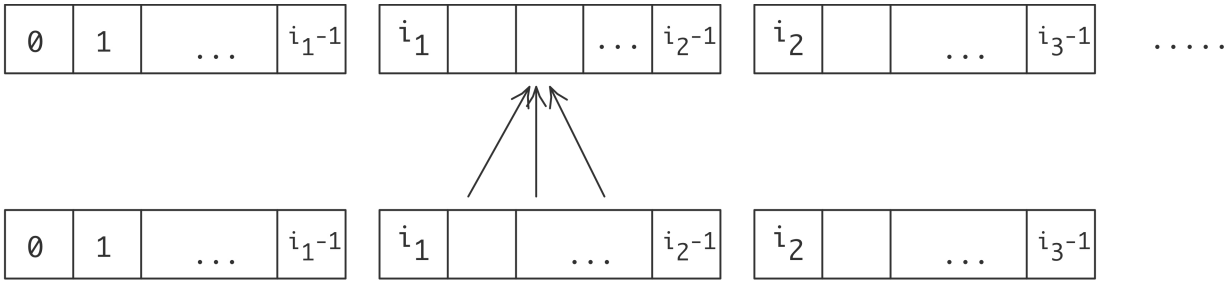


Figure 4.1: Three point averaging in parallel

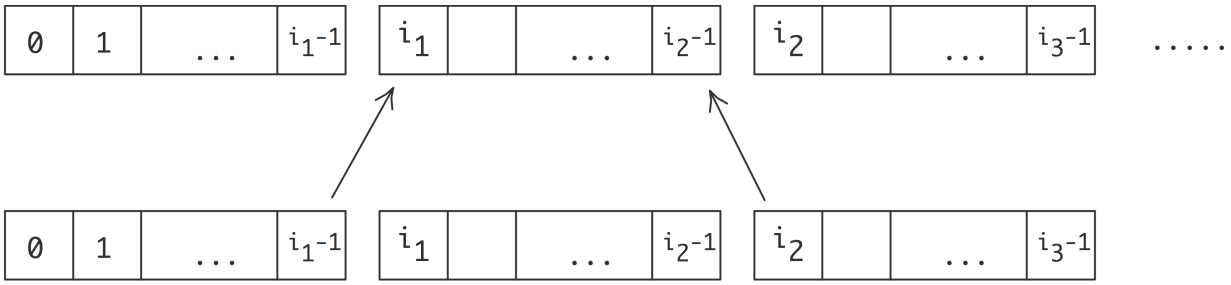


Figure 4.2: Three point averaging in parallel, case of edge points

4.1.1 Send example: ping-pong

A simple scenario for information exchange between just two processes is the *ping-pong*: process A sends data to process B, which sends data back to A. This means that process A executes the code

```
MPI_Send( /* to: */ B ..... );
MPI_Recv( /* from: */ B ... );
```

while process B executes

```
MPI_Recv( /* from: */ A ... );
MPI_Send( /* to: */ A ..... );
```

Since we are programming in SPMD mode, this means our program looks like:

```
if ( /* I am process A */ ) {
  MPI_Send( /* to: */ B ..... );
  MPI_Recv( /* from: */ B ... );
} else if ( /* I am process B */ ) {
  MPI_Recv( /* from: */ A ... );
  MPI_Send( /* to: */ A ..... );
}
```

Remark 6 *The structure of the send and receive calls shows the symmetric nature of MPI: every target process is reached with the same send call, no matter whether it's running on the same multicore chip as the sender, or on a computational node halfway across the machine room, taking several network hops to reach. Of course,*

Figure 4.1 `MPI_Send`

Name	Param name	C type	F type	inout
`mpi_send (`				
	`buf`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	initial address of send buffer			
	`count`	`int`	`INTEGER`	`in`
	number of elements in send buffer			
	`datatype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	datatype of each send buffer element			
	`dest`	`int`	`INTEGER`	`in`
	rank of destination			
	`tag`	`int`	`INTEGER`	`in`
	message tag			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T >
void mpl::communicator::send
   ( const T scalar&,int dest,tag = tag(0) ) const
   ( const T *buffer,const layout< T > &,int dest,tag = tag(0) ) const
   ( iterT begin,iterT end,int dest,tag = tag(0) ) const
T : scalar type
begin : begin iterator
end : end iterator
```

Python:

Name	Param name	param type
`Comm.send(`		
	`self`	
	`obj`	`Any`
	`dest`	`int`
	`tag`	`int = 0`
`) -> None`		

any self-respecting MPI implementation optimizes for the case where sender and receiver have access to the same shared memory. This means that a send/recv pair is realized as a copy operation from the sender buffer to the receiver buffer, rather than a network transfer.

4.1.1.1 Send call

The blocking send command is `MPI_Send` (figure 4.1). Example:

```
// sendandrecv.c
double send_data = 1.;
MPI_Send
  ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
    /* to: */ receiver, /* tag: */ 0,
    /* communicator: */ comm);
```

For the full source of this example, see section 4.5.2

The send call has the following elements.

Buffer The *send buffer* is described by a trio of buffer/count/datatype. See section 3.2.4 for discussion.

Target The *messsage target* is an explicit process rank to send to. This rank is a number from zero up to the result of `MPI_Comm_size`. It is allowed for a process to send to itself, but this may lead to a runtime *deadlock*; see section 4.1.2 for discussion.

Tag Next, a message can have a *tag*. Many applications have each sender send only one message at a time to a given receiver. For the case where there are multiple simultaneous messages between the same sender / receiver pair, the tag can be used to disambiguate between the messages.

Often, a tag value of zero is safe to use. Indeed, OO interfaces to MPI typically have the tag as an optional parameter with value zero. If you do use tag values, you can use the key `MPI_TAG_UB` to query what the maximum value is that can be used; see section 14.1.2.

Communicator Finally, in common with the vast majority of MPI calls, there is a communicator argument that provides a context for the send transaction.

MPL note 23. MPL uses a default value for the tag, and it can deduce the type of the buffer. Sending a scalar becomes:

```
// sendscalar.cxx
if (comm_world.rank()==0) {
  double pi=3.14;
  comm_world.send(pi, 1);  // send to rank 1
  cout << "sent: " << pi << '\n';
} else if (comm_world.rank()==1) {
  double pi=0;
  comm_world.recv(pi, 0);  // receive from rank 0
  cout << "got : " << pi << '\n';
}
```

For the full source of this example, see section 4.5.3

End of MPL note

MPL note 24. MPL can send *static array*s without further layout specification:

```
// sendarray.cxx
double v[2][2][2];
  comm_world.send(v, 1);  // send to rank 1
  comm_world.recv(v, 0);  // receive from rank 0
```

For the full source of this example, see section 4.5.4

Sending vectors uses a general mechanism:

Figure 4.2 `MPI_Recv`

Name	Param name	C type	F type	inout
`mpi_recv (`				
	`buf`	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	initial address of receive buffer			
	`count`	`int`	`INTEGER`	`in`
	number of elements in receive buffer			
	`datatype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	datatype of each receive buffer element			
	`source`	`int`	`INTEGER`	`in`
	rank of source or `MPI_ANY_SOURCE`			
	`tag`	`int`	`INTEGER`	`in`
	message tag or `MPI_ANY_TAG`			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`status`	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T >
status mpl::communicator::recv
   ( T &,int,tag = tag(0) ) const inline
   ( T *,const layout< T > &,int,tag = tag(0) ) const
   ( iterT  begin,iterT  end,int  source, tag  t = tag(0) ) const
```

Python:

Name	Param name	param type
`Comm.recv(`		
	`self`	
	`buf`	`Optional[Buffer] = None`
	`source`	`int = Ellipsis`
	`tag`	`int = Ellipsis`
	`status`	`Optional[Status] = None`
`) -> Any`		

```
// sendbuffer.cxx
std::vector<double> v(8);
mpl::contiguous_layout<double> v_layout(v.size());
  comm_world.send(v.data(), v_layout, 1);  // send to rank 1
  comm_world.recv(v.data(), v_layout, 0);  // receive from rank 0
```

For the full source of this example, see section 4.5.5

End of MPL note

4.1.1.2 Receive call

The basic blocking receive command is `MPI_Recv` (figure 4.2)

An example:

```
double recv_data;
MPI_Recv
```

```
( /* recv buffer/count/type: */ &recv_data,1,MPI_DOUBLE,
  /* from: */ sender, /* tag: */ 0,
  /* communicator: */ comm,
  /* recv status: */ MPI_STATUS_IGNORE);
```

For the full source of this example, see section 4.5.2

This is similar in structure to the send call, with some exceptions.

Buffer The *receive buffer* has the same buffer/count/data parameters as the send call. However, the `count` argument here indicates the size of the buffer, rather than the actual length of a message. This sets an upper bound on the length of the incoming message.

- For receiving messages with unknown length, use `MPI_Probe`; section 4.3.1.
- A message longer than the buffer size will give an overflow error, typically ending your program; see section 14.2.2.

The length of the received message can be determined from the status object; see section 4.3.2 for more detail.

Source Mirroring the target argument of the `MPI_Send` call, `MPI_Recv` has a *message source* argument. This can be either a specific rank, or it can be the `MPI_ANY_SOURCE` wildcard. In the latter case, the actual source can be determined after the message has been received; see section 4.3.2. A source value of `MPI_PROC_NULL` is also allowed, which makes the receive succeed immediately with no data received.

MPL note 25. The constant `mpl::any_source` equals `MPI_ANY_SOURCE` (by `constexpr`).

End of MPL note

Tag Similar to the messsage source, the message tag of a receive call can be a specific value or a wildcard, in this case `MPI_ANY_TAG`. Again, see below.

Communicator The communicator argument almost goes without remarking.

Status The `MPI_Recv` command has one parameter that the send call lacks: the `MPI_Status` object, describing the *message status*. This gives information about the message received, for instance if you used wildcards for source or tag. See section 4.3.2 for more about the status object.

Exercise 4.1. Implement the ping-pong program. Add a timer using `MPI_Wtime`. For the `status` argument of the receive call, use `MPI_STATUS_IGNORE`.

- Run multiple ping-pongs (say a thousand) and put the timer around the loop. The first run may take longer; try to discard it.
- Run your code with the two communicating processes first on the same node, then on different nodes. Do you see a difference?
- Then modify the program to use longer messages. How does the timing increase with message size?

For bonus points, can you do a regression to determine α, β?
(There is a skeleton for this exercise under the name `pingpong`.)

Exercise 4.2. Take your pingpong program and modify it to let half the processors be source and the other half the targets. Does the pingpong time increase?

4.1.2 Problems with blocking communication

You may be tempted to think that the send call puts the data somewhere in the network, and the sending code can progress, as in figure 4.3, left. But this ideal scenario is not realistic: it assumes that somewhere

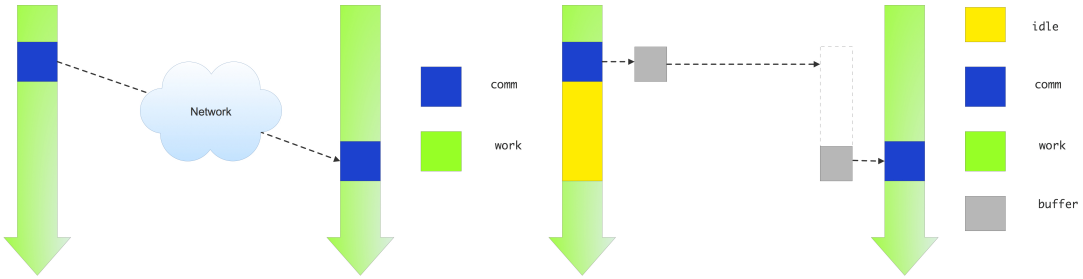


Figure 4.3: Illustration of an ideal (left) and actual (right) send-receive interaction

in the network there is buffer capacity for all messages that are in transit. This is not the case: data resides on the sender, and the sending call blocks, until the receiver has received all of it. (There is a exception for small messages, as explained in the next section.)

The use of `MPI_Send` and `MPI_Recv` is known as *blocking communication*: when your code reaches a send or receive call, it blocks until the call is succesfully completed.

Technically, blocking operations are called *non-local* since their execution depends on factors that are not local to the process. See section 5.4.

For a receive call it is clear that the receiving code will wait until the data has actually come in, but for a send call this is more subtle.

4.1.2.1 Deadlock

Suppose two process need to exchange data, and consider the following pseudo-code, which purports to exchange data between processes 0 and 1:

```
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
receive(source=other);
send(target=other);
```

Imagine that the two processes execute this code. They both issue the send call... and then can't go on, because they are both waiting for the other to issue the send call corresponding to their receive call. This is known as *deadlock*.

4.1.2.2 Eager limit

If you reverse the send and receive call:

```
other = 1-mytid; /* if I am 0, other is 1; and vice versa */
send(target=other);
receive(source=other);
```

you should get deadlock, since the send calls will be waiting for the receive operation to be posted.

In practice, however, this code will often work. The reason is that MPI implementations sometimes send small messages regardless of whether the receive has been posted. This relies on the availability of some amount of available buffer space. The size under which this behaviour is used is sometimes referred to as the *eager limit*.

The following code is guaranteed to block, since a `MPI_Recv` always blocks:

```
// recvblock.c
other = 1-procno;
MPI_Recv(&recvbuf,1,MPI_INT,other,0,comm,&status);
MPI_Send(&sendbuf,1,MPI_INT,other,0,comm);
printf("This statement will not be reached on %d\n",procno);
```

For the full source of this example, see section 4.5.6

On the other hand, if we put the send call before the receive, code may not block for small messages that fall under the eager limit.

To illustrate eager and blocking behavior in `MPI_Send`, consider an example where we send gradually larger messages. From the screen output you can see what the largest message was that fell under the eager limit; after that the code hangs because of a deadlock.

```
// sendblock.c
other = 1-procno;
/* loop over increasingly large messages */
for (int size=1; size<2000000000; size*=10) {
  sendbuf = (int*) malloc(size*sizeof(int));
  recvbuf = (int*) malloc(size*sizeof(int));
  if (!sendbuf || !recvbuf) {
    printf("Out of memory\n"); MPI_Abort(comm,1);
  }
  MPI_Send(sendbuf,size,MPI_INT,other,0,comm);
  MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
  /* If control reaches this point, the send call
     did not block. If the send call blocks,
     we do not reach this point, and the program will hang.
  */
  if (procno==0)
    printf("Send did not block for size %d\n",size);
  free(sendbuf); free(recvbuf);
}
```

For the full source of this example, see section 4.5.7

```
// sendblock.F90
```

```
other = 1-mytid
size = 1
do
   allocate(sendbuf(size)); allocate(recvbuf(size))
   print *,size
   call MPI_Send(sendbuf,size,MPI_INTEGER,other,0,comm,err)
   call MPI_Recv(recvbuf,size,MPI_INTEGER,other,0,comm,status,err)
   if (mytid==0) then
      print *,"MPI_Send did not block for size",size
   end if
   deallocate(sendbuf); deallocate(recvbuf)
   size = size*10
   if (size>2000000000) goto 20
end do
20   continue
```

For the full source of this example, see section 4.5.8

```
## sendblock.py
size = 1
while size<2000000000:
    sendbuf = np.empty(size, dtype=np.int)
    recvbuf = np.empty(size, dtype=np.int)
    comm.Send(sendbuf,dest=other)
    comm.Recv(sendbuf,source=other)
    if procid<other:
        print("Send did not block for",size)
    size *= 10
```

For the full source of this example, see section 4.5.9

If you want a code to exhibit the same blocking behavior for all message sizes, you force the send call to be blocking by using `MPI_Ssend`, which has the same calling sequence as `MPI_Send`, but which does not allow eager sends.

```
// ssendblock.c
other = 1-procno;
sendbuf = (int*) malloc(sizeof(int));
recvbuf = (int*) malloc(sizeof(int));
size = 1;
MPI_Ssend(sendbuf,size,MPI_INT,other,0,comm);
MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
printf("This statement is not reached\n");
```

For the full source of this example, see section 5.6.4

Formally you can describe deadlock as follows. Draw up a graph where every process is a node, and draw a directed arc from process A to B if A is waiting for B. There is deadlock if this directed graph has a loop.

The solution to the deadlock in the above example is to first do the send from 0 to 1, and then from 1 to 0 (or the other way around). So the code would look like:

```
if ( /* I am processor 0 */ ) {
  send(target=other);
  receive(source=other);
```

```
} else {
  receive(source=other);
  send(target=other);
}
```

The eager limit is implementation-specific. For instance, for *Intel MPI* there is a variable `I_MPI_EAGER_THRESHOLD`, for *mvapich2* it is `MV2_IBA_EAGER_THRESHOLD`, and for *OpenMPI* the `--mca` options `btl_openib_eager_limit` and `btl_openib_rndv_eager_limit`.

4.1.2.3 Serialization

There is a second, even more subtle problem with blocking communication. Consider the scenario where every processor needs to pass data to its successor, that is, the processor with the next higher rank. The basic idea would be to first send to your successor, then receive from your predecessor. Since the last processor does not have a successor it skips the send, and likewise the first processor skips the receive. The pseudo-code looks like:

```
successor = mytid+1; predecessor = mytid-1;
if ( /* I am not the last processor */ )
  send(target=successor);
if ( /* I am not the first processor */ )
  receive(source=predecessor)
```

Exercise 4.3. (Classroom exercise) Each student holds a piece of paper in the right hand – keep your left hand behind your back – and we want to execute:

1. Give the paper to your right neighbour;
2. Accept the paper from your left neighbour.

Including boundary conditions for first and last process, that becomes the following program:

1. If you are not the rightmost student, turn to the right and give the paper to your right neighbour.
2. If you are not the leftmost student, turn to your left and accept the paper from your left neighbour.

This code does not deadlock. All processors but the last one block on the send call, but the last processor executes the receive call. Thus, the processor before the last one can do its send, and subsequently continue to its receive, which enables another send, et cetera.

In one way this code does what you intended to do: it will terminate (instead of hanging forever on a deadlock) and exchange data the right way. However, the execution now suffers from unexpected *serialization*: only one processor is active at any time, so what should have been a parallel operation becomes a sequential one. This is illustrated in figure 4.4.

Exercise 4.4. Implement the above algorithm using `MPI_Send` and `MPI_Recv` calls. Run the code, and use TAU to reproduce the trace output of figure 4.4. If you don't have TAU, can you show this serialization behaviour using timings, for instance running it on an increasing number of processes?
(There is a skeleton for this exercise under the name `rightsend`.)

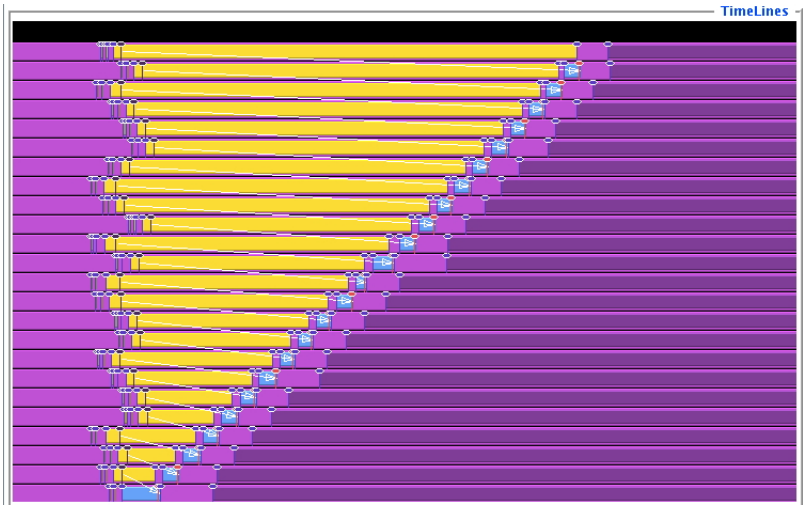


Figure 4.4: Trace of a simple send-recv code

It is possible to orchestrate your processes to get an efficient and deadlock-free execution, but doing so is a bit cumbersome.

Exercise 4.5. The above solution treated every processor equally. Can you come up with a solution that uses blocking sends and receives, but does not suffer from the serialization behaviour?

There are better solutions which we will explore in the next section.

4.1.3 Bucket brigade

The problem with the previous exercise was that an operation that was conceptually parallel, became serial in execution. On the other hand, sometimes the operation is actually serial in nature. One example is the *bucket brigade* operation, where a piece of data is successively passed down a sequence of processors.

Exercise 4.6. Take the code of exercise 4.4 and modify it so that the data from process zero gets propagated to every process. Specifically, compute all partial sums $\sum_{i=0}^{p} i^2$:

$$\begin{cases} x_0 = 1 & \text{on process zero} \\ x_p = x_{p-1} + (p+1)^2 & \text{on process } p \end{cases}$$

Use `MPI_Send` and `MPI_Recv`; make sure to get the order right.

Remark 7 *All quantities involved here are integers. Is it a good idea to use the integer datatype here?*

(There is a skeleton for this exercise under the name `bucketblock`.)

Remark 8 *There is an `MPI_Scan` routine (section 3.4) that performs the same computation, but computationally more efficiently. Thus, this exercise only serves to illustrate the principle.*

4.1.4 Pairwise exchange

Above you saw that with blocking sends the precise ordering of the send and receive calls is crucial. Use the wrong ordering and you get either deadlock, or something that is not efficient at all in parallel. MPI has a way out of this problem that is sufficient for many purposes: the combined send/recv call `MPI_Sendrecv` (figure 4.3).

The sendrecv call works great if every process is paired up. You would then write

```
sendrecv( ....from... ...to... );
```

with the right choice of source and destination. For instance, to send data to your right neighbour:

```
MPI_Comm_rank(comm,&procno);
MPI_Sendrecv( ....
    /* from: */ procno-1
     ... ...
    /* to:   */ procno+1
     ... );
```

This scheme is correct for all processes but the first and last. In order to use the sendrecv call on these processes, we use `MPI_PROC_NULL` for the non-existing processes that the endpoints communicate with.

```
MPI_Comm_rank( .... &mytid );
if ( /* I am not the first processor */ )
  predecessor = mytid-1;
else
  predecessor = MPI_PROC_NULL;
if ( /* I am not the last processor */ )
  successor = mytid+1;
else
  successor = MPI_PROC_NULL;
sendrecv(from=predecessor,to=successor);
```

where the sendrecv call is executed by all processors.

All processors but the last one send to their neighbour; the target value of `MPI_PROC_NULL` for the last processor means a 'send to the null processor': no actual send is done. The null processor value is also of use with the `MPI_Sendrecv` call; section 4.1.4.

Likewise, receive from `MPI_PROC_NULL` succeeds without altering the receive buffer. The corresponding `MPI_Status` object has source `MPI_PROC_NULL`, tag `MPI_ANY_TAG`, and count zero.

MPL note 26. The send-recv call in MPL has the same possibilities for specifying the send and receive buffer as the separate send and recv calls: scalar, layout, iterator. However, out of the nine conceivably possible routine prototypes, only the versions are available where the send and receive buffer are specified the same way. Also, the send and receive tag need to be specified; they do not have default values.

```
// sendrecv.cxx
mpl::tag t0(0);
comm_world.sendrecv
  ( mydata,sendto,t0,
    leftdata,recvfrom,t0 );
```

Figure 4.3 `MPI_Sendrecv`

Name	Param name	C type	F type	inout
`mpi_sendrecv (`				
	`sendbuf` *initial address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`sendcount` *number of elements in send buffer*	`int`	`INTEGER`	`in`
	`sendtype` *type of elements in send buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`dest` *rank of destination*	`int`	`INTEGER`	`in`
	`sendtag` *send tag*	`int`	`INTEGER`	`in`
	`recvbuf` *initial address of receive buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`recvcount` *number of elements in receive buffer*	`int`	`INTEGER`	`in`
	`recvtype` *type of elements receive buffer element*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`source` *rank of source or* `MPI_ANY_SOURCE`	`int`	`INTEGER`	`in`
	`recvtag` *receive tag or* `MPI_ANY_TAG`	`int`	`INTEGER`	`in`
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`status`	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T >
status mpl::communicator::sendrecv
   ( const T & senddata, int dest,   tag sendtag,
           T & recvdata, int source, tag recvtag
   ) const
   ( const T * senddata, const layout< T > & sendl, int dest,   tag sendtag,
           T * recvdata, const layout< T > & recvl, int source, tag recvtag
   ) const
   ( iterT1 begin1, iterT1 end1, int dest,   tag sendtag,
     iterT2 begin2, iterT2 end2, int source, tag recvtag
   ) const
```

Python:

Name	Param name	param type
`Comm.sendrecv(`		
	`self`	
	`sendobj`	`Any`
	`dest`	`int`
	`sendtag`	`int = 0`
	`recvbuf`	`Optional[Buffer] = None`
	`source`	`int = Ellipsis`
	`recvtag`	`int = Ellipsis`
	`status`	`Optional[Status] = None`
`) -> Any`		

End of MPL note

Exercise 4.7. Revisit exercise 4.3 and solve it using `MPI_Sendrecv`.
If you have TAU installed, make a trace. Does it look different from the serialized send/recv code? If you don't have TAU, run your code with different numbers of processes and show that the runtime is essentially constant.
(There is a skeleton for this exercise under the name `rightsend`.)

This call makes it easy to exchange data between two processors: both specify the other as both target and source. However, there need not be any such relation between target and source: it is possible to receive from a predecessor in some ordering, and send to a successor in that ordering; see figure 4.5.

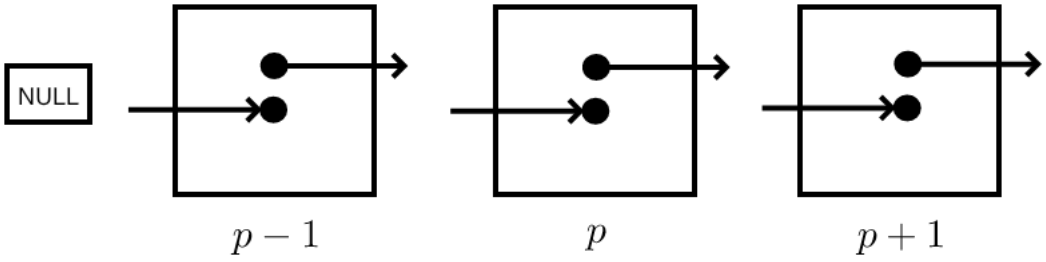


Figure 4.5: An MPI Sendrecv call

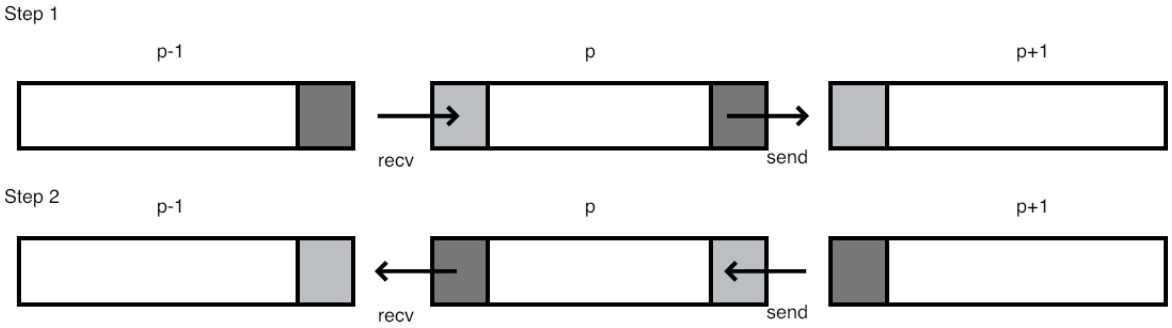


Figure 4.6: Two steps of send/recv to do a three-point combination

.

For the above three-point combination scheme you need to move data both left right, so you need two `MPI_Sendrecv` calls; see figure 4.6.

Exercise 4.8. Implement the above three-point combination scheme using `MPI_Sendrecv`; every processor only has a single number to send to its neighbour.
(There is a skeleton for this exercise under the name `sendrecv`.)

Hints for this exercise:

- Each process does one send and one receive; if a process needs to skip one or the other, you can specify `MPI_PROC_NULL` as the other process in the send or receive specification. In that case the corresponding action is not taken.

Figure 4.4 `MPI_Sendrecv_replace`

Name	Param name	C type	F type	inout
`mpi_sendrecv_replace (`				
	`buf`	`void*`	`TYPE(*), DIMENSION(..)`	`inout`
	initial address of send and receive buffer			
	`count`	`int`	`INTEGER`	`in`
	number of elements in send and receive buffer			
	`datatype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	type of elements in send and receive buffer			
	`dest`	`int`	`INTEGER`	`in`
	rank of destination			
	`sendtag`	`int`	`INTEGER`	`in`
	send message tag			
	`source`	`int`	`INTEGER`	`in`
	rank of source or `MPI_ANY_SOURCE`			
	`recvtag`	`int`	`INTEGER`	`in`
	receive message tag or `MPI_ANY_TAG`			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`status`	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Sendrecv_replace(`		
	`self`	
	`buf`	`BufSpec`
	`dest`	`int`
	`sendtag`	`int = 0`
	`source`	`int = Ellipsis`
	`recvtag`	`int = Ellipsis`
	`status`	`Optional[Status] = None`
`) -> None`		

- As with the simple send/recv calls, processes have to match up: if process p specifies p' as the destination of the send part of the call, p' needs to specify p as the source of the recv part.

If the send and receive buffer have the same size, the routine `MPI_Sendrecv_replace` (figure 4.4) will do an in-place replacement.

The following exercise lets you implement a sorting algorithm with the send-receive call[1].

Exercise 4.9. A very simple sorting algorithm is *swap sort* or *odd-even transposition sort*: pairs of processors compare data, and if necessary exchange. The elementary step is called a *compare-and-swap*: in a pair of processors each sends their data to the other; one keeps the minimum values, and the other the maximum. For simplicity, in this exercise we give each processor just a single number.
The exchange sort algorithm is split in even and odd stages, where in the even stage, processors $2i$ and $2i+1$ compare and swap data, and in the odd stage, processors $2i+1$ and $2i+2$ compare and swap. You need to repeat this $P/2$ times, where P is the number of processors; see figure 4.7.

1. There is an `MPI_Compare_and_swap` call. Do not use that.

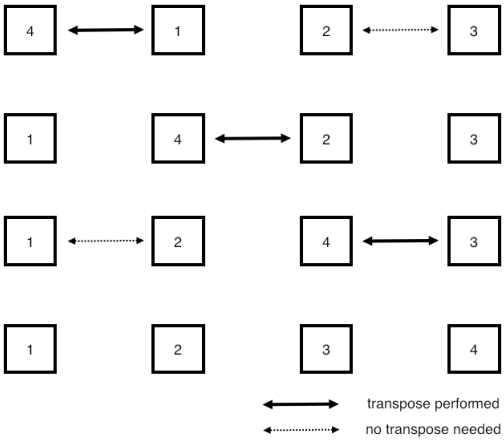


Figure 4.7: Odd-even transposition sort on 4 elements.

Implement this algorithm using `MPI_Sendrecv`. (Use `MPI_PROC_NULL` for the edge cases if needed.) Use a gather call to print the global state of the distributed array at the beginning and end of the sorting process.

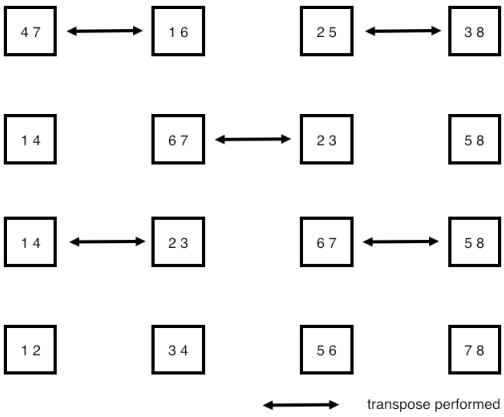


Figure 4.8: Odd-even transposition sort on 4 processes, holding 2 elements each.

Exercise 4.10. Extend this exercise to the case where each process hold an equal number of elements, more than 1. Consider figure 4.8 for inspiration. Is it coincidence that the algorithm takes the same number of steps as in the single scalar case?

The following material is for the (unreleased) MPI-4 standard only

There are *non-blocking* and *persistent* versions of this routine: `MPI_Isendrecv`, `MPI_Sendrecv_init`, `MPI_Isendrecv_replace`, `MPI_Sendrecv_replace_init`.

End of MPI-4 material

4.2 Nonblocking point-to-point operations

The structure of communication is often a reflection of the structure of the operation. With some regular applications we also get a regular communication pattern. Consider again the above operation:

$$y_i = x_{i-1} + x_i + x_{i+1} : i = 1, \ldots, N-1$$

Doing this in parallel induces communication, as pictured in figure **??**.

We note:

- The data is one-dimensional, and we have a linear ordering of the processors.
- The operation involves neighbouring data points, and we communicate with neighbouring processors.

Above you saw how you can use information exchange between pairs of processors

- using `MPI_Send` and `MPI_Recv`, if you are careful; or
- using `MPI_Sendrecv`, as long as there is indeed some sort of pairing of processors.

However, there are circumstances where it is not possible, not efficient, or simply not convenient, to have such a deterministic setup of the send and receive calls. Figure 4.9 illustrates such a case, where processors

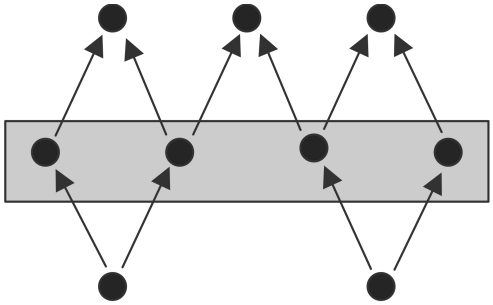


Figure 4.9: Processors with unbalanced send/receive patterns

are organized in a general graph pattern. Here, the numbers of sends and receive of a processor do not need to match.

In such cases, one wants a possibility to state 'these are the expected incoming messages', without having to wait for them in sequence. Likewise, one wants to declare the outgoing messages without having to do them in any particular sequence. Imposing any sequence on the sends and receives is likely to run into the serialization behaviour observed above, or at least be inefficient since processors will be waiting for messages.

4.2.1 Nonblocking send and receive calls

In the previous section you saw that blocking communication makes programming tricky if you want to avoid *deadlock* and performance problems. The main advantage of these routines is that you have full control about where the data is: if the send call returns the data has been successfully received, and the send buffer can be used for other purposes or de-allocated.

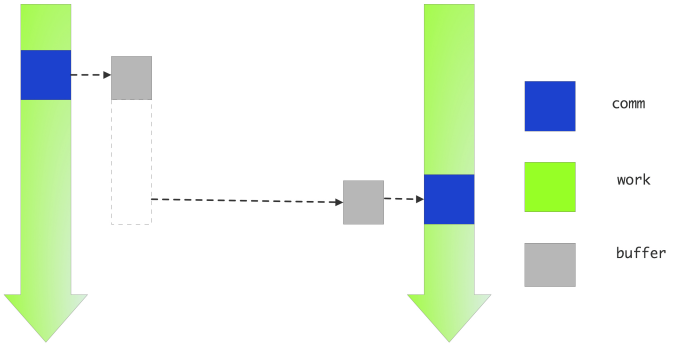


Figure 4.10: Nonblocking send

By contrast, the nonblocking calls `MPI_Isend` (figure 4.5) and `MPI_Irecv` (figure 4.6) (where the 'I' stands for '*immediate*' or '*incomplete*') do not wait for their counterpart: in effect they tell the runtime system 'here is some data and please send it as follows' or 'here is some buffer space, and expect such-and-such data to come'. This is illustrated in figure 4.10.

```
// isendandirecv.c
double send_data = 1.;
MPI_Request request;
MPI_Isend
  ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
       /* to: */ receiver, /* tag: */ 0,
       /* communicator: */ comm,
       /* request: */ &request);
MPI_Wait(&request,MPI_STATUS_IGNORE);
```

For the full source of this example, see section 4.5.11

```
double recv_data;
MPI_Request request;
MPI_Irecv
  ( /* recv buffer/count/type: */ &recv_data,1,MPI_DOUBLE,
       /* from: */ sender, /* tag: */ 0,
       /* communicator: */ comm,
       /* request: */ &request);
MPI_Wait(&request,MPI_STATUS_IGNORE);
```

For the full source of this example, see section 4.5.11

Issuing the `MPI_Isend` / `MPI_Irecv` call is sometimes referred to as *posting* a send/receive.

4.2.2 Request completion: wait calls

From the definition of `MPI_Isend` / `MPI_Irecv`, you seen that nonblocking routine yields an `MPI_Request` object. This request can then be used to query whether the operation has concluded. You may also notice that the `MPI_Irecv` routine does not yield an `MPI_Status` object. This makes sense: the status object describes the actually received data, and at the completion of the `MPI_Irecv` call there is no received data yet.

Waiting for the request is done with a number of routines. We first consider `MPI_Wait` (figure 4.7). It takes

Figure 4.5 `MPI_Isend`

Name	Param name	C type	F type	inout
`mpi_isend (`				
	`buf`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	initial address of send buffer			
	`count`	`int`	`INTEGER`	`in`
	number of elements in send buffer			
	`datatype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	datatype of each send buffer element			
	`dest`	`int`	`INTEGER`	`in`
	rank of destination			
	`tag`	`int`	`INTEGER`	`in`
	message tag			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`request`	`MPI_Request*`	`TYPE(MPI_Request)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T >
irequest mpl::communicator::isend
   ( const T & data, int dest, tag t = tag(0) ) const;
   ( const T * data, const layout< T > &  l, int  dest, tag  t = tag(0) ) const;
   ( iterT  begin, iterT  end, int  dest, tag  t = tag(0) ) const;
```

Python:

Name	Param name	param type
`Comm.isend(`		
	`self`	
	`obj`	`Any`
	`dest`	`int`
	`tag`	`int = 0`
`) -> Request`		

the request as input, and gives an `MPI_Status` as output. If you don't need the status object, you can pass `MPI_STATUS_IGNORE`.

```
// hangwait.c
if (procno==sender) {
  for (int p=0; p<nprocs-1; p++) {
    double send = 1.;
    MPI_Send( &send,1,MPI_DOUBLE,p,0,comm);
  }
} else {
  double recv=0.;
  MPI_Request request;
  MPI_Irecv( &recv,1,MPI_DOUBLE,sender,0,comm,&request);
  MPI_Wait(&request,MPI_STATUS_IGNORE);
}
```

For the full source of this example, see section 4.5.12

Note that the request is passed by reference, so that the wait routine can free it.

Figure 4.6 `MPI_Irecv`

Name	Param name	C type	F type	inout
`mpi_irecv (`				
	`buf` *initial address of receive buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`count` *number of elements in receive buffer*	`int`	`INTEGER`	`in`
	`datatype` *datatype of each receive buffer element*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`source` *rank of source or* `MPI_ANY_SOURCE`	`int`	`INTEGER`	`in`
	`tag` *message tag or* `MPI_ANY_TAG`	`int`	`INTEGER`	`in`
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`request`	`MPI_Request*`	`TYPE(MPI_Request)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T >
irequest mpl::communicator::irecv
   ( const T & data, int src, tag t = tag(0) ) const;
   ( const T * data, const layout< T > &  l, int  src, tag  t = tag(0) ) const;
   ( iterT  begin, iterT  end, int  src, tag  t = tag(0) ) const;
```

Python:

Name	Param name	param type
`Comm.irecv(`		
	`self`	
	`buf`	`Optional[Buffer] = None`
	`source`	`int = Ellipsis`
	`tag`	`int = Ellipsis`
`) -> Request`		

Figure 4.7 `MPI_Wait`

Name	Param name	C type	F type	inout
`mpi_wait (`				
	`request` *request*	`MPI_Request*`	`TYPE(MPI_Request)`	`inout`
	`status`	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Wait(`		
	`self`	
`) -> Literal[True]`		

MPL note 27. Nonblocking routines have an `irequest` as function result. Note: not a parameter passed by reference, as in the C interface. The various wait calls are methods of the `irequest` class.

```
double recv_data;
mpl::irequest recv_request =
  comm_world.irecv( recv_data,sender );
recv_request.wait();
```

For the full source of this example, see section 4.5.13

You can not default-construct the request variable:

```
// DOES NOT COMPILE:
mpl::irequest recv_request;
recv_request = comm.irecv( ... );
```

This means that the normal sequence of first declaring, and then filling in, the request variable is not possible.

MPL implementation note: The wait call always returns a `status` object; not assigning it means that the destructor is called on it.

End of MPL note

4.2.2.1 More wait calls

Here we discuss in some detail the various wait calls. These are blocking; for the nonblocking versions see section 4.2.3.

4.2.2.1.1 Wait for one request

`MPI_Wait` waits for a a single request. If you are indeed waiting for a single nonblocking communication to complete, this is the right routine. If you are waiting for multiple requests you could call this routine in a loop.

```
for (p=0; p<nrequests ; p++) // Not efficient!
  MPI_Wait(request[p],&(status[p]));
```

However, this would be inefficient if the first request is fulfilled much later than the others: your waiting process would have lots of idle time. In that case, use one of the following routines.

4.2.2.1.2 Wait for all requests

`MPI_Waitall` (figure 4.8) allows you to wait for a number of requests, and it does not matter in what sequence they are satisfied. Using this routine is easier to code than the loop above, and it could be more efficient.

MPI has two types of routines for handling requests; we will start with the `MPI_Wait...` routines. These calls are blocking: when you issue such a call, your execution will wait until the specified requests have been completed. Typically you use `MPI_Waitall` to wait for all requests:

```
// irecvloop.c
MPI_Request requests =
  (MPI_Request*) malloc( 2*nprocs*sizeof(MPI_Request) );
```

Figure 4.8 `MPI_Waitall`

Name	Param name	C type	F type	inout
`mpi_waitall (`				
	`count` *lists length*	`int`	`INTEGER`	`in`

	`array_of_requests`
	`length: count`
	array of requests
		`MPI_Request[]`	`TYPE(MPI_Request)`	`inout`

	`array_of_statuses`
	`length: *`
	array of status objects
		`MPI_Status[]`	`TYPE(MPI_Status)`	`out`

`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Request.waitall(`		
	`cls`	
	`requests`	`Sequence[Request]`
	`statuses`	`Optional[List[Status]] = None`
`) -> List[Any]`		

```
recv_buffers = (int*) malloc( nprocs*sizeof(int) );
send_buffers = (int*) malloc( nprocs*sizeof(int) );
for (int p=0; p<nprocs; p++) {
  int
    left_p  = (p-1+nprocs) % nprocs,
    right_p = (p+1) % nprocs;
  send_buffer[p] = nprocs-p;
  MPI_Isend(sendbuffer+p,1,MPI_INT, right_p,0, requests+2*p);
  MPI_Irecv(recvbuffer+p,1,MPI_INT, left_p,0, requests+2*p+1);
}
/* your useful code here */
MPI_Waitall(2*nprocs,requests,MPI_STATUSES_IGNORE);
```

The output argument is an array or `MPI_Status` object. If you don't need the status objects, you can pass `MPI_STATUSES_IGNORE`.

Exercise 4.11. Revisit exercise 4.6 and consider replacing the blocking calls by nonblocking ones. How far apart can you put the `MPI_Isend` / `MPI_Irecv` calls and the corresponding `MPI_Wait`s?
(There is a skeleton for this exercise under the name `bucketpipenonblock`.)

Exercise 4.12. Create two distributed arrays of positive integers. Take the set difference of the two: the first array needs to be transformed to remove from it those numbers that are in the second array.
How could you solve this with an `MPI_Allgather` call? Why is it not a good idea to do so? Solve this exercise instead with a circular bucket brigade algorithm.
(There is a skeleton for this exercise under the name `setdiff`.)

Python note. In python creating the array for the returned requests is somewhat tricky.

Figure 4.9 `MPI_Waitany`

Name	Param name	C type	F type	inout
`mpi_waitany (`				
	`count` *list length*	`int`	`INTEGER`	in

	`array_of_requests`
	`length: count`
	array of requests
		`MPI_Request[]`	`TYPE(MPI_Request)`	inout

	`index` *index of handle for operation that completed*	`int*`	`INTEGER`	out
	`status`	`MPI_Status*`	`TYPE(MPI_Status)`	out
`(opt)`	`ierror`		`INTEGER`	out
`)`				

Python:

Name	Param name	param type
`Request.waitany(`		
	`cls`	
	`requests`	`Sequence[Request]`
	`status`	`Optional[Status] = None`
`) -> Tuple[int, Any]`		

```
## irecvloop.py
requests = [ None ] * (2*nprocs)
sendbuffer = np.empty( nprocs, dtype=np.int )
recvbuffer = np.empty( nprocs, dtype=np.int )

for p in range(nprocs):
    left_p = (p-1) % nprocs
    right_p = (p+1) % nprocs
    requests[2*p] = comm.Isend\
        ( sendbuffer[p:p+1],dest=left_p)
    requests[2*p+1] = comm.Irecv\
        ( sendbuffer[p:p+1],source=right_p)
MPI.Request.Waitall(requests)
```

For the full source of this example, see section 4.5.14

4.2.2.1.3 Wait for any requests The 'waitall' routine is good if you need all nonblocking communications to be finished before you can proceed with the rest of the program. However, sometimes it is possible to take action as each request is satisfied. In that case you could use `MPI_Waitany` (figure 4.9) and write:

```
for (p=0; p<nrequests; p++) {
  MPI_Irecv(buffer+index, /* ... */, requests+index);
}
for (p=0; p<nrequests; p++) {
  MPI_Waitany(nrequests,request_array,&index,&status);
  // operate on buffer[index]
}
```

Note that this routine takes a single status argument, passed by reference, and not an array of statuses!

Fortran note. The `index` parameter is the index in the array of requests, so it uses *1-based indexing*.

```
// irecvsource.F90
if (mytid==ntids-1) then
   do p=1,ntids-1
      print *,"post"
      call MPI_Irecv(recv_buffer(p),1,MPI_INTEGER,p-1,0,comm,&
           requests(p),err)
   end do
   do p=1,ntids-1
      call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE,err)
      write(*,'("Message from",i3,":",i5)') index,recv_buffer(index)
   end do
```

For the full source of this example, see section 4.5.15

```
// waitnull.F90
Type(MPI_Request),dimension(:),allocatable :: requests
allocate(requests(ntids-1))
      call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE)
      if ( .not. requests(index)==MPI_REQUEST_NULL) then
           print *,"This request should be null:",index
```

For the full source of this example, see section 4.5.16

```
// waitnull.F90
Type(MPI_Request),dimension(:),allocatable :: requests
allocate(requests(ntids-1))
      call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE)
      if ( .not. requests(index)==MPI_REQUEST_NULL) then
           print *,"This request should be null:",index
```

For the full source of this example, see section 4.5.16

MPL note 28. Instead of an array of requests, use an `irequest_pool` object, which acts like a vector of requests, meaning that you can `push` onto it.

```
// irecvsource.cxx
mpl::irequest_pool recv_requests;
for (int p=0; p<nprocs-1; p++) {
  recv_requests.push( comm_world.irecv( recv_buffer[p], p ) );
}
```

For the full source of this example, see section 4.5.17

You can not declare a pool of a fixed size and assign elements.

End of MPL note

MPL note 29. The `irequest_pool` class has methods `waitany`, `waitall`, `testany`, `testall`, `waitsome`, `testsome`.

The 'any' methods return a `std::pair<bool,size_t>`, with `false` meaning `index==MPI_UNDEFINED+` meaning no more requests to be satisfied.

```
  auto [success,index] = recv_requests.waitany();
  if (success) {
        auto recv_status = recv_requests.get_status(index);
```

For the full source of this example, see section 4.5.17

Same for `testany`, then false means no requests test true.

End of MPL note

4.2.2.1.4 Polling with MPI Wait any The `MPI_Waitany` routine can be used to implement *polling*: occasionally check for incoming messages while other work is going on.

```
// irecvsource.c
if (procno==nprocs-1) {
  int *recv_buffer;
  MPI_Request *request; MPI_Status status;
  recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));
  request = (MPI_Request*) malloc((nprocs-1)*sizeof(MPI_Request));

  for (int p=0; p<nprocs-1; p++) {
    ierr = MPI_Irecv(recv_buffer+p,1,MPI_INT, p,0,comm,
                     request+p); CHK(ierr);
  }
  for (int p=0; p<nprocs-1; p++) {
    int index,sender;
    MPI_Waitany(nprocs-1,request,&index,&status); //MPI_STATUS_IGNORE);
    if (index!=status.MPI_SOURCE)
        printf("Mismatch index %d vs source %d\n",index,status.MPI_SOURCE);
    printf("Message from %d: %d\n",index,recv_buffer[index]);
  }
} else {
  ierr = MPI_Send(&procno,1,MPI_INT, nprocs-1,0,comm); CHK(ierr);
}
```

For the full source of this example, see section 4.5.18

```
## irecvsource.py
if procid==nprocs-1:
    receive_buffer = np.empty(nprocs-1,dtype=np.int)
    requests = [ None ] * (nprocs-1)
    for sender in range(nprocs-1):
        requests[sender] = comm.Irecv(receive_buffer[sender:sender+1],source=sender)
    # alternatively: requests = [ comm.Irecv(s) for s in .... ]
    status = MPI.Status()
    for sender in range(nprocs-1):
        ind = MPI.Request.Waitany(requests,status=status)
        if ind!=status.Get_source():
            print("sender mismatch: %d vs %d" % (ind,status.Get_source()))
        print("received from",ind)
else:
    mywait = random.randint(1,2*nprocs)
    print("[%d] wait for %d seconds" % (procid,mywait))
    time.sleep(mywait)
    mydata = np.empty(1,dtype=np.int)
    mydata[0] = procid
    comm.Send([mydata,MPI.INT],dest=nprocs-1)
```

For the full source of this example, see section 4.5.19

Each process except for the root does a blocking send; the root posts `MPI_Irecv` from all other processors, then loops with `MPI_Waitany` until all requests have come in. Use `MPI_SOURCE` to test the index parameter of the wait call.

Note the `MPI_STATUS_IGNORE` parameter: we know everything about the incoming message, so we do not need to query a status object. Contrast this with the example in section 46.

4.2.2.1.5 Wait for some requests Finally, `MPI_Waitsome` is very much like `MPI_Waitany`, except that it returns multiple numbers, if multiple requests are satisfied. Now the status argument is an array of `MPI_Status` objects.

Figure 4.11 shows the trace of a nonblocking execution using `MPI_Waitall`.

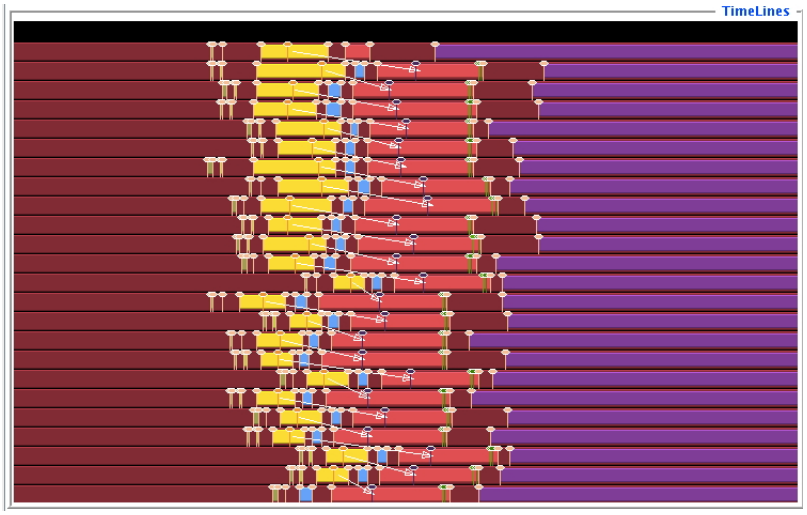


Figure 4.11: A trace of a nonblocking send between neighbouring processors

4.2.2.2 Receive status of the wait calls

The `MPI_Wait...` routines have the `MPI_Status` objects as output. If you are not interested in the status information, you can use the values `MPI_STATUS_IGNORE` for `MPI_Wait` and `MPI_Waitany`, or `MPI_STATUSES_IGNORE` for `MPI_Waitall` and `MPI_Waitsome`.

Exercise 4.13.

(There is a skeleton for this exercise under the name `isendirecv`.) Now use nonblocking send/receive routines to implement the three-point averaging operation

$$y_i = \bigl(x_{i-1} + x_i + x_{i+1}\bigr)/3 : \; i = 1, \ldots, N-1$$

on a distributed array. (Hint: use `MPI_PROC_NULL` at the ends.)

There is a second motivation for the `Isend/Irecv` calls: if your hardware supports it, the communication can happen while your program can continue to do useful work:

```
// start nonblocking communication
MPI_Isend( ... ); MPI_Irecv( ... );
// do work that does not depend on incoming data
....
// wait for the Isend/Irecv calls to finish
MPI_Wait( ... );
// now do the work that absolutely needs the incoming data
....
```

This is known as *overlapping computation and communication*, or *latency hiding*. See also *asynchronous progress*; section 5.4.1.

Unfortunately, a lot of this communication involves activity in user space, so the solution would have been to let it be handled by a separate thread. Until recently, processors were not efficient at doing such multi-threading, so true overlap stayed a promise for the future. Some network cards have support for this overlap, but it requires a nontrivial combination of hardware, firmware, and MPI implementation.

Exercise 4.14.

(There is a skeleton for this exercise under the name `isendirecvarray`.) Take your code of exercise 4.13 and modify it to use latency hiding. Operations that can be performed without needing data from neighbours should be performed in between the `MPI_Isend` / `MPI_Irecv` calls and the corresponding `MPI_Wait` calls.

Remark 9 *You have now seen various send types: blocking, nonblocking, synchronous. Can a receiver see what kind of message was sent? Are different receive routines needed? The answer is that, on the receiving end, there is nothing to distinguish a nonblocking or synchronous message. The `MPI_Recv` call can match any of the send routines you have seen so far (but not `MPI_Sendrecv`), and conversely a message sent with `MPI_Send` can be received by `MPI_Irecv`.*

4.2.2.3 *Buffer issues in nonblocking communication*

While the use of nonblocking routines prevents deadlock, it introduces problems of its own.

- With a blocking send call, you could repeatedly fill the send buffer and send it off.

```
double *buffer;
for ( ... p ... ) {
   buffer = // fill in the data
   MPI_Send( buffer, ... /* to: */ p );
```

- On the other hand, when a nonblocking send call returns, the actual send may not have been executed, so the send buffer may not be safe to overwrite. Similarly, when the recv call returns, you do not know for sure that the expected data is in it. Only after the corresponding wait call are you use that the buffer has been sent, or has received its contents.
- To send multiple messages with nonblocking calls you therefore have to allocate multiple buffers.

```
double **buffers;
for ( ... p ... ) {
```

```
    buffers[p] = // fill in the data
    MPI_Send( buffers[p], ... /* to: */ p );
}
MPI_Wait( /* the requests */ );
```

```
// irecvloop.c
MPI_Request requests =
  (MPI_Request*) malloc( 2*nprocs*sizeof(MPI_Request) );
recv_buffers = (int*) malloc( nprocs*sizeof(int) );
send_buffers = (int*) malloc( nprocs*sizeof(int) );
for (int p=0; p<nprocs; p++) {
  int
    left_p  = (p-1+nprocs) % nprocs,
    right_p = (p+1) % nprocs;
  send_buffer[p] = nprocs-p;
  MPI_Isend(sendbuffer+p,1,MPI_INT, right_p,0, requests+2*p);
  MPI_Irecv(recvbuffer+p,1,MPI_INT, left_p,0, requests+2*p+1);
}
/* your useful code here */
MPI_Waitall(2*nprocs,requests,MPI_STATUSES_IGNORE);
```

For the full source of this example, see section 4.5.20

The last example we explicitly noted the possibility of overlapping computation and communication.

4.2.3 Wait and test calls

The `MPI_Wait...` routines are blocking. Thus, they are a good solution if the receiving process can not do anything until the data (or at least *some* data) is actually received. The `MPI_Test...` calls are themselves nonblocking: they test for whether one or more requests have been fullfilled, but otherwise immediately return. It is also a *local operation*: it does not force progress.

The `MPI_Test` call can be used in the *manager-worker* model: the manager process creates tasks, and sends them to whichever worker process has finished its work. (This uses a receive from `MPI_ANY_SOURCE`, and a subsequent test on the `MPI_SOURCE` field of the receive status.) While waiting for the workers, the manager can do useful work too, which requires a periodic check on incoming message.

Pseudo-code:

```
while ( not done ) {
  // create new inputs for a while
  ....
  // see if anyone has finished
  MPI_Test( .... &index, &flag );
  if ( flag ) {
    // receive processed data and send new
}
```

If the test is true, the request is deallocated and set to `MPI_REQUEST_NULL`, or, in the case of an active persistent request, set to inactive.

Analogous to `MPI_Wait`, `MPI_Waitany`, `MPI_Waitall`, `MPI_Waitsome`, there are `MPI_Test` (figure 4.10), `MPI_Testany`, `MPI_Testall`, *MPI_Testsome*.

Figure 4.10 `MPI_Test`

Name	Param name	C type	F type	inout
`mpi_test (`				
	`request`	`MPI_Request*`	`TYPE(MPI_Request)`	`inout`
	`flag` *true if operation completed*	`int*`	`LOGICAL`	`out`
	`status`	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Test(`		
	`self`	
`) -> bool`		

Figure 4.11 `MPI_Request_free`

Name	Param name	C type	F type	inout
`mpi_request_free (`				
	`request`	`MPI_Request*`	`TYPE(MPI_Request)`	`inout`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Request.Free(`		
	`self`	
`) -> None`		

Exercise 4.15. Read section HPSC-6.5 and give pseudo-code for the distributed sparse matrix-vector product using the above idiom for using `MPI_Test...` calls. Discuss the advantages and disadvantages of this approach. The answer is not going to be black and white: discuss when you expect which approach to be preferable.

4.2.4 More about requests

Every nonblocking call allocates an `MPI_Request` object. Unlike `MPI_Status`, an `MPI_Request` variable is not actually an object, but instead it is an (opaque) pointer. This meeans that when you call, for instance, `MPI_Irecv`, MPI will allocate an actual request object, and return its address in the `MPI_Request` variable.

Correspondingly, calls to `MPI_Wait...` or `MPI_Test` free this object, setting the handle to `MPI_REQUEST_NULL`. Thus, it is wise to issue wait calls even if you know that the operation has succeeded. For instance, if all receive calls are concluded, you know that the corresponding send calls are finished and there is no strict need to wait for their requests. However, omitting the wait calls would lead to a *memory leak*.

Another way around this is to call `MPI_Request_free` (figure 4.11), which sets the request variable to `MPI_REQUEST_NULL`, and marks the object for deallocation after completion of the operation. Conceivably, one could issue a nonblocking call, and immediately call `MPI_Request_free`, dispensing with any wait call.

Figure 4.12 `MPI_Request_get_status`

Name	Param name	C type	F type	inout
`mpi_request_get_status (`				
	`request` *request*	`MPI_Request`	`TYPE(MPI_Request)`	`in`
	`flag` *boolean flag, same as from* `MPI_TEST`	`int*`	`LOGICAL`	`out`
	`status` *status object if flag is true*	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Figure 4.13 `MPI_Probe`

Name	Param name	C type	F type	inout
`mpi_probe (`				
	`source` *rank of source or* `MPI_ANY_SOURCE`	`int`	`INTEGER`	`in`
	`tag` *message tag or* `MPI_ANY_TAG`	`int`	`INTEGER`	`in`
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`status`	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.probe(`		
	`self`	
	`source`	`int = Ellipsis`
	`tag`	`int = Ellipsis`
	`status`	`Optional[Status] = None`
`) -> Literal[True]`		

However, this makes it hard to know when the operation is concluded and when the buffer is safe to reuse [24].

You can inspect the status of a request without freeing the request object with `MPI_Request_get_status` (figure 4.12).

4.3 More about point-to-point communication

4.3.1 Message probing

MPI receive calls specify a receive buffer, and its size has to be enough for any data sent. In case you really have no idea how much data is being sent, and you don't want to overallocate the receive buffer, you can use a 'probe' call.

The routine `MPI_Probe` (figure 4.13) (and `MPI_Iprobe`, for which see section 5.4.1), accepts a message, but does not copy the data. Instead, when probing tells you that there is a message, you can use `MPI_Get_count`

Figure 4.14 `MPI_Mprobe`

Name	Param name	C type	F type	inout
`mpi_mprobe (`				
	`source`	`int`	`INTEGER`	`in`
	rank of source or `MPI_ANY_SOURCE`			
	`tag`	`int`	`INTEGER`	`in`
	message tag or `MPI_ANY_TAG`			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`message`	`MPI_Message*`	`TYPE(MPI_Message)`	`out`
	returned message			
	`status`	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.mprobe(`		
	`self`	
	`source`	`int = Ellipsis`
	`tag`	`int = Ellipsis`
	`status`	`Optional[Status] = None`
`) -> Message`		

to determine its size, allocate a large enough receive buffer, and do a regular receive to have the data copied.

```
// probe.c
if (procno==receiver) {
  MPI_Status status;
  MPI_Probe(sender,0,comm,&status);
  int count;
  MPI_Get_count(&status,MPI_FLOAT,&count);
  float recv_buffer[count];
  MPI_Recv(recv_buffer,count,MPI_FLOAT, sender,0,comm,MPI_STATUS_IGNORE);
} else if (procno==sender) {
  float buffer[buffer_size];
  ierr = MPI_Send(buffer,buffer_size,MPI_FLOAT, receiver,0,comm); CHK(ierr);
}
```

For the full source of this example, see section 4.5.21

There is a problem with the `MPI_Probe` call in a multithreaded environment: the following scenario can happen.

1. A thread determines by probing that a certain message has come in.
2. It issues a blocking receive call for that message...
3. But in between the probe and the receive call another thread has already received the message.
4. ... Leaving the first thread in a blocked state with not message to receive.

This is solved by `MPI_Mprobe` (figure 4.14), which after a successful probe removes the message from the *matching queue*: the list of messages that can be matched by a receive call. The thread that matched the probe now issues an `MPI_Mrecv` (figure 4.15) call on that message through an object of type `MPI_Message`.

Figure 4.15 `MPI_Mrecv`

Name	Param name	C type	F type	inout
`mpi_mrecv (`				
	`buf`	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	initial address of receive buffer			
	`count`	`int`	`INTEGER`	`in`
	number of elements in receive buffer			
	`datatype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	datatype of each receive buffer element			
	`message`	`MPI_Message*`	`TYPE(MPI_Message)`	`inout`
	message			
	`status`	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

4.3.2 The Status object and wildcards

In section 4.1.1 you saw that `MPI_Recv` has a 'status' argument of type `MPI_Status` that `MPI_Send` lacks. (The various `MPI_Wait...` routines also have a status argument; see section 4.2.1.) Often you specify `MPI_STATUS_IGNORE` for this argument: commonly you know what data is coming in and where it is coming from.

However, in some circumstances the recipient may not know all details of a message when you make the receive call, so MPI has a way of querying the *status* of the message:

- If you are expecting multiple incoming messages, it may be most efficient to deal with them in the order in which they arrive. So, instead of waiting for specific message, you would specify `MPI_ANY_SOURCE` or `MPI_ANY_TAG` in the description of the receive message. Now you have to be able to ask 'who did this message come from, and what is in it'.
- Maybe you know the sender of a message, but the amount of data is unknown. In that case you can overallocate your receive buffer, and after the message is received ask how big it was, or you can 'probe' an incoming message and allocate enough data when you find out how much data is being sent.

To do this, the receive call has a `MPI_Status` parameter.

This status is a property of the actually received messsage, so `MPI_Irecv` does not have a status parameter, but `MPI_Wait` does.

MPL note 30. The `mpl::status` object is created by the receive (or wait) call:

```
mpl::contiguous_layout<double> target_layout(count),
mpl::status recv_status =
  comm_world.recv(target.data(),target_layout, the_other);
recv_count = recv_status.get_count<double>();
```

For the full source of this example, see section 6.8.7

End of MPL note

The `MPI_Status` object is a structure with freely accessible members, as discussed below.

4.3.2.1 Source

In some applications it makes sense that a message can come from one of a number of processes. In this case, it is possible to specify `MPI_ANY_SOURCE` as the source. To find out the *source* where the message actually came from, you would use the `MPI_SOURCE` field of the status object that is delivered by `MPI_Recv` or the `MPI_Wait...` call after an `MPI_Irecv`.

```
MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
         &status);
sender = status.MPI_SOURCE;
```

The source of a message can be obttained as the `MPI_SOURCE` member of the status structure.

There are various scenarios where receiving from 'any source' makes sense. One is that of the *manager-worker* model. The manager task would first send data to the worker tasks, then issues a blocking wait for the data of whichever process finishes first.

This code snippet is a simple model for this: all workers processes wait a random amount of time. For efficiency, the manager process accepts message from any source.

```
// anysource.c
if (procno==nprocs-1) {
  /*
   * The last process receives from every other process
   */
  int *recv_buffer;
  MPI_Status status;

  recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

  /*
   * Messages can come in in any order, so use MPI_ANY_SOURCE
   */
  for (int p=0; p<nprocs-1; p++) {
    err = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
                   &status); CHK(err);
    int sender = status.MPI_SOURCE;
    printf("Message from sender=%d: %d\n",
           sender,recv_buffer[p]);
  }
} else {
  /*
   * Each rank waits an unpredictable amount of time,
   * then sends to the last process in line.
   */
  float randomfraction = (rand() / (double)RAND_MAX);
  int randomwait = (int) ( nprocs * randomfraction );
  printf("process %d waits for %e/%d=%d\n",
         procno,randomfraction,nprocs,randomwait);
  sleep(randomwait);
  err = MPI_Send(&randomwait,1,MPI_INT, nprocs-1,0,comm); CHK(err);
}
```

For the full source of this example, see section 4.5.23

In Fortran2008 style, the source is a member of the `Status` type.

```
// anysource.F90
allocate(recv_buffer(ntids-1))
do p=0,ntids-2
   call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
        MPI_ANY_SOURCE,0,comm,status)
   sender = status%MPI_SOURCE
```

For the full source of this example, see section 4.5.24

In Fortran90 style, the source is an index in the `Status` array.

```
// anysource.F90
allocate(recv_buffer(ntids-1))
do p=0,ntids-2
   call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
        MPI_ANY_SOURCE,0,comm,status,err)
   sender = status(MPI_SOURCE)
```

For the full source of this example, see section 4.5.25

MPL note 31. The `status` object can be queried:

```
int source = recv_status.source();
```

End of MPL note

4.3.2.2 Tag

If a processor is expecting more than one messsage from a single other processor, message tags are used to distinguish between them. In that case, a value of `MPI_ANY_TAG` can be used, and the actual *tag* of a message can be retrieved as the `MPI_TAG` member in the status structure. See the section about `MPI_SOURCE` for how to use this.

MPL note 32. MPL differs from other APIs in its treatment of tags: a tag is not directly an integer, but an object of class `tag`.

```
// sendrecv.cxx
mpl::tag t0(0);
comm_world.sendrecv
  ( mydata,sendto,t0,
    leftdata,recvfrom,t0 );
```

The `tag` class has a couple of methods such as `mpl::tag::any()` (for the `MPI_ANY_TAG` wildcard in receive calls) and `mpl::tag::up()` (maximal tag, found from the `MPI_TAG_UB` attribute).

End of MPL note

4.3.2.3 Error

Any *errors* during the receive operation can be found as the `MPI_ERROR` member of the status structure. This field is only set by functions that return multiple statuses, such as `MPI_Waitall`. For functions that return a single status, any error is returned as the function result. For a function returning multiple statuses, the presence of any error is indicated by a result of `MPI_ERR_IN_STATUS`.

Figure 4.16 `MPI_Get_count`

Name	Param name	C type	F type	inout
`mpi_get_count (`				
	`status`	`const MPI_Status*`	`TYPE(MPI_Status)`	`in`
	return status of receive operation			
	`datatype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	datatype of each receive buffer entry			
	`count`	`int*`	`INTEGER`	`out`
	number of received entries			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
template<typename T>
int mpl::status::get_count () const

template<typename T>
int mpl::status::get_count (const layout<T> &l) const
```

Python:

Name	Param name	param type
`Status.Get_count(`		
	`self`	
	`datatype`	`Datatype = Ellipsis`
`) -> int`		

4.3.2.4 Count

If the amount of data received is not known a priori, the *count* of elements received can be found by `MPI_Get_count` (figure 4.16):

This may be necessary since the `count` argument to `MPI_Recv` is the buffer size, not an indication of the actually received number of data items.

Remarks.

- Unlike the above, this is not directly a member of the status structure.
- The 'count' returned is the number of elements of the specified datatype. If this is a derived type (section 6.3) this is not the same as the number of elementary datatype elements. For that, use `MPI_Get_elements` (figure 4.17) or `MPI_Get_elements_x` which returns the number of basic elements.

MPL note 33. The `get_count` function is a method of the status object. The argument type is handled through templating:

```
// recvstatus.cxx
double pi=0;
auto s = comm_world.recv(pi, 0);  // receive from rank 0
int c = s.get_count<double>();
std::cout << "got : " << c << " scalar(s): " << pi << '\n';
```

For the full source of this example, see section 4.5.26

End of MPL note

Figure 4.17 `MPI_Get_elements`

Name	Param name	C type	F type	inout
`mpi_get_elements (`				
	`status`	`const MPI_Status*`	`TYPE(MPI_Status)`	`in`
	return status of receive operation			
	`datatype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	datatype used by receive operation			
	`count`	`int*`	`INTEGER`	`out`
	number of received basic elements			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Status.Get_elements(`		
	`self`	
	`datatype`	`Datatype`
`) -> int`		

4.3.2.5 *Example: receiving from any source*

Consider an example where the last process receives from every other process. We could implement this as a loop

```
for (int p=0; p<nprocs-1; p++)
  MPI_Recv( /* from source= */ p );
```

but this may incur idle time if the messages arrive out of order.

Instead, we use the `MPI_ANY_SOURCE` specifier to give a wildcard behavior to the receive call: using this value for the 'source' value means that we accept mesages from any source within the communicator, and messages are only matched by tag value. (Note that size and type of the receive buffer is not used for message matching!)

We then retrieve the actual source from the `MPI_Status` object through the `MPI_SOURCE` field.

```
// anysource.c
if (procno==nprocs-1) {
  /*
   * The last process receives from every other process
   */
  int *recv_buffer;
  MPI_Status status;

  recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

  /*
   * Messages can come in in any order, so use MPI_ANY_SOURCE
   */
  for (int p=0; p<nprocs-1; p++) {
    err = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
                   &status); CHK(err);
    int sender = status.MPI_SOURCE;
```

```
    printf("Message from sender=%d: %d\n",
           sender,recv_buffer[p]);
  }
} else {
  /*
   * Each rank waits an unpredictable amount of time,
   * then sends to the last process in line.
   */
  float randomfraction = (rand() / (double)RAND_MAX);
  int randomwait = (int) ( nprocs * randomfraction );
  printf("process %d waits for %e/%d=%d\n",
         procno,randomfraction,nprocs,randomwait);
  sleep(randomwait);
  err = MPI_Send(&randomwait,1,MPI_INT, nprocs-1,0,comm); CHK(err);
}
```

For the full source of this example, see section 4.5.23

```
## anysource.py
rstatus = MPI.Status()
comm.Recv(rbuf,source=MPI.ANY_SOURCE,status=rstatus)
print("Message came from %d" % rstatus.Get_source())
```

For the full source of this example, see section 4.5.27

In sections 46 and 4.2.3 we explained the *manager-worker* model, and how it offers an opportunity for inspecting the `MPI_SOURCE` field of the `MPI_Status` object describing the data that was received.

4.4 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

1. Describe a deadlock scenario involving three processors.
2. True or false: a message sent with `MPI_Isend` from one processor can be received with an `MPI_Recv` call on another processor.
3. True or false: a message sent with `MPI_Send` from one processor can be received with an `MPI_Irecv` on another processor.
4. Why does the `MPI_Irecv` call not have an `MPI_Status` argument?
5. Suppose you are testing ping-pong timings. Why is it generally not a good idea to use processes 0 and 1 for the source and target processor? Can you come up with a better guess?
6. What is the relation between the concepts of 'origin', 'target', 'fence', and 'window' in one-sided communication.
7. What are the three routines for one-sided data transfer?
8. In the following fragments assume that all buffers have been allocated with sufficient size. For each fragment note whether it deadlocks or not. Discuss performance issues.

```
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
for (int p=0; p<nprocs; p++)
```

```
  if (p!=procid)
  MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
```

```
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
for (int p=0; p<nprocs; p++)
  if (p!=procid)
  MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
```

```
int ireq = 0;
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Isend(sbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE);
MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);
```

```
int ireq = 0;
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Irecv(rbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);
```

```
int ireq = 0;
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Irecv(rbuffers[p],buflen,MPI_INT,p,0,comm,&(requests[ireq++]));
MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE);
for (int p=0; p<nprocs; p++)
  if (p!=procid)
    MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm);
```

Fortran codes:

```
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
   end if
end do
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)
   end if
end do
```

```
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)
   end if
end do
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
   end if
end do
```

```
ireq = 0
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Isend(sbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
           requests(ireq+1),ierr)
      ireq = ireq+1
   end if
end do
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Recv(rbuffer,buflen,MPI_INT,p,0,comm,MPI_STATUS_IGNORE,ierr)
   end if
end do
call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)
```

```
ireq = 0
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Irecv(rbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
           requests(ireq+1),ierr)
      ireq = ireq+1
   end if
end do
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
   end if
end do
call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)
```

```
// block5.F90
ireq = 0
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Irecv(rbuffers(1,p+1),buflen,MPI_INT,p,0,comm,&
           requests(ireq+1),ierr)
      ireq = ireq+1
   end if
end do
call MPI_Waitall(nprocs-1,requests,MPI_STATUSES_IGNORE,ierr)
do p=0,nprocs-1
   if (p/=procid) then
      call MPI_Send(sbuffer,buflen,MPI_INT,p,0,comm,ierr)
   end if
end do
```

9. Consider a ring-wise communication where

```
int
  next = (mytid+1) % ntids,
  prev = (mytid+ntids-1) % ntids;
```

and each process sends to *next*, and receives from *prev*.
The normal solution for preventing deadlock is to use both `MPI_Isend` and `MPI_Irecv`. The send and receive complete at the wait call. But does it matter in what sequence you do the wait calls?

```
// ring3.c
MPI_Request req1,req2;
MPI_Irecv(&y,1,MPI_DOUBLE,prev,0,
    comm,&req1);
MPI_Isend(&x,1,MPI_DOUBLE,next,0,
    comm,&req2);
MPI_Wait(&req1,MPI_STATUS_IGNORE);
MPI_Wait(&req2,MPI_STATUS_IGNORE);
```

```
// ring4.c
MPI_Request req1,req2;
MPI_Irecv(&y,1,MPI_DOUBLE,prev,0,
    comm,&req1);
MPI_Isend(&x,1,MPI_DOUBLE,next,0,
    comm,&req2);
MPI_Wait(&req2,MPI_STATUS_IGNORE);
MPI_Wait(&req1,MPI_STATUS_IGNORE);
```

Can we have one nonblocking and one blocking call? Do these scenarios block?

```
// ring1.c
MPI_Request req;
MPI_Issend(&x,1,MPI_DOUBLE,next,0,
    comm,&req);
MPI_Recv(&y,1,MPI_DOUBLE,prev,0,
    comm,
         MPI_STATUS_IGNORE);
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

```
// ring2.c
MPI_Request req;
MPI_Irecv(&y,1,MPI_DOUBLE,prev,0,
    comm,&req);
MPI_Ssend(&x,1,MPI_DOUBLE,next,0,
    comm);
MPI_Wait(&req,MPI_STATUS_IGNORE);
```

4.5 Sources used in this chapter

4.5.1 Listing of code header

4.5.2 Listing of code examples/mpi/c/sendandrecv.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

  MPI_Comm comm = MPI_COMM_WORLD;
  int nprocs, procno;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_rank(comm,&procno);

  /*
   * We set up a single communication between
   * the first and last process
   */
  int sender,receiver;
  sender = 0; receiver = nprocs-1;

  if (procno==sender) {
    double send_data = 1.;
    MPI_Send
      ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
        /* to: */ receiver, /* tag: */ 0,
        /* communicator: */ comm);
    printf("[%d] Send successfully concluded\n",procno);
  } else if (procno==receiver) {
    double recv_data;
    MPI_Recv
      ( /* recv buffer/count/type: */ &recv_data,1,MPI_DOUBLE,
        /* from: */ sender, /* tag: */ 0,
        /* communicator: */ comm,
        /* recv status: */ MPI_STATUS_IGNORE);
    printf("[%d] Receive successfully concluded\n",procno);
  }

  MPI_Finalize();
  return 0;
}
```

4.5.3 Listing of code examples/mpi/mpl/sendscalar.cxx

```
#include <cstdlib>
```

```
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  if (comm_world.size()<2)
    return EXIT_FAILURE;

  // send and receive a single floating point number
  if (comm_world.rank()==0) {
    double pi=3.14;
    comm_world.send(pi, 1);  // send to rank 1
    cout << "sent: " << pi << '\n';
  } else if (comm_world.rank()==1) {
    double pi=0;
    comm_world.recv(pi, 0);  // receive from rank 0
    cout << "got : " << pi << '\n';
  }
  return EXIT_SUCCESS;
}
```

4.5.4 Listing of code examples/mpi/mpl/sendarray.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <sstream>
using std::stringstream;

#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  if (comm_world.size()<2)
    return EXIT_FAILURE;

  /*
   * The compiler knows about arrays so we can send them `as is'
   */
  double v[2][2][2];

  // Initialize the data
  if (comm_world.rank()==0) {
    double *vt = &(v[0][0][0]);
    for (int i=0; i<8; i++)
      *vt++ = i;
```

```
    /*
     * Send and report
     */
    comm_world.send(v, 1);  // send to rank 1

    stringstream s;
    s << "sent: ";
    vt = &(v[0][0][0]);
    for (int i=0; i<8; i++)
      s << " " << *(vt+i);
    cout << s.str() << '\n';

    // std::cout << "sent: ";
    // for (double &x : v)
    //   std::cout << x << ' ';
    // std::cout << '\n';

  } else if (comm_world.rank()==1) {

    /*
     * Receive data and report
     */
    comm_world.recv(v, 0);  // receive from rank 0

    stringstream s;
    s << "got : ";
    double *vt = &(v[0][0][0]);
    for (int i=0; i<8; i++)
      s << " " << *(vt+i);
    cout << s.str() << '\n';

  }
  return EXIT_SUCCESS;
}
```

4.5.5 Listing of code examples/mpi/mpl/sendbuffer.cxx

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  if (comm_world.size()<2)
    return EXIT_FAILURE;

  /*
   * To send a std::vector we declare a contiguous layout
   */
  std::vector<double> v(8);
  mpl::contiguous_layout<double> v_layout(v.size());
```

```
  // Initialize the data
  if (comm_world.rank()==0) {
    double init=0;
    for (double &x : v) {
      x=init;
      ++init;
    }

    /*
     * Send and report
     */
    comm_world.send(v.data(), v_layout, 1);  // send to rank 1
    std::cout << "sent: ";
    for (double &x : v)
      std::cout << x << ' ';
    std::cout << '\n';

  } else if (comm_world.rank()==1) {

    /*
     * Receive data and report
     */
    comm_world.recv(v.data(), v_layout, 0);  // receive from rank 0
    std::cout << "got : ";
    for (double &x : v)
      std::cout << x << ' ';
    std::cout << '\n';

  }
  return EXIT_SUCCESS;
}
```

4.5.6 Listing of code examples/mpi/c/recvblock.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
  int other, recvbuf=2, sendbuf=3;

#include "globalinit.c"
  MPI_Status status;

  other = 1-procno;
  if (procno>1) goto skip;
  MPI_Recv(&recvbuf,1,MPI_INT,other,0,comm,&status);
  MPI_Send(&sendbuf,1,MPI_INT,other,0,comm);
  printf("This statement will not be reached on %d\n",procno);
```

```
 skip:
  MPI_Finalize();
  return 0;
}
```

4.5.7 Listing of code examples/mpi/c/sendblock.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

/** This program shows deadlocking behaviour
    when two processes exchange data with blocking
    send and receive calls.
    Under a certain limit MPI_Send may actually not be
    blocking; we loop, increasing the message size,
    to find roughly the crossover point.
*/
int main(int argc,char **argv) {
  int  *recvbuf, *sendbuf;
  MPI_Status status;

#include "globalinit.c"

  /* we only use processors 0 and 1 */
  int other;
  if (procno>1) goto skip;
  other = 1-procno;
  /* loop over increasingly large messages */
  for (int size=1; size<2000000000; size*=10) {
    sendbuf = (int*) malloc(size*sizeof(int));
    recvbuf = (int*) malloc(size*sizeof(int));
    if (!sendbuf || !recvbuf) {
      printf("Out of memory\n"); MPI_Abort(comm,1);
    }
    MPI_Send(sendbuf,size,MPI_INT,other,0,comm);
    MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
    /* If control reaches this point, the send call
       did not block. If the send call blocks,
       we do not reach this point, and the program will hang.
    */
    if (procno==0)
      printf("Send did not block for size %d\n",size);
    free(sendbuf); free(recvbuf);
  }

 skip:
  MPI_Finalize();
  return 0;
}
```

4.5.8 Listing of code examples/mpi/f/sendblock.F90

```
Program SendBlock

#include "mpif.h"

  integer :: other,size,status(MPI_STATUS_SIZE)
  integer,dimension(:),allocatable :: sendbuf,recvbuf
#include "globalinit.F90"

  if (mytid>1) goto 10
     other = 1-mytid
     size = 1
     do
        allocate(sendbuf(size)); allocate(recvbuf(size))
        print *,size
        call MPI_Send(sendbuf,size,MPI_INTEGER,other,0,comm,err)
        call MPI_Recv(recvbuf,size,MPI_INTEGER,other,0,comm,status,err)
        if (mytid==0) then
           print *,"MPI_Send did not block for size",size
        end if
        deallocate(sendbuf); deallocate(recvbuf)
        size = size*10
        if (size>2000000000) goto 20
     end do
20   continue

10   call MPI_Finalize(err)


end program SendBlock
```

4.5.9 Listing of code examples/mpi/p/sendblock.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

if procid in [0,nprocs-1]:
    other = nprocs-1-procid
    size = 1
    while size<2000000000:
        sendbuf = np.empty(size, dtype=np.int)
        recvbuf = np.empty(size, dtype=np.int)
        comm.Send(sendbuf,dest=other)
        comm.Recv(sendbuf,source=other)
```

```
        if procid<other:
            print("Send did not block for",size)
        size *= 10
```

4.5.10 Listing of code examples/mpi/c/ssendblock.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
  int other, size, *recvbuf, *sendbuf;
  MPI_Status status;

#include "globalinit.c"

  if (procno>1) goto skip;
  other = 1-procno;
  sendbuf = (int*) malloc(sizeof(int));
  recvbuf = (int*) malloc(sizeof(int));
  size = 1;
  MPI_Ssend(sendbuf,size,MPI_INT,other,0,comm);
  MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
  printf("This statement is not reached\n");
  free(sendbuf); free(recvbuf);

 skip:
  MPI_Finalize();
  return 0;
}
```

4.5.11 Listing of code examples/mpi/c/isendandirecv.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  /*
   * We set up a single communication between
   * the first and last process
   */
  int sender,receiver;
  sender = 0; receiver = nprocs-1;

  if (procno==sender) {
    double send_data = 1.;
```

```
    MPI_Request request;
    MPI_Isend
      ( /* send buffer/count/type: */ &send_data,1,MPI_DOUBLE,
/* to: */ receiver, /* tag: */ 0,
/* communicator: */ comm,
/* request: */ &request);
    MPI_Wait(&request,MPI_STATUS_IGNORE);
    printf("[%d] Isend successfully concluded\n",procno);
  } else if (procno==receiver) {
    double recv_data;
    MPI_Request request;
    MPI_Irecv
      ( /* recv buffer/count/type: */ &recv_data,1,MPI_DOUBLE,
/* from: */ sender, /* tag: */ 0,
/* communicator: */ comm,
/* request: */ &request);
    MPI_Wait(&request,MPI_STATUS_IGNORE);
    printf("[%d] Ireceive successfully concluded\n",procno);
  }

  MPI_Finalize();
  return 0;
}
```

4.5.12 Listing of code examples/mpi/c/hangwait.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  double
    mydata=procno;
  int sender = nprocs-1;

  if (procno==sender) {
    for (int p=0; p<nprocs-1; p++) {
      double send = 1.;
      MPI_Send( &send,1,MPI_DOUBLE,p,0,comm);
    }
  } else {
    double recv=0.;
    MPI_Request request;
    MPI_Irecv( &recv,1,MPI_DOUBLE,sender,0,comm,&request);
    MPI_Wait(&request,MPI_STATUS_IGNORE);
  }
```

```
  MPI_Finalize();
  return 0;
}
```

4.5.13 Listing of code examples/mpi/mpl/isendandirecv.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

  const mpl::communicator &comm_world=mpl::environment::comm_world();
  int
    nprocs = comm_world.size(),
    procno = comm_world.rank();
  if (comm_world.size()<2)
    return EXIT_FAILURE;

  int sender = 0,receiver = nprocs-1;

  if (procno==sender) {
    double send_data = 1.;
    mpl::irequest send_request
      ( comm_world.isend( send_data, receiver ) );
    send_request.wait();
    printf("[%d] Isend successfully concluded\n",procno);
  } else if (procno==receiver) {
    double recv_data;
    mpl::irequest recv_request =
      comm_world.irecv( recv_data,sender );
    recv_request.wait();
    printf("[%d] Ireceive successfully concluded\n",procno);
  }

  return 0;
}
```

4.5.14 Listing of code examples/mpi/p/irecvloop.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
```

```
    sys.exit(1)

requests = [ None ] * (2*nprocs)
sendbuffer = np.empty( nprocs, dtype=np.int )
recvbuffer = np.empty( nprocs, dtype=np.int )

for p in range(nprocs):
    left_p = (p-1) % nprocs
    right_p = (p+1) % nprocs
    requests[2*p] = comm.Isend\
        ( sendbuffer[p:p+1],dest=left_p)
    requests[2*p+1] = comm.Irecv\
        ( sendbuffer[p:p+1],source=right_p)
MPI.Request.Waitall(requests)
```

4.5.15 Listing of code examples/mpi/f/irecvsource.F90

```
Program Irecv_source

  use mpi

  implicit none
  integer,dimension(:),allocatable :: recv_buffer,requests
  integer :: index,randomint
  real :: randomvalue
#include "globalinit.F90"
  allocate(recv_buffer(ntids-1))
  allocate(requests(ntids-1))

  if (mytid==ntids-1) then
     do p=1,ntids-1
        print *,"post"
        call MPI_Irecv(recv_buffer(p),1,MPI_INTEGER,p-1,0,comm,&
             requests(p),err)
     end do
     do p=1,ntids-1
        call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE,err)
        write(*,'("Message from",i3,":",i5)') index,recv_buffer(index)
     end do
  else
     call sleep(6)
     call random_number(randomvalue)
     randomint = randomvalue
     randomint = 30+mytid
     call MPI_Send(randomint,1,MPI_INTEGER, ntids-1,0,comm,err)
  end if

end Program Irecv_source
```

4.5.16 Listing of code examples/mpi/f08/waitnull.F90

```
Program Waitnull
```

```
use mpi_f08

implicit none

!!
!! General stuff
!!
Type(MPI_Comm) :: comm;
integer :: mytid,ntids,i,p,err;

!!
!! random number generatoe
!!
integer :: randsize
integer,allocatable,dimension(:) :: randseed

!!
!! data for this program
!!
Type(MPI_Request),dimension(:),allocatable :: requests
integer,dimension(:),allocatable :: recv_buffer
integer :: index,randomint,success = 1
real :: randomvalue

call MPI_Init()
comm = MPI_COMM_WORLD
call MPI_Comm_rank(comm,mytid)
call MPI_Comm_size(comm,ntids)
call MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN)

!!
!! seed the random number generator
!!
call random_seed(size=randsize)
allocate(randseed(randsize))
do i=1,randsize
   randseed(i) = 1023*mytid
end do
call random_seed(put=randseed)

allocate(recv_buffer(ntids-1))
allocate(requests(ntids-1))

if (mytid==ntids-1) then
   !
   ! the last process posts a receive
   ! from every other process
   !
   do p=0,ntids-2
      call MPI_Irecv(recv_buffer(p+1),1,MPI_INTEGER,p,0,comm,&
           requests(p+1))
   end do
```

```
    !
    ! then wait to see what comes in
    !
    do p=0,ntids-2
       call MPI_Waitany(ntids-1,requests,index,MPI_STATUS_IGNORE)
       if ( .not. requests(index)==MPI_REQUEST_NULL) then
            print *,"This request should be null:",index
            success = 0
         end if
       !write(*,'("Message from",i3,":",i5)') index,recv_buffer(index)
    end do

  else
    !
    ! everyone else sends one number to the last
    ! after some random wait
    !
    call sleep(6)
    call random_number(randomvalue)
    randomint = randomvalue
    randomint = 30+mytid
    call MPI_Send(randomint,1,MPI_INTEGER, ntids-1,0,comm)
  end if

  call MPI_Allreduce(MPI_IN_PLACE,success,1,MPI_INTEGER,MPI_SUM,comm)
  if (mytid==0) then
     if (success==ntids) then
        print *,"All processes successfully concluded"
     else
        print *,"Failure on",ntids-success,"processes"
     end if
  end if

  call MPI_Finalize()

end Program Waitnull
```

4.5.17 Listing of code examples/mpi/mpl/irecvsource.cxx

```
#include <cstdlib>
#include <unistd.h>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;
#include <vector>
using std::vector;

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {
```

```
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  int
    nprocs = comm_world.size(),
    procno = comm_world.rank();
  if (comm_world.size()<2)
    return EXIT_FAILURE;

  // Initialize the random number generator
  srand((int)(procno*(double)RAND_MAX/nprocs));

  if (procno==nprocs-1) {
    mpl::irequest_pool recv_requests;
    vector<int> recv_buffer(nprocs-1);
    for (int p=0; p<nprocs-1; p++) {
      recv_requests.push( comm_world.irecv( recv_buffer[p], p ) );
    }
    printf("Outstanding request #=%d\n",recv_requests.size());
    for (int p=0; p<nprocs-1; p++) {
      auto [success,index] = recv_requests.waitany();
      if (success) {
     auto recv_status = recv_requests.get_status(index);
     int source = recv_status.source();
     if (index!=source)
       printf("Mismatch index %lu vs source %d\n",index,source);
     printf("Message from %lu: %d\n",index,recv_buffer[index]);
      } else
     break;
    }
  } else {
    float randomfraction = (rand() / (double)RAND_MAX);
    int randomwait = (int) ( 2* nprocs * randomfraction );
    printf("process %d waits for %d\n",procno,randomwait);
    sleep(randomwait);
    comm_world.send( procno, nprocs-1 );
  }

  return 0;
}
```

4.5.18 Listing of code examples/mpi/c/irecvsource.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  // Initialize the random number generator
  srand((int)(procno*(double)RAND_MAX/nprocs));
```

```
  if (procno==nprocs-1) {
    int *recv_buffer;
    MPI_Request *request; MPI_Status status;
    recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));
    request = (MPI_Request*) malloc((nprocs-1)*sizeof(MPI_Request));

    for (int p=0; p<nprocs-1; p++) {
      ierr = MPI_Irecv(recv_buffer+p,1,MPI_INT, p,0,comm,
      request+p); CHK(ierr);
    }
    for (int p=0; p<nprocs-1; p++) {
      int index,sender;
      MPI_Waitany(nprocs-1,request,&index,&status); //MPI_STATUS_IGNORE);
      if (index!=status.MPI_SOURCE)
printf("Mismatch index %d vs source %d\n",index,status.MPI_SOURCE);
      printf("Message from %d: %d\n",index,recv_buffer[index]);
    }
  } else {
    float randomfraction = (rand() / (double)RAND_MAX);
    int randomwait = (int) ( 2* nprocs * randomfraction );
    printf("process %d waits for %d\n",procno,randomwait);
    sleep(randomwait);
    ierr = MPI_Send(&procno,1,MPI_INT, nprocs-1,0,comm); CHK(ierr);
  }

  MPI_Finalize();
  return 0;
}
```

4.5.19 Listing of code examples/mpi/p/irecvsource.py

```
import numpy as np
import random
import time
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

if procid==nprocs-1:
    receive_buffer = np.empty(nprocs-1,dtype=np.int)
    requests = [ None ] * (nprocs-1)
    for sender in range(nprocs-1):
        requests[sender] = comm.Irecv(receive_buffer[sender:sender+1],source=sender)
    # alternatively: requests = [ comm.Irecv(s) for s in .... ]
    status = MPI.Status()
    for sender in range(nprocs-1):
        ind = MPI.Request.Waitany(requests,status=status)
```

```
        if ind!=status.Get_source():
            print("sender mismatch: %d vs %d" % (ind,status.Get_source()))
        print("received from",ind)
else:
    mywait = random.randint(1,2*nprocs)
    print("[%d] wait for %d seconds" % (procid,mywait))
    time.sleep(mywait)
    mydata = np.empty(1,dtype=np.int)
    mydata[0] = procid
    comm.Send([mydata,MPI.INT],dest=nprocs-1)
```

4.5.20 Listing of code examples/mpi/c/irecvloop.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  int recvbuf=2, sendbuf=3, other;
  MPI_Request requests =
    (MPI_Request*) malloc( 2*nprocs*sizeof(MPI_Request) );
  recv_buffers = (int*) malloc( nprocs*sizeof(int) );
  send_buffers = (int*) malloc( nprocs*sizeof(int) );
  for (int p=0; p<nprocs; p++) {
    int
      left_p  = (p-1+nprocs) % nprocs,
      right_p = (p+1) % nprocs;
    send_buffer[p] = nprocs-p;
    MPI_Isend(sendbuffer+p,1,MPI_INT, right_p,0, requests+2*p);
    MPI_Irecv(recvbuffer+p,1,MPI_INT, left_p,0, requests+2*p+1);
  }
  /* your useful code here */
  MPI_Waitall(2*nprocs,requests,MPI_STATUSES_IGNORE);

 skip:
  MPI_Finalize();
  return 0;
}
```

4.5.21 Listing of code examples/mpi/c/probe.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc,char **argv) {
```

```
#include "globalinit.c"

  // Initialize the random number generator
  srand((int)(procno*(double)RAND_MAX/nprocs));

  int sender = 0, receiver = nprocs-1;
  if (procno==receiver) {
    MPI_Status status;
    MPI_Probe(sender,0,comm,&status);
    int count;
    MPI_Get_count(&status,MPI_FLOAT,&count);
    printf("Receiving %d floats\n",count);
    float recv_buffer[count];
    MPI_Recv(recv_buffer,count,MPI_FLOAT, sender,0,comm,MPI_STATUS_IGNORE);
  } else if (procno==sender) {
    float randomfraction = (rand() / (double)RAND_MAX);
    int buffer_size = (int) ( 10 * nprocs * randomfraction );
    printf("Sending %d floats\n",buffer_size);
    float buffer[buffer_size];
    ierr = MPI_Send(buffer,buffer_size,MPI_FLOAT, receiver,0,comm); CHK(ierr);
  }

  MPI_Finalize();
  return 0;
}
```

4.5.22 Listing of code examples/mpi/mpl/vector.cxx

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <vector>
using std::vector;
#include <cassert>

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

  const mpl::communicator &comm_world = mpl::environment::comm_world();
  int nprocs,procno;
  // compute communicator rank and size
  nprocs = comm_world.size();
  procno = comm_world.rank();

  if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
  }
  int sender = 0, receiver = 1, the_other = 1-procno,
    count = 5,stride=2;
  vector<double>
```

```
    source(stride*count);
  vector<double>
    target(count);

  for (int i=0; i<stride*count; i++)
    source[i] = i+.5;

  if (procno==sender) {
    mpl::strided_vector_layout<double>
      newvectortype(count,1,stride);
    comm_world.send
      (source.data(),newvectortype,the_other);
  }
  else if (procno==receiver) {
    int recv_count;
    mpl::contiguous_layout<double> target_layout(count),
    mpl::status recv_status =
      comm_world.recv(target.data(),target_layout, the_other);
    recv_count = recv_status.get_count<double>();
    assert(recv_count==count);
  }

  if (procno==receiver) {
    for (int i=0; i<count; i++)
      if (target[i]!=source[stride*i])
printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
  }

  if (procno==0)
    printf("Finished\n");

  return 0;
}
```

4.5.23 Listing of code examples/mpi/c/anysource.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (procno==nprocs-1) {
    /*
     * The last process receives from every other process
     */
    int *recv_buffer;
    MPI_Status status;
```

```
    recv_buffer = (int*) malloc((nprocs-1)*sizeof(int));

    /*
     * Messages can come in in any order, so use MPI_ANY_SOURCE
     */
    for (int p=0; p<nprocs-1; p++) {
      err = MPI_Recv(recv_buffer+p,1,MPI_INT, MPI_ANY_SOURCE,0,comm,
      &status); CHK(err);
      int sender = status.MPI_SOURCE;
      printf("Message from sender=%d: %d\n",
     sender,recv_buffer[p]);
    }
  } else {
    /*
     * Each rank waits an unpredictable amount of time,
     * then sends to the last process in line.
     */
    float randomfraction = (rand() / (double)RAND_MAX);
    int randomwait = (int) ( nprocs * randomfraction );
    printf("process %d waits for %e/%d=%d\n",
   procno,randomfraction,nprocs,randomwait);
    sleep(randomwait);
    err = MPI_Send(&randomwait,1,MPI_INT, nprocs-1,0,comm); CHK(err);
  }

  MPI_Finalize();
  return 0;
}
```

4.5.24 Listing of code examples/mpi/f08/anysource.F90

```
Program AnySource

  use mpi_f08

  implicit none

  integer,dimension(:),allocatable :: recv_buffer
  Type(MPI_Status)  :: status
  real :: randomvalue
  integer :: randomint,sender

#include "globalinit.F90"

      if (mytid.eq.ntids-1) then
         allocate(recv_buffer(ntids-1))
         do p=0,ntids-2
            call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
                 MPI_ANY_SOURCE,0,comm,status)
            sender = status%MPI_SOURCE
            print *,"Message from",sender
         end do
      else
```

```
         call random_number(randomvalue)
         randomint = randomvalue*ntids
         call sleep(randomint)
         print *,mytid,"waits for",randomint
         call MPI_Send(p,1,MPI_INTEGER,ntids-1,0,comm)
      end if

  call MPI_Finalize(err)


end program AnySource
```

4.5.25 Listing of code examples/mpi/f/anysource.F90

```
Program AnySource

  implicit none

#include "mpif.h"

  integer,dimension(:),allocatable :: recv_buffer
  integer :: status(MPI_STATUS_SIZE)
  real :: randomvalue
  integer :: randomint,sender

#include "globalinit.F90"

      if (mytid.eq.ntids-1) then
         allocate(recv_buffer(ntids-1))
         do p=0,ntids-2
            call MPI_Recv(recv_buffer(p+1),1,MPI_INTEGER,&
                 MPI_ANY_SOURCE,0,comm,status,err)
            sender = status(MPI_SOURCE)
            print *,"Message from",sender
         end do
      else
         call random_number(randomvalue)
         randomint = randomvalue*ntids
         call sleep(randomint)
         print *,mytid,"waits for",randomint
         call MPI_Send(p,1,MPI_INTEGER,ntids-1,0,comm,err)
      end if

  call MPI_Finalize(err)


end program AnySource
```

4.5.26 Listing of code examples/mpi/mpl/recvstatus.cxx

```
#include <cstdlib>
#include <complex>
```

```
#include <iostream>
#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  if (comm_world.size()<2)
    return EXIT_FAILURE;
  // send and recieve a single floating point number
  if (comm_world.rank()==0) {
    double pi=3.14;
    comm_world.send(pi, 1);  // send to rank 1
    std::cout << "sent: " << pi << '\n';
  } else if (comm_world.rank()==1) {
    double pi=0;
    auto s = comm_world.recv(pi, 0);  // receive from rank 0
    int c = s.get_count<double>();
    std::cout << "got : " << c << " scalar(s): " << pi << '\n';
  }
  return EXIT_SUCCESS;
}
```

4.5.27 Listing of code examples/mpi/p/anysource.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

if procid==nprocs-1:
    rbuf = np.empty(1,dtype=np.float64)
    for p in range(procid):
        rstatus = MPI.Status()
        comm.Recv(rbuf,source=MPI.ANY_SOURCE,status=rstatus)
        print("Message came from %d" % rstatus.Get_source())
else:
    sbuf = np.empty(1,dtype=np.float64)
    sbuf = np.empty(1,dtype=np.float64)
    sbuf[0] = 1.
    comm.Send(sbuf,dest=nprocs-1)
```

Chapter 5

MPI topic: Communication modes

5.1 Persistent communication requests

Persistent communication is a mechanism for dealing with a repeating communication transaction, where the parameters of the transaction, such as sender, receiver, tag, root, and buffer type and size, stay the same. Only the contents of the buffers involved changes between the transactions.

You can imagine that setting up a communication carries some overhead, and if the same communication structure is repeated many times, this overhead may be avoided.

1. For nonblocking communications `MPI_Ixxx` (both point-to-point and collective) there is a persistent variant `MPI_Xxx_init` with the same calling sequence. The 'init' call produces an `MPI_Request` output parameter, which can be used to test for completion of the communication.
2. The 'init' routine does not start the actual communication: that is done in `MPI_Start`, or `MPI_Startall` for multiple requests.
3. Any of the MPI 'wait' calls can then be used to conclude the communication.
4. The communication can then be restarted with another 'start' call.
5. The wait call does not release the request object, since it can be used for repeat occurrences of this transaction. The request object is freed, as usual, with `MPI_Request_free`.

```
MPI_Send_init( /* ... */ &request);
while ( /* ... */ ) {
  MPI_Start( request );
  MPI_Wait( request, &status );
}
MPI_Request_free( & request );
```

MPL note 34. MPL returns a `prequest` from persistent 'init' routines, rather than an `irequest` (MPL note 27):

```
template<typename T >
prequest send_init (const T &data, int dest, tag t=tag(0)) const;
```

Likewise, there is a `prequest_pool` instead of an `irequest_pool` (note 28).

End of MPL note

5.1.1 Persistent point-to-point communication

The main persistent point-to-point routines are `MPI_Send_init`, which has the same calling sequence as `MPI_Isend`, and `MPI_Recv_init`, which has the same calling sequence as `MPI_Irecv`.

In the following example a ping-pong is implemented with persistent communication. Since we use persistent operations for both send and receive on the 'ping' process, we use `MPI_Startall` to start both at the same time, and `MPI_Waitall` to test their completion.

```
// persist.c
if (procno==src) {
  MPI_Send_init(send,s,MPI_DOUBLE,tgt,0,comm,requests+0);
  MPI_Recv_init(recv,s,MPI_DOUBLE,tgt,0,comm,requests+1);
  printf("Size %d\n",s);
  t[cnt] = MPI_Wtime();
  for (int n=0; n<NEXPERIMENTS; n++) {
        fill_buffer(send,s,n);
        MPI_Startall(2,requests);
        MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
        int r = chck_buffer(send,s,n);
        if (!r) printf("buffer problem %d\n",s);
  }
  t[cnt] = MPI_Wtime()-t[cnt];
  MPI_Request_free(requests+0); MPI_Request_free(requests+1);
} else if (procno==tgt) {
  for (int n=0; n<NEXPERIMENTS; n++) {
        MPI_Recv(recv,s,MPI_DOUBLE,src,0,comm,MPI_STATUS_IGNORE);
        MPI_Send(recv,s,MPI_DOUBLE,src,0,comm);
  }
}
```

For the full source of this example, see section 5.6.2

```
## persist.py
sendbuf = np.ones(size,dtype=np.int)
recvbuf = np.ones(size,dtype=np.int)
if procid==src:
    print("Size:",size)
    times[isize] = MPI.Wtime()
    for n in range(nexperiments):
        requests[0] = comm.Isend(sendbuf[0:size],dest=tgt)
        requests[1] = comm.Irecv(recvbuf[0:size],source=tgt)
        MPI.Request.Waitall(requests)
        sendbuf[0] = sendbuf[0]+1
    times[isize] = MPI.Wtime()-times[isize]
elif procid==tgt:
    for n in range(nexperiments):
        comm.Recv(recvbuf[0:size],source=src)
        comm.Send(recvbuf[0:size],dest=src)
```

For the full source of this example, see section 5.6.3

As with ordinary send commands, there are persistent variants

- `MPI_Bsend_init` for buffered communication, section 5.5;

- `MPI_Ssend_init` for synchronous communication, section 5.3.1;
- `MPI_Rsend_init`.

5.1.2 Persistent collectives

The following material is for the (unreleased) MPI-4 standard only

For each collective call, there is a persistent variant (MPI-4). As with persistent point-to-point calls (section 5.1.1), these have the same calling sequence as the nonpersistent variants, except for an added final `MPI_Request` parameter. This request (or an array of requests) can then be used by `MPI_Start` (or `MPI_Startall`) to initiate the actual communication.

Some points.

- Metadata arrays, such as of counts and datatypes, must not be altered until the `MPI_Request_free` call.
- The initialization call is nonlocal, so it can block until all processes have performed it.
- Multiple persistent collective can be initialized, in which case they satisfy the same restrictions as ordinary collectives, in particular on ordering. Thus, the following code is incorrect:

```
// WRONG
if (procid==0) {
  MPI_Reduce_init( /* ... */ &req1);
  MPI_Bcast_init( /* ... */ &req2);
} else {
  MPI_Bcast_init( /* ... */ &req2);
  MPI_Reduce_init( /* ... */ &req1);
}
```

  However, after initialization the start calls can be in arbitrary order, and in different order among the processes.

Available persistent collectives are: `MPI_Barrier_init` `MPI_Bcast_init` `MPI_Reduce_init` `MPI_Allreduce_init` `MPI_Reduce_scatter_init` `MPI_Reduce_scatter_block_init` `MPI_Gather_init` `MPI_Gatherv_init` `MPI_Allgather_init` `MPI_Allgatherv_init` `MPI_Scatter_init` `MPI_Scatterv_init` `MPI_Alltoall_init` `MPI_Alltoallv_init` `MPI_Alltoallw_init` `MPI_Scan_init` `MPI_Exscan_init`

End of MPI-4 material

5.1.3 Persistent neighbor communications

The following material is for the (unreleased) MPI-4 standard only

`MPI_Neighbor_allgather_init`, `MPI_Neighbor_allgatherv_init`, `MPI_Neighbor_alltoall_init`, `MPI_Neighbor_alltoallv_init`, `MPI_Neighbor_alltoallw_init`,

End of MPI-4 material

5.2 Partitioned communication

The following material is for the (unreleased) MPI-4 standard only

Partitioned communication is a variant on *persistent communication*, where a message is constructed in partitions.

- The normal `MPI_Send_init` is replaced by `MPI_Psend_init`.
- After this, the `MPI_Start` does not actually start the transfer; instead:
- Each partition of the message is separately declared as read-to-be-sent with `MPI_Pready`.

A common scenario for this is in multi-threaded environments, where each thread can construct its own part of a message. Having partitioned messages means that partially constructed message buffers can be sent off without having to wait for all threads to finish.

Indicating that parts of a message are ready for sending is done by one of the following calls:

- `MPI_Pready` for a single partition;
- `MPI_Pready_range` for a range of partitions; and
- `MPI_Pready_list` for an explicitly enumerated list of partitions.

The `MPI_Psend_init` call yields an `MPI_Request` object that can be used to test for completion (see sections 4.2.2.1 and 4.2.3) of the full operation started with `MPI_Start`.

On the receiving side:

- A call to `MPI_Recv_init` is replaced by `MPI_Precv_init`.
- Arrival of a partition can be tested with `MPI_Parrived`.

Again, the `MPI_Request` object from the receive-init call can be used to test for completion of the full receive operation.
End of MPI-4 material

5.3 Synchronous and asynchronous communication

It is easiest to think of blocking as a form of synchronization with the other process, but that is not quite true. Synchronization is a concept in itself, and we talk about *synchronous* communication if there is actual coordination going on with the other process, and *asynchronous* communication if there is not. Blocking then only refers to the program waiting until the user data is safe to reuse; in the synchronous case a blocking call means that the data is indeed transferred, in the asynchronous case it only means that the data has been transferred to some system buffer. The four possible cases are illustrated in figure 5.1.

5.3.1 Synchronous send operations

MPI has a number of routines for synchronous communication, such as `MPI_Ssend`. Driving home the point that nonblocking and asynchronous are different concepts, there is a routine `MPI_Issend`, which is synchronous but nonblocking. These routines have the same calling sequence as their not-explicitly synchronous variants, and only differ in their semantics.

```
// ssendblock.c
other = 1-procno;
sendbuf = (int*) malloc(sizeof(int));
recvbuf = (int*) malloc(sizeof(int));
size = 1;
MPI_Ssend(sendbuf,size,MPI_INT,other,0,comm);
MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
printf("This statement is not reached\n");
```

For the full source of this example, see section 5.6.4

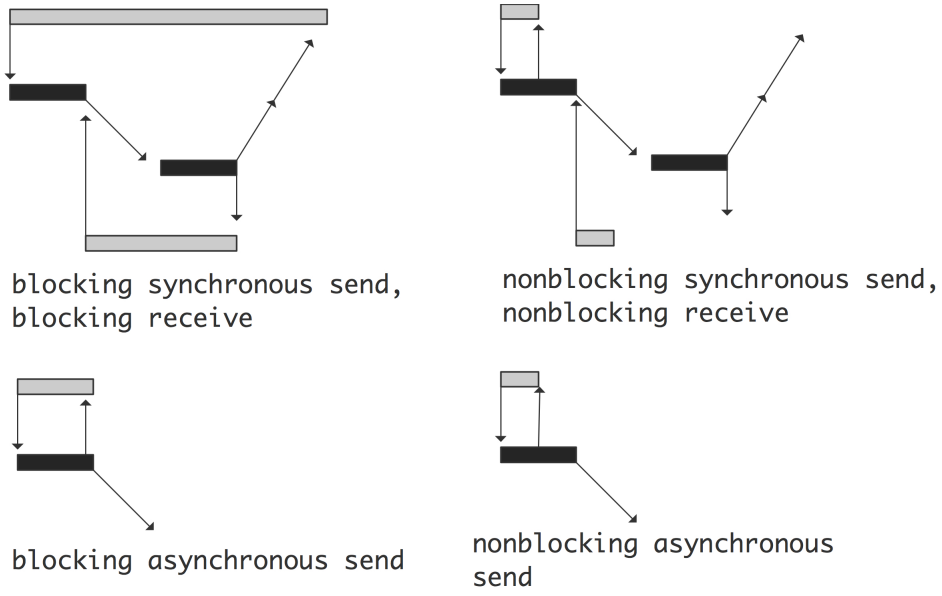


Figure 5.1: Blocking and synchronicity

5.4 Local and nonlocal operations

The MPI standard does not dictate whether communication is buffered. If a message is buffered, a send call can complete, even if no corresponding send has been posted yet. See section 4.1.2.2. Thus, in the standard communication, a send operation is *nonlocal*: its completion may be depend on whether the corresponding receive has been posted. A *local operation* is one that is not nonlocal.

On the other hand, *buffered communication* (routines `MPI_Bsend`, `MPI_Ibsend`, `MPI_Bsend_init`; section 5.5) is *local*: the presence of an explicit buffer means that a send operation can complete no matter whether the receive has been posted.

The *synchronous send* (routines `MPI_Ssend`, `MPI_Issend`, `MPI_Ssend_init`; section 14.7) is again nonlocal (even in the nonblocking variant) since it will only complete when the receive call has completed.

Finally, the *ready mode send* (`MPI_Rsend`, `MPI_Irsend`) is nonlocal in the sense that its only correct use is when the corresponding receive has been issued.

5.4.1 Asynchronous progress

The concept *asynchronous progress* describes that MPI messages continue on their way through the network, while the application is otherwise busy.

The problem here is that, unlike straight `MPI_Send` and `MPI_Recv` calls, communication of this sort can typically not be off-loaded to the network card, so different mechanisms are needed.

This can happen in a number of ways:

- Compute nodes may have a dedicated communications processor. The *Intel Paragon* was of this design; modern multicore processors are a more efficient realization of this idea.
- The MPI library may reserve a core or thread for communications processing. This is implementation dependent; for instance, *Intel MPI* has a number of `I_MPI_ASYNC_PROGRESS_...` variables.
- Absent such dedicated resources, the application can force MPI to make progress by occasional calls to a *polling* routine such as `MPI_Iprobe`.

Remark 10 *The `MPI_Probe` call is somewhat similar, in spirit if not quite in functionality, as `MPI_Test`. However, they behave differently with respect to progress. Quoting the standard:*

> *The MPI implementation of `MPI_Probe` and `MPI_Iprobe` needs to guarantee progress: if a call to `MPI_Probe` has been issued by a process, and a send that matches the probe has been initiated by some process, then the call to `MPI_Probe` will return.*

In other words: probing causes MPI to make progress. On the other hand,

> *A call to `MPI_Test` returns `flag = true` if the operation identified by request is complete.*

In other words, if progress has been made, then testing will report completion, but by itself it does not cause completion.

A similar problem arises with passive target synchronization: it is possible that the origin process may hang until the target process makes an MPI call.

Intel note. Only available with the `release_mt` and `debug_mt` versions of the Intel MPI library. Set `I_MPI_ASYNC_PROGRESS` to 1 to enable asynchronous progress threads, and `I_MPI_ASYNC_PROGRESS_THREADS` to set the number of progress threads.

See https://software.intel.com/en-us/mpi-developer-guide-linux-asynchronous-progress-control, https://software.intel.com/en-us/mpi-developer-reference-linux-environment-variables-for-asynchronous-progress-control

5.5 Buffered communication

By now you have probably got the notion that managing buffer space in MPI is important: data has to be somewhere, either in user-allocated arrays or in system buffers. Using *buffered communication* is yet another way of managing buffer space.

1. You allocate your own buffer space, and you attach it to your process. This buffer is not a send buffer: it is a replacement for buffer space used inside the MPI library or on the network card; figure 5.2. If high-bandwdith memory is available, you could create your buffer there.
2. You use the `MPI_Bsend` (figure 5.1) (or its *local* variant `MPI_Ibsend`) call for sending, using otherwise normal send and receive buffers;
3. You detach the buffer when you're done with the buffered sends.

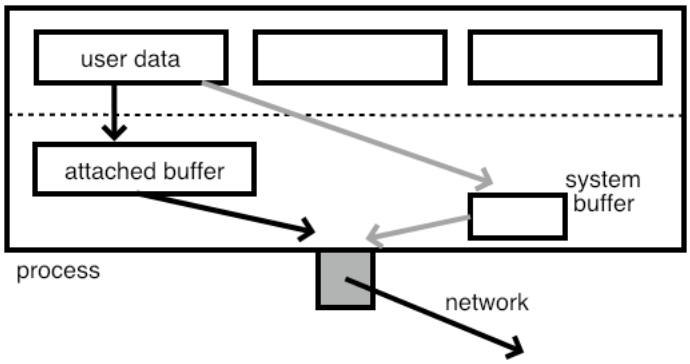


Figure 5.2: User communication routed through an attached buffer

Figure 5.1 `MPI_Bsend`

Name	Param name	C type	F type	inout
`mpi_bsend (`				
	`buf` *initial address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`count` *number of elements in send buffer*	`int`	`INTEGER`	`in`
	`datatype` *datatype of each send buffer element*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`dest` *rank of destination*	`int`	`INTEGER`	`in`
	`tag` *message tag*	`int`	`INTEGER`	`in`
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.bsend(`		
	`self`	
	`obj`	`Any`
	`dest`	`int`
	`tag`	`int = 0`
`) -> None`		

One advantage of buffered sends is that they are nonblocking: since there is a guaranteed buffer long enough to contain the message, it is not necessary to wait for the receiving process.

We illustrate the use of buffered sends:

```
// bufring.c
int bsize = BUFLEN*sizeof(float);
float
  *sbuf = (float*) malloc( bsize ),
  *rbuf = (float*) malloc( bsize );
MPI_Pack_size( BUFLEN,MPI_FLOAT,comm,&bsize);
bsize += MPI_BSEND_OVERHEAD;
```

Figure 5.2 `MPI_Buffer_attach`

Name	Param name	C type	F type	inout
`mpi_buffer_attach (`				
	`buffer` *initial buffer address*	`void*`	`TYPE(*), DIMENSION(..)`	`in`
	`size` *buffer size, in bytes*	`int`	`INTEGER`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

```
float
  *buffer = (float*) malloc( bsize );

MPI_Buffer_attach( buffer,bsize );
err = MPI_Bsend(sbuf,BUFLEN,MPI_FLOAT,next,0,comm);
MPI_Recv (rbuf,BUFLEN,MPI_FLOAT,prev,0,comm,MPI_STATUS_IGNORE);
MPI_Buffer_detach( &buffer,&bsize );
```

5.5.1 Buffer treatment

There can be only one buffer per process, attached with `MPI_Buffer_attach` (figure 5.2). Its size should be enough for all `MPI_Bsend` calls that are simultaneously outstanding. You can compute the needed size of the buffer with `MPI_Pack_size`; see section 6.6.2. Additionally, a term of `MPI_BSEND_OVERHEAD` is needed. See the above code fragment.

The buffer is detached with `MPI_Buffer_detach`:

```
int MPI_Buffer_detach(
  void *buffer, int *size);
```

This returns the address and size of the buffer; the call blocks until all buffered messages have been delivered.

Note that both `MPI_Buffer_attach` and `MPI_Buffer_detach` have a `void*` argument for the buffer, but

- in the attach routine this is the address of the buffer,
- while the detach routine it is the address of the buffer pointer.

This is done so that the detach routine can zero the buffer pointer.

While the buffered send is nonblocking like an `MPI_Isend`, there is no corresponding wait call. You can force delivery by

```
MPI_Buffer_detach( &b, &n );
MPI_Buffer_attach( b, n );
```

MPL note 35. Creating and attaching a buffer is done through `bsend_buffer` and a support routine `bsend_size` helps in calculating the buffer size:

```
// bufring.cxx
vector<float> sbuf(BUFLEN), rbuf(BUFLEN);
```

```
int size{ comm_world.bsend_size<float>(mpl::contiguous_layout<float>(BUFLEN)) };
mpl::bsend_buffer<> buff(size);
comm_world.bsend(sbuf.data(),mpl::contiguous_layout<float>(BUFLEN), next);
```

Constant: `mpl::bsend_overhead` is *constexpr*'d to the MPI constant `MPI_BSEND_OVERHEAD`.

End of MPL note

MPL note 36. There is a separate attach routine, but normally this is called by the constructor of the `bsend_buffer`. Likewise, the detach routine is called in the buffer destructor.

```
void mpl::environment::buffer_attach (void *buff, int size);
std::pair< void *, int > mpl::environment::buffer_detach ();
```

End of MPL note

5.5.2 Bufferend send calls

The possible error codes are

- `MPI_SUCCESS` the routine completed successfully.
- `MPI_ERR_BUFFER` The buffer pointer is invalid; this typically means that you have supplied a null pointer.
- `MPI_ERR_INTERN` An internal error in MPI has been detected.

The asynchronous version is `MPI_Ibsend`, the persistent (see section 5.1) call is `MPI_Bsend_init`.

5.5.3 Persistent buffered communication

There is a persistent variant `MPI_Bsend_init` (figure 5.3) of buffered sends, as with regular sends (section 5.1).

Figure 5.3 `MPI_Bsend_init`

Name	Param name	C type	F type	inout
`mpi_bsend_init (`				
	`buf` *initial address of send buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`count` *number of elements sent*	`int`	`INTEGER`	`in`
	`datatype` *type of each element*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`dest` *rank of destination*	`int`	`INTEGER`	`in`
	`tag` *message tag*	`int`	`INTEGER`	`in`
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`request`	`MPI_Request*`	`TYPE(MPI_Request)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Bsend_init(`		
	`self`	
	`buf`	`BufSpec`
	`dest`	`int`
	`tag`	`int = 0`
`) -> Request`		

5.6 Sources used in this chapter

5.6.1 Listing of code header

5.6.2 Listing of code examples/mpi/c/persist.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

void fill_buffer(double *send,int s,int n) {
  for (int i=0; i<s; i++)
    send[i] = n;
}
int chck_buffer(double *recv,int s,int n) {
  int r = 1;
  for (int i=0; i<s; i++)
    r = r && recv[i]==n;
  return r;
}

int main(int argc,char **argv) {
```

```
#include "globalinit.c"

#define NEXPERIMENTS 10
#define NSIZES 6
  int src = 0,tgt = nprocs-1,maxsize=1000000;
  double t[NSIZES], send[maxsize],recv[maxsize];

  MPI_Request requests[2];

  // First ordinary communication
  for (int cnt=0,s=1; cnt<NSIZES && s<maxsize; s*=10,cnt++) {
    if (procno==src) {
      printf("Size %d\n",s);
      t[cnt] = MPI_Wtime();
      for (int n=0; n<NEXPERIMENTS; n++) {
MPI_Isend(send,s,MPI_DOUBLE,tgt,0,comm,requests+0);
MPI_Irecv(recv,s,MPI_DOUBLE,tgt,0,comm,requests+1);
MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
      }
      t[cnt] = MPI_Wtime()-t[cnt];
    } else if (procno==tgt) {
      for (int n=0; n<NEXPERIMENTS; n++) {
MPI_Recv(recv,s,MPI_DOUBLE,src,0,comm,MPI_STATUS_IGNORE);
MPI_Send(recv,s,MPI_DOUBLE,src,0,comm);
      }
    }
  }
  if (procno==src) {
    for (int cnt=0,s=1; cnt<NSIZES; s*=10,cnt++) {
      t[cnt] /= NEXPERIMENTS;
      printf("Time for pingpong of size %d: %e\n",s,t[cnt]);
    }
  }

  // Now persistent communication
  for (int cnt=0,s=1; cnt<NSIZES; s*=10,cnt++) {
    if (procno==src) {
      MPI_Send_init(send,s,MPI_DOUBLE,tgt,0,comm,requests+0);
      MPI_Recv_init(recv,s,MPI_DOUBLE,tgt,0,comm,requests+1);
      printf("Size %d\n",s);
      t[cnt] = MPI_Wtime();
      for (int n=0; n<NEXPERIMENTS; n++) {
fill_buffer(send,s,n);
MPI_Startall(2,requests);
MPI_Waitall(2,requests,MPI_STATUSES_IGNORE);
int r = chck_buffer(send,s,n);
if (!r) printf("buffer problem %d\n",s);
      }
      t[cnt] = MPI_Wtime()-t[cnt];
      MPI_Request_free(requests+0); MPI_Request_free(requests+1);
    } else if (procno==tgt) {
      for (int n=0; n<NEXPERIMENTS; n++) {
MPI_Recv(recv,s,MPI_DOUBLE,src,0,comm,MPI_STATUS_IGNORE);
```

```
MPI_Send(recv,s,MPI_DOUBLE,src,0,comm);
      }
    }
  }
  if (procno==src) {
    for (int cnt=0,s=1; cnt<NSIZES; s*=10,cnt++) {
      t[cnt] /= NEXPERIMENTS;
      printf("Time for persistent pingpong of size %d: %e\n",s,t[cnt]);
    }
  }

  MPI_Finalize();
  return 0;
}
```

5.6.3 Listing of code examples/mpi/p/persist.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

nexperiments = 10
nsizes = 6
times = np.empty(nsizes,dtype=np.float64)
src = 0; tgt = nprocs-1

#
# ordinary communication
#
size = 1
requests = [ None ] * 2
if procid==src:
    print("Ordinary send/recv")
for isize in range(nsizes):
    sendbuf = np.ones(size,dtype=np.int)
    recvbuf = np.ones(size,dtype=np.int)
    if procid==src:
        print("Size:",size)
        times[isize] = MPI.Wtime()
        for n in range(nexperiments):
            requests[0] = comm.Isend(sendbuf[0:size],dest=tgt)
            requests[1] = comm.Irecv(recvbuf[0:size],source=tgt)
            MPI.Request.Waitall(requests)
            sendbuf[0] = sendbuf[0]+1
        times[isize] = MPI.Wtime()-times[isize]
    elif procid==tgt:
```

```
        for n in range(nexperiments):
            comm.Recv(recvbuf[0:size],source=src)
            comm.Send(recvbuf[0:size],dest=src)
    size *= 10
if procid==src:
    print("Timings:",times)

#
# ordinary communication
#
size = 1
requests = [ None ] * 2
if procid==src:
    print("Persistent send/recv")
for isize in range(nsizes):
    sendbuf = np.ones(size,dtype=np.int)
    recvbuf = np.ones(size,dtype=np.int)
    if procid==src:
        print("Size:",size)
        requests[0] = comm.Send_init(sendbuf[0:size],dest=tgt)
        requests[1] = comm.Recv_init(recvbuf[0:size],source=tgt)
        times[isize] = MPI.Wtime()
        for n in range(nexperiments):
            MPI.Prequest.Startall(requests)
            MPI.Prequest.Waitall(requests)
            sendbuf[0] = sendbuf[0]+1
        times[isize] = MPI.Wtime()-times[isize]
    elif procid==tgt:
        for n in range(nexperiments):
            comm.Recv(recvbuf[0:size],source=src)
            comm.Send(recvbuf[0:size],dest=src)
    size *= 10
if procid==src:
    print("Timings:",times)
```

5.6.4 Listing of code examples/mpi/c/ssendblock.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {
  int other, size, *recvbuf, *sendbuf;
  MPI_Status status;

#include "globalinit.c"

  if (procno>1) goto skip;
  other = 1-procno;
  sendbuf = (int*) malloc(sizeof(int));
  recvbuf = (int*) malloc(sizeof(int));
```

```
  size = 1;
  MPI_Ssend(sendbuf,size,MPI_INT,other,0,comm);
  MPI_Recv(recvbuf,size,MPI_INT,other,0,comm,&status);
  printf("This statement is not reached\n");
  free(sendbuf); free(recvbuf);

 skip:
  MPI_Finalize();
  return 0;
}
```

Chapter 6

MPI topic: Data types

In the examples you have seen so far, every time data was sent, it was as a contiguous buffer with elements of a single type. In practice you may want to send heterogeneous data, or noncontiguous data.

- Communicating the real parts of an array of complex numbers means specifying every other number.
- Communicating a C structure or Fortran type with more than one type of element is not equivalent to sending an array of elements of a single type.

The datatypes you have dealt with so far are known as *elementary datatypes*; irregular objects are known as *derived datatypes*.

6.1 Data type handling

Datatypes such as `MPI_INT` are values of the type `MPI_Datatype`. This type is handled differently in different languages.

Fortran note. In Fortran before 2008, datatypes variables are stored in `Integer` variables. With the 2008 standard, datatypes are Fortran derived types:

```
Type(Datatype) :: mytype
```

Python note. There is a class

```
mpi4py.MPI.Datatype
```

with predefined values such as

```
mpi4py.MPI.Datatype.DOUBLE
```

which are themselves objects with methods for creating derived types; see section 6.3.1.

MPL note 37.

```
// sendlong.cxx
mpl::contiguous_layout<long long> v_layout(v.size());
  comm.send(v.data(), v_layout, 1); // send to rank 1
```

Also works with complex of float and double.

End of MPL note

MPL note 38. MPL routines are templated over the data type. The data types, where MPL can infer their internal representation, are enumeration types, C arrays of constant size and the template classes `std::array`, `std::pair` and `std::tuple` of the C++ Standard Template Library. The only limitation is, that the C array and the mentioned template classes hold data elements of types that can be sent or received by MPL.

End of MPL note

6.2 Elementary data types

MPI has a number of elementary data types, corresponding to the simple data types of programming languages. The names are made to resemble the types of C and Fortran, for instance `MPI_FLOAT` and `MPI_DOUBLE` in C, versus `MPI_REAL` and `MPI_DOUBLE_PRECISION` in Fortran.

6.2.1 C/C++

Here we illustrate the correspondence between a type used to declare a variable, and how this type appears in MPI communication routines:

```
long int i;
MPI_Send(&i,1,MPI_LONG_INT,target,tag,comm);
```

C type	MPI type
`char`	`MPI_CHAR`
`unsigned char`	`MPI_UNSIGNED_CHAR`
`char`	`MPI_SIGNED_CHAR`
`short`	`MPI_SHORT`
`unsigned short`	`MPI_UNSIGNED_SHORT`
`int`	`MPI_INT`
`unsigned int`	`MPI_UNSIGNED`
`long int`	`MPI_LONG`
`unsigned long int`	`MPI_UNSIGNED_LONG`
`long long int`	`MPI_LONG_LONG_INT`
`float`	`MPI_FLOAT`
`double`	`MPI_DOUBLE`
`long double`	`MPI_LONG_DOUBLE`
`unsigned char`	`MPI_BYTE`
(does not correspond to a C type)	`MPI_PACKED`
`MPI_Aint`	`MPI_AINT`

There is some, but not complete, support for *C99* types.

See section 6.2.4 for `MPI_Aint` and more about byte counting.

6.2.2 Fortran

MPI_CHARACTER	Character(Len=1)
MPI_INTEGER	
MPI_INTEGER1	
MPI_INTEGER2	
MPI_INTEGER4	
MPI_INTEGER8	(common compiler extension; not standard)
MPI_INTEGER16	
MPI_REAL	
MPI_DOUBLE_PRECISION	
MPI_REAL2	
MPI_REAL4	
MPI_REAL8	
MPI_COMPLEX	
MPI_DOUBLE_COMPLEX	Complex(Kind=Kind(0.d0))
MPI_LOGICAL	
MPI_PACKED	

Not all these types need be supported, for instance `MPI_INTEGER16` may not exist, in which case it will be equivalent to `MPI_DATATYPE_NULL`.

The default integer type `MPI_INTEGER` is equivalent to `INTEGER(KIND=MPI_INTEGER_KIND)`.

The C type `MPI_Count` corresponds to an integer of type `MPI_COUNT_KIND`, used most prominently in 'big data' routines such as `MPI_Type_size_x` (section 6.5):

```
Integer(kind=MPI_COUNT_KIND) :: count
call MPI_Type_size_x(my_type, count)
```

Kind `MPI_ADDRESS_KIND` is used for `MPI_Aint` quantities, used in Remote Memory Access (RMA) windows; see section 9.3.1.

The `MPI_OFFSET_KIND` is used to define `MPI_Offset` quantities, used in file I/O; section 10.2.2.

6.2.2.1 *Fortran90 kind-defined types*

If your Fortran code uses `KIND` to define scalar types with specified precision, these do not in general correspond to any predefined MPI datatypes. Hence the following routines exist to make *MPI equivalences of Fortran scalar types*: `MPI_Type_create_f90_integer` (figure 6.1) `MPI_Type_create_f90_real` (figure 6.2) `MPI_Type_create_f90_complex` (figure 6.3).

Examples:

```
INTEGER ( KIND = SELECTED_INTEGER_KIND(15) ) , &
 DIMENSION(100) :: array INTEGER :: root , integertype , error

CALL MPI_Type_create_f90_integer( 15 , integertype , error )
CALL MPI_Bcast ( array , 100 , &
 integertype , root , MPI_COMM_WORLD , error )
```

Figure 6.1 `MPI_Type_create_f90_integer`

Name	Param name	C type	F type	inout
`mpi_type_create_f90_integer (`				
	`r`	`int`	`INTEGER`	`in`
	decimal exponent range, i.e., number of decimal digits			
	`newtype`	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`out`
	the requested MPI datatype			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Create_f90_integer(`		
	`cls`	
	`r`	`int`
`) -> Datatype`		

Figure 6.2 `MPI_Type_create_f90_real`

Name	Param name	C type	F type	inout
`mpi_type_create_f90_real (`				
	`p`	`int`	`INTEGER`	`in`
	precision, in decimal digits			
	`r`	`int`	`INTEGER`	`in`
	decimal exponent range			
	`newtype`	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`out`
	the requested MPI datatype			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Create_f90_real(`		
	`cls`	
	`p`	`int`
	`r`	`int`
`) -> Datatype`		

```
REAL ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &
 DIMENSION(100) :: array
CALL MPI_Type_create_f90_real( 15 , 300 , realtype , error )

COMPLEX ( KIND = SELECTED_REAL_KIND(15 ,300) ) , &
 DIMENSION(100) :: array
CALL MPI_Type_create_f90_complex( 15 , 300 , complextype , error )
```

6.2.3 Python

In python, all buffer data comes from *Numpy*.

Figure 6.3 `MPI_Type_create_f90_complex`

Name	Param name	C type	F type	inout
`mpi_type_create_f90_complex (`				
	`p` *precision, in decimal digits*	`int`	`INTEGER`	`in`
	`r` *decimal exponent range*	`int`	`INTEGER`	`in`
	`newtype` *the requested MPI datatype*	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Create_f90_complex(`		
	`cls`	
	`p`	`int`
	`r`	`int`
`) -> Datatype`		

mpi4py type	NumPy type
`MPI.INT`	`np.intc`
`MPI.LONG`	`np.int`
`MPI.FLOAT`	`np.float32`
`MPI.DOUBLE`	`np.float64`

Note that numpy native integers are C longs, and therefore 8 bytes. This is no problem if you send and receive contiguous buffers of integers, but it may trip you up in cases where the actual size of the integer matters, such as in derived types, or window definitions.

Examples:

```
## inttype.py
sizeofint = np.dtype('int').itemsize
print("Size of numpy int: {}".format(sizeofint))
sizeofint = np.dtype('intc').itemsize
print("Size of C int: {}".format(sizeofint))
```

For the full source of this example, see section 6.8.2

```
## allgatherv.py
mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)
```

For the full source of this example, see section 6.8.3

6.2.4 Byte addressing type

So far we have mostly been taking about datatypes in the context of sending them. The `MPI_Aint` type is not so much for sending, as it is for describing the size of objects, such as the size of an `MPI_Win` object; section 9.1.

Figure 6.4 `MPI_Sizeof`

Name	Param name	C type	F type	inout
`mpi_sizeof (`				
	`x`	`void*`	`TYPE(*), DIMENSION(..)`	`in`
	a Fortran variable of numeric intrinsic type			
	`size`	`int*`	`INTEGER`	`out`
	size of machine representation of that type			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Addresses have type `MPI_Aint` The start of the address range is given in `MPI_BOTTOM`.

Variables of type `MPI_Aint` can be sent as `MPI_AINT`:

```
MPI_Aint address;
MPI_Send( address,1,MPI_AINT, ... );
```

See section 9.5.2 for an example.

In order to prevent overflow errors in byte calculations there are support routines `MPI_Aint_add`

```
MPI_Aint MPI_Aint_add(MPI_Aint base, MPI_Aint disp)
```

and similarly `MPI_Aint_diff`.

See also the `MPI_Sizeof` (section 6.2.6) and `MPI_Get_address` routines.

6.2.4.1 *Fortran*

The equivalent of `MPI_Aint` in Fortran is an integer of kind `MPI_ADDRESS_KIND`:

```
integer(kind=MPI_ADDRESS_KIND) :: winsize
```

Fortran lacks a `sizeof` operator to query the sizes of datatypes. Since sometimes exact byte counts are necessary, for instance in one-sided communication, Fortran can use the (deprecated) `MPI_Sizeof` (figure 6.4) routine. See section 6.2.6 for details.

Example usage in `MPI_Win_create`:

```
call MPI_Sizeof(windowdata,window_element_size,ierr)
window_size = window_element_size*500
call MPI_Win_create( windowdata,window_size,window_element_size,... )
```

6.2.4.2 *Python*

Here is a good way for finding the size of *numpy* datatypes in bytes:

```
## putfence.py
intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=np.int)
win = MPI.Win.Create(window_data,intsize,comm=comm)
```

Figure 6.5 `MPI_Type_match_size`

Name	Param name	C type	F type	inout
`mpi_type_match_size (`				
	`typeclass`	`int`	`INTEGER`	`in`
	generic type specifier			
	`size`	`int`	`INTEGER`	`in`
	size, in bytes, of representation			
	`datatype`	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`out`
	datatype with correct type, size			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Match_size(`		
	`cls`	
	`typeclass`	`int`
	`size`	`int`
`) -> Datatype`		

6.2.5 Translating language type to MPI type

In some circumstances you may want to find the MPI type that corresponds to a type in your programming language.

- In C++ functions and classes can be templated, meaning that the type is not fully known:

```
template<typename T> {
class something<T> {
public:
  void dosend(T input) {
    MPI_Send( &input,1,/* ????? */ );
  };
};
```

  (Note that in MPL this is hardly ever needed because MPI calls are templated there.)
- Petsc installations use a generic identifier `PetscScalar` (or `PetscReal`) with a configuration-dependent realization.

6.2.6 Matching MPI and language type sizes

The size of a datatype is not always statically known, for instance if the Fortran `KIND` keyword is used. The translation of datatypes in the source language can be translated to MPI types with `MPI_Type_match_size` (figure 6.5) where the *typeclass* argument is one of `MPI_TYPECLASS_REAL`, `MPI_TYPECLASS_INTEGER`, `MPI_TYPECLASS_COMPLEX` .

```
// typematch.c
float x5;
double x10;
int s5,s10;
MPI_Datatype mpi_x5,mpi_x10;
```

Figure 6.6 `MPI_Type_size`

Name	Param name	C type	F type	inout
`mpi_type_size (`				
	`datatype` *datatype*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`size` *datatype size*	`int*`	`INTEGER`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

```
MPI_Type_match_size(MPI_TYPECLASS_REAL,sizeof(x5),&mpi_x5);
MPI_Type_match_size(MPI_TYPECLASS_REAL,sizeof(x10),&mpi_x10);
MPI_Type_size(mpi_x5,&s5);
MPI_Type_size(mpi_x10,&s10);
```

For the full source of this example, see section 6.8.4

In Fortran, the size of the datatype in the language can be obtained with `MPI_Sizeof` (note the nonoptional error parameter!). This routine is deprecated in MPI-4: use of `storage_size` and/or `c_sizeof` is recommended.

```
// matchkind.F90
call MPI_Sizeof(x10,s10,ierr)
call MPI_Type_match_size(MPI_TYPECLASS_REAL,s10,mpi_x10)
call MPI_Type_size(mpi_x10,s10)
print *,"10 positions supported, MPI type size is",s10
```

For the full source of this example, see section 6.8.4

The space that MPI takes for a structure type can be queried in a variety of ways. First of all `MPI_Type_size` (figure 6.6) counts the *datatype size* as the number of bytes occupied by the data in a type. That means that in an *MPI vector datatype* it does not count the gaps.

```
// typesize.c
MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
MPI_Type_commit(&newtype);
MPI_Type_size(newtype,&size);
ASSERT( size==(count*bs)*sizeof(double) );
```

For the full source of this example, see section 6.8.5

Petsc has its own translation mechanism; see section 29.2.

6.3 Derived datatypes

MPI allows you to create your own data types, somewhat (but not completely...) analogous to defining structures in a programming language. MPI data types are mostly of use if you want to send multiple items in one message.

There are two problems with using only elementary datatypes as you have seen so far.

- MPI communication routines can only send multiples of a single data type: it is not possible to send items of different types, even if they are contiguous in memory. It would be possible to use the `MPI_BYTE` data type, but this is not advisable.
- It is also ordinarily not possible to send items of one type if they are not contiguous in memory. You could of course send a contiguous memory area that contains the items you want to send, but that is wasteful of bandwidth.

With MPI data types you can solve these problems in several ways.

- You can create a new *contiguous data type* consisting of an array of elements of another data type. There is no essential difference between sending one element of such a type and multiple elements of the component type.
- You can create a *vector data type* consisting of regularly spaced blocks of elements of a component type. This is a first solution to the problem of sending noncontiguous data.
- For not regularly spaced data, there is the *indexed data type*, where you specify an array of index locations for blocks of elements of a component type. The blocks can each be of a different size.
- The *struct data type* can accomodate multiple data types.

And you can combine these mechanisms to get irregularly spaced heterogeneous data, et cetera.

6.3.1 Basic calls

The typical sequence of calls for creating a new datatype is as follows:

- You need a variable for the datatype;
- There is a create call, followed by a 'commit' call where MPI performs internal bookkeeping and optimizations;
- The datatype is used, possibly multiple times;
- When the datatype is no longer needed, it must be freed to prevent memory leaks.

In code:

```
MPI_Datatype newtype;
MPI_Type_something( < oldtype specifications >, &newtype );
MPI_Type_commit( &newtype );
/* code that uses your new type */
MPI_Type_free( &newtype );
```

In Fortran2008:

```
Type(MPI_Datatype) :: newvectortype
call MPI_Type_something( <oldtype specification>, &
          newvectortype)
call MPI_Type_commit(newvectortype)
!! code that uses your type
call MPI_Type_free(newvectortype)
```

Python note. The various type creation routines are methods of the datatype classes, after which commit and free are methods on the new type.

```
## vector.py
source = np.empty(stride*count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newvectortype = MPI.DOUBLE.Create_vector(count,1,stride)
    newvectortype.Commit()
    comm.Send([source,1,newvectortype],dest=the_other)
    newvectortype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

For the full source of this example, see section 6.8.6

MPL note 39. In MPL type creation routines are in the main namespace, templated over the datatypes.

```
mpl::contiguous_layout<double> target_layout(count),
mpl::status recv_status =
  comm_world.recv(target.data(),target_layout, the_other);
recv_count = recv_status.get_count<double>();
```

For the full source of this example, see section 6.8.7

The commit call is part of the type creation, and freeing is done in the destructor.

End of MPL note

6.3.1.1 Datatype objects

MPI derived data types are stored in variables of type `MPI_Datatype`; section 6.3.1.1.

6.3.1.2 Create calls

The `MPI_Datatype` varriable gets its value by a call to one of the following routines:

- `MPI_Type_contiguous` for contiguous blocks of data; section 6.3.2;
- `MPI_Type_vector` for regularly strided data; section 6.3.3;
- `MPI_Type_create_subarray` for subsets out higher dimensional block; section 6.3.4;
- `MPI_Type_create_struct` for heterogeneous irregular data; section 6.3.6;
- `MPI_Type_indexed` and `MPI_Type_hindexed` for irregularly strided data; section 6.3.5.

These calls take an existing type, whether elementary or also derived, and produce a new type.

6.3.1.3 Commit and free

It is necessary to call `MPI_Type_commit` (figure 6.7) on a new data type, which makes MPI do the indexing calculations for the data type.

When you no longer need the data type, you call `MPI_Type_free` (figure 6.8). (This is typically not needed in OO APIs.) This has the following effects:

- The definition of the datatype identifier will be changed to `MPI_DATATYPE_NULL`.
- Any communication using this data type, that was already started, will be completed succesfully.
- Datatypes that are defined in terms of this data type will still be usable.

Figure 6.7 `MPI_Type_commit`

Name	Param name	C type	F type	inout
`mpi_type_commit (`				
	`datatype`	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`inout`
	datatype that is committed			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Commit(`		
	`self`	
`) -> Datatype`		

Figure 6.8 `MPI_Type_free`

Name	Param name	C type	F type	inout
`mpi_type_free (`				
	`datatype`	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`inout`
	datatype that is freed			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Free(`		
	`self`	
`) -> None`		

6.3.2 Contiguous type

The simplest derived type is the 'contiguous' type, constructed with `MPI_Type_contiguous` (figure 6.9).

A contigous type describes an array of items of an elementary or earlier defined type. There is no difference between sending one item of a contiguous type and multiple items of the constituent type. This is illustrated in figure 6.1.

```
// contiguous.c
MPI_Datatype newvectortype;
if (procno==sender) {
  MPI_Type_contiguous(count,MPI_DOUBLE,&newvectortype);
  MPI_Type_commit(&newvectortype);
  MPI_Send(source,1,newvectortype,receiver,0,comm);
  MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,count,MPI_DOUBLE,sender,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
  ASSERT(count==recv_count);
}
```

Figure 6.9 `MPI_Type_contiguous`

Name	Param name	C type	F type	inout
`mpi_type_contiguous (`				
	`count` *replication count*	`int`	`INTEGER`	`in`
	`oldtype` *old datatype*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`newtype` *new datatype*	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

```
Create_contiguous(self, int count)
```

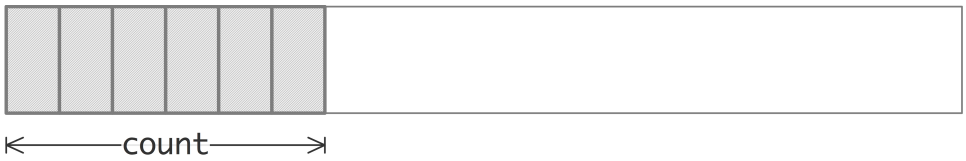


Figure 6.1: A contiguous datatype is built up out of elements of a constituent type

For the full source of this example, see section 6.8.8

```
// contiguous.F90
integer :: newvectortype
if (mytid==sender) then
   call MPI_Type_contiguous(count,MPI_DOUBLE_PRECISION,newvectortype,err)
   call MPI_Type_commit(newvectortype,err)
   call MPI_Send(source,1,newvectortype,receiver,0,comm,err)
   call MPI_Type_free(newvectortype,err)
else if (mytid==receiver) then
   call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
        recv_status,err)
   call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count,err)
   !ASSERT(count==recv_count);
end if
```

For the full source of this example, see section 6.8.9

```
## contiguous.py
source = np.empty(count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newcontiguoustype = MPI.DOUBLE.Create_contiguous(count)
    newcontiguoustype.Commit()
    comm.Send([source,1,newcontiguoustype],dest=the_other)
    newcontiguoustype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

For the full source of this example, see section 6.8.10

Figure 6.10 `MPI_Type_vector`

Name	Param name	C type	F type	inout
`mpi_type_vector (`				
	`count` *number of blocks*	`int`	`INTEGER`	`in`
	`blocklength` *number of elements in each block*	`int`	`INTEGER`	`in`
	`stride` *number of elements between start of each block*	`int`	`INTEGER`	`in`
	`oldtype` *old datatype*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`newtype` *new datatype*	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

```
MPI.Datatype.Create_vector(self, int count, int blocklength, int stride)
```

MPL note 40. The MPL interface makes extensive use of `contiguous_layout`, as it is the main way to declare a nonscalar buffer; see note 11.

End of MPL note

MPL note 41. Contiguous layouts can only use elementary types or other contiguous layouts as their 'old' type. To make a contiguous type for other layouts, use `vector_layout`:

```
// contiguous.cxx
mpl::contiguous_layout<int> type1(7);
mpl::vector_layout<int> type2(8,type1);
```

(Contrast this with `strided_vector_layout`; note 42.)

End of MPL note

6.3.3 Vector type

The simplest noncontiguous datatype is the 'vector' type, constructed with `MPI_Type_vector` (figure 6.10).

A vector type describes a series of blocks, all of equal size, spaced with a constant stride. This is illustrated in figure 6.2.

The vector datatype gives the first nontrivial illustration that datatypes can be *different on the sender and receiver*. If the sender sends `b` blocks of length `l` each, the receiver can receive them as `bl` contiguous elements, either as a contiguous datatype, or as a contiguous buffer of an elementary type; see figure 6.3. In this case, the receiver has no knowledge of the stride of the datatype on the sender.

In this example a vector type is created only on the sender, in order to send a strided subset of an array; the receiver receives the data as a contiguous block.

```
// vector.c
source = (double*) malloc(stride*count*sizeof(double));
target = (double*) malloc(count*sizeof(double));
```

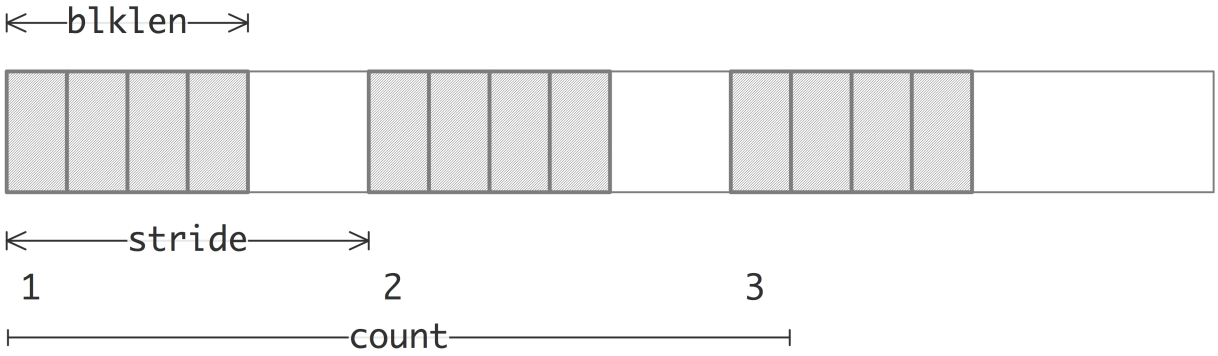


Figure 6.2: A vector datatype is built up out of strided blocks of elements of a constituent type

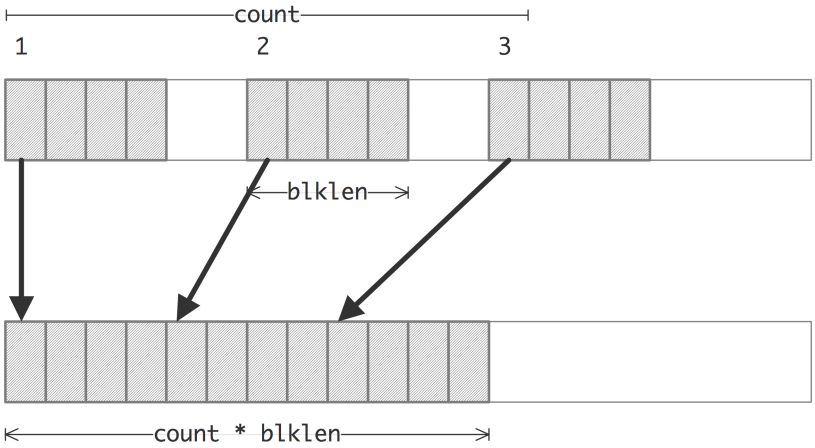


Figure 6.3: Sending a vector datatype and receiving it as elementary or contiguous

```
MPI_Datatype newvectortype;
if (procno==sender) {
  MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
  MPI_Type_commit(&newvectortype);
  MPI_Send(source,1,newvectortype,the_other,0,comm);
  MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
  ASSERT(recv_count==count);
}
```

For the full source of this example, see section 6.8.11

We illustrate Fortran2008:

```
// vector.F90
Type(MPI_Datatype) :: newvectortype
if (mytid==sender) then
   call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
        newvectortype)
   call MPI_Type_commit(newvectortype)
   call MPI_Send(source,1,newvectortype,receiver,0,comm)
   call MPI_Type_free(newvectortype)
   if ( .not. newvectortype==MPI_DATATYPE_NULL) then
      print *,"Trouble freeing datatype"
   else
      print *,"Datatype successfully freed"
   end if
else if (mytid==receiver) then
   call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
        recv_status)
   call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
end if
```

For the full source of this example, see section 6.8.12

In legacy mode, Fortran code stays the same except that the type is declared as `Integer`:

```
// vector.F90
integer :: newvectortype
integer :: recv_status(MPI_STATUS_SIZE),recv_count
   call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
        newvectortype,err)
   call MPI_Type_commit(newvectortype,err)
```

For the full source of this example, see section 6.8.13

Python note. The vector creation routine is a method of the `MPI.Datatype` class. For the general discussion, see section 6.3.1.

```
## vector.py
source = np.empty(stride*count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newvectortype = MPI.DOUBLE.Create_vector(count,1,stride)
    newvectortype.Commit()
    comm.Send([source,1,newvectortype],dest=the_other)
    newvectortype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

For the full source of this example, see section 6.8.6

MPL note 42. MPL has the `strided_vector_layout` class as equivalent of the vector type:

```
// vector.cxx
vector<double>
  source(stride*count);
if (procno==sender) {
  mpl::strided_vector_layout<double>
    newvectortype(count,1,stride);
```

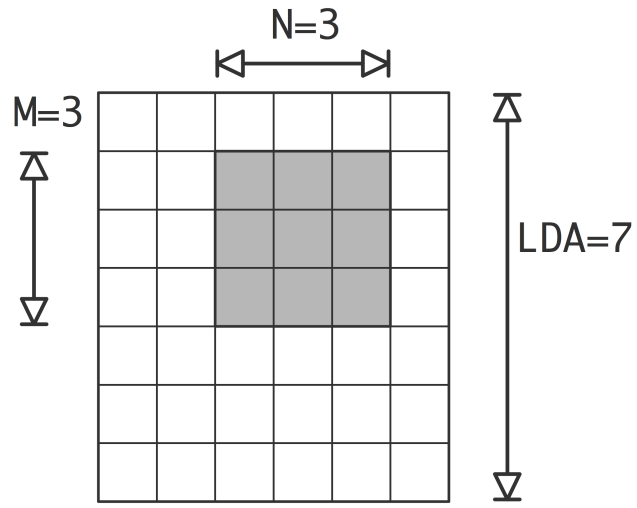


Figure 6.4: Memory layout of a row and column of a matrix in column-major storage

```
  comm_world.send
    (source.data(),newvectortype,the_other);
}
```

For the full source of this example, see section 6.8.7

(See note 41 for nonstrided vectors.)

End of MPL note

MPL note 43. It is possible to to send containers by iterators

```
// sendrange.cxx
vector<double> v(15);
  comm_world.send(v.begin(), v.end(), 1);  // send to rank 1
  comm_world.recv(v.begin(), v.end(), 0);  // receive from rank 0
```

For the full source of this example, see section 6.8.14

End of MPL note

MPL note 44. Noncontiguous iteratable objects can be send with a `iterator_layout`:

```
std::list<int> v(20, 0);
mpl::iterator_layout<int> l(v.begin(), v.end());
comm_world.recv(&(*v.begin()), l, 0);
```

End of MPL note

Figure 6.4 indicates one source of irregular data: with a matrix on *column-major storage*, a column is stored in contiguous memory. However, a row of such a matrix is not contiguous; its elements being separated by a *stride* equal to the column length.

Exercise 6.1. How would you describe the memory layout of a submatrix, if the whole matrix has size $M \times N$ and the submatrix $m \times n$?

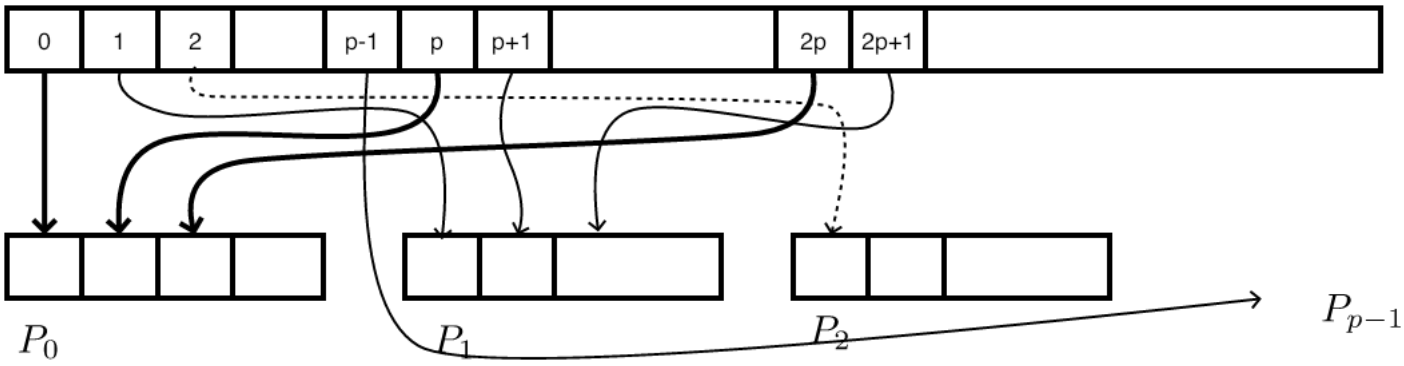


Figure 6.5: Send strided data from process zero to all others

As an example of this datatype, consider the example of transposing a matrix, for instance to convert between C and Fortran arrays (see section HPSC-28.3). Suppose that a processor has a matrix stored in C, row-major, layout, and it needs to send a column to another processor. If the matrix is declared as

```
int M,N; double mat[M][N]
```

then a column has M blocks of one element, spaced N locations apart. In other words:

```
MPI_Datatype MPI_column;
MPI_Type_vector(
    /* count= */ M, /* blocklength= */ 1, /* stride= */ N,
    MPI_DOUBLE, &MPI_column );
```

Sending the first column is easy:

```
MPI_Send( mat, 1,MPI_column, ... );
```

The second column is just a little trickier: you now need to pick out elements with the same stride, but starting at `A[0][1]`.

```
MPI_Send( &(mat[0][1]), 1,MPI_column, ... );
```

You can make this marginally more efficient (and harder to read) by replacing the index expression by `mat+1`.

Exercise 6.2. Suppose you have a matrix of size $4N \times 4N$, and you want to send the elements `A[4*i][4*j]` with $i, j = 0, \ldots, N-1$. How would you send these elements with a single transfer?

Exercise 6.3. Allocate a matrix on processor zero, using Fortran column-major storage. Using P sendrecv calls, distribute the rows of this matrix among the processors.

Exercise 6.4. Let processor 0 have an array x of length $10P$, where P is the number of processors. Elements $0, P, 2P, \ldots, 9P$ should go to processor zero, $1, P+1, 2P+1, \ldots$ to processor 1, et cetera. Code this as a sequence of send/recv calls, using a vector datatype for the send, and a contiguous buffer for the receive.

Figure 6.11 `MPI_Type_create_subarray`

Name	Param name	C type	F type	inout
`mpi_type_create_subarray (`				
	`ndims`	`int`	`INTEGER`	`in`
	number of array dimensions			
	`array_of_sizes`	`const int[]`	`INTEGER`	`in`
	`length: ndims`			
	number of elements of type oldtype in each dimension of the full array			
	`array_of_subsizes`	`const int[]`	`INTEGER`	`in`
	`length: ndims`			
	number of elements of type oldtype in each dimension of the subarray			
	`array_of_starts`	`const int[]`	`INTEGER`	`in`
	`length: ndims`			
	starting coordinates of the subarray in each dimension			
	`order`	`int`	`INTEGER`	`in`
	array storage order flag			
	`oldtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	old datatype			
	`newtype`	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`out`
	new datatype			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Create_subarray(`		
	`self`	
	`sizes`	`Sequence[int]`
	`subsizes`	`Sequence[int]`
	`starts`	`Sequence[int]`
	`order`	`int = Ellipsis`
`) -> Datatype`		

For simplicity, skip the send to/from zero. What is the most elegant solution if you want to include that case?
For testing, define the array as $x[i] = i$.
(There is a skeleton for this exercise under the name `stridesend`.)

Exercise 6.5. Write code to compare the time it takes to send a strided subset from an array: copy the elements by hand to a smaller buffer, or use a vector data type. What do you find? You may need to test on fairly large arrays.

6.3.4 Subarray type

The vector datatype can be used for blocks in an array of dimension more than 2 by using it recursively. However, this gets tedious. Instead, there is an explicit subarray type `MPI_Type_create_subarray` (figure 6.11) This describes the dimensionality and extent of the array, and the starting point (the 'upper left corner') and extent of the subarray.

MPL note 45. The templated `subarray_layout` class is constructed from a vector of triplets of global size / subblock size / first coordinate.

```
mpl::subarray_layout<int>(
 { {ny, ny_1, ny_0}, {nx, nx_1, nx_0} }
);
```

End of MPL note

Exercise 6.6.

(There is a skeleton for this exercise under the name `cubegather`.)
Assume that your number of processors is $P = Q^3$, and that each process has an array of identical size. Use `MPI_Type_create_subarray` to gather all data onto a root process. Use a sequence of send and receive calls; `MPI_Gather` does not work here.

Subarrays are naturally supported in Fortran through array sections:

```
// section.F90
integer,parameter :: siz=20
real,dimension(siz,siz) :: matrix = [ ((j+(i-1)*siz,i=1,siz),j=1,siz) ]
real,dimension(2,2) :: submatrix
if (procno==0) then
   call MPI_Send(matrix(1:2,1:2),4,MPI_REAL,1,0,comm)
else if (procno==1) then
   call MPI_Recv(submatrix,4,MPI_REAL,0,0,comm,MPI_STATUS_IGNORE)
   if (submatrix(2,2)==22) then
      print *,"Yay"
   else
      print *,"nay...."
   end if
end if
```

For the full source of this example, see section 6.8.15

at least, since MPI-3.

The possibilities for the `order` parameter are `MPI_ORDER_C` and `MPI_ORDER_FORTRAN`. However, this has nothing to do with the order of traversal of elements; it determines how the bounds of the subarray are interpreted. As an example, we fill a 4×4 array in C order with the numbers $0\cdots15$, and send the $[0,1]\times[0\cdots4]$ slice two ways, first C order, then Fortran order:

```
// row2col.c
#define SIZE 4
  int
    sizes[2], subsizes[2], starts[2];
  sizes[0] = SIZE; sizes[1] = SIZE;
  subsizes[0] = SIZE/2;  subsizes[1] = SIZE;
  starts[0] = starts[1] = 0;
    MPI_Type_create_subarray
      (2,sizes,subsizes,starts,
       MPI_ORDER_C,MPI_DOUBLE,&rowtype);
    MPI_Type_create_subarray
      (2,sizes,subsizes,starts,
       MPI_ORDER_FORTRAN,MPI_DOUBLE,&coltype);
```

For the full source of this example, see section 6.8.16

The receiver receives the following, formatted to bring out where the numbers originate:

Figure 6.12 `MPI_Type_indexed`

Name	Param name	C type	F type	inout
`mpi_type_indexed (`				
	`count` *number of blocks – also number of entries in array_of_displacements and array_of_blocklengths*	`int`	`INTEGER`	`in`

	`array_of_blocklengths`
	`length: count`
	number of elements per block
		`const int[]`	`INTEGER`	`in`

	`array_of_displacements`
	`length: count`
	displacement for each block, in multiples of oldtype
		`const int[]`	`INTEGER`	`in`

	`oldtype` *old datatype*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`newtype` *new datatype*	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

```
MPI.Datatype.Create_vector(self, blocklengths,displacements )
```

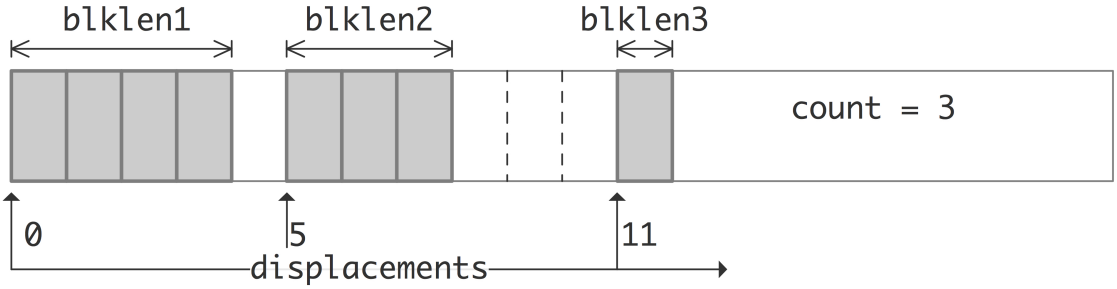


Figure 6.6: The elements of an MPI Indexed datatype

```
Received C order:
 0.000 1.000 2.000 3.000
 4.000 5.000 6.000 7.000
Received F order:
 0.000 1.000
 4.000 5.000
 8.000 9.000
 12.000 13.000
```

6.3.5 Indexed type

The indexed datatype, constructed with `MPI_Type_indexed` (figure 6.12) can send arbitrarily located elements from an array of a single datatype. You need to supply an array of index locations, plus an array of blocklengths with a separate blocklength for each index. The total number of elements sent is the sum of the blocklengths.

The following example picks items that are on prime number-indexed locations.

```
// indexed.c
displacements = (int*) malloc(count*sizeof(int));
blocklengths = (int*) malloc(count*sizeof(int));
source = (int*) malloc(totalcount*sizeof(int));
target = (int*) malloc(targetbuffersize*sizeof(int));
MPI_Datatype newvectortype;
if (procno==sender) {
  MPI_Type_indexed(count,blocklengths,displacements,MPI_INT,&newvectortype);
  MPI_Type_commit(&newvectortype);
  MPI_Send(source,1,newvectortype,the_other,0,comm);
  MPI_Type_free(&newvectortype);
} else if (procno==receiver) {
  MPI_Status recv_status;
  int recv_count;
  MPI_Recv(target,targetbuffersize,MPI_INT,the_other,0,comm,
    &recv_status);
  MPI_Get_count(&recv_status,MPI_INT,&recv_count);
  ASSERT(recv_count==count);
}
```

For the full source of this example, see section 6.8.17

For Fortran we show the legacy syntax for once:

```
// indexed.F90
 integer :: newvectortype;
 ALLOCATE(indices(count))
 ALLOCATE(blocklengths(count))
 ALLOCATE(source(totalcount))
 ALLOCATE(targt(count))
 if (mytid==sender) then
    call MPI_Type_indexed(count,blocklengths,indices,MPI_INT,&
         newvectortype,err)
    call MPI_Type_commit(newvectortype,err)
    call MPI_Send(source,1,newvectortype,receiver,0,comm,err)
    call MPI_Type_free(newvectortype,err)
 else if (mytid==receiver) then
   call MPI_Recv(targt,count,MPI_INT,sender,0,comm,&
        recv_status,err)
   call MPI_Get_count(recv_status,MPI_INT,recv_count,err)
   !    ASSERT(recv_count==count);
end if
```

For the full source of this example, see section 6.8.18

```
## indexed.py
displacements = np.empty(count,dtype=np.int)
blocklengths = np.empty(count,dtype=np.int)
source = np.empty(totalcount,dtype=np.float64)
target = np.empty(count,dtype=np.float64)
if procid==sender:
    newindextype = MPI.DOUBLE.Create_indexed(blocklengths,displacements)
    newindextype.Commit()
    comm.Send([source,1,newindextype],dest=the_other)
```

```
    newindextype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)
```

For the full source of this example, see section 6.8.19

MPL note 46. In MPL, the `indexed_layout` is based on a vector of 2-tuples denoting block length / block location.

```
// indexed.cxx
const int count = 5;
mpl::contiguous_layout<int>
  fiveints(count);
mpl::indexed_layout<int>
  indexed_where{ { {1,2}, {1,3}, {1,5}, {1,7}, {1,11} } };

if (procno==sender) {
  comm_world.send( source_buffer.data(),indexed_where, receiver );
} else if (procno==receiver) {
  auto recv_status =
    comm_world.recv( target_buffer.data(),fiveints, sender );
  int recv_count = recv_status.get_count<int>();
  assert(recv_count==count);
}
```

For the full source of this example, see section 6.8.20

End of MPL note

MPL note 47. The size/displacement arrays for `MPI_Gatherv` / `MPI_Alltoallv` are handled through a `layouts` object, which is basically a vector of `layout` objects.

```
mpl::layouts<int> receive_layout;
for ( int iproc=0,loc=0; iproc<nprocs; iproc++ ) {
  auto siz = size_buffer.at(iproc);
  receive_layout.push_back
        ( mpl::indexed_layout<int>( {{ siz,loc }} ) );
  loc += siz;
}
```

End of MPL note

MPL note 48. For the case where all block lengths are the same, use `indexed_block_layout`:

```
// indexedblock.cxx
const int count = 5;
mpl::contiguous_layout<int>
  fiveints(count);
mpl::indexed_block_layout<int>
  indexed_where(  1, {2,3,5,7,11} );
```

For the full source of this example, see section 6.8.21

End of MPL note

You can also `MPI_Type_create_hindexed` which describes blocks of a single old type, but with index locations in bytes, rather than in multiples of the old type.

Figure 6.13 `MPI_Type_create_hindexed_block`

Name	Param name	C type	F type	inout
`mpi_type_create_hindexed_block (`				
	`count` *length of array of displacements*	`int`	`INTEGER`	`in`
	`blocklength` *size of block*	`int`	`INTEGER`	`in`
	`array_of_displacements`	`const MPI_Aint[]`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`
	`length: count` *byte displacement of each block*			
	`oldtype` *old datatype*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`newtype` *new datatype*	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Create_hindexed_block(`		
	`self`	
	`blocklength`	`int`
	`displacements`	`Sequence[int]`
`) -> Datatype`		

Figure 6.14 `MPI_Get_address`

Name	Param name	C type	F type	inout
`mpi_get_address (`				
	`location` *location in caller memory*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`address` *address of location*	`MPI_Aint*`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Get_address(`		
	`location`	`Union[Buffer, Bottom]`
`) -> int`		

```
int MPI_Type_create_hindexed
 (int count, int blocklens[], MPI_Aint indices[],
  MPI_Datatype old_type,MPI_Datatype *newtype)
```

A slightly simpler version, `MPI_Type_create_hindexed_block` (figure 6.13) assumes constant block length.

There is an important difference between the `hindexed` and the above `MPI_Type_indexed`: that one described offsets from a base location; these routines describes absolute memory addresses. You can use this to send for instance the elements of a linked list. You would traverse the list, recording the addresses of the elements with `MPI_Get_address` (figure 6.14).

Figure 6.15 `MPI_Type_create_struct`

Name	Param name	C type	F type	inout
`mpi_type_create_struct (`				
	`count` *number of blocks also number of entries in arrays array_of_types, array_of_displacements, and array_of_blocklengths*	`int`	`INTEGER`	`in`

	`array_of_blocklengths`
	`length: count`
	number of elements in each block
		`const int[]`	`INTEGER`	`in`

	`array_of_displacements`
	`length: count`
	byte displacement of each block
		`const MPI_Aint[]`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`

	`array_of_types`
	`length: count`
	types of elements in each block
		`const MPI_Datatype[]`	`TYPE(MPI_Datatype)`	`in`

	`newtype` *new datatype*	`MPI_Datatype*`	`TYPE(MPI_Datatype)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Create_struct(`		
	`cls`	
	`blocklengths`	`Sequence[int]`
	`displacements`	`Sequence[int]`
	`datatypes`	`Sequence[Datatype]`
`) -> Datatype`		

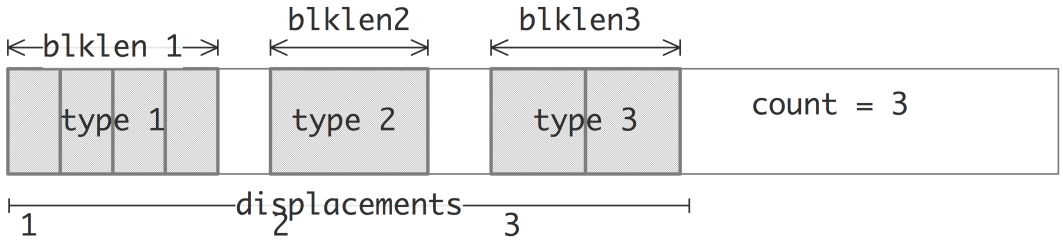


Figure 6.7: The elements of an MPI Struct datatype

In C++ you can use this to send an `std::<vector>`, that is, a vector object from the *C++ standard library*, if the component type is a pointer.

6.3.6 Struct type

The structure type, created with `MPI_Type_create_struct` (figure 6.15), can contain multiple data types. (The routine `MPI_Type_struct` is deprecated with MPI-3.) The specification contains a 'count' parameter that specifies how many blocks there are in a single structure. For instance,

```
struct {
 int i;
 float x,y;
```

```
} point;
```

has two blocks, one of a single integer, and one of two floats. This is illustrated in figure 6.7.

count The number of blocks in this datatype. The `blocklengths`, `displacements`, `types` arguments have to be at least of this length.

blocklengths array containing the lengths of the blocks of each datatype.

displacements array describing the relative location of the blocks of each datatype.

types array containing the datatypes; each block in the new type is of a single datatype; there can be multiple blocks consisting of the same type.

In this example, unlike the previous ones, both sender and receiver create the structure type. With structures it is no longer possible to send as a derived type and receive as a array of a simple type. (It would be possible to send as one structure type and receive as another, as long as they have the same *datatype signature*.)

```
// struct.c
struct object {
  char c;
  double x[2];
  int i;
};
MPI_Datatype newstructuretype;
int structlen = 3;
int blocklengths[structlen]; MPI_Datatype types[structlen];
MPI_Aint displacements[structlen];

/*
 * where are the components relative to the structure?
 */
MPI_Aint current_displacement=0;

// one character
blocklengths[0] = 1; types[0] = MPI_CHAR;
displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;

// two doubles
blocklengths[1] = 2; types[1] = MPI_DOUBLE;
displacements[1] = (size_t)&(myobject.x) - (size_t)&myobject;

// one int
blocklengths[2] = 1; types[2] = MPI_INT;
displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;

MPI_Type_create_struct(structlen,blocklengths,displacements,types,&newstructuretype);
MPI_Type_commit(&newstructuretype);
if (procno==sender) {
  MPI_Send(&myobject,1,newstructuretype,the_other,0,comm);
} else if (procno==receiver) {
  MPI_Recv(&myobject,1,newstructuretype,the_other,0,comm,MPI_STATUS_IGNORE);
}
MPI_Type_free(&newstructuretype);
```

For the full source of this example, see section 6.8.22

Note the `displacement` calculations in this example, which involve some not so elegant pointer arithmetic. The following Fortran code uses `MPI_Get_address`, which is more elegant, and in fact the only way address calculations can be done in Fortran.

```
// struct.F90
Type object
   character :: c
   real*8,dimension(2) :: x
   integer :: i
end type object
type(object) :: myobject
integer,parameter :: structlen = 3
type(MPI_Datatype) :: newstructuretype
integer,dimension(structlen) :: blocklengths
type(MPI_Datatype),dimension(structlen) :: types;
MPI_Aint,dimension(structlen) :: displacements
MPI_Aint :: base_displacement, next_displacement
if (procno==sender) then
   myobject%c = 'x'
   myobject%x(0) = 2.7; myobject%x(1) = 1.5
   myobject%i = 37

!! component 1: one character
blocklengths(1) = 1; types(1) = MPI_CHAR
call MPI_Get_address(myobject,base_displacement)
call MPI_Get_address(myobject%c,next_displacement)
displacements(1) = next_displacement-base_displacement

!! component 2: two doubles
blocklengths(2) = 2; types(2) = MPI_DOUBLE
call MPI_Get_address(myobject%x,next_displacement)
displacements(2) = next_displacement-base_displacement

!! component 3: one int
blocklengths(3) = 1; types(3) = MPI_INT
call MPI_Get_address(myobject%i,next_displacement)
displacements(3) = next_displacement-base_displacement

if (procno==sender) then
   call MPI_Send(myobject,1,newstructuretype,receiver,0,comm)
else if (procno==receiver) then
   call MPI_Recv(myobject,1,newstructuretype,sender,0,comm,MPI_STATUS_IGNORE)
end if
call MPI_Type_free(newstructuretype)
```

For the full source of this example, see section 6.8.23

It would have been incorrect to write

```
displacement[0] = 0;
displacement[1] = displacement[0] + sizeof(char);
```

since you do not know the way the *compiler* lays out the structure in memory[1].

If you want to send more than one structure, you have to worry more about padding in the structure. You can solve this by adding an extra type `MPI_UB` for the 'upper bound' on the structure:

```
displacements[3] = sizeof(myobject); types[3] = MPI_UB;
MPI_Type_create_struct(struclen+1,.....);
```

MPL note 49. One could describe the MPI struct type as a collection of displacements, to be applied to any set of items that conforms to the specifications. An MPL `heterogeneous_layout` on the other hand, incorporates the actual data. Thus you could write

```
// structscalar.cxx
char c;
double x;
int i;
if (procno==sender) {
  c = 'x'; x = 2.4; i = 37;
}
mpl::heterogeneous_layout object( c,x,i );
if (procno==sender) {
  comm_world.send( mpl::absolute,object,receiver );
} else if (procno==receiver) {
  comm_world.recv( mpl::absolute,object,sender );
}
```

For the full source of this example, see section 6.8.24

Here, the `absolute` indicates the lack of an implicit buffer: the layout is absolute rather than a relative description.

End of MPL note

MPL note 50. More complicated data than scalars takes more work:

```
// struct.cxx
char c;
vector<double> x(2);
int i;
if (procno==sender) {
  c = 'x';
  x[0] = 2.7; x[1] = 1.5;
  i = 37;
}
mpl::heterogeneous_layout object
  ( c,
    mpl::make_absolute( x.data(),mpl::vector_layout<double>(2) ),
    i );
if (procno==sender) {
  comm_world.send( mpl::absolute,object,receiver );
} else if (procno==receiver) {
  comm_world.recv( mpl::absolute,object,sender );
}
```

1. Homework question: what does the language standard say about this?

For the full source of this example, see section 6.8.24

Note the `make_absolute` in addition to `absolute` mentioned above.

End of MPL note

6.4 Type maps and type matching

With derived types, you saw that it was not necessary for the type of the sender and receiver to match. Howver, when the send buffer is constructed, and the receive buffer unpacked, it is necessary for the successive types in that buffer to match.

The types in the send and receive buffers also need to match the datatypes of the underlying architecture, with two exceptions. The `MPI_PACKED` and `MPI_BYTE` types can match any underlying type. However, this still does not mean that these types can be used on only sender or receiver, and a specific type on the other.

6.5 Type extent

See section 6.2.6 about the related issue of type sizes.

6.5.1 Extent and true extent

The *datatype extent*, measured with `MPI_Type_get_extent` (figure 6.16), is strictly the distance from the first to the last data item of the type, that is, with counting the gaps in the type. It is measured in bytes so the output parameters are of type `MPI_Aint`.

In the following example we measure the extent of a vector type. Note that the extent is not the stride times the number of blocks, because that would count a 'trailing gap'.

```
MPI_Aint lb,asize;
MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
MPI_Type_commit(&newtype);
MPI_Type_get_extent(newtype,&lb,&asize);
ASSERT( lb==0 );
ASSERT( asize==((count-1)*stride+bs)*sizeof(double) );
MPI_Type_free(&newtype);
```

For the full source of this example, see section 6.8.5

Similarly, using `MPI_Type_get_extent` counts the gaps in a `struct` induced by *alignment* issues.

```
size_t size_of_struct = sizeof(struct object);
MPI_Aint typesize,typelb;
MPI_Type_get_extent(newstructuretype,&typelb,&typesize);
assert( typesize==size_of_struct );
```

For the full source of this example, see section 6.8.22

See section 6.3.6 for the code defining the structure type.

Figure 6.16 `MPI_Type_get_true_extent`

Name	Param name	C type	F type	inout
`mpi_type_get_true_extent (`				
	`datatype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	datatype to get information on			
	`true_lb`	`MPI_Aint*`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`out`
	true lower bound of datatype			
	`true_extent`	`MPI_Aint*`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`out`
	true size of datatype			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Get_true_extent(`		
	`self`	
`) -> Tuple[int, int]`		

Remark 11 *Routine* `MPI_Type_get_extent` *replaces deprecated functions* `MPI_Type_extent`, `MPI_Type_lb`, `MPI_Type_ub` .

The *subarray datatype* need not start at the first element of the buffer, so the extent is an overstatement of how much data is involved. The routine `MPI_Type_get_true_extent` (figure 6.16) returns the lower bound, indicating where the data starts, and the extent from that point.

```
// trueextent.c
int sender = 0, receiver = 1, the_other = 1-procno,
  count = 4;
int sizes[2] = {4,6},subsizes[2] = {2,3},starts[2] = {1,2};
MPI_Datatype subarraytype;
if (procno==sender) {
  MPI_Type_create_subarray
    (2,sizes,subsizes,starts,MPI_ORDER_C,MPI_DOUBLE,&subarraytype);
  MPI_Type_commit(&subarraytype);

  MPI_Aint true_lb,true_extent,extent;
//    MPI_Type_get_extent(subarraytype,&extent);
  MPI_Type_get_true_extent
    (subarraytype,&true_lb,&true_extent);
  MPI_Aint
    comp_lb = sizeof(double) *
        ( starts[0]*sizes[1]+starts[1] );
    comp_extent = sizeof(double) *
        ( (starts[0]+subsizes[0]-1)*sizes[1] + starts[1]+subsizes[1] )
        - comp_lb;
//    ASSERT(extent==true_lb+extent);
  ASSERT(true_lb==comp_lb);
  ASSERT(true_extent==comp_extent);

  MPI_Send(source,1,subarraytype,the_other,0,comm);
  MPI_Type_free(&subarraytype);
```

For the full source of this example, see section 6.8.25

There is also a 'big data' routine `MPI_Type_get_true_extent_x` that has an `MPI_Count` as output.

6.5.2 Extent resizing

A type is partly characterized by its lower bound and extent, or equivalently lower bound and upperbound. Somewhat miraculously, you can actually change these to achieve special effects. This is needed for some cases of gather/scatter operations, or when the count of items in a buffer is more than one.

To understand the latter case, consider the vector type from the previous section. It is clear that

```
MPI_Type_vector( 2*count,bs,stride,oldtype,&two_n_type );
```

will not give the same result as

```
MPI_Type_vector( count,bs,stride,oldtype,&one_n_type );
MPI_Type_contiguous( 2,&one_n_type,&two_n_type );
```

(Can you draw a picture of the two cases?)

For the former case, consider figure 6.5 and exercise 6.4. There, strided data was sent in individual transactions. Would it be possible to address all these interleaved packets in one gather or scatter call?

The problem here is that MPI uses the extent of the send type in a scatter, or the receive type in a gather: if that type is 20 bytes big from its first to its last element, then data will be read out 20 bytes apart in a scatter, or written 20 bytes apart in a gather. This ignores the 'gaps' in the type!

The technicality on which the solution hinges is that you can 'resize' a type to give it a different extent, while not affecting how much data there actually is in it.

Let's consider an example where each process makes a buffer of integers that will be interleaved in a gather call:

```
int *mydata = (int*) malloc( localsize*sizeof(int) );
for (int i=0; i<localsize; i++)
  mydata[i] = i*nprocs+procno;
MPI_Gather( mydata,localsize,MPI_INT,
    /* rest to be determined */ );
```

An ordinary gather call will of course not interleave, but put the data end-to-end:

```
MPI_Gather( mydata,localsize,MPI_INT,
            gathered,localsize,MPI_INT, // abutting
            root,comm );
```

```
gather 4 elements from 3 procs:
 0 3 6 9 1 4 7 10 2 5 8 11
```

Using a strided type still puts data end-to-end, but now there are unwritten gaps in the gather buffer:

```
MPI_Gather( mydata,localsize,MPI_INT,
            gathered,1,stridetype, // abut with gaps
            root,comm );
```

```
0 1879048192 1100361260 3 3 0 6 0 0 9 1 198654
```

The trick is to use `MPI_Type_create_resized` to make the extent of the type only one int long:

```
// interleavegather.c
MPI_Datatype interleavetype;
MPI_Type_create_resized(stridetype,0,sizeof(int),&interleavetype);
MPI_Type_commit(&interleavetype);
MPI_Gather( mydata,localsize,MPI_INT,
              gathered,1,interleavetype, // shrunk extent
              root,comm );
```

For the full source of this example, see section 6.8.26

Now data is written with the same stride, but at starting points equal to the shrunk extent:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

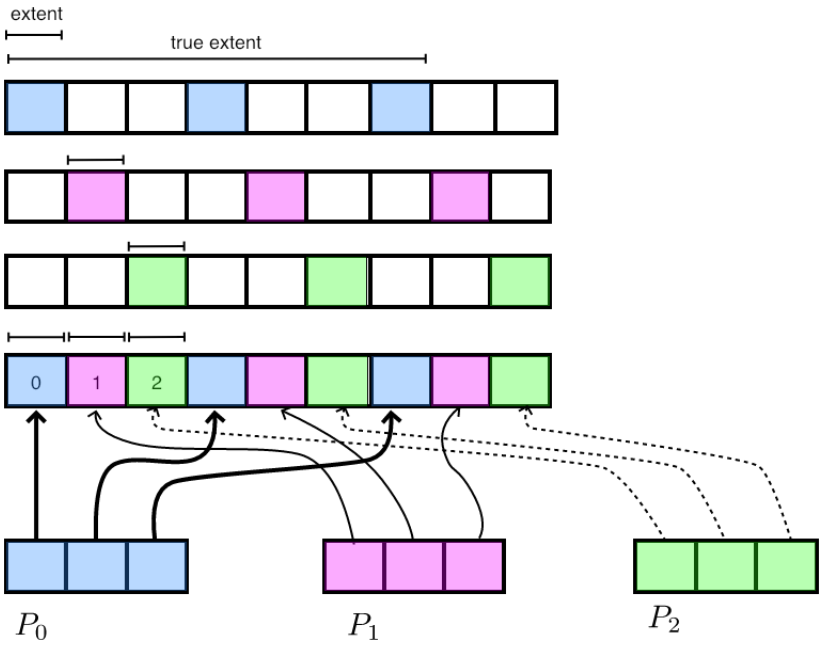


Figure 6.8: Interleaved gather from data with resized extent

This is illustrated in figure 6.8.

MPL note 51. Resizing a datatype does not give a new type, but does the resize 'in place':

```
void layout::resize(ssize_t lb, ssize_t extent);
```

End of MPL note

6.5.2.1 Example: dynamic vectors

Does it bother you (a little) that in the vector type you have to specify explicitly how many blocks there are? It would be nice if you could create a 'block with padding' and then send however many of those.

Well, you can introduce that padding by resizing a type, making it a little larger.

```
// stridestretch.c
MPI_Datatype oneblock;
MPI_Type_vector(1,1,stride,MPI_DOUBLE,&oneblock);
MPI_Type_commit(&oneblock);
MPI_Aint block_lb,block_x;
MPI_Type_get_extent(oneblock,&block_lb,&block_x);
printf("One block has extent: %ld\n",block_x);

MPI_Datatype paddedblock;
MPI_Type_create_resized(oneblock,0,stride*sizeof(double),&paddedblock);
MPI_Type_commit(&paddedblock);
MPI_Type_get_extent(paddedblock,&block_lb,&block_x);
printf("Padded block has extent: %ld\n",block_x);

// now send a bunch of these padded blocks
MPI_Send(source,count,paddedblock,the_other,0,comm);
```

For the full source of this example, see section 6.8.27

There is a second solution to this problem, using a structure type. This does not use resizing, but rather indicates a displacement that reaches to the end of the structure. We do this by putting a type `MPI_UB` at this displacement:

```
int blens[2]; MPI_Aint displs[2];
MPI_Datatype types[2], paddedblock;
blens[0] = 1; blens[1] = 1;
displs[0] = 0; displs[1] = 2 * sizeof(double);
types[0] = MPI_DOUBLE; types[1] = MPI_UB;
MPI_Type_struct(2, blens, displs, types, &paddedblock);
MPI_Type_commit(&paddedblock);
MPI_Status recv_status;
MPI_Recv(target,count,paddedblock,the_other,0,comm,&recv_status);
```

For the full source of this example, see section 6.8.27

6.5.2.2 *Example: transpose*

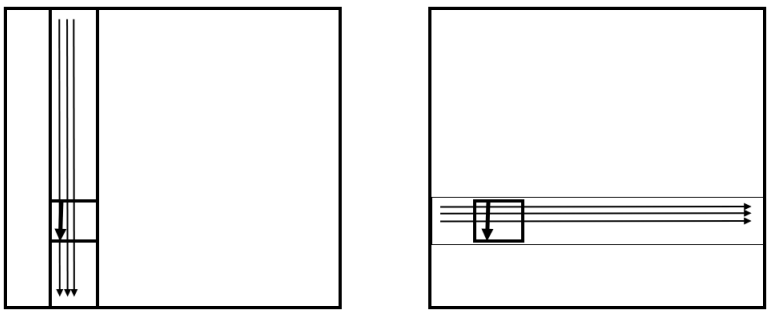


Figure 6.9: Transposing a 1D partitioned array

Transposing data is an important part of such operations as the FFT. We develop this in steps. Refer to figure 6.9.

The source data is easily described as a vector type defined as:

- there are b blocks,
- of blocksize b,
- spaced apart by the global i-size of the array.

```
// transposeblock.cxx
MPI_Datatype sourceblock;
MPI_Type_vector( blocksize_j,blocksize_i,isize,MPI_INT,&sourceblock);
MPI_Type_commit( &sourceblock);
```

The target type is harder to describe. First we note that each contiguous block from the source type can be described as a vector type with:

- b blocks,
- of size 1 each,
- stided by the global j-size of the matrix.

```
MPI_Datatype targetcolumn;
MPI_Type_vector( blocksize_i,1,jsize, MPI_INT,&targetcolumn);
MPI_Type_commit( &targetcolumn );
```

For the full type at the receiving process we now need to pack b of these lines together.

Exercise 6.7. Finish the code.

- What is the extent of the `targetcolumn` type?
- What is the spacing of the first elements of the blocks? How do you therefore resize the `targetcolumn` type?

6.6 More about data

6.6.1 Big data types

The `size` parameter in MPI send and receive calls is of type integer, meaning that it's maximally $2^{31}-1$. These day computers are big enough that this is a limitation. Derived types offer some way out: to send a *big data type* of 10^{40} elements you would

- create a contiguous type with 10^{20} elements, and
- send 10^{20} elements of that type.

This often works, but it's not perfect. For instance, the routine `MPI_Get_elements` returns the total number of basic elements sent (as opposed to `MPI_Get_count` which would return the number of elements of the derived type). Since its output argument is of integer type, it can't store the right value.

The MPI-3 standard has addressed this through the introduction of an `MPI_Count` datatype, and new routines that return that type of count. (The alternative would be to break backwards compatibility and use `MPI_Count` parameter in all existing routines.)

Let us consider an example.

Allocating a buffer of more than 4Gbyte is not hard:

```
// vectorx.c
float *source=NULL,*target=NULL;
int mediumsize = 1<<30;
int nblocks = 8;
size_t datasize = (size_t)mediumsize * nblocks * sizeof(float);
if (procno==sender) {
  source = (float*) malloc(datasize);
```

For the full source of this example, see section 6.8.28

We use the trick with sending elements of a derived type:

```
MPI_Datatype blocktype;
MPI_Type_contiguous(mediumsize,MPI_FLOAT,&blocktype);
MPI_Type_commit(&blocktype);
if (procno==sender) {
  MPI_Send(source,nblocks,blocktype,receiver,0,comm);
```

For the full source of this example, see section 6.8.28

We use the same trick for the receive call, but now we catch the status parameter which will later tell us how many elements of the basic type were sent:

```
} else if (procno==receiver) {
  MPI_Status recv_status;
  MPI_Recv(target,nblocks,blocktype,sender,0,comm,
    &recv_status);
```

For the full source of this example, see section 6.8.28

When we query how many of the basic elements are in the buffer (remember that in the receive call the buffer length is an upper bound on the number of elements received) do we need a counter that is larger than an integer. MPI has introduced a type `MPI_Count` for this, and new routines such as `MPI_Get_elements_x` (figure 4.17) that return a count of this type:

```
MPI_Count recv_count;
MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
```

For the full source of this example, see section 6.8.28

Remark 12 *Computing a big number to allocate is not entirely simple.*

```
// getx.c
int gig = 1<<30;
int nblocks = 8;
size_t big1 = gig * nblocks * sizeof(double);
size_t big2 = (size_t)1 * gig * nblocks * sizeof(double);
size_t big3 = (size_t) gig * nblocks * sizeof(double);
size_t big4 = gig * nblocks * (size_t) ( sizeof(double) );
size_t big5 = sizeof(double) * gig * nblocks;
;
```

For the full source of this example, see section 6.8.29

gives as output:

```
size of size_t = 8
0 68719476736 68719476736 0 68719476736
```

Clearly, not only do operations go left-to-right, but casting is done that way too: the computed subexpressions are only cast to `size_t` *if one operand is.*

Above, we did not actually create a datatype that was bigger than 2G, but if you do so, you can query its extent by `MPI_Type_get_extent_x` (figure 6.16) and `MPI_Type_get_true_extent_x` (figure 6.16).

Python note. Since python has unlimited size integers there is no explicit need for the 'x' variants of routines. Internally, *`MPI.Status.Get_count`* is implemented in terms of `MPI_Get_count_x`.

6.6.2 Packing

One of the reasons for derived datatypes is dealing with noncontiguous data. In older communication libraries this could only be done by *packing* data from its original containers into a buffer, and likewise unpacking it at the receiver into its destination data structures.

MPI offers this packing facility, partly for compatibility with such libraries, but also for reasons of flexibility. Unlike with derived datatypes, which transfers data atomically, packing routines add data sequentially to the buffer and unpacking takes them sequentially.

This means that one could pack an integer describing how many floating point numbers are in the rest of the packed message. Correspondingly, the unpack routine could then investigate the first integer and based on it unpack the right number of floating point numbers.

MPI offers the following:

- The `MPI_Pack` command adds data to a send buffer;
- the `MPI_Unpack` command retrieves data from a receive buffer;
- the buffer is sent with a datatype of `MPI_PACKED`.

With `MPI_Pack` data elements can be added to a buffer one at a time. The `position` parameter is updated each time by the packing routine.

```
int MPI_Pack(
  void *inbuf, int incount, MPI_Datatype datatype,
  void *outbuf, int outcount, int *position,
  MPI_Comm comm);
```

Conversely, `MPI_Unpack` retrieves one element from the buffer at a time. You need to specify the MPI datatype.

```
int MPI_Unpack(
  void *inbuf, int insize, int *position,
  void *outbuf, int outcount, MPI_Datatype datatype,
  MPI_Comm comm);
```

A packed buffer is sent or received with a datatype of `MPI_PACKED`. The sending routine uses the `position` parameter to specify how much data is sent, but the receiving routine does not know this value a priori, so has to specify an upper bound.

Figure 6.17 `MPI_Pack_size`

Name	Param name	C type	F type	inout
`mpi_pack_size (`				
	`incount`	`int`	`INTEGER`	`in`
	count argument to packing call			
	`datatype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	datatype argument to packing call			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator argument to packing call			
	`size`	`int*`	`INTEGER`	`out`
	upper bound on size of packed message, in bytes			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Datatype.Pack_size(`		
	`self`	
	`count`	`int`
	`comm`	`Comm`
`) -> int`		

```
// pack.c
if (procno==sender) {
  MPI_Pack(&nsends,1,MPI_INT,buffer,buflen,&position,comm);
  for (int i=0; i<nsends; i++) {
    double value = rand()/(double)RAND_MAX;
    MPI_Pack(&value,1,MPI_DOUBLE,buffer,buflen,&position,comm);
  }
  MPI_Pack(&nsends,1,MPI_INT,buffer,buflen,&position,comm);
  MPI_Send(buffer,position,MPI_PACKED,other,0,comm);
} else if (procno==receiver) {
  int irecv_value;
  double xrecv_value;
  MPI_Recv(buffer,buflen,MPI_PACKED,other,0,comm,MPI_STATUS_IGNORE);
  MPI_Unpack(buffer,buflen,&position,&nsends,1,MPI_INT,comm);
  for (int i=0; i<nsends; i++) {
    MPI_Unpack(buffer,buflen,&position,&xrecv_value,1,MPI_DOUBLE,comm);
  }
  MPI_Unpack(buffer,buflen,&position,&irecv_value,1,MPI_INT,comm);
  ASSERT(irecv_value==nsends);
}
```

For the full source of this example, see section 6.8.30

You can precompute the size of the required buffer with `MPI_Pack_size` (figure 6.17) Add one time `MPI_BSEND_OVERHEAD` .

Exercise 6.8. Suppose you have a 'structure of arrays'

```
struct aos {
  int length;
  double *reals;
  double *imags;
```

```
};
```

with dynamically created arrays. Write code to send and receive this structure.

6.7 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

1. Give two examples of MPI derived datatypes. What parameters are used to describe them?
2. Give a practical example where the sender uses a different type to send than the receiver uses in the corresponding receive call. Name the types involved.
3. Fortran only. True or false?
   (a) Array indices can be different between the send and receive buffer arrays.
   (b) It is allowed to send an array section.
   (c) You need to `Reshape` a multi-dimensional array to linear shape before you can send it.
   (d) An allocatable array, when dimensioned and allocated, is treated by MPI as if it were a normal static array, when used as send buffer.
   (e) An allocatable array is allocated if you use it as the receive buffer: it is filled with the incoming data.
4. Fortran only: how do you handle the case where you want to use an allocatable array as receive buffer, but it has not been allocated yet, and you do not know the size of the incoming data?

6.8 Sources used in this chapter

6.8.1 Listing of code header

6.8.2 Listing of code examples/mpi/p/inttype.py

```
from mpi4py import MPI
import numpy as np
import sys

comm = MPI.COMM_WORLD

nprocs = comm.Get_size()
procno = comm.Get_rank()

if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

## set up sender and receiver
sender = 0; receiver = nprocs-1
## how many elements to each process?
count = 6

## Send a contiguous buffer as numpy ints
if procno==sender:
    data = np.empty(count,dtype=np.int)
    for i in range(count):
        data[i] = i
    comm.Send( data, receiver )
elif procno==receiver:
    data = np.empty(count,dtype=np.int)
    comm.Recv( data, sender )
    print(data)

## Send a strided buffer as numpy ints
## this is wrong because numpy ints are not C ints
if procno==sender:
    sizeofint = np.dtype('int').itemsize
    print("Size of numpy int: {}".format(sizeofint))
    data = np.empty(2*count,dtype=np.int)
    for i in range(2*count):
        data[i] = i
    vectortype = MPI.INT.Create_vector(count,1,2)
    vectortype.Commit()
    comm.Send( [data,1,vectortype], receiver )
elif procno==receiver:
    data = np.empty(count,dtype=np.int)
    comm.Recv( data, sender )
    print(data)

## Send strided buffer as C ints
```

```
if procno==sender:
    sizeofint = np.dtype('intc').itemsize
    print("Size of C int: {}".format(sizeofint))
    data = np.empty(2*count,dtype=np.intc)
    for i in range(2*count):
        data[i] = i
    vectortype = MPI.INT.Create_vector(count,1,2)
    vectortype.Commit()
    comm.Send( [data,1,vectortype], receiver )
elif procno==receiver:
    data = np.empty(count,dtype=np.intc)
    comm.Recv( data, sender )
    print(data)
```

6.8.3 Listing of code examples/mpi/p/allgatherv.py

```
import numpy as np
import random # random.randint(1,N), random.random()
import sys
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

mycount = procid+1
my_array = np.empty(mycount,dtype=np.float64)

for i in range(mycount):
    my_array[i] = procid
recv_counts = np.empty(nprocs,dtype=np.int)
recv_displs = np.empty(nprocs,dtype=np.int)

my_count = np.empty(1,dtype=np.int)
my_count[0] = mycount
comm.Allgather( my_count,recv_counts )

accumulate = 0
for p in range(nprocs):
    recv_displs[p] = accumulate; accumulate += recv_counts[p]
global_array = np.empty(accumulate,dtype=np.float64)
comm.Allgatherv( my_array, [global_array,recv_counts,recv_displs,MPI.DOUBLE] )

# other syntax:
# comm.Allgatherv( [my_array,mycount,0,MPI.DOUBLE], [global_array,recv_counts,recv_displs,MPI.DOUBLE] )

if procid==0:
    #print(procid,global_array)
    for p in range(nprocs):
        if recv_counts[p]!=p+1:
```

```
            print( "recv count[%d] should be %d, not %d" \
                % (p,p+1,recv_counts[p]) )
    c = 0
    for p in range(nprocs):
        for q in range(p+1):
            if global_array[c]!=p:
                print( "p=%d, q=%d should be %d, not %d" \
                    % (p,q,p,global_array[c]) )
            c += 1
    print "finished"
```

6.8.4 Listing of code examples/mpi/c/typematch.c

```
int main() {

  MPI_Init(0,0);

  float x5;
  double x10;
  int s5,s10;
  MPI_Datatype mpi_x5,mpi_x10;

  MPI_Type_match_size(MPI_TYPECLASS_REAL,sizeof(x5),&mpi_x5);
  MPI_Type_match_size(MPI_TYPECLASS_REAL,sizeof(x10),&mpi_x10);
  MPI_Type_size(mpi_x5,&s5);
  MPI_Type_size(mpi_x10,&s10);

  printf("%d, %d\n",s5,s10);

  MPI_Finalize();

  return 0;
}
```

6.8.5 Listing of code examples/mpi/c/typesize.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  int size, count, stride, bs;

  MPI_Datatype newtype;

  count = 3; bs = 2; stride = 5;
```

```
  MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
  MPI_Type_commit(&newtype);
  MPI_Type_size(newtype,&size);
  ASSERT( size==(count*bs)*sizeof(double) );
  MPI_Type_free(&newtype);

  printf("count=%d, stride=%d, bs=%d, size=%d\n",count,stride,bs,size);

  MPI_Aint lb,asize;
  MPI_Type_vector(count,bs,stride,MPI_DOUBLE,&newtype);
  MPI_Type_commit(&newtype);
  MPI_Type_get_extent(newtype,&lb,&asize);
  ASSERT( lb==0 );
  ASSERT( asize==((count-1)*stride+bs)*sizeof(double) );
  MPI_Type_free(&newtype);

  printf("count=%d, stride=%d, bs=%d: lb=%ld, extent=%ld\n",count,stride,bs,lb,asize);

  if (procno==0)
    printf("Finished\n");

  MPI_Finalize();
  return 0;
}
```

6.8.6 Listing of code examples/mpi/p/vector.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

sender = 0; receiver = 1; the_other = 1-procid
count = 5; stride = 2

source = np.empty(stride*count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)

for i in range(stride*count):
    source[i] = i+.5

if procid==sender:
    newvectortype = MPI.DOUBLE.Create_vector(count,1,stride)
    newvectortype.Commit()
    comm.Send([source,1,newvectortype],dest=the_other)
    newvectortype.Free()
elif procid==receiver:
```

```
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)

if procid==sender:
    print("finished")
if procid==receiver:
    for i in range(count):
        if target[i]!=source[stride*i]:
            print("error in location %d: %e s/b %e" % (i,target[i],source[stride*i]))
```

6.8.7 Listing of code examples/mpi/mpl/vector.cxx

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <vector>
using std::vector;
#include <cassert>

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

  const mpl::communicator &comm_world = mpl::environment::comm_world();
  int nprocs,procno;
  // compute communicator rank and size
  nprocs = comm_world.size();
  procno = comm_world.rank();

  if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
  }
  int sender = 0, receiver = 1, the_other = 1-procno,
    count = 5,stride=2;
  vector<double>
    source(stride*count);
  vector<double>
    target(count);

  for (int i=0; i<stride*count; i++)
    source[i] = i+.5;

  if (procno==sender) {
    mpl::strided_vector_layout<double>
      newvectortype(count,1,stride);
    comm_world.send
      (source.data(),newvectortype,the_other);
  }
  else if (procno==receiver) {
    int recv_count;
    mpl::contiguous_layout<double> target_layout(count),
    mpl::status recv_status =
      comm_world.recv(target.data(),target_layout, the_other);
```

```
    recv_count = recv_status.get_count<double>();
    assert(recv_count==count);
  }

  if (procno==receiver) {
    for (int i=0; i<count; i++)
      if (target[i]!=source[stride*i])
printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
  }

  if (procno==0)
    printf("Finished\n");

  return 0;
}
```

6.8.8 Listing of code examples/mpi/c/contiguous.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
  }
  int sender = 0, receiver = 1, count = 5;
  double *source,*target;
  source = (double*) malloc(count*sizeof(double));
  target = (double*) malloc(count*sizeof(double));

  for (int i=0; i<count; i++)
    source[i] = i+.5;

  MPI_Datatype newvectortype;
  if (procno==sender) {
    MPI_Type_contiguous(count,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,receiver,0,comm);
    MPI_Type_free(&newvectortype);
  } else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,sender,0,comm,
      &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(count==recv_count);
  }
```

```
  if (procno==receiver) {
    for (int i=0; i<count; i++)
      if (target[i]!=source[i])
printf("location %d %e s/b %e\n",i,target[i],source[i]);
  }

  if (procno==0)
    printf("Finished\n");

  MPI_Finalize();
  return 0;
}
```

6.8.9 Listing of code examples/mpi/f/contiguous.F90

```
Program Contiguous

  use mpi
  implicit none

  integer :: sender = 0, receiver = 1, count = 5
  double precision, dimension(:),allocatable :: source,target

  integer :: newvectortype
  integer :: recv_status(MPI_STATUS_SIZE),recv_count

#include "globalinit.F90"

  if (ntids<2) then
    print *,"This program needs at least two processes"
    stop
  end if

  ALLOCATE(source(count))
  ALLOCATE(target(count))

  do i=1,count
     source(i) = i+.5;
  end do

  if (mytid==sender) then
     call MPI_Type_contiguous(count,MPI_DOUBLE_PRECISION,newvectortype,err)
     call MPI_Type_commit(newvectortype,err)
     call MPI_Send(source,1,newvectortype,receiver,0,comm,err)
     call MPI_Type_free(newvectortype,err)
  else if (mytid==receiver) then
     call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
          recv_status,err)
     call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count,err)
     !ASSERT(count==recv_count);
  end if
```

```
  if (mytid==receiver) then
    ! for (i=0; i<count; i++)
    !   if (target[i]!=source[i])
    !     printf("location %d %e s/b %e\n",i,target[i],source[i]);
  end if

  call MPI_Finalize(err)

end Program Contiguous
```

6.8.10 Listing of code examples/mpi/p/contiguous.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

sender = 0; receiver = 1; the_other = 1-procid
count = 5

source = np.empty(count,dtype=np.float64)
target = np.empty(count,dtype=np.float64)

for i in range(count):
    source[i] = i+.5

if procid==sender:
    newcontiguoustype = MPI.DOUBLE.Create_contiguous(count)
    newcontiguoustype.Commit()
    comm.Send([source,1,newcontiguoustype],dest=the_other)
    newcontiguoustype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)

if procid==sender:
    print("finished")
if procid==receiver:
    for i in range(count):
        if target[i]!=source[i]:
            print("error in location %d: %e s/b %e" % (i,target[i],source[i]))
```

6.8.11 Listing of code examples/mpi/c/vector.c

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
  }
  int sender = 0, receiver = 1, the_other = 1-procno,
    count = 5,stride=2;
  double *source,*target;
  source = (double*) malloc(stride*count*sizeof(double));
  target = (double*) malloc(count*sizeof(double));

  for (int i=0; i<stride*count; i++)
    source[i] = i+.5;

  MPI_Datatype newvectortype;
  if (procno==sender) {
    MPI_Type_vector(count,1,stride,MPI_DOUBLE,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    MPI_Type_free(&newvectortype);
  } else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
      &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
  }

  if (procno==receiver) {
    for (int i=0; i<count; i++)
      if (target[i]!=source[stride*i])
printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
  }

  if (procno==0)
    printf("Finished\n");

  MPI_Finalize();
  return 0;
}
```

6.8.12 Listing of code examples/mpi/f08/vector.F90

```
Program Vector

  use mpi_f08
```

```
implicit none

double precision, dimension(:),allocatable :: source,target
integer :: sender = 0,receiver = 1, count = 5, stride = 2

Type(MPI_Datatype) :: newvectortype
integer :: recv_count
Type(MPI_Status) :: recv_status

Type(MPI_Comm) :: comm;
integer :: mytid,ntids,i,p,err;

call MPI_Init()
comm = MPI_COMM_WORLD
call MPI_Comm_rank(comm,mytid)
call MPI_Comm_size(comm,ntids)
call MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN)


if (ntids<2) then
  print *,"This program needs at least two processes"
  stop
end if

ALLOCATE(source(stride*count))
ALLOCATE(target(stride*count))

do i=1,stride*count
   source(i) = i+.5;
end do

if (mytid==sender) then
   call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
        newvectortype)
   call MPI_Type_commit(newvectortype)
   call MPI_Send(source,1,newvectortype,receiver,0,comm)
   call MPI_Type_free(newvectortype)
   if ( .not. newvectortype==MPI_DATATYPE_NULL) then
      print *,"Trouble freeing datatype"
   else
      print *,"Datatype successfully freed"
   end if
else if (mytid==receiver) then
   call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
        recv_status)
   call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count)
end if

if (mytid==receiver) then
  ! for (i=0; i<count; i++)
  !   if (target[i]!=source[stride*i])
  !     printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
end if
```

```
  call MPI_Finalize(err)

end Program Vector
```

6.8.13 Listing of code examples/mpi/f/vector.F90

```
Program Vector

  use mpi
  implicit none

  double precision, dimension(:),allocatable :: source,target
  integer :: sender = 0,receiver = 1, count = 5, stride = 2

  integer :: newvectortype
  integer :: recv_status(MPI_STATUS_SIZE),recv_count

#include "globalinit.F90"

  if (ntids<2) then
    print *,"This program needs at least two processes"
    stop
  end if

  ALLOCATE(source(stride*count))
  ALLOCATE(target(stride*count))

  do i=1,stride*count
     source(i) = i+.5;
  end do

  if (mytid==sender) then
     call MPI_Type_vector(count,1,stride,MPI_DOUBLE_PRECISION,&
          newvectortype,err)
     call MPI_Type_commit(newvectortype,err)
     call MPI_Send(source,1,newvectortype,receiver,0,comm,err)
     call MPI_Type_free(newvectortype,err)
  else if (mytid==receiver) then
     call MPI_Recv(target,count,MPI_DOUBLE_PRECISION,sender,0,comm,&
          recv_status,err)
     call MPI_Get_count(recv_status,MPI_DOUBLE_PRECISION,recv_count,err)
  end if

  if (mytid==receiver) then
    ! for (i=0; i<count; i++)
    !   if (target[i]!=source[stride*i])
    !     printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
  end if

  call MPI_Finalize(err)

end Program Vector
```

6.8.14 Listing of code examples/mpi/mpl/sendrange.cxx

```
#include <cstdlib>
#include <complex>
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <mpl/mpl.hpp>

int main() {
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  if (comm_world.size()<2)
    return EXIT_FAILURE;

  vector<double> v(15);

  if (comm_world.rank()==0) {

    // initialize
    for ( auto &x : v ) x = 1.41;

    /*
     * Send and report
     */
    comm_world.send(v.begin(), v.end(), 1);  // send to rank 1

  } else if (comm_world.rank()==1) {

    /*
     * Receive data and report
     */
    comm_world.recv(v.begin(), v.end(), 0);  // receive from rank 0

    cout << "Got:";
    for ( auto x : v )
      cout << " " << x;
    cout << endl;
  }
  return EXIT_SUCCESS;
}
```

6.8.15 Listing of code examples/mpi/f08/section.F90

```
Program F90Section
  use mpi_f08
  implicit none

  integer :: i,j, nprocs,procno
```

```
  integer,parameter :: siz=20
  real,dimension(siz,siz) :: matrix = [ ((j+(i-1)*siz,i=1,siz),j=1,siz) ]
  real,dimension(2,2) :: submatrix
  Type(MPI_Comm) :: comm

  call MPI_Init()
  comm = MPI_COMM_WORLD

  call MPI_Comm_size(comm,nprocs)
  call MPI_Comm_rank(comm,procno)
  if (nprocs<2) then
     print *,"This example really needs 2 processors"
     call MPI_Abort(comm,0)
  end if
  if (procno==0) then
     call MPI_Send(matrix(1:2,1:2),4,MPI_REAL,1,0,comm)
  else if (procno==1) then
     call MPI_Recv(submatrix,4,MPI_REAL,0,0,comm,MPI_STATUS_IGNORE)
     if (submatrix(2,2)==22) then
        print *,"Yay"
     else
        print *,"nay...."
     end if
  end if

  call MPI_Finalize()

end Program F90Section
```

6.8.16 Listing of code code/mpi/c/row2col.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#define ERR(c,m) if (c) { printf("Error: %s\n",m); MPI_Abort(MPI_COMM_WORLD,c); }

int main(int argc,char **argv) {

#define MOD(i,n) (i+n)%(n)
#define ABS(x) ((x)<0 ? (-x) : (x))
#define MAX(x,y) ( (x)>(y) ? (x) : (y) )

#define MALLOC( t,v,n ) \
  t *v = (t*) malloc ( (n)*sizeof(t) ); \
  if (!v) { printf("Allocation failed in line %d\n",__LINE__); MPI_Abort(comm,0); }

  MPI_Comm comm;
  int procno=-1,nprocs,ierr;
  MPI_Init(&argc,&argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_rank(comm,&procno);
```

```
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);

  if (nprocs==1) {
    printf("This program needs at least 2 procs\n");
    MPI_Abort(comm,0);
  }
  int sender = 0, receiver = nprocs-1;
#define SIZE 4
  int
    sizes[2], subsizes[2], starts[2];
  sizes[0] = SIZE; sizes[1] = SIZE;
  subsizes[0] = SIZE/2;  subsizes[1] = SIZE;
  starts[0] = starts[1] = 0;

  MPI_Request req;

  if (procno==sender) {
    /*
     * Write lexicographic test data
     */
    double data[SIZE][SIZE];
    for (int i=0; i<SIZE; i++)
      for (int j=0; j<SIZE; j++)
data[i][j] = j+i*SIZE;
    /*
     * Make a datatype that enumerates the storage in C order
     */
    MPI_Datatype rowtype;
    ierr =
    MPI_Type_create_subarray
      (2,sizes,subsizes,starts,
       MPI_ORDER_C,MPI_DOUBLE,&rowtype);
    ERR(ierr,"creating rowtype");
    MPI_Type_commit(&rowtype);
    MPI_Send(data,1,rowtype, receiver,0,comm);
    MPI_Type_free(&rowtype);

    /*
     * Make a datatype that enumerates the storage in F order
     */
    MPI_Datatype coltype;
    ierr =
    MPI_Type_create_subarray
      (2,sizes,subsizes,starts,
       MPI_ORDER_FORTRAN,MPI_DOUBLE,&coltype);
    ERR(ierr,"creating rowtype");
    MPI_Type_commit(&coltype);
    MPI_Send(data,1,coltype, receiver,0,comm);
    MPI_Type_free(&coltype);

  } else if (procno==receiver) {
    int linearsize = SIZE * SIZE/2;
```

```
    double lineardata[ linearsize ];

    /*
     * Receive msg in C order:
     */
     MPI_Recv(lineardata,linearsize,MPI_DOUBLE, sender,0,comm, MPI_STATUS_IGNORE);
    printf("Received C order:\n");
    for (int i=0; i<SIZE/2; i++) {
      for (int j=0; j<SIZE; j++)
printf(" %5.3f",lineardata[j+i*SIZE]);
      printf("\n");
    }

    /*
     * Receive msg in F order:
     */
     MPI_Recv(lineardata,linearsize,MPI_DOUBLE, sender,0,comm, MPI_STATUS_IGNORE);
    printf("Received F order:\n");
    for (int j=0; j<SIZE; j++) {
    for (int i=0; i<SIZE/2; i++)
printf(" %5.3f",lineardata[i+j*SIZE/2]);
      printf("\n");
    }

  }

  /* MPI_Finalize(); */
  return 0;
}
```

6.8.17 Listing of code examples/mpi/c/indexed.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
  }
  int sender = 0, receiver = 1, the_other = 1-procno,
    count = 5,totalcount = 15, targetbuffersize = 2*totalcount;
  int *source,*target;
  int *displacements,*blocklengths;

  displacements = (int*) malloc(count*sizeof(int));
  blocklengths = (int*) malloc(count*sizeof(int));
  source = (int*) malloc(totalcount*sizeof(int));
```

```
  target = (int*) malloc(targetbuffersize*sizeof(int));

  displacements[0] = 2; displacements[1] = 3; displacements[2] = 5;
  displacements[3] = 7; displacements[4] = 11;
  for (int i=0; i<count; ++i)
    blocklengths[i] = 1;
  for (int i=0; i<totalcount; ++i)
    source[i] = i;

  MPI_Datatype newvectortype;
  if (procno==sender) {
    MPI_Type_indexed(count,blocklengths,displacements,MPI_INT,&newvectortype);
    MPI_Type_commit(&newvectortype);
    MPI_Send(source,1,newvectortype,the_other,0,comm);
    MPI_Type_free(&newvectortype);
  } else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,targetbuffersize,MPI_INT,the_other,0,comm,
      &recv_status);
    MPI_Get_count(&recv_status,MPI_INT,&recv_count);
    ASSERT(recv_count==count);
  }

  if (procno==receiver) {
    int i=3,val=7;
    if (target[i]!=val)
      printf("location %d %d s/b %d\n",i,target[i],val);
    i=4; val=11;
    if (target[i]!=val)
      printf("location %d %d s/b %d\n",i,target[i],val);
  }
  if (procno==0)
    printf("Finished\n");

  MPI_Finalize();
  return 0;
}
```

6.8.18 Listing of code examples/mpi/f/indexed.F90

```
Program Indexed

  use mpi
  implicit none

  integer :: newvectortype;
  integer,dimension(:),allocatable :: indices,blocklengths,&
       source,targt
  integer :: sender = 0, receiver = 1, count = 5,totalcount = 15
  integer :: recv_status(MPI_STATUS_SIZE),recv_count

#include "globalinit.F90"
```

```
  if (ntids<2) then
    print *,"This program needs at least two processes"
    stop
  end if

  ALLOCATE(indices(count))
  ALLOCATE(blocklengths(count))
  ALLOCATE(source(totalcount))
  ALLOCATE(targt(count))

  indices(0) = 2; indices(1) = 3; indices(2) = 5;
  indices(3) = 7; indices(4) = 11;
  do i=1,count
     blocklengths(i) = 1
  end do
  do i=1,totalcount
     source(i) = i
  end do

  if (mytid==sender) then
     call MPI_Type_indexed(count,blocklengths,indices,MPI_INT,&
          newvectortype,err)
     call MPI_Type_commit(newvectortype,err)
     call MPI_Send(source,1,newvectortype,receiver,0,comm,err)
     call MPI_Type_free(newvectortype,err)
  else if (mytid==receiver) then
    call MPI_Recv(targt,count,MPI_INT,sender,0,comm,&
         recv_status,err)
    call MPI_Get_count(recv_status,MPI_INT,recv_count,err)
    !    ASSERT(recv_count==count);
 end if

  ! if (mytid==receiver) {
  !   int i=3,val=7;
  !   if (targt(i)!=val)
  !     printf("location %d %d s/b %d\n",i,targt(i),val);
  !   i=4; val=11;
  !   if (targt(i)!=val)
  !     printf("location %d %d s/b %d\n",i,targt(i),val);
  ! }

 call MPI_Finalize(err)

end Program Indexed
```

6.8.19 Listing of code examples/mpi/p/indexed.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI
```

```
comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

sender = 0; receiver = 1; the_other = 1-procid
count = 5; totalcount = 15

displacements = np.empty(count,dtype=np.int)
blocklengths = np.empty(count,dtype=np.int)
source = np.empty(totalcount,dtype=np.float64)
target = np.empty(count,dtype=np.float64)

idcs = [2,3,5,7,11]
for i in range(len(idcs)):
    displacements[i] = idcs[i]
    blocklengths[i] = 1
for i in range(totalcount):
    source[i] = i+.5

if procid==sender:
    newindextype = MPI.DOUBLE.Create_indexed(blocklengths,displacements)
    newindextype.Commit()
    comm.Send([source,1,newindextype],dest=the_other)
    newindextype.Free()
elif procid==receiver:
    comm.Recv([target,count,MPI.DOUBLE],source=the_other)

if procid==sender:
    print("finished")
if procid==receiver:
    target_loc = 0
    for block in range(count):
        for element in range(blocklengths[block]):
            source_loc = displacements[block]+element
            if target[target_loc]!=source[source_loc]:
                print("error in src/tar location %d/%d: %e s/b %e" \
                    % (source_loc,target_loc,target[target_loc],source[source_loc]) )
            target_loc += 1
```

6.8.20 Listing of code examples/mpi/mpl/indexed.cxx

```
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;

#include <cassert>
```

```
#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

  // MPI Comm world
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  int nprocs = comm_world.size(), procno = comm_world.rank();

  int sender = 0, receiver = 1, the_other = 1-procno,
    totalcount = 15, targetbuffersize = 2*totalcount;

  vector<int>
    source_buffer(totalcount),
    target_buffer(targetbuffersize);
  for (int i=0; i<totalcount; ++i)
    source_buffer[i] = i;

  const int count = 5;
  mpl::contiguous_layout<int>
    fiveints(count);
  mpl::indexed_layout<int>
    indexed_where{ { {1,2}, {1,3}, {1,5}, {1,7}, {1,11} } };

  if (procno==sender) {
    comm_world.send( source_buffer.data(),indexed_where, receiver );
  } else if (procno==receiver) {
    auto recv_status =
      comm_world.recv( target_buffer.data(),fiveints, sender );
    int recv_count = recv_status.get_count<int>();
    assert(recv_count==count);
  }

  if (procno==receiver) {
    int i=3,val=7;
    if (target_buffer[i]!=val)
      printf("Error: location %d %d s/b %d\n",i,target_buffer[i],val);
    i=4; val=11;
    if (target_buffer[i]!=val)
      printf("Error: location %d %d s/b %d\n",i,target_buffer[i],val);
    printf("Finished. Correctly sent indexed primes.\n");
  }

  return 0;
}
```

6.8.21 Listing of code examples/mpi/mpl/indexedblock.cxx

```
#include <iostream>
using std::cout;
using std::endl;

#include <vector>
using std::vector;
```

```
#include <cassert>

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

  // MPI Comm world
  const mpl::communicator &comm_world=mpl::environment::comm_world();
  int nprocs = comm_world.size(), procno = comm_world.rank();

  int sender = 0, receiver = 1, the_other = 1-procno,
    totalcount = 15, targetbuffersize = 2*totalcount;

  vector<int>
    source_buffer(totalcount),
    target_buffer(targetbuffersize);
  for (int i=0; i<totalcount; ++i)
    source_buffer[i] = i;

  const int count = 5;
  mpl::contiguous_layout<int>
    fiveints(count);
  mpl::indexed_block_layout<int>
    indexed_where(  1, {2,3,5,7,11} );
  if (procno==sender) {
    comm_world.send( source_buffer.data(),indexed_where, receiver );
  } else if (procno==receiver) {
    auto recv_status =
      comm_world.recv( target_buffer.data(),fiveints, sender );
    int recv_count =
      recv_status.get_count<int>();
    assert(recv_count==count);
  }

  if (procno==receiver) {
    int i=3,val=7;
    if (target_buffer[i]!=val)
      printf("Error: location %d %d s/b %d\n",i,target_buffer[i],val);
    i=4; val=11;
    if (target_buffer[i]!=val)
      printf("Error: location %d %d s/b %d\n",i,target_buffer[i],val);
    printf("Finished. Correctly sent indexed primes.\n");
  }

  return 0;
}
```

6.8.22 Listing of code examples/mpi/c/struct.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
#include <assert.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
  }
  int sender = 0, receiver = 1, the_other = 1-procno;
  struct object {
    char c;
    double x[2];
    int i;
  };

  size_t size_of_struct = sizeof(struct object);
  if (procno==sender)
    printf("Structure has size %ld, naive size %ld\n",
   size_of_struct,
   sizeof(char)+2*sizeof(double)+sizeof(int));
  struct object myobject;
  if (procno==sender) {
    myobject.c = 'x';
    myobject.x[0] = 2.7; myobject.x[1] = 1.5;
    myobject.i = 37;
  }

  MPI_Datatype newstructuretype;
  int structlen = 3;
  int blocklengths[structlen]; MPI_Datatype types[structlen];
  MPI_Aint displacements[structlen];

  /*
   * where are the components relative to the structure?
   */
  MPI_Aint current_displacement=0;

  // one character
  blocklengths[0] = 1; types[0] = MPI_CHAR;
  displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;

  // two doubles
  blocklengths[1] = 2; types[1] = MPI_DOUBLE;
  displacements[1] = (size_t)&(myobject.x) - (size_t)&myobject;

  // one int
  blocklengths[2] = 1; types[2] = MPI_INT;
  displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;

  MPI_Type_create_struct(structlen,blocklengths,displacements,types,&newstructuretype);
```

```
  MPI_Type_commit(&newstructuretype);

  MPI_Aint typesize,typelb;
  MPI_Type_get_extent(newstructuretype,&typelb,&typesize);
  assert( typesize==size_of_struct );
  if (procno==sender) {
    printf("Type extent: %ld bytes; displacements: %ld %ld %ld\n",
   typesize,displacements[0],displacements[1],displacements[2]);
  }
  if (procno==sender) {
    MPI_Send(&myobject,1,newstructuretype,the_other,0,comm);
  } else if (procno==receiver) {
    MPI_Recv(&myobject,1,newstructuretype,the_other,0,comm,MPI_STATUS_IGNORE);
  }
  MPI_Type_free(&newstructuretype);

  /* if (procno==sender) */
  /*   printf("char x=%ld, l=%ld; double x=%ld, l=%ld, int x=%ld, l=%ld\n", */
  /*     char_extent,char_lb,double_extent,double_lb,int_extent,int_lb); */

  if (procno==receiver) {
    printf("Char '%c' double0=%e double1=%e int=%d\n",
   myobject.c,myobject.x[0],myobject.x[1],myobject.i);
    ASSERT(myobject.x[1]==1.5);
    ASSERT(myobject.i==37);
  }

  if (procno==0)
    printf("Finished\n");

  MPI_Finalize();
  return 0;
}

#if 0
  blocklengths[0] = 1; types[0] = MPI_CHAR;
  displacements[0] = (size_t)&(myobject.c) - (size_t)&myobject;
  blocklengths[1] = 2; types[1] = MPI_DOUBLE;
  displacements[1] = (size_t)&(myobject.x[0]) - (size_t)&myobject;
  blocklengths[2] = 1; types[2] = MPI_INT;
  displacements[2] = (size_t)&(myobject.i) - (size_t)&myobject;

  MPI_Aint char_extent,char_lb;
  MPI_Type_get_extent(MPI_CHAR,&char_lb,&char_extent);
  /* if (procno==0) */
  /*   printf("CHAR lb=%ld xt=%ld disp=%ld\n",char_lb,char_extent,current_displacement); */
  MPI_Aint double_extent,double_lb;
  MPI_Type_get_extent(MPI_DOUBLE,&double_lb,&double_extent);
  /* if (procno==0) */
  /*   printf("DOUBLE lb=%ld xt=%ld disp=%ld\n",double_lb,double_extent,current_displacement); */
  MPI_Aint int_extent,int_lb;
  MPI_Type_get_extent(MPI_INT,&int_lb,&int_extent);
  /* if (procno==0) */
```

```
  /*   printf("INT lb=%ld xt=%ld disp=%ld\n",int_lb,int_extent,current_displacement); */
#endif
```

6.8.23 Listing of code examples/mpi/f/struct.F90

```
Program StructType

  use mpi_f08
  implicit none

  type(MPI_Comm) :: comm;
  integer :: nprocs,procno;

  !!
  !! The Type that we are going to send
  !!
  Type object
     character :: c
     real*8,dimension(2) :: x
     integer :: i
  end type object

#define MPI_Aint integer(kind=MPI_ADDRESS_KIND)

  !!
  !! local data
  !!
  integer :: sender=0,receiver=1
  type(object) :: myobject
  integer,parameter :: structlen = 3
  type(MPI_Datatype) :: newstructuretype
  integer,dimension(structlen) :: blocklengths
  type(MPI_Datatype),dimension(structlen) :: types;
  MPI_Aint,dimension(structlen) :: displacements
  MPI_Aint :: base_displacement, next_displacement

  !!
  !! Initial setup
  !!
  call MPI_Init()
  comm = MPI_COMM_WORLD;
  call MPI_Comm_rank(comm,procno)
  call MPI_Comm_size(comm,nprocs)
  call MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN)

  if (nprocs<2) then
     print *,"This program needs at least two processes"
     call MPI_Abort(comm,1)
  end if

  !!
  !! Fill in meaningful data only on the sender
```

```
!!
if (procno==sender) then
   myobject%c = 'x'
   myobject%x(0) = 2.7; myobject%x(1) = 1.5
   myobject%i = 37

   print '("Set: char=",a1,x,", double0=",x,f9.5,x,"double1=",x,f9.5,x,", int=",i5)', &
        myobject%c,myobject%x(0),myobject%x(1),myobject%i
else
   myobject%c = ' '
   myobject%x(0) = 0.0; myobject%x(1) = 0.0
   myobject%i = 0
end if

!!
!! Where are the components relative to the structure?
!!

!! component 1: one character
blocklengths(1) = 1; types(1) = MPI_CHAR
call MPI_Get_address(myobject,base_displacement)
call MPI_Get_address(myobject%c,next_displacement)
displacements(1) = next_displacement-base_displacement

!! component 2: two doubles
blocklengths(2) = 2; types(2) = MPI_DOUBLE
call MPI_Get_address(myobject%x,next_displacement)
displacements(2) = next_displacement-base_displacement

!! component 3: one int
blocklengths(3) = 1; types(3) = MPI_INT
call MPI_Get_address(myobject%i,next_displacement)
displacements(3) = next_displacement-base_displacement

if (procno==sender) then
   print '("Displacements:",3(1x,i0))',displacements
end if

!!
!! Create the structure type
!!
call MPI_Type_create_struct(structlen,blocklengths,displacements,types,newstructuretype)
call MPI_Type_commit(newstructuretype)

!!
!! Send and receive call, both using the structure type
!!
if (procno==sender) then
   call MPI_Send(myobject,1,newstructuretype,receiver,0,comm)
else if (procno==receiver) then
   call MPI_Recv(myobject,1,newstructuretype,sender,0,comm,MPI_STATUS_IGNORE)
end if
call MPI_Type_free(newstructuretype)
```

```
  !!
  !! Print out what we sent and received
  !!
  if (procno==sender) then
     print '("Sent: char=",a1,x,", double0=",x,f9.5,x,"double1=",x,f9.5,x,", int=",i5)', &
          myobject%c,myobject%x(0),myobject%x(1),myobject%i
  else if (procno==receiver) then
     print '("Received: char=",a1,x,", double0=",x,f9.5,x,"double1=",x,f9.5,x,", int=",i5)', &
          myobject%c,myobject%x(0),myobject%x(1),myobject%i
  end if

  if (procno==0) then
     print *,"Finished"
  end if

  call MPI_Finalize()


end Program StructType
```

6.8.24 Listing of code examples/mpi/mpl/struct.cxx

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <vector>
using std::vector;
#include <cassert>

#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

  const mpl::communicator &comm_world = mpl::environment::comm_world();
  int nprocs,procno;
  // compute communicator rank and size
  nprocs = comm_world.size();
  procno = comm_world.rank();

  if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
  }
  int sender = 0, receiver = 1, the_other = 1-procno;
  char c;
  vector<double> x(2);
  int i;
  if (procno==sender) {
    c = 'x';
    x[0] = 2.7; x[1] = 1.5;
    i = 37;
  }
```

```
  mpl::heterogeneous_layout object
    ( c,
      mpl::make_absolute( x.data(),mpl::vector_layout<double>(2) ),
      i );
  if (procno==sender) {
    comm_world.send( mpl::absolute,object,receiver );
  } else if (procno==receiver) {
    comm_world.recv( mpl::absolute,object,sender );
  }

  if (procno==receiver) {
    printf("Char '%c' double0=%e double1=%e int=%d\n",
   c,x[0],x[1],i);
    assert(x[1]==1.5);
    assert(i==37);
  }

  if (procno==0)
    printf("Finished\n");

  return 0;
}
```

6.8.25 Listing of code examples/mpi/c/trueextent.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
  }
  int sender = 0, receiver = 1, the_other = 1-procno,
    count = 4;
  int sizes[2] = {4,6},subsizes[2] = {2,3},starts[2] = {1,2};
  double *source,*target;
  source = (double*) malloc(sizes[0]*sizes[1]*sizeof(double));
  target = (double*) malloc(subsizes[0]*subsizes[1]*sizeof(double));

  for (int i=0; i<sizes[0]*sizes[1]; i++)
    source[i] = i+.5;

  MPI_Datatype subarraytype;
  if (procno==sender) {
    MPI_Type_create_subarray
      (2,sizes,subsizes,starts,MPI_ORDER_C,MPI_DOUBLE,&subarraytype);
```

```
    MPI_Type_commit(&subarraytype);

    MPI_Aint true_lb,true_extent,extent;
    //    MPI_Type_get_extent(subarraytype,&extent);
    MPI_Type_get_true_extent
      (subarraytype,&true_lb,&true_extent);
    MPI_Aint
      comp_lb = sizeof(double) *
          ( starts[0]*sizes[1]+starts[1] );
      comp_extent = sizeof(double) *
          ( (starts[0]+subsizes[0]-1)*sizes[1] + starts[1]+subsizes[1] )
          - comp_lb;
    printf("Found lb=%ld, extent=%ld\n",true_lb,true_extent);
    printf("Computing lb=%ld extent=%ld\n",comp_lb,comp_extent);
    //    ASSERT(extent==true_lb+extent);
    ASSERT(true_lb==comp_lb);
    ASSERT(true_extent==comp_extent);

    MPI_Send(source,1,subarraytype,the_other,0,comm);
    MPI_Type_free(&subarraytype);
  } else if (procno==receiver) {
    MPI_Status recv_status;
    int recv_count;
    MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm,
      &recv_status);
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
  }

  if (procno==receiver) {
    printf("received:");
    for (int i=0; i<count; i++)
      printf(" %6.3f",target[i]);
    printf("\n");
    int icnt = 0;
    for (int i=starts[0]; i<starts[0]+subsizes[0]; i++) {
      for (int j=starts[1]; j<starts[1]+subsizes[1]; j++) {
printf("%d,%d\n",i,j);
ASSERTm(icnt<count,"icnt too hight");
int isrc = i*sizes[1]+j;
if (source[isrc]!=target[icnt])
  printf("target location (%d,%d)->%d %e s/b %e\n",i,j,icnt,target[icnt],source[isrc]);
icnt ++;
      }
    }
  }

  if (procno==0)
    printf("Finished\n");

  /* MPI_Finalize(); */
  return 0;
}
```

6.8.26 Listing of code examples/mpi/c/interleavegather

6.8.27 Listing of code examples/mpi/c/stridestretch.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
  }
  int sender = 0, receiver = 1, the_other = 1-procno,
    count = 5,stride=2;
  double *source,*target;
  source = (double*) malloc(stride*count*sizeof(double));
  target = (double*) malloc(stride*count*sizeof(double));

  for (int i=0; i<stride*count; i++)
    source[i] = i+.5;

  if (procno==sender) {
    MPI_Datatype oneblock;
    MPI_Type_vector(1,1,stride,MPI_DOUBLE,&oneblock);
    MPI_Type_commit(&oneblock);
    MPI_Aint block_lb,block_x;
    MPI_Type_get_extent(oneblock,&block_lb,&block_x);
    printf("One block has extent: %ld\n",block_x);

    MPI_Datatype paddedblock;
    MPI_Type_create_resized(oneblock,0,stride*sizeof(double),&paddedblock);
    MPI_Type_commit(&paddedblock);
    MPI_Type_get_extent(paddedblock,&block_lb,&block_x);
    printf("Padded block has extent: %ld\n",block_x);

    // now send a bunch of these padded blocks
    MPI_Send(source,count,paddedblock,the_other,0,comm);
    MPI_Type_free(&oneblock);
    MPI_Type_free(&paddedblock);
  } else if (procno==receiver) {
    int blens[2]; MPI_Aint displs[2];
    MPI_Datatype types[2], paddedblock;
    blens[0] = 1; blens[1] = 1;
    displs[0] = 0; displs[1] = 2 * sizeof(double);
    types[0] = MPI_DOUBLE; types[1] = MPI_UB;
    MPI_Type_struct(2, blens, displs, types, &paddedblock);
```

```
    MPI_Type_commit(&paddedblock);
    MPI_Status recv_status;
    MPI_Recv(target,count,paddedblock,the_other,0,comm,&recv_status);
    /* MPI_Recv(target,count,MPI_DOUBLE,the_other,0,comm, */
    /*   &recv_status); */
    int recv_count;
    MPI_Get_count(&recv_status,MPI_DOUBLE,&recv_count);
    ASSERT(recv_count==count);
  }

  if (procno==receiver) {
    for (int i=0; i<count; i++)
      if (target[i*stride]!=source[i*stride])
printf("location %d %e s/b %e\n",i,target[i*stride],source[stride*i]);
  }

  if (procno==0)
    printf("Finished\n");

  MPI_Finalize();
  return 0;
}
```

6.8.28 Listing of code examples/mpi/c/vectorx.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (nprocs<2) {
    printf("This program needs at least two processes\n");
    return -1;
  }
  int sender = 0, receiver = 1, the_other = 1-procno,
    count = 5,stride=2;

  if (procno==sender) printf("size of size_t = %d\n",sizeof(size_t));

  float *source=NULL,*target=NULL;
  int mediumsize = 1<<30;
  int nblocks = 8;
  size_t datasize = (size_t)mediumsize * nblocks * sizeof(float);

  if (procno==sender)
    printf("datasize = %lld bytes =%7.3f giga-bytes = %7.3f gfloats\n",
   datasize,datasize*1.e-9,datasize*1.e-9/sizeof(float));

  if (procno==sender) {
```

```
    source = (float*) malloc(datasize);
    if (source) {
      printf("Source allocated\n");
    } else {
      printf("Could not allocate source data\n"); MPI_Abort(comm,1); }
    long int idx = 0;
    for (int iblock=0; iblock<nblocks; iblock++) {
      for (int element=0; element<mediumsize; element++) {
source[idx] = idx+.5; idx++;
if (idx<0) { printf("source index wrap\n"); MPI_Abort(comm,0); }
      }
    }
  }

  if (procno==receiver) {
    target = (float*) malloc(datasize);
    if (target) {
      printf("Target allocated\n");
    } else {
      printf("Could not allocate target data\n"); MPI_Abort(comm,1); }
  }


  MPI_Datatype blocktype;
  MPI_Type_contiguous(mediumsize,MPI_FLOAT,&blocktype);
  MPI_Type_commit(&blocktype);
  if (procno==sender) {
    MPI_Send(source,nblocks,blocktype,receiver,0,comm);
  } else if (procno==receiver) {
    MPI_Status recv_status;
    MPI_Recv(target,nblocks,blocktype,sender,0,comm,
      &recv_status);
    MPI_Count recv_count;
    MPI_Get_elements_x(&recv_status,MPI_FLOAT,&recv_count);
    printf("Received %7.3f medium size elements\n",recv_count * 1e-9);
  }
  MPI_Type_free(&blocktype);

  if (0 && procno==receiver) {
    for (int i=0; i<count; i++)
      if (target[i]!=source[stride*i])
printf("location %d %e s/b %e\n",i,target[i],source[stride*i]);
  }

  if (procno==0)
    printf("Finished\n");

  if (procno==sender)
    free(source);
  if (procno==receiver)
    free(target);

  MPI_Finalize();
```

```
  return 0;
}
```

6.8.29 Listing of code examples/mpi/c/getx.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (procno==0) {
    printf("size of size_t = %d\n",sizeof(size_t));

    int gig = 1<<30;
    int nblocks = 8;
    size_t big1 = gig * nblocks * sizeof(double);
    size_t big2 = (size_t)1 * gig * nblocks * sizeof(double);
    size_t big3 = (size_t) gig * nblocks * sizeof(double);
    size_t big4 = gig * nblocks * (size_t) ( sizeof(double) );
    size_t big5 = sizeof(double) * gig * nblocks;
;
    printf("%lld %lld %lld %lld %lld\n",big1,big2,big3,big4,big5);
  }

  MPI_Finalize();
  return 0;
}
```

6.8.30 Listing of code examples/mpi/pack/pack.c

Chapter 7

MPI topic: Communicators

A communicator is an object describing a group of processes. In many applications all processes work together closely coupled, and the only communicator you need is `MPI_COMM_WORLD`, the group describing all processes that your job starts with.

In this chapter you will see ways to make new groups of MPI processes: subgroups of the original world communicator. Chapter 8 discusses dynamic process management, which, while not extending `MPI_COMM_WORLD` does extend the set of available processes.

7.1 Basic communicators

There are three predefined communicators:

- `MPI_COMM_WORLD` comprises all processes that were started together by *mpiexec* (or some related program).
- `MPI_COMM_SELF` is the communicator that contains only the current process.
- `MPI_COMM_NULL` is the invalid communicator. Routines that construct communicators can give this as result if an error occurs.

These values are constants, though not necessarily compile-time constants. Thus, they can not be used in switch statements, array declarations, or `constexpr` evaluations.

If you don't want to write `MPI_COMM_WORLD` repeatedly, you can assign that value to a variable of type `MPI_Comm`.

Examples:

```
// C:
#include <mpi.h>
MPI_Comm comm = MPI_COMM_WORLD;
```

```
!! Fortran 2008 interface
use mpi_f08
Type(MPI_Comm) :: comm = MPI_COMM_WORLD
```

Figure 7.1 MPI_Comm_dup

Name	Param name	C type	F type	inout
mpi_comm_dup (				
	comm *communicator*	MPI_Comm	TYPE(MPI_Comm)	in
	newcomm *copy of comm*	MPI_Comm*	TYPE(MPI_Comm)	out
(opt)	ierror		INTEGER	out
)				

Python:

Name	Param name	param type
Comm.Dup(		
	self	
	info	Optional[Info] = None
) -> Comm		

```
!! Fortran legacy interface
#include <mpif.h>
Integer :: comm = MPI_COMM_WORLD
```

Python note.

```
comm = MPI.COMM_WORLD
```

MPL note 52. The `environment` namespace has the equivalents of `MPI_COMM_WORLD` and `MPI_COMM_SELF`:

```
const communicator& mpl::environment::comm_world();
const communicator& mpl::environment::comm_self();
```

There doesn't seem to be an equivalent of `MPI_COMM_NULL`.

End of MPL note

You can name your communicators with `MPI_Comm_set_name`, which could improve the quality of error messages when they arise.

7.2 Duplicating communicators

With `MPI_Comm_dup` (figure 7.1) you can make an exact duplicate of a communicator. There is a nonblocking variant `MPI_Comm_idup` (figure 7.2).

These calls do not propagate info hints (sections 14.1.1 and 14.1.1.2); to achieve this, use `MPI_Comm_dup_with_info` and `MPI_Comm_idup_with_info`.

MPL note 53. Communicators can be duplicated but only during initialization. Copy assignment has been deleted. Thus:

```
// LEGAL:
mpl::communicator init = comm;
```

Figure 7.2 `MPI_Comm_idup`

Name	Param name	C type	F type	inout
`mpi_comm_idup (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`newcomm` *copy of comm*	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
	`request` *communication request*	`MPI_Request*`	`TYPE(MPI_Request)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Idup(`		
	`self`	
`) -> Tuple[Comm, Request]`		

```
// WRONG:
mpl::communicator init;
init = comm;
```

End of MPL note

7.2.1 Communicator comparing

You may wonder what 'an exact copy' means precisely. For this, think of a communicator as a context label that you can attach to, among others, operations such as sends and receives. A send and a receive 'belong together' if they have the same communicator context. Conversely, a send in one communicator can not be matched to a receive in a duplicate communicator, made by `MPI_Comm_dup`.

Testing whether two communicators are really the same is then more than testing if they comprise the same processes. The call `MPI_Comm_compare` returns `MPI_IDENT` if two communicator values are the same, and not if one is derived from the other by duplication:

```
// commcompare.c
int result;
MPI_Comm copy = comm;
MPI_Comm_compare(comm,copy,&result);
printf("assign:    comm==copy: %d \n",
    result==MPI_IDENT);
printf("            congruent: %d \n",
    result==MPI_CONGRUENT);
printf("            not equal: %d \n",
    result==MPI_UNEQUAL);

MPI_Comm_dup(comm,&copy);
MPI_Comm_compare(comm,copy,&result);
printf("duplicate: comm==copy: %d \n",
    result==MPI_IDENT);
printf("            congruent: %d \n",
    result==MPI_CONGRUENT);
printf("            not equal: %d \n",
    result==MPI_UNEQUAL);
```

```
Output:                                                  not equal: 0
                                              duplicate: comm==copy: 0
assign:    comm==copy: 1                                 congruent: 1
            congruent: 0                                 not equal: 0
```

Communicators that are not actually the same can be

- consisting of the same processes, in the same order, giving `MPI_CONGRUENT`;
- merely consisting of the same processes, but not in the same order, giving `MPI_SIMILAR`;
- different, giving `MPI_UNEQUAL`.

Comparing against `MPI_COMM_NULL` is not allowed.

7.2.2 Communicator duplication for library use

Duplicating a communicator may seem pointless, but it is actually very useful for the design of software libraries. Image that you have a code

```
MPI_Isend(...); MPI_Irecv(...);
// library call
MPI_Waitall(...);
```

and suppose that the library has receive calls. Now it is possible that the receive in the library inadvertently catches the message that was sent in the outer environment.

In section 14.6 it was explained that MPI messages are nonovertaking. This may lead to confusing situations, witness the following. First of all, here is code where the library stores the communicator of the calling program:

```
// commdupwrong.cxx
class library {
private:
  MPI_Comm comm;
  int procno,nprocs,other;
  MPI_Request request[2];
public:
  library(MPI_Comm incomm) {
    comm = incomm;
    MPI_Comm_rank(comm,&procno);
    other = 1-procno;
  };
  int communication_start();
  int communication_end();
};
```

For the full source of this example, see section 7.8.2

This models a main program that does a simple message exchange, and it makes two calls to library routines. Unbeknown to the user, the library also issues send and receive calls, and they turn out to interfere.

Here

- The main program does a send,
- the library call `function_start` does a send and a receive; because the receive can match either send, it is paired with the first one;
- the main program does a receive, which will be paired with the send of the library call;
- both the main program and the library do a wait call, and in both cases all requests are successfully fulfilled, just not the way you intended.

To prevent this confusion, the library should duplicate the outer communicator with `MPI_Comm_dup` and send all messages with respect to its duplicate. Now messages from the user code can never reach the library software, since they are on different communicators.

```
// commdupright.cxx
class library {
private:
  MPI_Comm comm;
  int procno,nprocs,other;
  MPI_Request request[2];
public:
  library(MPI_Comm incomm) {
    MPI_Comm_dup(incomm,&comm);
    MPI_Comm_rank(comm,&procno);
    other = 1-procno;
  };
  ~library() {
    MPI_Comm_free(&comm);
  }
  int communication_start();
  int communication_end();
};
```

For the full source of this example, see section 7.8.3

Note how the preceding example performs the `MPI_Comm_free` cal in a C++ *destructor*.

```
## commdup.py
class Library():
    def __init__(self,comm):
        # wrong: self.comm = comm
        self.comm = comm.Dup()
        self.other = self.comm.Get_size()-self.comm.Get_rank()-1
        self.requests = [ None ] * 2
    def communication_start(self):
        sendbuf = np.empty(1,dtype=np.int); sendbuf[0] = 37
        recvbuf = np.empty(1,dtype=np.int)
        self.requests[0] = self.comm.Isend( sendbuf, dest=other,tag=2 )
        self.requests[1] = self.comm.Irecv( recvbuf, source=other )
    def communication_end(self):
        MPI.Request.Waitall(self.requests)

mylibrary = Library(comm)
my_requests[0] = comm.Isend( sendbuffer,dest=other,tag=1 )
mylibrary.communication_start()
my_requests[1] = comm.Irecv( recvbuffer,source=other )
MPI.Request.Waitall(my_requests,my_status)
```

```
mylibrary.communication_end()
```

For the full source of this example, see section 7.8.4

7.3 Sub-communicators

In many scenarios you divide a large job over all the available processors. However, your job may have two or more parts that can be considered as jobs by themselves. In that case it makes sense to divide your processors into subgroups accordingly.

Suppose for instance that you are running a simulation where inputs are generated, a computation is performed on them, and the results of this computation are analyzed or rendered graphically. You could then consider dividing your processors in three groups corresponding to generation, computation, rendering. As long as you only do sends and receives, this division works fine. However, if one group of processes needs to perform a collective operation, you don't want the other groups involved in this. Thus, you really want the three groups to be really distinct from each other.

In order to make such subsets of processes, MPI has the mechanism of taking a subset of `MPI_COMM_WORLD` and turning that subset into a new communicator.

Now you understand why the MPI collective calls had an argument for the communicator: a collective involves all processes of that communicator. By making a communicator that contains a subset of all available processes, you can do a collective on that subset.

The usage is as follows:

- You create a new communicator with `MPI_Comm_dup` (section 7.2), `MPI_Comm_split` (section 7.4), `MPI_Comm_create` (section 7.5), `MPI_Intercomm_create` (section 7.6), `MPI_Comm_spawn` (section 8.1);
- you use that communiator for a while;
- and you call `MPI_Comm_free` when you are done with it; this also sets the communicator variable to `MPI_COMM_NULL`.

7.3.1 Scenario: distributed linear algebra

For *scalability* reasons, matrices should often be distributed in a 2D manner, that is, each process receives a subblock that is not a block of columns or rows. This means that the processors themselves are, at least logically, organized in a 2D grid. Operations then involve reductions or broadcasts inside rows or columns. For this, a row or column of processors needs to be in a subcommunicator.

7.3.2 Scenario: climate model

A climate simulation code has several components, for instance corresponding to land, air, ocean, and ice. You can imagine that each needs a different set of equations and algorithms to simulate. You can then divide your processes, where each subset simulates one component of the climate, occasionally communicating with the other components.

Figure 7.3 `MPI_Comm_split`

Name	Param name	C type	F type	inout
`mpi_comm_split (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`color` *control of subset assignment*	`int`	`INTEGER`	`in`
	`key` *control of rank assignment*	`int`	`INTEGER`	`in`
	`newcomm` *new communicator*	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Split(`		
	`self`	
	`color`	`int = 0`
	`key`	`int = 0`
`) -> Comm`		

7.3.3 Scenario: quicksort

The popular quicksort algorithm works by splitting the data into two subsets that each can be sorted individually. If you want to sort in parallel, you could implement this by making two subcommunicators, and sorting the data on these, creating recursively more subcommunicators.

7.3.4 Shared memory

There is an important application of communicator splitting in the context of one-sided communication, grouping processes by whether they access the same shared memory area; see section 12.1.

7.3.5 Process spawning

Finally, newly created communicators do not always need to be subset of the initial `MPI_COMM_WORLD`. MPI can dynamically spawn new processes (see chapter 8) which start in a `MPI_COMM_WORLD` of their own. However, another communicator will be created that spawns the old and new worlds so that you can communicate with the new processes.

7.4 Splitting a communicator

Above we saw several scenarios where it makes sense to divide `MPI_COMM_WORLD` into disjoint subcommunicators. The command `MPI_Comm_split` (figure 7.3) uses a 'colour' to define these subcommunicators: all processes in the old communicator with the same colour wind up in a new communicator together. The old communicator still exists, so processes now have two different contexts in which to communicate.

The ranking of processes in the new communicator is determined by a 'key' value. Most of the time, there is no reason to use a relative ranking that is different from the global ranking, so the `MPI_Comm_rank` value of the global communicator is a good choice.

Here is one example of communicator splitting. Suppose your processors are in a two-dimensional grid:

```
MPI_Comm_rank( MPI_COMM_WORLD, &mytid );
proc_i = mytid % proc_column_length;
proc_j = mytid / proc_column_length;
```

You can now create a communicator per column:

```
MPI_Comm column_comm;
MPI_Comm_split( MPI_COMM_WORLD, proc_j, mytid, &column_comm );
```

and do a broadcast in that column:

```
MPI_Bcast( data, /* tag: */ 0, column_comm );
```

Because of the SPMD nature of the program, you are now doing in parallel a broadcast in every processor column. Such operations often appear in *dense linear algebra*.

The `MPI_Comm_split` routine has a 'key' parameter, which controls how the processes in the new communicator are ordered. By supplying the rank from the original communicator you let them be arranged in the same order.

Python note. In Python, the 'key' argument is optional:

```
## commsplit.py
mydata = procid

# communicator modulo 2
color = procid%2
mod2comm = comm.Split(color)
new_procid = mod2comm.Get_rank()

# communicator modulo 4 recursively
color = new_procid%2
mod4comm = mod2comm.Split(color)
new_procid = mod4comm.Get_rank()
```

For the full source of this example, see section 7.8.5

MPL note 54. In MPL, splitting a communicator is done as one of the overloads of the communicator constructor;

```
// commsplit.cxx
// create sub communicator modulo 2
int color2 = procno % 2;
mpl::communicator comm2( mpl::communicator::split(), comm_world, color2 );
auto procno2 = comm2.rank();

// create sub communicator modulo 4 recursively
int color4 = procno2 % 2;
mpl::communicator comm4( mpl::communicator::split(), comm2, color4 );
auto procno4 = comm4.rank();
```

For the full source of this example, see section 7.8.6

MPL implementation note: The `communicator::split` identifier itself is an otherwise empty subclass of `communicator`.

End of MPL note

There is also a routine `MPI_Comm_split_type` which uses a type rather than a key to split the communicator. We will see this in action in section 12.1.

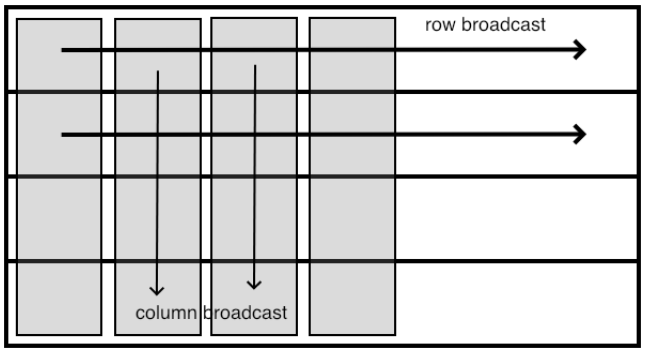


Figure 7.1: Row and column broadcasts in subcommunicators

7.4.1 Examples

One application of communicator splitting is setting up a processor grid, with the possibility of using MPI solely within one row or column; see figure 7.1.

Exercise 7.1. Organize your processes in a grid, and make subcommunicators for the rows and columns. For this compute the row and column number of each process.
In the row and column communicator, compute the rank. For instance, on a 2×3 processor grid you should find:

```
Global ranks:  Ranks in row:  Ranks in colum:
  0  1  2       0  1  2        0  0  0
  3  4  5       0  1  2        1  1  1
```

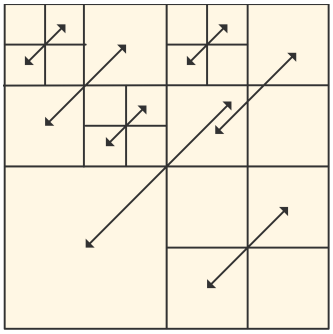
Check that the rank in the row communicator is the column number, and the other way around.
Run your code on different number of processes, for instance a number of rows and columns that is a power of 2, or that is a prime number.
(There is a skeleton for this exercise under the name `procgrid`.)

As another example of communicator splitting, consider the recursive algorithm for *matrix transposition*. Processors are organized in a square grid. The matrix is divided on 2×2 block form.

Exercise 7.2. Implement a recursive algorithm for matrix transposition:

Figure 7.4 `MPI_Comm_group`

Name	Param name	C type	F type	inout
`mpi_comm_group (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`group` *group corresponding to comm*	`MPI_Group*`	`TYPE(MPI_Group)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				



- Swap blocks $(1, 2)$ and $(2, 1)$; then
- Divide the processors into four subcommunicators, and apply this algorithm recursively on each;
- If the communicator has only one process, transpose the matrix in place.

7.5 Communicators and groups

You saw in section 7.4 that it is possible derive communicators that have a subset of the processes of another communicator. There is a more general mechanism, using `MPI_Group` objects.

Using groups, it takes three steps to create a new communicator:

1. Access the `MPI_Group` of a communicator object using `MPI_Comm_group` (figure 7.4).
2. Use various routines, discussed next, to form a new group.
3. Make a new communicator object from the group with `MPI_Group`, using `MPI_Comm_create` (figure 7.5)

Creating a new communicator from a group is collective on the old communicator. There is also a routine `MPI_Comm_create_group` that only needs to be called on the group that constitutes the new communicator.

7.5.1 Process groups

Groups are manipulated with `MPI_Group_incl` (figure 7.6), `MPI_Group_excl` (figure 7.7), `MPI_Group_difference` and a few more.

```
MPI_Comm_group (comm, group, ierr)
MPI_Comm_create (MPI_Comm comm,MPI_Group group, MPI_Comm newcomm, ierr)
```

Figure 7.5 `MPI_Comm_create`

Name	Param name	C type	F type	inout
`mpi_comm_create (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`group` *group, which is a subset of the group of comm*	`MPI_Group`	`TYPE(MPI_Group)`	`in`
	`newcomm` *new communicator*	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Create(`		
	`self`	
	`group`	`Group`
`) -> Comm`		

Figure 7.6 `MPI_Group_incl`

Name	Param name	C type	F type	inout
`mpi_group_incl (`				
	`group` *group*	`MPI_Group`	`TYPE(MPI_Group)`	`in`
	`n` *number of elements in array ranks (and size of newgroup)*	`int`	`INTEGER`	`in`

	`ranks`
	`length: n`
	ranks of processes in group to appear in newgroup
		`const int[]`	`INTEGER`	`in`

	`newgroup` *new group derived from above, in the order defined by ranks*	`MPI_Group*`	`TYPE(MPI_Group)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Group.Incl(`		
	`self`	
	`ranks`	`Sequence[int]`
`) -> Group`		

```
MPI_Group_union(group1, group2, newgroup, ierr)
MPI_Group_intersection(group1, group2, newgroup, ierr)
MPI_Group_difference(group1, group2, newgroup, ierr)
```

```
MPI_Group_size(group, size, ierr)
MPI_Group_rank(group, rank, ierr)
```

Figure 7.7 `MPI_Group_excl`

Name	Param name	C type	F type	inout
`mpi_group_excl (`				
	`group` *group*	`MPI_Group`	`TYPE(MPI_Group)`	`in`
	`n` *number of elements in array ranks*	`int`	`INTEGER`	`in`

	`ranks`
	`length: n`
	array of integer ranks of processes in group not to appear in newgroup
		`const int[]`	`INTEGER`	`in`

	`newgroup` *new group derived from above, preserving the order defined by group*	`MPI_Group*`	`TYPE(MPI_Group)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Group.Excl(`		
	`self`	
	`ranks`	`Sequence[int]`
`) -> Group`		

7.5.2 Example

Suppose you want to split the world communicator into one manager process, with the remaining processes workers.

```
// portapp.c
MPI_Comm comm_work;
{
  MPI_Group group_world,group_work;
  MPI_Comm_group( comm_world,&group_world );
  int manager[] = {0};
  MPI_Group_excl( group_world,1,manager,&group_work );
  MPI_Comm_create( comm_world,group_work,&comm_work );
  MPI_Group_free( &group_world ); MPI_Group_free( &group_work );
}
```

7.6 Inter-communicators

In several scenarios it may be desirable to have a way to communicate between communicators. For instance, an application can have clearly functionally separated modules (preprocessor, simulation, postprocessor) that need to stream data pairwise. In another example, dynamically spawned processes (section 8.1) get their own value of `MPI_COMM_WORLD`, but still need to communicate with the process(es) that spawned them. In this section we will discuss the *inter-communicator* mechanism that serves such use cases.

Communicating between disjoint communicators can of course be done by having a communicator that overlaps them, but this would be complicated: since the 'inter' communication happens in the overlap

Figure 7.8 `MPI_Intercomm_create`

Name	Param name	C type	F type	inout
`mpi_intercomm_create (`				
	`local_comm` *local intra-communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`local_leader` *rank of local group leader in local_comm*	`int`	`INTEGER`	`in`
	`peer_comm` *"peer" communicator; significant only at the local_leader*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`remote_leader` *rank of remote group leader in peer_comm; significant only at the local_leader*	`int`	`INTEGER`	`in`
	`tag` *tag*	`int`	`INTEGER`	`in`
	`newintercomm` *new inter-communicator*	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

communicator, you have to translate its ordering into those of the two worker communicators. It would be easier to express messages directly in terms of those communicators, and this is what happens in an *inter-communicator*.

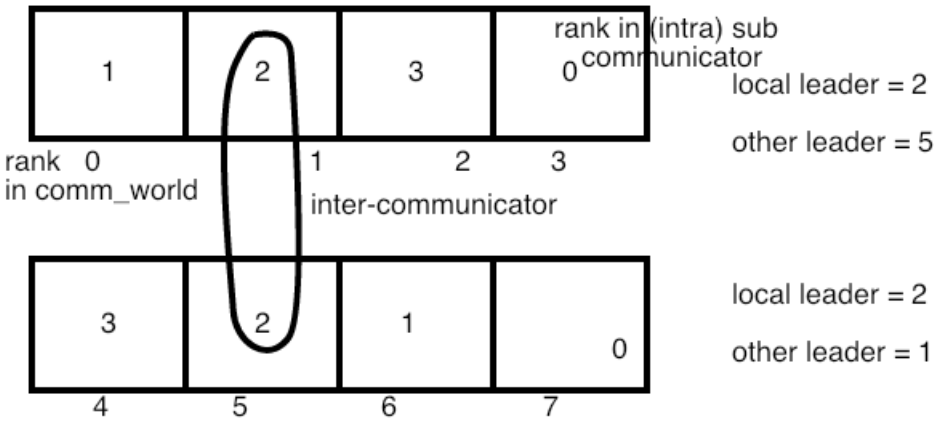


Figure 7.2: Illustration of ranks in an inter-communicator setup

A call to `MPI_Intercomm_create` (figure 7.8) involves the following communicators:

- Two local communicators, which in this context are known as *intra-communicator*s: one process in each will act as the local leader, connected to the remote leader;
- The *peer communicator*, often `MPI_COMM_WORLD`, that contains the local communicators;
- An *inter-communicator* that allows the leaders of the subcommunicators to communicate with the other subcommunicator.

Even though the intercommunicator connects only two proceses, it is collective on the peer communicator.

7.6.1 Inter-communicator point-to-point

The local leaders can now communicate with each other.

- The sender specifies as target the local number of the other leader in the other sub-communicator;
- Likewise, the receiver specifies as source the local number of the sender in its sub-communicator.

In one way, this design makes sense: processors are referred to in their natural, local, numbering. On the other hand, it means that each group needs to know how the local ordering of the other group is arranged. Using a complicated *key* value makes this difficult.

```
if (i_am_local_leader) {
  if (color==0) {
    interdata = 1.2;
    int inter_target = local_number_of_other_leader;
    printf("[%d] sending interdata %e to %d\n",
             procno,interdata,inter_target);
    MPI_Send(&interdata,1,MPI_DOUBLE,inter_target,0,intercomm);
  } else {
    MPI_Status status;
    MPI_Recv(&interdata,1,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,intercomm,&status);
    int inter_source = status.MPI_SOURCE;
    printf("[%d] received interdata %e from %d\n",
             procno,interdata,inter_source);
    if (inter_source!=local_number_of_other_leader)
        fprintf(stderr,
                "Got inter communication from unexpected %d; s/b %d\n",
                inter_source,local_number_of_other_leader);
  }
}
```

For the full source of this example, see section 7.8.7

7.6.2 Inter-communicator collectives

The intercommunicator can be used in collectives such as a broadcast.

- In the sending group, the root process passes `MPI_ROOT` as 'root' value; all others use `MPI_PROC_NULL` .
- In the receiving group, all processes use a 'root' value that is the rank of the root process in the root group. Note: this is not the global rank!

Gather and scatter behave similarly; the allgather is different: all send buffers of group A are concatenated in rank order, and places on all processes of group B.

Inter-communicators can be used if two groups of process work asynchronously with respect to each other; another application is fault tolerance (section 14.4).

```
if (color==0) { // sending group: the local leader sends
  if (i_am_local_leader)
    root = MPI_ROOT;
  else
    root = MPI_PROC_NULL;
} else { // receiving group: everyone indicates leader of other group
  root = local_number_of_other_leader;
}
if (DEBUG) fprintf(stderr,"[%d] using root value %d\n",procno,root);
MPI_Bcast(&bcast_data,1,MPI_INT,root,intercomm);
```

Figure 7.9 `MPI_Comm_get_parent`

Name	Param name	C type	F type	inout
`mpi_comm_get_parent (`				
	`parent` *the parent communicator*	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Get_parent(`		
	`cls`	
`) -> Intercomm`		

Figure 7.10 `MPI_Comm_test_inter`

Name	Param name	C type	F type	inout
`mpi_comm_test_inter (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`flag` *true if comm is an inter-communicator*	`int*`	`LOGICAL`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

For the full source of this example, see section 7.8.7

7.6.3 Inter-communicator querying

Some of the operations you have seen before for *intra-communicator*s behave differently with inter-communicator:

- `MPI_Comm_size` returns the size of the local group, not the size of the inter-communicator.
- `MPI_Comm_rank` returns the rank in the local group.
- `MPI_Comm_group` returns the local group.

Spawned processes can find their parent communicator with `MPI_Comm_get_parent` (figure 7.9) (see examples in section 8.1). On other processes this returns `MPI_COMM_NULL`.

Test whether a communicator is intra or inter: `MPI_Comm_test_inter` (figure 7.10).

`MPI_Comm_compare` works for inter-communicators.

Processes connected through an intercommunicator can query the size of the 'other' communicator with `MPI_Comm_remote_size` (figure 7.11). The actual group can be obtained with `MPI_Comm_remote_group` (figure 7.12).

Virtual topologies (chapter 11) cannot be created with an intercommunicator. To set up virtual topologies, first transform the intercommunicator to an intracommunicator with the function `MPI_Intercomm_merge` (figure 7.13).

Figure 7.11 `MPI_Comm_remote_size`

Name	Param name	C type	F type	inout
`mpi_comm_remote_size (`				
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	inter-communicator			
	`size`	`int*`	`INTEGER`	`out`
	number of processes in the remote group of comm			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

```
Intercomm.Get_remote_size(self)
```

Figure 7.12 `MPI_Comm_remote_group`

Name	Param name	C type	F type	inout
`mpi_comm_remote_group (`				
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	inter-communicator			
	`group`	`MPI_Group*`	`TYPE(MPI_Group)`	`out`
	remote group corresponding to comm			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

```
Intercomm.Get_remote_group(self)
```

Figure 7.13 `MPI_Intercomm_merge`

Name	Param name	C type	F type	inout
`mpi_intercomm_merge (`				
	`intercomm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	inter-communicator			
	`high`	`int`	`LOGICAL`	`in`
	ordering of the local and remote groups in the new intra-communicator			
	`newintracomm`	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
	new intra-communicator			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Intercomm.Merge(`		
	`self`	
	`high`	`bool = False`
`) -> Intracomm`		

7.7 Review questions

For all true/false questions, if you answer that a statement is false, give a one-line explanation.

1. True or false: in each communicator, processes are numbered consecutively from zero.
2. If a process is in two communicators, it has the same rank in both.
3. Any communicator that is not `MPI_COMM_WORLD` is a strict subset of it.
4. The subcommunicators derived by `MPI_Comm_split` are disjoint.
5. If two processes have ranks $p < q$ in some communicator, and they are in the same subcommunicator, then their ranks p', q' in the subcommunicator also obey $p' < q'$.

7.8 Sources used in this chapter

7.8.1 Listing of code header

7.8.2 Listing of code examples/mpi/c/commdupwrong.cxx

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

class library {
private:
  MPI_Comm comm;
  int procno,nprocs,other;
  MPI_Request request[2];
public:
  library(MPI_Comm incomm) {
    comm = incomm;
    MPI_Comm_rank(comm,&procno);
    other = 1-procno;
  };
  int communication_start();
  int communication_end();
};

int main(int argc,char **argv) {

#include "globalinit.c"

  int other = 1-procno, sdata=5,rdata;
  MPI_Request request[2];
  MPI_Status status[2];

  if (procno>1) { MPI_Finalize(); return 0; }

  library my_library(comm);
  MPI_Isend(&sdata,1,MPI_INT,other,1,comm,&(request[0]));
  my_library.communication_start();
  MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,&(request[1]));
  MPI_Waitall(2,request,status);
  my_library.communication_end();

  if (status[1].MPI_TAG==2)
    printf("wrong!\n");

  MPI_Finalize();
  return 0;
}

int library::communication_start() {
```

```
  int sdata=6,rdata;
  MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0]));
  MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,
    comm,&(request[1]));
  return 0;
}

int library::communication_end() {
  MPI_Status status[2];
  MPI_Waitall(2,request,status);
  return 0;
}
```

7.8.3 Listing of code examples/mpi/c/commdupright.cxx

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

class library {
private:
  MPI_Comm comm;
  int procno,nprocs,other;
  MPI_Request request[2];
public:
  library(MPI_Comm incomm) {
    MPI_Comm_dup(incomm,&comm);
    MPI_Comm_rank(comm,&procno);
    other = 1-procno;
  };
  ~library() {
    MPI_Comm_free(&comm);
  }
  int communication_start();
  int communication_end();
};

int main(int argc,char **argv) {

#include "globalinit.c"

  int other = 1-procno, sdata=5,rdata;
  MPI_Request request[2];
  MPI_Status status[2];

  if (procno>1) { MPI_Finalize(); return 0; }

  library my_library(comm);
  MPI_Isend(&sdata,1,MPI_INT,other,1,comm,&(request[0]));
  my_library.communication_start();
```

```
  MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,
    comm,&(request[1]));
  MPI_Waitall(2,request,status);
  my_library.communication_end();

  if (status[1].MPI_TAG==2)
    printf("wrong!\n");

  MPI_Finalize();
  return 0;
}

int library::communication_start() {
  int sdata=6,rdata, ierr;
  MPI_Isend(&sdata,1,MPI_INT,other,2,comm,&(request[0]));
  MPI_Irecv(&rdata,1,MPI_INT,other,MPI_ANY_TAG,comm,&(request[1]));
  return 0;
}

int library::communication_end() {
  MPI_Status status[2];
  int ierr;
  ierr = MPI_Waitall(2,request,status); CHK(ierr);
  return 0;
}
```

7.8.4 Listing of code examples/mpi/p/commdup.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

other = nprocs-procid-1
my_requests = [ None ] * 2
my_status = [ MPI.Status() ] * 2
sendbuffer = np.empty(1,dtype=np.int)
recvbuffer = np.empty(1,dtype=np.int)

class Library():
    def __init__(self,comm):
        # wrong: self.comm = comm
        self.comm = comm.Dup()
        self.other = self.comm.Get_size()-self.comm.Get_rank()-1
        self.requests = [ None ] * 2
```

```
    def communication_start(self):
        sendbuf = np.empty(1,dtype=np.int); sendbuf[0] = 37
        recvbuf = np.empty(1,dtype=np.int)
        self.requests[0] = self.comm.Isend( sendbuf, dest=other,tag=2 )
        self.requests[1] = self.comm.Irecv( recvbuf, source=other )
    def communication_end(self):
        MPI.Request.Waitall(self.requests)

mylibrary = Library(comm)
my_requests[0] = comm.Isend( sendbuffer,dest=other,tag=1 )
mylibrary.communication_start()
my_requests[1] = comm.Irecv( recvbuffer,source=other )
MPI.Request.Waitall(my_requests,my_status)
mylibrary.communication_end()

if my_status[1].Get_tag()==2:
    print("Caught wrong message!")
```

7.8.5 Listing of code examples/mpi/p/commsplit.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<4:
    prin( "Need 4 procs at least")
    sys.exit(1)

mydata = procid

# communicator modulo 2
color = procid%2
mod2comm = comm.Split(color)
new_procid = mod2comm.Get_rank()

# communicator modulo 4 recursively
color = new_procid%2
mod4comm = mod2comm.Split(color)
new_procid = mod4comm.Get_rank()

if mydata/4!=new_procid:
    print("Error",procid,new_procid,mydata/4)

if procid==0:
    print("Finished")
```

7.8.6 Listing of code examples/mpi/mpl/commsplit.cxx

```
#include <iostream>
```

```
using std::cout;
using std::endl;
#include <mpl/mpl.hpp>

int main(int argc,char **argv) {

  const mpl::communicator &comm_world=mpl::environment::comm_world();
  auto procno = comm_world.rank();
  auto nprocs = comm_world.size();

  // create sub communicator modulo 2
  int color2 = procno % 2;
  mpl::communicator comm2( mpl::communicator::split(), comm_world, color2 );
  auto procno2 = comm2.rank();

  // create sub communicator modulo 4 recursively
  int color4 = procno2 % 2;
  mpl::communicator comm4( mpl::communicator::split(), comm2, color4 );
  auto procno4 = comm4.rank();

  int mod4ranks[nprocs];
  comm_world.gather( 0, procno4,mod4ranks );
  if (procno==0) {
    cout << "Ranks mod 4:";
    for (int ip=0; ip<nprocs; ip++)
      cout << " " << mod4ranks[ip];
    cout << endl;
  }

  if (procno/4!=procno4)
    printf("Error %d %d\n",procno,procno4);

  if (procno==0)
    printf("Finished\n");

  return 0;
}
```

7.8.7 Listing of code examples/mpi/c/intercomm.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#ifndef DEBUG
#define DEBUG 0
#endif

int main(int argc,char **argv) {

#include "globalinit.c"
```

```
if (nprocs<4) {
  fprintf(stderr,"This program needs at least four processes\n");
  return -1;
}
if (nprocs%2>0) {
  fprintf(stderr,"This program needs an even number of processes\n");
  return -1;
}

int color,colors=2;
MPI_Comm split_half_comm;

int mydata = procno;
// create sub communicator first & second half
color = procno<nprocs/2 ? 0 : 1;
int key=procno;
if (color==0)
  // first half rotated
  key = ( procno+1 ) % (nprocs/2);
else
  // second half numbered backwards
  key = nprocs-procno;
MPI_Comm_split(MPI_COMM_WORLD,color,key,&split_half_comm);
int sub_procno,sub_nprocs;
MPI_Comm_rank(split_half_comm,&sub_procno);
MPI_Comm_size(split_half_comm,&sub_nprocs);
if (DEBUG) fprintf(stderr,"[%d] key=%d local rank: %d\n",procno,key,sub_procno);

int
  local_leader_in_inter_comm
  = color==0 ? 2 : (sub_nprocs-2)
  ,
  local_number_of_other_leader
  = color==1 ? 2 : (sub_nprocs-2)
  ;

if (local_leader_in_inter_comm<0 || local_leader_in_inter_comm>=sub_nprocs) {
  fprintf(stderr,
  "[%d] invalid local member: %d\n",
  procno,local_leader_in_inter_comm);
  MPI_Abort(2,comm);
}
int
  global_rank_of_other_leader =
  1 + ( procno<nprocs/2 ? nprocs/2 : 0 )
  ;

int
  i_am_local_leader = sub_procno==local_leader_in_inter_comm,
  inter_tag = 314;
if (i_am_local_leader)
  fprintf(stderr,"[%d] creating intercomm with %d\n",
```

```
   procno,global_rank_of_other_leader);
  MPI_Comm intercomm;
  MPI_Intercomm_create
    (/* local_comm:       */ split_half_comm,
     /* local_leader:     */ local_leader_in_inter_comm,
     /* peer_comm:        */ MPI_COMM_WORLD,
     /* remote_peer_rank: */ global_rank_of_other_leader,
     /* tag:              */ inter_tag,
     /* newintercomm:     */ &intercomm );
  if (DEBUG) fprintf(stderr,"[%d] intercomm created.\n",procno);

  if (i_am_local_leader) {
    int inter_rank,inter_size;
    MPI_Comm_size(intercomm,&inter_size);
    MPI_Comm_rank(intercomm,&inter_rank);
    if (DEBUG) fprintf(stderr,"[%d] inter rank/size: %d/%d\n",procno,inter_rank,inter_size);
  }

  double interdata=0.;
  if (i_am_local_leader) {
    if (color==0) {
      interdata = 1.2;
      int inter_target = local_number_of_other_leader;
      printf("[%d] sending interdata %e to %d\n",
     procno,interdata,inter_target);
      MPI_Send(&interdata,1,MPI_DOUBLE,inter_target,0,intercomm);
    } else {
      MPI_Status status;
      MPI_Recv(&interdata,1,MPI_DOUBLE,MPI_ANY_SOURCE,MPI_ANY_TAG,intercomm,&status);
      int inter_source = status.MPI_SOURCE;
      printf("[%d] received interdata %e from %d\n",
     procno,interdata,inter_source);
      if (inter_source!=local_number_of_other_leader)
fprintf(stderr,
"Got inter communication from unexpected %d; s/b %d\n",
inter_source,local_number_of_other_leader);
    }
  }

  int root; int bcast_data = procno;
  if (color==0) { // sending group: the local leader sends
    if (i_am_local_leader)
      root = MPI_ROOT;
    else
      root = MPI_PROC_NULL;
  } else { // receiving group: everyone indicates leader of other group
    root = local_number_of_other_leader;
  }
  if (DEBUG) fprintf(stderr,"[%d] using root value %d\n",procno,root);
  MPI_Bcast(&bcast_data,1,MPI_INT,root,intercomm);
  fprintf(stderr,"[%d] bcast data: %d\n",procno,bcast_data);

  if (procno==0)
```

```
    fprintf(stderr,"Finished\n");

  MPI_Finalize();
  return 0;
}
```

Chapter 8

MPI topic: Process management

In this course we have up to now only considered the SPMD model of running MPI programs. In some rare cases you may want to run in an MPMD mode, rather than SPMD. This can be achieved either on the OS level, using options of the *mpiexec* mechanism, or you can use MPI's built-in process management. Read on if you're interested in the latter.

8.1 Process spawning

The first version of MPI did not contain any process management routines, even though the earlier *PVM* project did have that functionality. Process management was later added with MPI-2.

Unlike what you might think, newly added processes do not become part of `MPI_COMM_WORLD`; rather, they get their own communicator, and an *inter-communicator* (section 7.6) is established between this new group and the existing one. The first routine is `MPI_Comm_spawn` (figure 8.1), which tries to fire up multiple copies of a single named executable. Errors in starting up these codes are returned in an array of integers, or if you're feeling sure of yourself, specify `MPI_ERRCODES_IGNORE`.

It is not immediately clear whether there is opportunity for spawning new executables; after all, `MPI_COMM_WORLD` contains all your available processors. You can probably tell your job starter to reserve space for a few extra processes, but that is installation-dependent (see below). However, there is a standard mechanism for querying whether such space has been reserved. The attribute `MPI_UNIVERSE_SIZE`, retrieved with `MPI_Comm_get_attr` (section 14.1.2), will tell you to the total number of hosts available.

If this option is not supported, you can determine yourself how many processes you want to spawn. If you exceed the hardware resources, your multi-tasking operating system (which is some variant of Unix for almost everyone) will use *time-slicing* to start the spawned processes, but you will not gain any performance.

Here is an example of a work manager. First we query how much space we have for new processes:

```
// spawnmanager.c
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI_Comm_rank(MPI_COMM_WORLD, &manager_rank);

err = MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
```

Figure 8.1 `MPI_Comm_spawn`

Name	Param name	C type	F type	inout
`mpi_comm_spawn (`				
	`command`	`const char*`	`CHARACTER`	`in`
	name of program to be spawned			
	`argv`	`char[]*`	`CHARACTER`	`in`
	arguments to command			
	`maxprocs`	`int`	`INTEGER`	`in`
	maximum number of processes to start			
	`info`	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	a set of key-value pairs telling the runtime system where and how to start the processes			
	`root`	`int`	`INTEGER`	`in`
	rank of process in which previous arguments are examined			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	intra-communicator containing group of spawning processes			
	`intercomm`	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
	inter-communicator between original group and the newly spawned group			
	`array_of_errcodes`	`int[]`	`INTEGER`	`out`
	`length:`			
	one code per process (array of integer)			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Intracomm.Spawn(`		
	`self`	
	`command`	`str`
	`args`	`Optional[Sequence[str]] = None`
	`maxprocs`	`int = 1`
	`info`	`Info = Ellipsis`
	`root`	`int = 0`
	`errcodes`	`Optional[list] = None`
`) -> Intercomm`		

```
                (void*)&universe_sizep, &flag);
  universe_size = *universe_sizep;
```

For the full source of this example, see section 8.5.2

Then we actually spawn them:

```
int nworkers = universe_size-world_size;
const char *worker_program = "spawnworker";
int errorcodes[nworkers];
MPI_Comm inter_to_workers;           /* intercommunicator */
MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, nworkers,
                MPI_INFO_NULL, 0, MPI_COMM_WORLD, &inter_to_workers,
                errorcodes);
```

For the full source of this example, see section 8.5.2

```
## spawnmanager.py
try :
    universe_size = comm.Get_attr(MPI.UNIVERSE_SIZE)
    if universe_size is None:
```

```
        print("Universe query returned None")
        universe_size = nprocs + 4
    else:
        print("World has {} ranks in a universe of {}"\
              .format(nprocs,universe_size))
except :
    print("Exception querying universe size")
    universe_size = nprocs + 4
nworkers = universe_size - nprocs

itercomm = comm.Spawn("./spawn_worker.py", maxprocs=nworkers)
```

For the full source of this example, see section 8.5.3

You could start up a single copy of this program with

```
mpirun -np 1 spawnmanager
```

but with a hostfile that has more than one host.

TACC note. *Intel MPI* requires you to pass an option `-usize` to `mpiexec` indicating the size of the comm universe. With the TACC jobs starter `ibrun` do the following:

```
export FI_MLX_ENABLE_SPAWN=yes
# specific
MY_MPIRUN_OPTIONS="-usize 8" ibrun -np 4 spawnmanager
# more generic
MY_MPIRUN_OPTIONS="-usize ${SLURM_NPROCS}" ibrun -np 4 spawnmanager
# using mpiexec:
mpiexec -np 2 -usize ${SLURM_NPROCS} spawnmanager
```

The spawned program looks very much like a regular MPI program, with its own initialization and finalize calls.

```
// spawnworker.c
MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
MPI_Comm_get_parent(&parent);
MPI_Comm_remote_size(parent, &remotesize);
if (workerno==0) {
  printf("Worker deduces %d workers and %d parents\n",nworkers,remotesize);
}
```

For the full source of this example, see section 8.5.4

```
## spawnworker.py
parentcomm = comm.Get_parent()
nparents = parentcomm.Get_remote_size()
```

For the full source of this example, see section 8.5.5

Spawned processes wind up with a value of `MPI_COMM_WORLD` of their own, but managers and workers can find each other regardless. The spawn routine returns the intercommunicator to the parent; the children can find it through `MPI_Comm_get_parent` (section 7.6.3). The number of spawning processes can be found through `MPI_Comm_remote_size` on the parent communicator.

Figure 8.2 `MPI_Open_port`

Name	Param name	C type	F type	inout
`mpi_open_port (`				
	`info`	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	implementation-specific information on how to establish an address			
	`port_name`	`char*`	`CHARACTER`	`out`
	`length: MPI_MAX_PORT_NAME`			
	newly established port			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Open_port(`		
	`info`	`Info = Ellipsis`
`) -> str`		

8.1.1 MPMD

Instead of spawning a single executable, you can spawn multiple with `MPI_Comm_spawn_multiple`.

8.2 Socket-style communications

It is possible to establish connections with running MPI programs that have their own world communicator.

- The *server* process establishes a port with `MPI_Open_port`, and calls `MPI_Comm_accept` to accept connections to its port.
- The *client* process specifies that port in an `MPI_Comm_connect` call. This establishes the connection.

8.2.1 Server calls

The server calls `MPI_Open_port` (figure 8.2), yielding a port name. Port names are generated by the system and copied into a character buffer of length at most `MPI_MAX_PORT_NAME`.

The server then needs to call `MPI_Comm_accept` (figure 8.3) prior to the client doing a connect call. This is collective over the calling communicator. It returns an intercommunicator (section 7.6) that allows communication with the client.

```
MPI_Comm intercomm;
char myport[MPI_MAX_PORT_NAME];
MPI_Open_port( MPI_INFO_NULL,myport );
int portlen = strlen(myport);
MPI_Send( myport,portlen+1,MPI_CHAR,1,0,comm_world );
printf("Host sent port <<%s>>\n",myport);
MPI_Comm_accept( myport,MPI_INFO_NULL,0,comm_self,&intercomm );
printf("host accepted connection\n");
```

For the full source of this example, see section 8.5.6

The port can be closed with `MPI_Close_port`.

Figure 8.3 `MPI_Comm_accept`

Name	Param name	C type	F type	inout
`mpi_comm_accept (`				
	`port_name` *port name*	`const char*`	`CHARACTER`	`in`
	`info` *implementation-dependent information*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`root` *rank in comm of root node*	`int`	`INTEGER`	`in`
	`comm` *intra-communicator over which call is collective*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`newcomm` *inter-communicator with client as remote group*	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Intracomm.Accept(`		
	`self`	
	`port_name`	`str`
	`info`	`Info = Ellipsis`
	`root`	`int = 0`
`) -> Intercomm`		

8.2.2 Client calls

After the server has generated a port name, the client needs to connect to it with `MPI_Comm_connect` (figure 8.4), again specifying the port through a character buffer. The connect call is collective over its communicator.

```
char myport[MPI_MAX_PORT_NAME];
if (work_p==0) {
  MPI_Recv( myport,MPI_MAX_PORT_NAME,MPI_CHAR,
            MPI_ANY_SOURCE,0, comm_world,MPI_STATUS_IGNORE );
  printf("Worker received port <<%s>>\n",myport);
}
MPI_Bcast( myport,MPI_MAX_PORT_NAME,MPI_CHAR,0,comm_work );
MPI_Comm intercomm;
MPI_Comm_connect( myport,MPI_INFO_NULL,0,comm_work,&intercomm );
if (work_p==0) {
  int manage_n;
  MPI_Comm_remote_size(intercomm,&manage_n);
  printf("%d workers connected to %d managers\n",work_n,manage_n);
}
```

For the full source of this example, see section 8.5.6

If the named port does not exist (or has been closed), `MPI_Comm_connect` raises an error of class `MPI_ERR_PORT`.

The client can sever the connection with `MPI_Comm_disconnect`

Running the above code on 5 processes gives:

```
# exchange port name:
```

Figure 8.4 `MPI_Comm_connect`

Name	Param name	C type	F type	inout
`mpi_comm_connect (`				
	`port_name` *network address*	`const char*`	`CHARACTER`	`in`
	`info` *implementation-dependent information*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`root` *rank in comm of root node*	`int`	`INTEGER`	`in`
	`comm` *intra-communicator over which call is collective*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`newcomm` *inter-communicator with server as remote group*	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Intracomm.Connect(`		
	`self`	
	`port_name`	`str`
	`info`	`Info = Ellipsis`
	`root`	`int = 0`
`) -> Intercomm`		

```
Host sent port <<tag#0$OFA#000010e1:0001cde9:0001cdee$rdma_port#1024$rdma_host#10:16:225:0:1:205:199:25
Worker received port <<tag#0$OFA#000010e1:0001cde9:0001cdee$rdma_port#1024$rdma_host#10:16:225:0:1:205:

# Comm accept/connect
host accepted connection
4 workers connected to 1 managers

# Send/recv over the intercommunicator
Manager sent 4 items over intercomm
Worker zero received data
```

8.2.3 Published service names

`MPI_Publish_name` (figure 8.5)

`MPI_Unpublish_name`

Unpublishing a nonexisting or already unpublished service gives an error code of `MPI_ERR_SERVICE`.

`MPI_Comm_join`

MPI provides no guarantee of fairness in servicing connection attempts. That is, connection attempts are not necessarily satisfied in the order in which they were initiated, and competition from other connection attempts may prevent a particular connection attempt from being satisfied.

The following material is for the (unreleased) MPI-4 standard only

Figure 8.5 `MPI_Publish_name`

Name	Param name	C type	F type	inout
`mpi_publish_name (`				
	`service_name`	`const char*`	`CHARACTER`	`in`
	a service name to associate with the port			
	`info`	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	implementation-specific information			
	`port_name`	`const char*`	`CHARACTER`	`in`
	a port name			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Publish_name(`		
	`service_name`	`str`
	`port_name`	`str`
	`info`	`Info = Ellipsis`
`) -> None`		

8.3 Sessions

The most common way of initializing MPI, with `MPI_Init` (or `MPI_Init_thread`) and `MPI_Finalize`, is known as the *world model*. Additionally, there is the *session model*, which can be described as doing multiple initializations and finalizations. The two models can be used in the same program, but there are limitations on how they can mix.

8.3.1 World model versus sessions model

The *world model* of using MPI can be described as:

1. There is a single call to `MPI_Init` or `MPI_Init_thread`;
2. There is a single call to `MPI_Finalize`;
3. With very few exceptions, all MPI calls appear in between the initialize and finalize calls.

In the *session model*, the world model has become a single session, and it is possible to start multiple sessions, each on their own set of processes, possibly identical or overlapping.

An MPI *session* is initialized and finalized with `MPI_Session_init` and `MPI_Session_finalize`, somewhat similar to `MPI_Init` and `MPI_Finalize`.

```
MPI_Info       info;
MPI_Errhandler errhandler;
MPI_Session    session;
MPI_Session_init(info,errhandler,&session);

MPI_Info info_used;
MPI_Session_get_info(session,&info_used);
MPI_Info_free(&info_used);

MPI_Session_finalize(&session);
```

The info object can contain implementation-specific data, but the key *mpi_thread_support_level* is predefined.

You can not mix in a single call objects from different sessions, from a session and from the world model, or from a session and from MPI_Comm_get_parent or MPI_Comm_join.

8.3.2 Process sets

Process sets are indicated with a Uniform Resource Identifier (URI), where the URIs *mpi*://WORLD and *mpi*://SELF are always defined.

The following partial code creates a communicator equivalent to MPI_COMM_WORLD in the session model:

```
const char pset_name[] = "mpi://WORLD";
MPI_Group_from_session_pset
   (lib_shandle,pset_name,&wgroup);
MPI_Comm_create_from_group
   (wgroup,"parcompbook-example",
    MPI_INFO_NULL,MPI_ERRORS_RETURN,&world_comm);
```

Further process sets can be found: MPI_Session_get_num_psets.

Get a specific one: MPI_Session_get_nth_pset.

Get the info object (section 14.1.1) from a process set: MPI_Session_get_pset_info. This info object always has the key *mpi_size*.
End of MPI-4 material

8.4 Functionality available outside init/finalize

MPI_Initialized MPI_Finalized MPI_Get_version MPI_Get_library_version MPI_Info_create MPI_Info_create_env MPI_Info_set MPI_Info_delete MPI_Info_get MPI_Info_get_valuelen MPI_Info_get_nkeys MPI_Info_get_nthkey MPI_Info_dup MPI_Info_free MPI_Info_f2c MPI_Info_c2f MPI_Session_create_errhandler MPI_Session_call_errhandler MPI_Errhandler_free MPI_Errhandler_f2c MPI_Errhandler_c2f MPI_Error_string MPI_Error_class

Also all routines starting with MPI_Txxx.

8.5 Sources used in this chapter

8.5.1 Listing of code header

8.5.2 Listing of code examples/mpi/c/spawnmanager.c

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
int main(int argc, char *argv[])
{

#define ASSERT(p) if (!(p)) {printf("Assertion failed for proc %d at line %d\n",procno,__LINE__); return -1;}
#define ASSERTm(p,m) if (!(p)) {printf("Message<<%s>> for proc %d at line %d\n",m,procno,__LINE__); return -1;}

  MPI_Comm comm;
  int procno=-1,nprocs,err;
  MPI_Init(&argc,&argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_rank(comm,&procno);
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);

  /*
   * To investigate process placement, get host name
   */
  {
    int namelen = MPI_MAX_PROCESSOR_NAME;
    char procname[namelen];
    MPI_Get_processor_name(procname,&namelen);
    printf("[%d] manager process runs on <<%s>>\n",procno,procname);
  }

  int world_size,manager_rank, universe_size, *universe_sizep, flag;

  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  MPI_Comm_rank(MPI_COMM_WORLD, &manager_rank);

  err = MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_UNIVERSE_SIZE,
       (void*)&universe_sizep, &flag);
  if (err==MPI_ERR_KEYVAL) {
    printf("MPI_ERR_KEYVAL on proc %d\n",procno);
    MPI_Abort(comm,0); }

  if (!flag) {
    if (manager_rank==0) {
      printf("This MPI does not support UNIVERSE_SIZE.\nHow many processes total?");
      scanf("%d", &universe_size);
    }
    MPI_Bcast(&universe_size,1,MPI_INTEGER,0,MPI_COMM_WORLD);
  } else {
    universe_size = *universe_sizep;
```

```
    if (manager_rank==0)
      printf("Universe size deduced as %d\n",universe_size);
  }
  ASSERTm(universe_size>world_size,"No room to start workers");

  int nworkers = universe_size-world_size;

  /*
   * Now spawn the workers. Note that there is a run-time determination
   * of what type of worker to spawn, and presumably this calculation must
   * be done at run time and cannot be calculated before starting
   * the program. If everything is known when the application is
   * first started, it is generally better to start them all at once
   * in a single MPI_COMM_WORLD.
   */

  if (manager_rank==0)
    printf("Now spawning %d workers\n",nworkers);
  const char *worker_program = "spawnworker";
  int errorcodes[nworkers];
  MPI_Comm inter_to_workers;           /* intercommunicator */
  MPI_Comm_spawn(worker_program, MPI_ARGV_NULL, nworkers,
 MPI_INFO_NULL, 0, MPI_COMM_WORLD, &inter_to_workers,
 errorcodes);
  for (int ie=0; ie<nworkers; ie++)
    if (errorcodes[ie]!=0)
      printf("Error %d in spawning worker %d\n",errorcodes[ie],ie);

  /*
   * Parallel code here. The communicator "inter_to_workers" can be used
   * to communicate with the spawned processes, which have ranks 0,..
   * MPI_UNIVERSE_SIZE-1 in the remote group of the intercommunicator
   * "inter_to_workers".
   */

  MPI_Finalize();
  return 0;
}
```

8.5.3 Listing of code examples/mpi/p/spawnmanager.py

```
import numpy as np
import random # random.randint(1,N), random.random()
import sys

from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()

try :
```

```
    universe_size = comm.Get_attr(MPI.UNIVERSE_SIZE)
    if universe_size is None:
        print("Universe query returned None")
        universe_size = nprocs + 4
    else:
        print("World has {} ranks in a universe of {}"\
              .format(nprocs,universe_size))
except :
    print("Exception querying universe size")
    universe_size = nprocs + 4
nworkers = universe_size - nprocs

itercomm = comm.Spawn("./spawn_worker.py", maxprocs=nworkers)
```

8.5.4 Listing of code examples/mpi/c/spawnworker.c

```
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{

#define ASSERT(p) if (!(p)) {printf("Assertion failed for proc %d at line %d\n",procno,__LINE__); return -1;}
#define ASSERTm(p,m) if (!(p)) {printf("Message<<%s>> for proc %d at line %d\n",m,procno,__LINE__); return -1;}

  MPI_Comm comm;
  int procno=-1,nprocs,err;
  MPI_Init(&argc,&argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_rank(comm,&procno);
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);

  int remotesize,nworkers,workerno;
  MPI_Comm parent;

  MPI_Comm_size(MPI_COMM_WORLD,&nworkers);
  MPI_Comm_rank(MPI_COMM_WORLD,&workerno);
  MPI_Comm_get_parent(&parent);
  ASSERTm(parent!=MPI_COMM_NULL,"No parent!");

  /*
   * To investigate process placement, get host name
   */
  {
    int namelen = MPI_MAX_PROCESSOR_NAME;
    char procname[namelen];
    MPI_Get_processor_name(procname,&namelen);
    printf("[%d] worker process runs on <<%s>>\n",workerno,procname);
  }

  MPI_Comm_remote_size(parent, &remotesize);
```

```
  if (workerno==0) {
    printf("Worker deduces %d workers and %d parents\n",nworkers,remotesize);
  }
  //  ASSERTm(nworkers==size-1,"nworkers mismatch. probably misunderstanding");

  /*
   * Parallel code here.
   * The manager is represented as the process with rank 0 in (the remote
   * group of) MPI_COMM_PARENT.  If the workers need to communicate among
   * themselves, they can use MPI_COMM_WORLD.
   */

  char hostname[256]; int namelen = 256;
  MPI_Get_processor_name(hostname,&namelen);
  printf("worker %d running on %s\n",workerno,hostname);

  MPI_Finalize();
  return 0;
}
```

8.5.5 Listing of code examples/mpi/p/spawnworker.py

```
import numpy as np
import random # random.randint(1,N), random.random()

from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()

if procid==0:
    print("#workers:",nprocs)

parentcomm = comm.Get_parent()
nparents = parentcomm.Get_remote_size()

print("#parents=",nparents)
```

8.5.6 Listing of code code/mpi/c/portapp.c

```
#include <stdlib.h>
#include <stdio.h>
#include <memory.h>
#include <mpi.h>

int main(int argc,char **argv) {

  MPI_Init(&argc,&argv);
  MPI_Comm
    comm_world = MPI_COMM_WORLD,
```

```
  comm_self = MPI_COMM_SELF;

int world_p,world_n;
MPI_Comm_size(comm_world,&world_n);
MPI_Comm_rank(comm_world,&world_p);

/*
 * Set up a communicator for all the worker ranks
 */
MPI_Comm comm_work;
{
  MPI_Group group_world,group_work;
  MPI_Comm_group( comm_world,&group_world );
  int manager[] = {0};
  MPI_Group_excl( group_world,1,manager,&group_work );
  MPI_Comm_create( comm_world,group_work,&comm_work );
  MPI_Group_free( &group_world ); MPI_Group_free( &group_work );
}

if (world_p==0) {
  MPI_Comm intercomm;
  char myport[MPI_MAX_PORT_NAME];
  MPI_Open_port( MPI_INFO_NULL,myport );
  int portlen = strlen(myport);
  MPI_Send( myport,portlen+1,MPI_CHAR,1,0,comm_world );
  printf("Host sent port <<%s>>\n",myport);
  MPI_Comm_accept( myport,MPI_INFO_NULL,0,comm_self,&intercomm );
  printf("host accepted connection\n");
  int work_n;
  MPI_Comm_remote_size(intercomm,&work_n);
  double work_data[work_n];
  MPI_Send( work_data,work_n,MPI_DOUBLE,
    /* to rank zero of worker comm */ 0,0,intercomm );
  printf("Manager sent %d items over intercomm\n",work_n);
  MPI_Close_port(myport);
} else {
  int work_p,work_n;
  MPI_Comm_size( comm_work,&work_n );
  MPI_Comm_rank( comm_work,&work_p );
  char myport[MPI_MAX_PORT_NAME];
  if (work_p==0) {
    MPI_Recv( myport,MPI_MAX_PORT_NAME,MPI_CHAR,
              MPI_ANY_SOURCE,0, comm_world,MPI_STATUS_IGNORE );
    printf("Worker received port <<%s>>\n",myport);
  }
  MPI_Bcast( myport,MPI_MAX_PORT_NAME,MPI_CHAR,0,comm_work );
  MPI_Comm intercomm;
  MPI_Comm_connect( myport,MPI_INFO_NULL,0,comm_work,&intercomm );
  if (work_p==0) {
    int manage_n;
    MPI_Comm_remote_size(intercomm,&manage_n);
    printf("%d workers connected to %d managers\n",work_n,manage_n);
  }
```

```
    if (work_p==0) {
      double work_data[work_n];
      MPI_Recv( work_data,work_n,MPI_DOUBLE,
/* from rank zero of manager comm */ 0,0,intercomm,MPI_STATUS_IGNORE );
      printf("Worker zero received data\n");
    }
    MPI_Close_port(myport);
  }

  MPI_Finalize();

  return 0;
}

#if 0
  printf("[S] server is %d/%d in the world: \n",world_p,world_n);

  char myport[MPI_MAX_PORT_NAME];
  MPI_Comm intercomm;

  MPI_Open_port(MPI_INFO_NULL, myport);
  printf("[S] port name is: %s\n", myport);
  MPI_Comm_accept(myport, MPI_INFO_NULL, 0, MPI_COMM_SELF, &intercomm);
  printf("[S] accepting\n");

  int inter_p,inter_n;
  MPI_Comm_size(intercomm,&inter_n);
  MPI_Comm_rank(intercomm,&inter_p);
  printf("[S] server is %d/%d in the intercomm\n",inter_p,inter_n);

  float x;
  MPI_Recv(&x,1,MPI_FLOAT,MPI_ANY_SOURCE,MPI_ANY_TAG,intercomm,MPI_STATUS_IGNORE);
  printf("[S] received from client %e\n",x);
  x *= 2;
  MPI_Send(&x,1,MPI_FLOAT,1,0,intercomm);

  MPI_Comm_disconnect(&intercomm);

#endif
```

Chapter 9

MPI topic: One-sided communication

Above, you saw point-to-point operations of the two-sided type: they require the co-operation of a sender and receiver. This co-operation could be loose: you can post a receive with `MPI_ANY_SOURCE` as sender, but there had to be both a send and receive call. This two-sidedness can be limiting. Consider code where the receiving process is a dynamic function of the data:

```
x = f();
p = hash(x);
MPI_Send( x, /* to: */ p );
```

The problem is now: how does p know to post a receive, and how does everyone else know not to?

In this section, you will see one-sided communication routines where a process can do a 'put' or 'get' operation, writing data to or reading it from another processor, without that other processor's involvement.

In one-sided MPI operations, also known as Remote Direct Memory Access (RDMA) or Remote Memory Access (RMA) operations, there are still two processes involved: the *origin*, which is the process that originates the transfer, whether this is a 'put' or a 'get', and the *target* whose memory is being accessed. Unlike with two-sided operations, the target does not perform an action that is the counterpart of the action on the origin.

That does not mean that the origin can access arbitrary data on the target at arbitrary times. First of all, one-sided communication in MPI is limited to accessing only a specifically declared memory area on the target: the target declares an area of user-space memory that is accessible to other processes. This is known as a *window*. Windows limit how origin processes can access the target's memory: you can only 'get' data from a window or 'put' it into a window; all the other memory is not reachable from other processes.

The alternative to having windows is to use *distributed shared memory* or *virtual shared memory*: memory is distributed but acts as if it shared. The so-called Partitioned Global Address Space (PGAS) languages such as Unified Parallel C (UPC) use this model. The MPI RMA model makes it possible to lock a window which makes programming slightly more cumbersome, but the implementation more efficient.

Within one-sided communication, MPI has two modes: active RMA and passive RMA. In *active RMA*, or *active target synchronization*, the target sets boundaries on the time period (the 'epoch') during which its window can be accessed. The main advantage of this mode is that the origin program can perform

many small transfers, which are aggregated behind the scenes. Active RMA acts much like asynchronous transfer with a concluding `MPI_Waitall`.

In *passive RMA*, or *passive target synchronization*, the target process puts no limitation on when its window can be accessed. (PGAS languages such as UPC are based on this model: data is simply read or written at will.) While intuitively it is attractive to be able to write to and read from a target at arbitrary time, there are problems. For instance, it requires a remote agent on the target, which may interfere with execution of the main thread, or conversely it may not be activated at the optimal time. Passive RMA is also very hard to debug and can lead to strange deadlocks.

9.1 Windows

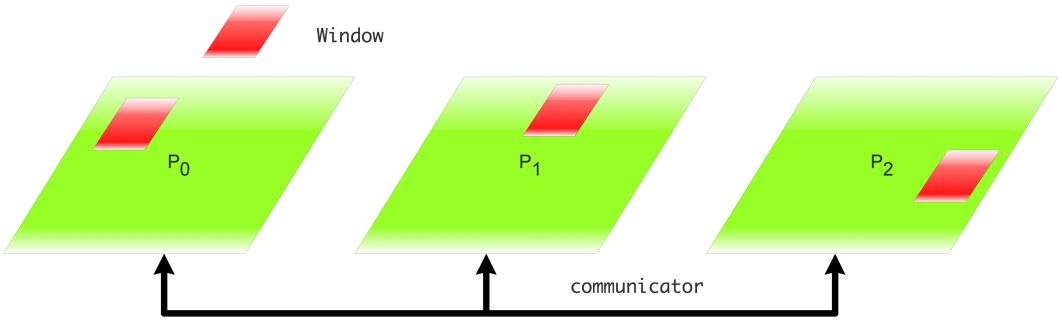


Figure 9.1: Collective definition of a window for one-sided data access

In one-sided communication, each processor can make an area of memory, called a *window*, available to one-sided transfers. This is stored in a variable of type `MPI_Win`. A process can put an arbitrary item from its own memory (not limited to any window) to the window of another process, or get something from the other process' window in its own memory.

A window can be characteristized as follows:

- The window is defined on a communicator, so the create call is collective; see figure 9.1.
- The window size can be set individually on each process. A zero size is allowed, but since window creation is collective, it is not possible to skip the create call.
- You can set a 'displacement unit' for the window: this is a number of bytes that will be used as the indexing unit. For example if you use `sizeof(double)` as the displacement unit, an `MPI_Put` to location 8 will go to the 8th double. That's easier than having to specify the 64th byte.
- The window is the target of data in a put operation, or the source of data in a get operation; see figure 9.2.
- There can be memory associated with a window, so it needs to be freed explicitly.

The typical calls involved are:

```
MPI_Info info;
MPI_Win window;
MPI_Win_allocate( /* size info */, info, comm, &memory, &window );
// do put and get calls
MPI_Win_free( &window );
```

Figure 9.1 `MPI_Win_create`

Name	Param name	C type	F type	inout
`mpi_win_create (`				
	`base`	`void*`	`TYPE(*), DIMENSION(..)`	`in`
	initial address of window			
	`size`	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`
	size of window in bytes			
	`disp_unit`	`int`	`INTEGER`	`in`
	local unit size for displacements, in bytes			
	`info`	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	intra-communicator			
	`win`	`MPI_Win*`	`TYPE(MPI_Win)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Create(`		
	`cls`	
	`memory`	`Union[Buffer, Bottom, None]`
	`disp_unit`	`int = 1`
	`info`	`Info = Ellipsis`
	`comm`	`Intracomm = Ellipsis`
`) -> Win`		

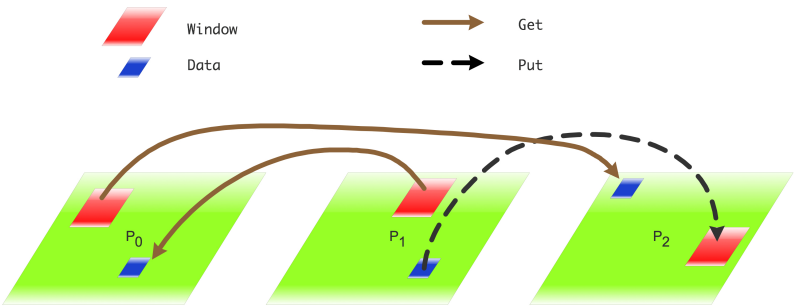


Figure 9.2: Put and get between process memory and windows

9.1.1 Window creation and allocation

The memory for a window is at first sight ordinary data in user space. There are multiple ways you can associate data with a window:

1. You can pass a user buffer to `MPI_Win_create` (figure 9.1). This buffer can be an ordinary array, or it can be created with `MPI_Alloc_mem`. (In the former case, it may not be possible to lock the window; section 9.4.)
2. You can let MPI do the allocation, so that MPI can perform various optimizations regarding placement of the memory. The user code then receives the pointer to the data from MPI. This can again be done in two ways:

Figure 9.2 `MPI_Win_allocate`

Name	Param name	C type	F type	inout
`mpi_win_allocate (`				
	`size`	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`
	size of window in bytes			
	`disp_unit`	`int`	`INTEGER`	`in`
	local unit size for displacements, in bytes			
	`info`	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	intra-communicator			
	`baseptr`	`void*`	`TYPE(C_PTR)`	`out`
	initial address of window			
	`win`	`MPI_Win*`	`TYPE(MPI_Win)`	`out`
	window object returned by call			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Allocate(`		
	`cls`	
	`size`	`int`
	`disp_unit`	`int = 1`
	`info`	`Info = Ellipsis`
	`comm`	`Intracomm = Ellipsis`
`) -> Win`		

- Use `MPI_Win_allocate` (figure 9.2) to create the data and the window in one call.
- If a communicator is on a shared memory (see section 12.1) you can create a window in that shared memory with `MPI_Win_allocate_shared`. This will be useful for *MPI shared memory*; see chapter 12.

3. Finally, you can create a window with `MPI_Win_create_dynamic` which postpones the allocation; see section 9.5.2.

First of all, `MPI_Win_create` creates a window from a pointer to memory. The data array must not be `PARAMETER` or `static const`.

The size parameter is measured in bytes. In C this is easily done with the `sizeof` operator;

```
// putfencealloc.c
MPI_Win the_window;
int *window_data;
MPI_Win_allocate(2*sizeof(int),sizeof(int),
                 MPI_INFO_NULL,comm,
                 &window_data,&the_window);
```

For the full source of this example, see section 9.9.2

for doing this calculation in Fortran, see section 14.3.1.

Python note. For computing the displacement in bytes, here is a good way for finding the size of *numpy* datatypes:

Figure 9.3 `MPI_Alloc_mem`

Name	Param name	C type	F type	inout
`mpi_alloc_mem (`				
	`size`	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`
	size of memory segment in bytes			
	`info`	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`baseptr`	`void*`	`TYPE(C_PTR)`	`out`
	pointer to beginning of memory segment allocated			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Alloc_mem(`		
	`size`	`int`
	`info`	`Info = Ellipsis`
`) -> memory`		

```
## putfence.py
intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=np.int)
win = MPI.Win.Create(window_data,intsize,comm=comm)
```

For the full source of this example, see section 9.9.3

Next, one can obtain the memory from MPI by using `MPI_Win_allocate`, which has the data pointer as output. Note the `void*` in the C prototype; it is still necessary to pass a pointer to a pointer:

```
double *window_data;
MPI_Win_allocate( ... &window_data ... );
```

The routine `MPI_Alloc_mem` (figure 9.3) performs only the allocation part of `MPI_Win_allocate`, after which you need to `MPI_Win_create`.

- An error of `MPI_ERR_NO_MEM` indicates that no memory could be allocated.

The following material is for the (unreleased) MPI-4 standard only

- Allocated memory can be aligned by specifying an `MPI_Info` key of *`mpi_minimum_memory_alignment`*
  .

End of MPI-4 material

This memory is freed with

```
MPI_Free_mem()
```

These calls reduce to *`malloc`* and *`free`* if there is no special memory area; SGI is an example where such memory does exist.

There will be more discussion of window memory in section 9.5.1.

Python note. Unlike in C, the python window allocate call does not return a pointer to the buffer memory. Should you need this, there are the following options:

- Window objects expose the Python buffer interface. So you can do Pythonic things like

Figure 9.4 `MPI_Win_fence`

Name	Param name	C type	F type	inout
`mpi_win_fence (`				
	`assert`	`int`	`INTEGER`	`in`
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Fence(`		
	`self`	
	`assertion`	`int = 0`
`) -> None`		

```
mview = memoryview(win)
array = numpy.frombuffer(win, dtype='i4')
```

- If you really want the raw base pointer (as an integer), you can do any of these:

```
base, size, disp_unit = win.atts
base = win.Get_attr(MPI.WIN_BASE)
```

- You can use mpi4py's builtin memoryview/buffer-like type, but I do not recommend it, much better to use NumPy as above:

```
mem = win.tomemory() # type(mem) is MPI.memory, similar to memoryview, but
    quite limited in functionality
base = mem.address
size = mem.nbytes
```

9.2 Active target synchronization: epochs

One-sided communication has an obvious complication over two-sided: if you do a put call instead of a send, how does the recipient know that the data is there? This process of letting the target know the state of affairs is called 'synchronization', and there are various mechanisms for it. First of all we will consider *active target synchronization*. Here the target knows when the transfer may happen (the *communication epoch*), but does not do any data-related calls.

In this section we look at the first mechanism, which is to use a *fence* operation: `MPI_Win_fence` (figure 9.4). This operation is collective on the communicator of the window. It is comparable to `MPI_Wait` calls for nonblocking communication.

The use of fences is somewhat complicated. The interval between two fences is known as an *epoch*. You can give various hints to the system about this epoch versus the ones before and after through the `assert` parameter.

```
MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
MPI_Get( /* operands */, win);
MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
```

In between the two fences the window is exposed, and while it is you should not access it locally. If you absolutely need to access it locally, you can use an RMA operation for that. Also, there can be only one remote process that does a `put`; multiple `accumulate` accesses are allowed.

Fences are, together with other window calls, collective operations. That means they imply some amount of synchronization between processes. Consider:

```
MPI_Win_fence( ... win ... ); // start an epoch
if (mytid==0) // do lots of work
MPI_Win_fence( ... win ... ); // end the epoch
```

and assume that all processes execute the first fence more or less at the same time. The zero process does work before it can do the second fence call, but all other processes can call it immediately. However, they can not finish that second fence call until all one-sided communication is finished, which means they wait for the zero process.

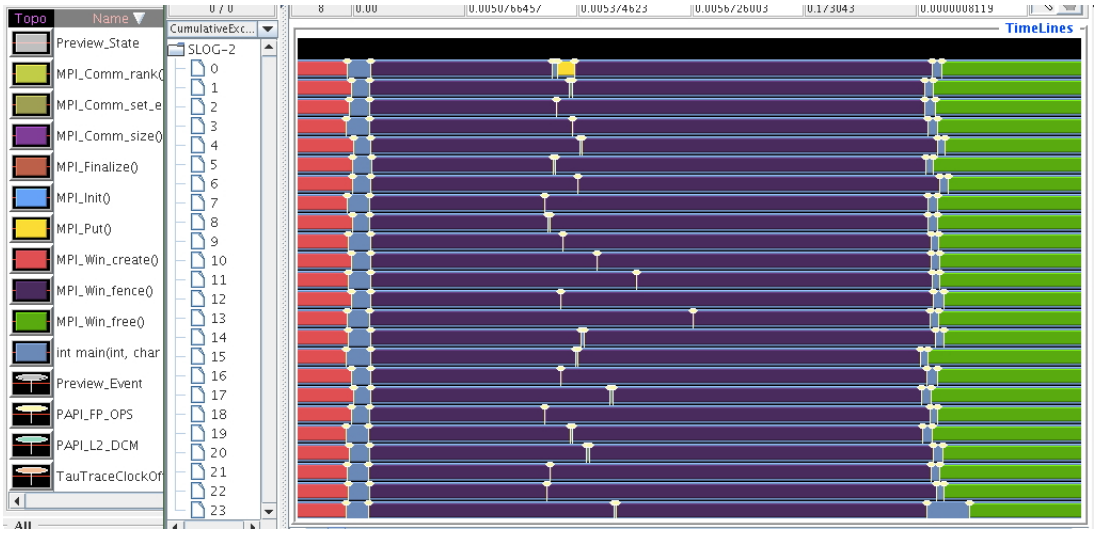


Figure 9.3: A trace of a one-sided communication epoch where process zero only originates a one-sided transfer

As a further restriction, you can not mix `MPI_Get` with `MPI_Put` or `MPI_Accumulate` calls in a single epoch. Hence, we can characterize an epoch as an *access epoch* on the origin, and as an *exposure epoch* on the target.

Assertions are an integer parameter: you can combine assertions by adding them or using logical-or. The value zero is always correct. For further information, see section 9.6.

9.3 Put, get, accumulate

We will now look at the first three routines for doing one-sided operations: the Put, Get, and Accumulate call. (We will look at so-called 'atomic' operations in section 9.3.8.) These calls are somewhat similar to a

Figure 9.5 `MPI_Put`

Name	Param name	C type	F type	inout
`mpi_put (`				
	`origin_addr` *initial address of origin buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`origin_count` *number of entries in origin buffer*	`int`	`INTEGER`	`in`
	`origin_datatype` *datatype of each entry in origin buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`target_rank` *rank of target*	`int`	`INTEGER`	`in`
	`target_disp` *displacement from start of window to target buffer*	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`
	`target_count` *number of entries in target buffer*	`int`	`INTEGER`	`in`
	`target_datatype` *datatype of each entry in target buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`win` *window object used for communication*	`MPI_Win`	`TYPE(MPI_Win)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

```
win.Put(self, origin, int target_rank, target=None)
```

Send, Receive and Reduce, except that of course only one process makes a call. Since one process does all the work, its calling sequence contains both a description of the data on the origin (the calling process) and the target (the affected other process).

As in the two-sided case, `MPI_PROC_NULL` can be used as a target rank.

The Put/Get/Accumulate routines have an `MPI_Op` argument that can be any of the usual operators, but no user-defined ones (see section 3.10.1).

9.3.1 Put

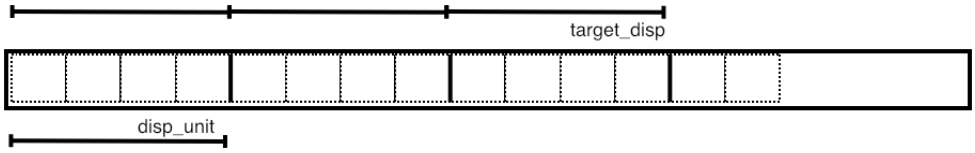
The `MPI_Put` (figure 9.5) call can be considered as a one-sided send. As such, it needs to specify

- the target rank
- the data to be sent from the origin, and
- the location where it is to be written on the target.

The description of the data on the origin is the usual trio of buffer/count/datatype. However, the description of the data on the target is more complicated. It has a count and a datatype, but instead of an address it has a *displacement* with respect to the start of the window on the target. This displacement can be given in bytes, so its type is `MPI_Aint`, but strictly speaking it is a multiple of the displacement unit that was specified in the window definition.

Specifically, data is written starting at

$$\texttt{window_base} + \texttt{target_disp} \times \texttt{disp_unit}.$$



Here is a single put operation. Note that the window create and window fence calls are collective, so they have to be performed on all processors of the communicator that was used in the create call.

```
// putfence.c
MPI_Win the_window;
MPI_Win_create
  (&window_data,2*sizeof(int),sizeof(int),
   MPI_INFO_NULL,comm,&the_window);
MPI_Win_fence(0,the_window);
if (procno==0) {
  MPI_Put
    ( /* data on origin: */   &my_number, 1,MPI_INT,
        /* data on target: */   other,1,    1,MPI_INT,
        the_window);
}
MPI_Win_fence(0,the_window);
MPI_Win_free(&the_window);
```

For the full source of this example, see section 9.9.4

Fortran note. The `disp_unit` variable is declared as an integer of 'kind' `MPI_ADDRESS_KIND`:

```
// putfence.F90
integer(kind=MPI_ADDRESS_KIND) :: target_displacement
   target_displacement = 1
   call MPI_Put( my_number, 1,MPI_INTEGER, &
        other,target_displacement, &
        1,MPI_INTEGER, &
        the_window)
```

For the full source of this example, see section 9.9.5

Prior to *Fortran2008*, specifying a literal constant, such as 0, could lead to bizarre runtime errors; the solution was to specify a zero-valued variable of the right type. With the `mpi_f08` module this is no longer allowed. Instead you get an error such as

```
error #6285: There is no matching specific subroutine for this generic subroutine call.   [MPI
```

Exercise 9.1. Revisit exercise 4.3 and solve it using `MPI_Put`.
(There is a skeleton for this exercise under the name `rightput`.)

Exercise 9.2. Write code where:

- process 0 computes a random number r
- if $r < .5$, zero writes in the window on 1;
- if $r \geq .5$, zero writes in the window on 2.

(There is a skeleton for this exercise under the name `randomput`.)

Figure 9.6 `MPI_Get`

Name	Param name	C type	F type	inout
`mpi_get (`				
	`origin_addr` *initial address of origin buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`origin_count` *number of entries in origin buffer*	`int`	`INTEGER`	`in`
	`origin_datatype` *datatype of each entry in origin buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`target_rank` *rank of target*	`int`	`INTEGER`	`in`
	`target_disp` *displacement from window start to the beginning of the target buffer*	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`
	`target_count` *number of entries in target buffer*	`int`	`INTEGER`	`in`
	`target_datatype` *datatype of each entry in target buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`win` *window object used for communication*	`MPI_Win`	`TYPE(MPI_Win)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

```
win.Get(self, origin, int target_rank, target=None)
```

9.3.2 Get

The `MPI_Get` (figure 9.6) call is very similar.

Example:

```
// getfence.c
MPI_Win_create(&other_number,2*sizeof(int),sizeof(int),
               MPI_INFO_NULL,comm,&the_window);
MPI_Win_fence(0,the_window);
if (procno==0) {
  MPI_Get( /* data on origin: */   &my_number, 1,MPI_INT,
           /* data on target: */   other,1,    1,MPI_INT,
           the_window);
}
MPI_Win_fence(0,the_window);
```

For the full source of this example, see section 9.9.6

We make a null window on processes that do not participate.

```
## getfence.py
if procid==0 or procid==nprocs-1:
    win_mem = np.empty( 1,dtype=np.float64 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    win = MPI.Win.Create( None,comm=comm )

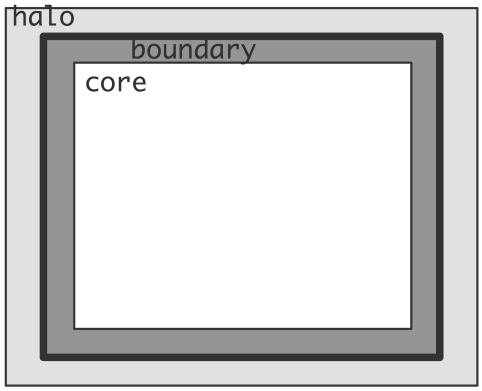
# put data on another process
win.Fence()
if procid==0 or procid==nprocs-1:
```

```
    putdata = np.empty( 1,dtype=np.float64 )
    putdata[0] = mydata
    print("[%d] putting %e" % (procid,mydata))
    win.Put( putdata,other )
win.Fence()
```

For the full source of this example, see section 9.9.7

9.3.3 Put and get example: halo update

As an example, let's look at *halo update*. The array `A` is updated using the local values and the halo that comes from bordering processors, either through Put or Get operations.



In a first version we separate computation and communication. Each iteration has two fences. Between the two fences in the loop body we do the `MPI_Put` operation; between the second and and first one of the next iteration there is only computation, so we add the `NOPRECEDE` and `NOSUCCEED` assertions. The `NOSTORE` assertion states that the local window was not updated: the Put operation only works on remote windows.

```
for ( .... ) {
  update(A);
  MPI_Win_fence(MPI_MODE_NOPRECEDE, win);
  for(i=0; i < toneighbors; i++)
    MPI_Put( ... );
  MPI_Win_fence((MPI_MODE_NOSTORE | MPI_MODE_NOSUCCEED), win);
  }
```

For much more about assertions, see section 9.6 below.

Next, we split the update in the core part, which can be done purely from local values, and the boundary, which needs local and halo values. Update of the core can overlap the communication of the halo.

```
for ( .... ) {
  update_boundary(A);
  MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);
  for(i=0; i < fromneighbors; i++)
    MPI_Get( ... );
  update_core(A);
  MPI_Win_fence(MPI_MODE_NOSUCCEED, win);
  }
```

The `NOPRECEDE` and `NOSUCCEED` assertions still hold, but the `Get` operation implies that instead of `NOSTORE` in the second fence, we use `NOPUT` in the first.

Figure 9.7 `MPI_Accumulate`

Name	Param name	C type	F type	inout
`mpi_accumulate (`				
	`origin_addr` *initial address of buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`origin_count` *number of entries in buffer*	`int`	`INTEGER`	`in`
	`origin_datatype` *datatype of each entry*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`target_rank` *rank of target*	`int`	`INTEGER`	`in`
	`target_disp` *displacement from start of window to beginning of target buffer*	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`
	`target_count` *number of entries in target buffer*	`int`	`INTEGER`	`in`
	`target_datatype` *datatype of each entry in target buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`op` *reduce operation*	`MPI_Op`	`TYPE(MPI_Op)`	`in`
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

```
MPI.Win.Accumulate(self, origin, int target_rank, target=None, Op op=SUM)
```

9.3.4 Accumulate

A third one-sided routine is `MPI_Accumulate` (figure 9.7) which does a reduction operation on the results that are being put.

Accumulate is an atomic reduction with remote result. This means that multiple accumulates to a single target gives the correct result. As with `MPI_Reduce`, the order in which the operands are accumulated is undefined.

The same predefined operators are available, but no user-defined ones. There is one extra operator: `MPI_REPLACE`, this has the effect that only the last result to arrive is retained.

Exercise 9.3. Implement an 'all-gather' operation using one-sided communication: each processor stores a single number, and you want each processor to build up an array that contains the values from all processors. Note that you do not need a special case for a processor collecting its own value: doing 'communication' between a processor and itself is perfectly legal.

Exercise 9.4.
Implement a shared counter:

- One process maintains a counter;
- Iterate: all others at random moments update this counter.
- When the counter is no longer positive, everyone stops iterating.

The problem here is data synchronization: does everyone see the counter the same way?

Figure 9.8 MPI_Rput

Name	Param name	C type	F type	inout
mpi_rput (				
	origin_addr *initial address of origin buffer*	const void*	TYPE(*), DIMENSION(..)	in
	origin_count *number of entries in origin buffer*	int	INTEGER	in
	origin_datatype *datatype of each entry in origin buffer*	MPI_Datatype	TYPE(MPI_Datatype)	in
	target_rank *rank of target*	int	INTEGER	in
	target_disp *displacement from start of window to target buffer*	MPI_Aint	INTEGER(KIND=MPI_ADDRESS_KIND)	in
	target_count *number of entries in target buffer*	int	INTEGER	in
	target_datatype *datatype of each entry in target buffer*	MPI_Datatype	TYPE(MPI_Datatype)	in
	win *window object used for communication*	MPI_Win	TYPE(MPI_Win)	in
	request *RMA request*	MPI_Request*	TYPE(MPI_Request)	out
(opt)	ierror		INTEGER	out
)				

9.3.5 Ordering and coherence of RMA operations

There are few guarantees about what happens inside one epoch.

- No ordering of Get and Put/Accumulate operations: if you do both, there is no guarantee whether the Get will find the value before or after the update.
- No ordering of multiple Puts. It is safer to do an Accumulate.

The following operations are well-defined inside one epoch:

- Instead of multiple Put operations, use Accumulate with MPI_REPLACE.
- MPI_Get_accumulate with MPI_NO_OP is safe.
- Multiple Accumulate operations from one origin are done in program order by default. To allow reordering, for instance to have all reads happen after all writes, use the info parameter when the window is created; section 9.5.3.

9.3.6 Request-based operations

Analogous to MPI_Isend there are request based one-sided operations: MPI_Rput (figure 9.8) and similarly MPI_Rget and MPI_Raccumulate and MPI_Rget_accumulate.

These only apply to passive target synchronization. Any MPI_Win_flush... call also terminates these transfers.

9.3.7 More active target synchronization

The 'fence' mechanism (section 9.2) uses a global synchronization on the communicator of the window. As such it is good for applications where the processes are largely synchronized, but it may lead to performance inefficiencies if processors are not in step which each other. There is a mechanism that is more

fine-grained, by using synchronization only on a processor *group*. This takes four different calls, two for starting and two for ending the epoch, separately for target and origin.

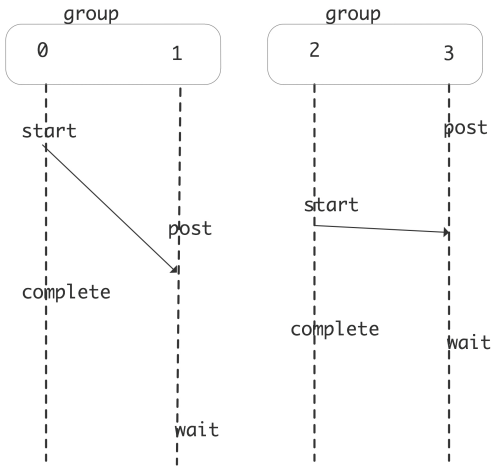


Figure 9.4: Window locking calls in fine-grained active target synchronization

You start and complete an *exposure epoch* with`MPI_Win_post``MPI_Win_wait`:

```
int MPI_Win_post(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_wait(MPI_Win win)
```

In other words, this turns your window into the *target* for a remote access.

You start and complete an *access epoch* with`MPI_Win_start``MPI_Win_complete`:

```
int MPI_Win_start(MPI_Group group, int assert, MPI_Win win)
int MPI_Win_complete(MPI_Win win)
```

In other words, these calls border the access to a remote window, with the current processor being the *origin* of the remote access.

In the following snippet a single processor puts data on one other. Note that they both have their own definition of the group, and that the receiving process only does the post and wait calls.

```
// postwaitwin.c
if (procno==origin) {
  MPI_Group_incl(all_group,1,&target,&two_group);
// access
  MPI_Win_start(two_group,0,the_window);
  MPI_Put( /* data on origin: */   &my_number, 1,MPI_INT,
           /* data on target: */   target,0,   1,MPI_INT,
               the_window);
  MPI_Win_complete(the_window);
}

if (procno==target) {
  MPI_Group_incl(all_group,1,&origin,&two_group);
```

Figure 9.9 `MPI_Fetch_and_op`

Name	Param name	C type	F type	inout
`mpi_fetch_and_op (`				
	`origin_addr` *initial address of buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	in
	`result_addr` *initial address of result buffer*	`void*`	`TYPE(*), DIMENSION(..)`	out
	`datatype` *datatype of the entry in origin, result, and target buffers*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	in
	`target_rank` *rank of target*	`int`	`INTEGER`	in
	`target_disp` *displacement from start of window to beginning of target buffer*	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	in
	`op` *reduce operation*	`MPI_Op`	`TYPE(MPI_Op)`	in
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	in
`(opt)`	`ierror`		`INTEGER`	out
`)`				

```
// exposure
  MPI_Win_post(two_group,0,the_window);
  MPI_Win_wait(the_window);
}
```

For the full source of this example, see section 9.9.8

Both pairs of operations declare a *group of processors*; see section 7.5.1 for how to get such a group from a communicator. On an origin processor you would specify a group that includes the targets you will interact with, on a target processor you specify a group that includes the possible origins.

9.3.8 Atomic operations

One-sided calls are said to emulate shared memory in MPI, but the put and get calls are not enough for certain scenarios with shared data. Consider the scenario where:

- One process stores a table of work descriptors, and a pointer to the first unprocessed descriptor;
- Each process reads the pointer, reads the corresponding descriptor, and increments the pointer; and
- A process that has read a descriptor then executes the corresponding task.

The problem is that reading and updating the pointer is not an *atomic operation*, so it is possible that multiple processes get hold of the same value; conversely, multiple updates of the pointer may lead to work descriptors being skipped. These inconsistent views of the data are called a *race condition*.

In MPI-3 some atomic routines have been added. Both `MPI_Fetch_and_op` (figure 9.9) and `MPI_Get_accumulate` (figure 9.10) atomically retrieve data from the window indicated, and apply an operator, combining the data on the target with the data on the origin. Unlike Put and Get, it is safe to have multiple atomic operations in the same epoch.

Both routines perform the same operations: return data before the operation, then atomically update data on the target, but `MPI_Get_accumulate` is more flexible in data type handling. The more simple routine,

Figure 9.10 `MPI_Get_accumulate`

Name	Param name	C type	F type	inout
`mpi_get_accumulate (`				
	`origin_addr` *initial address of buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	in
	`origin_count` *number of entries in origin buffer*	`int`	`INTEGER`	in
	`origin_datatype` *datatype of each entry in origin buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	in
	`result_addr` *initial address of result buffer*	`void*`	`TYPE(*), DIMENSION(..)`	out
	`result_count` *number of entries in result buffer*	`int`	`INTEGER`	in
	`result_datatype` *datatype of each entry in result buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	in
	`target_rank` *rank of target*	`int`	`INTEGER`	in
	`target_disp` *displacement from start of window to beginning of target buffer*	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	in
	`target_count` *number of entries in target buffer*	`int`	`INTEGER`	in
	`target_datatype` *datatype of each entry in target buffer*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	in
	`op` *reduce operation*	`MPI_Op`	`TYPE(MPI_Op)`	in
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	in
`(opt)`	`ierror`		`INTEGER`	out
`)`				

`MPI_Fetch_and_op`, which operates on only a single element, allows for faster implementations, in particular through hardware support.

Use of `MPI_NO_OP` as the `MPI_Op` turns these routines into an atomic Get. Similarly, using `MPI_REPLACE` turns them into an atomic Put.

Exercise 9.5. Redo exercise 9.4 using `MPI_Fetch_and_op`. The problem is again to make sure all processes have the same view of the shared counter.
Does it work to make the fetch-and-op conditional? Is there a way to do it unconditionally? What should the 'break' test be, seeing that multiple processes can update the counter at the same time?

Example. A root process has a table of data; the other processes do atomic gets and update of that data using *passive target synchronization* through `MPI_Win_lock`.

```
// passive.cxx
if (procno==repository) {
// Repository processor creates a table of inputs
// and associates that with the window
}
if (procno!=repository) {
  float contribution=(float)procno,table_element;
  int loc=0;
  MPI_Win_lock(MPI_LOCK_EXCLUSIVE,repository,0,the_window);
// read the table element by getting the result from adding zero
  MPI_Fetch_and_op
```

Figure 9.11 `MPI_Compare_and_swap`

Name	Param name	C type	F type	inout
`mpi_compare_and_swap (`				
	`origin_addr` *initial address of buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`compare_addr` *initial address of compare buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`result_addr` *initial address of result buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`datatype` *datatype of the element in all buffers*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`target_rank` *rank of target*	`int`	`INTEGER`	`in`
	`target_disp` *displacement from start of window to beginning of target buffer*	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

```
    (&contribution,&table_element,MPI_FLOAT,
     repository,loc,MPI_SUM,the_window);
  MPI_Win_unlock(repository,the_window);
}
```

For the full source of this example, see section 9.9.9

```
## passive.py
if procid==repository:
    # repository process creates a table of inputs
    # and associates it with the window
    win_mem = np.empty( ninputs,dtype=np.float32 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    # everyone else has an empty window
    win = MPI.Win.Create( None,comm=comm )
if procid!=repository:
    contribution = np.empty( 1,dtype=np.float32 )
    contribution[0] = 1.*procid
    table_element = np.empty( 1,dtype=np.float32 )
    win.Lock( repository,lock_type=MPI.LOCK_EXCLUSIVE )
    win.Fetch_and_op( contribution,table_element,repository,0,MPI.SUM)
    win.Unlock( repository )
```

For the full source of this example, see section 9.9.10

Finally, `MPI_Compare_and_swap` (figure 9.11) swaps the origin and target data if the target data equals some comparison value.

9.3.8.1 A case study in atomic operations

Let us consider an example where a process, identified by *counter_process*, has a table of work descriptors, and all processes, including the counter process, take items from it to work on. To avoid duplicate work, the counter process has as counter that indicates the highest numbered available item. The part of this application that we simulate is this:

1. a process reads the counter, to find an available work item; and
2. subsequently decrements the counter by one.

We initialize the window content, under the separate memory model:

```
// countdownop.c
MPI_Win_fence(0,the_window);
if (procno==counter_process)
  MPI_Put(&counter_init,1,MPI_INT,
          counter_process,0,1,MPI_INT,
          the_window);
MPI_Win_fence(0,the_window);
```

For the full source of this example, see section 9.9.11

We start by considering the naive approach, where we execute the above scheme literally with `MPI_Get` and `MPI_Put`:

```
// countdownput.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
         counter_process,0,1,MPI_INT,
         the_window);
MPI_Win_fence(0,the_window);
if (i_am_available) {
  my_counter_values[ n_my_counter_values++ ] = counter_value;
  total_decrement++;
  int decrement = -1;
  counter_value += decrement;
  MPI_Put
    ( &counter_value,   1,MPI_INT,
      counter_process,0,1,MPI_INT,
      the_window);
}
MPI_Win_fence(0,the_window);
```

For the full source of this example, see section 9.9.12

This scheme is correct if only process has a true value for `i_am_available`: that processes 'owns' the current counter values, and it correctly updates the counter through the `MPI_Put` operation. However, if more than one process is available, they get duplicate counter values, and the update is also incorrect. If we run this program, we see that the counter did not get decremented by the total number of 'put' calls.

Exercise 9.6. Supposing only one process is available, what is the function of the middle of the three fences? Can it be omitted?

We can fix the decrement of the counter by using `MPI_Accumulate` for the counter update, since it is atomic: multiple updates in the same epoch all get processed.

```
// countdownacc.c
MPI_Win_fence(0,the_window);
int counter_value;
MPI_Get( &counter_value,1,MPI_INT,
         counter_process,0,1,MPI_INT,
         the_window);
```

```
MPI_Win_fence(0,the_window);
if (i_am_available) {
  my_counter_values[n_my_counter_values++] = counter_value;
  total_decrement++;
  int decrement = -1;
  MPI_Accumulate
    ( &decrement,       1,MPI_INT,
      counter_process,0,1,MPI_INT,
      MPI_SUM,
      the_window);
}
MPI_Win_fence(0,the_window);
```

For the full source of this example, see section 9.9.13

This scheme still suffers from the problem that processes will obtain duplicate counter values. The true solution is to combine the 'get' and 'put' operations into one atomic action; in this case `MPI_Fetch_and_op`:

```
MPI_Win_fence(0,the_window);
int
  counter_value;
if (i_am_available) {
  int
    decrement = -1;
  total_decrement++;
  MPI_Fetch_and_op
    ( /* operate with data from origin: */   &decrement,
      /* retrieve data from target:     */   &counter_value,
      MPI_INT, counter_process, 0, MPI_SUM,
      the_window);
}
MPI_Win_fence(0,the_window);
if (i_am_available) {
  my_counter_values[n_my_counter_values++] = counter_value;
}
```

For the full source of this example, see section 9.9.11

Now, if there are multiple accesses, each retrieves the counter value and updates it in one atomic, that is, indivisible, action.

9.4 Passive target synchronization

In *passive target synchronization* only the origin is actively involved: the target makes no calls whatsoever. This means that the origin process remotely locks the window on the target, performs a one-sided transfer, and releases the window by unlocking it again.

During an access epoch, also called an *passive target epoch* in this case (the concept of 'exposure epoch' makes no sense with passive target synchronization), a process can initiate and finish a one-sided transfer. Typically it will lock the window with `MPI_Win_lock` (figure 9.12):

Figure 9.12 `MPI_Win_lock`

Name	Param name	C type	F type	inout
`mpi_win_lock (`				
	`lock_type`	`int`	`INTEGER`	`in`
	either `MPI_LOCK_EXCLUSIVE` *or* `MPI_LOCK_SHARED`			
	`rank`	`int`	`INTEGER`	`in`
	rank of locked window			
	`assert`	`int`	`INTEGER`	`in`
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Lock(`		
	`self`	
	`rank`	`int`
	`lock_type`	`int = Ellipsis`
	`assertion`	`int = 0`
`) -> None`		

```
if (rank == 0) {
  MPI_Win_lock (MPI_LOCK_EXCLUSIVE, 1, 0, win);
  MPI_Put (outbuf, n, MPI_INT, 1, 0, n, MPI_INT, win);
  MPI_Win_unlock (1, win);
}
```

Remark 13 *The possibility to lock a window is not guaranteed for windows that are not created (possibly internally) by `MPI_Alloc_mem`, that is, all but `MPI_Win_create`.*

9.4.1 Lock types

A lock is needed for an origin to acquire the capability to access a target. You can either acquire a lock on a specific process with `MPI_Win_lock`, or on all processes (in a communicator) with `MPI_Win_lock_all`. Unlike `MPI_Win_fence`, this is not a collective call.

The two lock types are:

- `MPI_LOCK_SHARED`: multiple processes can access the window on the same rank. If multiple processes perform a `MPI_Get` call there is no problem; with `MPI_Put` and similar calls there is a consistency problem; see below.
- `MPI_LOCK_EXCLUSIVE`: an origin gets exclusive access to the window on a certain target. Unlike the shared lock, this has no consistency problems.

To unlock a window, use `MPI_Win_unlock` (figure 9.13), respectively `MPI_Win_unlock_all` (figure 9.13).

Exercise 9.7.

(There is a skeleton for this exercise under the name `lockfetch`.) Investigate atomic updates using passive target synchronization. Use `MPI_Win_lock` with an

Figure 9.13 `MPI_Win_unlock`

Name	Param name	C type	F type	inout
`mpi_win_unlock (`				
	`rank` *rank of window*	`int`	`INTEGER`	`in`
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Unlock(`		
	`self`	
	`rank`	`int`
`) -> None`		

exclusive lock, which means that each process only acquires the lock when it absolutely has to.

- All processs but one update a window:

```
int one=1;
MPI_Fetch_and_op(&one, &readout,
    MPI_INT, repo, zero_disp, MPI_SUM,
    the_win);
```

- while the remaining process spins until the others have performed their update.

Use an atomic operation for the latter process to read out the shared value.

Exercise 9.8.

(There is a skeleton for this exercise under the name `lockfetchshared`.) As exercise 9.7, but now use a shared lock: all processes acquire the lock simultaneously and keep it as long as is needed.
The problem here is that coherence between window buffers and local variables is now not forced by a fence or releasing a lock. Use `MPI_Win_flush_local` to force coherence of a window (on another process) and the local variable from `MPI_Fetch_and_op`.

9.4.2 Lock all

To lock the windows of all processes in the group of the windows, use `MPI_Win_lock_all` (figure 9.14). This is not a collective call: the 'all' part refers to the fact that one process is locking the window on all processes.

- The assertion value can be zero, or `MPI_MODE_NOCHECK`, which asserts that no other process will acquire a competing lock.
- There is no 'locktype' parameter: this is a shared lock.

The corresponding unlock is `MPI_Win_unlock_all` (figure 9.13).

Figure 9.14 `MPI_Win_lock_all`

Name	Param name	C type	F type	inout
`mpi_win_lock_all (`				
	`assert`	`int`	`INTEGER`	`in`
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Lock_all(`		
	`self`	
	`assertion`	`int = 0`
`) -> None`		

The expected use of a 'lock/unlock all' is that they surround an extended epoch with get/put and flush calls.

9.4.3 Completion and consistency in passive target synchronization

In one-sided transfer one should keep straight the multiple instances of the data, and the various *completions* that effect their *consistency*.

- The user data. This is the buffer that is passed to a `Put` or `Get` call. For instance, after a `Put` call, but still in an access epoch, the user buffer is not safe to reuse. Making sure the buffer has been transferred is called *local completion*.
- The window data. While this may be publicly accessible, it is not necessarily always consistent with internal copies.
- The remote data. Even a successful `Put` does not guarantee that the other process has received the data. A successful transfer is a *remote completion*.

As observed, RMA operations are nonblocking, so we need mechanisms to ensure that an operation is completed, and to ensure *consistency* of the user and window data.

Completion of the RMA operations in a passive target epoch is ensured with `MPI_Win_unlock` or `MPI_Win_unlock_all`, similar to the use of `MPI_Win_fence` in active target synchronization.

If the passive target epoch is of greater duration, and no unlock operation is used to ensure completion, the following calls are available.

Remark 14 *Using flush routines with active target synchronization (or generally outside a passive target epoch) you are likely to get a message*

```
Wrong synchronization of RMA calls
```

9.4.3.1 Local completion

The call `MPI_Win_flush_local` (figure 9.15) ensure that all operations with a given target is completed

Figure 9.15 `MPI_Win_flush_local`

Name	Param name	C type	F type	inout
`mpi_win_flush_local (`				
	`rank` *rank of target window*	`int`	`INTEGER`	`in`
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Flush_local(`		
	`self`	
	`rank`	`int`
`) -> None`		

Figure 9.16 `MPI_Win_flush`

Name	Param name	C type	F type	inout
`mpi_win_flush (`				
	`rank` *rank of target window*	`int`	`INTEGER`	`in`
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Flush(`		
	`self`	
	`rank`	`int`
`) -> None`		

at the origin. For instance, for calls to `MPI_Get` or `MPI_Fetch_and_op` the local result is available after the `MPI_Win_flush_local`.

With `MPI_Win_flush_local_all` local operations are concluded for all targets. This will typically be used with `MPI_Win_lock_all` (section 9.4.2).

9.4.3.2 *Remote completion*

The calls `MPI_Win_flush` (figure 9.16) and `MPI_Win_flush_all` effect completion of all outstanding RMA operations on the target, so that other processes can access its data. This is useful for `MPI_Put` operations, but can also be used for atomic operations such as `MPI_Fetch_and_op`.

9.4.3.3 *Window synchronization*

Under the *separate memory model*, the user code can hold a buffer that is not coherent with the internal window data. The call `MPI_Win_sync` synchronizes private and public copies of the window.

Figure 9.17 `MPI_Win_create_dynamic`

Name	Param name	C type	F type	inout
`mpi_win_create_dynamic (`				
	`info`	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`comm` *intra-communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`win` *window object returned by the call*	`MPI_Win*`	`TYPE(MPI_Win)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Create_dynamic(`		
	`cls`	
	`info`	`Info = Ellipsis`
	`comm`	`Intracomm = Ellipsis`
`) -> Win`		

9.5 More about window memory

9.5.1 Memory models

You may think that the window memory is the same as the buffer you pass to `MPI_Win_create` or that you get from `MPI_Win_allocate` (section 9.1.1). This is not necessarily true, and the actual state of affairs is called the *memory model*. There are two memory models:

- Under the *unified* memory model, the buffer in process space is indeed the window memory, or at least they are kept *coherent*. This means that after *completion* of an epoch you can read the window contents from the buffer. To get this, the window needs to be created with `MPI_Win_allocate_shared`.
- Under the *separate* memory model, the buffer in process space is the *private window* and the target of put/get operations is the *public window* and the two are not the same and are not kept coherent. Under this model, you need to do an explicit get to read the window contents.

(Window models can be queried as attributes; see section 9.5.4.)

9.5.2 Dynamically attached memory

In section 9.1.1 we looked at simple ways to create a window and its memory.

It is also possible to have windows where the size is dynamically set. Create a dynamic window with `MPI_Win_create_dynamic` (figure 9.17) and attach memory to the window with `MPI_Win_attach` (figure 9.18).

At first sight, the code looks like splitting up a `MPI_Win_create` call into separate creation of the window and declaration of the buffer:

```
// windynamic.c
MPI_Win_create_dynamic(MPI_INFO_NULL,comm,&the_window);
if (procno==data_proc)
  window_buffer = (int*) malloc( 2*sizeof(int) );
  MPI_Win_attach(the_window,window_buffer,2*sizeof(int));
```

Figure 9.18 `MPI_Win_attach`

Name	Param name	C type	F type	inout
`mpi_win_attach (`				
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
	`base`	`void*`	`TYPE(*), DIMENSION(..)`	`in`
	initial address of memory to be attached			
	`size`	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`
	size of memory to be attached in bytes			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Attach(`		
	`self`	
	`memory`	`Buffer`
`) -> None`		

For the full source of this example, see section 9.9.14

(where the *window_buffer* represents memory that has been allocated.)

However, there is an important difference in how the window is addressed in RMA operations. With all other window models, the displacement parameter is measured relative in units from the start of the buffer, here the displacement is an absolute address. This means that we need to get the address of the window buffer with `MPI_Get_address` and communicate it to the other processes:

```
MPI_Aint data_address;
if (procno==data_proc) {
  MPI_Get_address(window_buffer,&data_address);
}
MPI_Bcast(&data_address,1,MPI_LONG,data_proc,comm);
```

For the full source of this example, see section 9.9.14

Location of the data, that is, the displacement parameter, is then given as an absolute location of the start of the buffer plus a count in bytes; in other words, the *displacement unit* is 1. In this example we use `MPI_Get` to find the second integer in a window buffer:

```
MPI_Aint disp = data_address+1*sizeof(int);
MPI_Get( /* data on origin: */           retrieve, 1,MPI_INT,
               /* data on target: */ data_proc,disp,     1,MPI_INT,
               the_window);
```

For the full source of this example, see section 9.9.14

Notes.

- The attached memory can be released with `MPI_Win_detach` (figure 9.19).
- The above fragments show that an origin process has the actual address of the window buffer. It is an error to use this if the buffer is not attached to a window.
- In particular, one has to make sure that the attach call is concluded before performing RMA operations on the window.

Figure 9.19 `MPI_Win_detach`

Name	Param name	C type	F type	inout
`mpi_win_detach (`				
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
	`base`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	initial address of memory to be detached			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Detach(`		
	`self`	
	`memory`	`Buffer`
`) -> None`		

9.5.3 Window usage hints

The following keys can be passed as info argument:

- *`no_locks`*: if set to true, passive target synchronization (section 9.4) will not be used on this window.
- *`accumulate_ordering`*: a comma-separated list of the keywords `rar`, `raw`, `war`, `waw` can be specified. This indicates that reads or writes from `MPI_Accumulate` or `MPI_Get_accumulate` can be reordered, subject to certain constraints.
- *`accumulate_ops`*: the value *`same_op`* indicates that concurrent Accumulate calls use the same operator; *`same_op_no_op`* indicates the same operator or `MPI_NO_OP`.

9.5.4 Window information

The `MPI_Info` parameter can be used to pass implementation-dependent information; see section 14.1.1.

A number of attributes are stored with a window when it is created.

Obtaining a pointer to the start of the window area:

```
void *base;
MPI_Win_get_attr(win, MPI_WIN_BASE, &base, &flag)
```

Obtaining the size and *window displacement unit*:

```
MPI_Aint *size;
MPI_Win_get_attr(win, MPI_WIN_SIZE, &size, &flag),
int *disp_unit;
MPI_Win_get_attr(win, MPI_WIN_DISP_UNIT, &disp_unit, &flag),
```

The type of create call used:

```
int *create_kind;
MPI_Win_get_attr(win, MPI_WIN_CREATE_FLAVOR, &create_kind, &flag)
```

with possible values:

- MPI_WIN_FLAVOR_CREATE if the window was create with MPI_Win_create;
- MPI_WIN_FLAVOR_ALLOCATE if the window was create with MPI_Win_allocate;
- MPI_WIN_FLAVOR_DYNAMIC if the window was create with MPI_Win_create_dynamic. In this case the base is MPI_BOTTOM and the size is zero;
- MPI_WIN_FLAVOR_SHARED if the window was create with MPI_Win_allocate_shared;

The window model:

```
int *memory_model;
MPI_Win_get_attr(win, MPI_WIN_MODEL, &memory_model, &flag);
```

with possible values:

- MPI_WIN_SEPARATE,
- MPI_WIN_UNIFIED,

Get the group of processes associated with a window:

```
int MPI_Win_get_group(MPI_Win win, MPI_Group *group)
MPI_Win_get_group(win, group, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
TYPE(MPI_Group), INTENT(OUT) :: group
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

```
int MPI_Win_set_info(MPI_Win win, MPI_Info info)
MPI_Win_set_info(win, info, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
TYPE(MPI_Info), INTENT(IN) :: info
INTEGER, OPTIONAL, INTENT(OUT) :: ierror

int MPI_Win_get_info(MPI_Win win, MPI_Info *info_used)
MPI_Win_get_info(win, info_used, ierror)
TYPE(MPI_Win), INTENT(IN) :: win
TYPE(MPI_Info), INTENT(OUT) :: info_used
INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

9.6 Assertions

The MPI_Win_fence call, as well MPI_Win_start and such, take an argument through which assertions can be passed about the activity before, after, and during the epoch. The value zero is always allowed, by you can make your program more efficient by specifying one or more of the following, combined by bitwise OR in C/C++ or IOR in Fortran.

- MPI_Win_start Supports the option:
  - MPI_MODE_NOCHECK the matching calls to MPI_Win_post have already completed on all target processes when the call to MPI_Win_start is made. The nocheck option can be specified in a start call if and only if it is specified in each matching post call. This is similar to the optimization of "ready-send" that may save a handshake when the handshake is implicit

in the code. (However, ready-send is matched by a regular receive, whereas both start and post must specify the nocheck option.)

- `MPI_Win_post` supports the following options:
  - `MPI_MODE_NOCHECK` the matching calls to `MPI_Win_start` have not yet occurred on any origin processes when the call to `MPI_Win_post` is made. The nocheck option can be specified by a post call if and only if it is specified by each matching start call.
  - `MPI_MODE_NOSTORE` the local window was not updated by local stores (or local get or receive calls) since last synchronization. This may avoid the need for cache synchronization at the post call.
  - `MPI_MODE_NOPUT` the local window will not be updated by put or accumulate calls after the post call, until the ensuing (wait) synchronization. This may avoid the need for cache synchronization at the wait call.
- `MPI_Win_fence` supports the following options:
  - `MPI_MODE_NOSTORE` the local window was not updated by local stores (or local get or receive calls) since last synchronization.
  - `MPI_MODE_NOPUT` the local window will not be updated by put or accumulate calls after the fence call, until the ensuing (fence) synchronization.
  - `MPI_MODE_NOPRECEDE` the fence does not complete any sequence of locally issued RMA calls. If this assertion is given by any process in the window group, then it must be given by all processes in the group.
  - `MPI_MODE_NOSUCCEED` the fence does not start any sequence of locally issued RMA calls. If the assertion is given by any process in the window group, then it must be given by all processes in the group.
- `MPI_Win_lock` and `MPI_Win_lock_all` support the following option:
  - `MPI_MODE_NOCHECK` no other process holds, or will attempt to acquire a conflicting lock, while the caller holds the window lock. This is useful when mutual exclusion is achieved by other means, but the coherence operations that may be attached to the lock and unlock calls are still required.

9.7 Implementation

You may wonder how one-sided communication is realized[1]. Can a processor somehow get at another processor's data? Unfortunately, no.

Active target synchronization is implemented in terms of two-sided communication. Imagine that the first fence operation does nothing, unless it concludes prior one-sided operations. The Put and Get calls do nothing involving communication, except for marking with what processors they exchange data. The concluding fence is where everything happens: first a global operation determines which targets need to issue send or receive calls, then the actual sends and receive are executed.

Exercise 9.9. Assume that only Get operations are performed during an epoch. Sketch how these are translated to send/receive pairs. The problem here is how the senders find

1. For more on this subject, see [25].

out that they need to send. Show that you can solve this with an `MPI_Reduce_scatter` call.

The previous paragraph noted that a collective operation was necessary to determine the two-sided traffic. Since collective operations induce some amount of synchronization, you may want to limit this.

Exercise 9.10. Argue that the mechanism with window post/wait/start/complete operations still needs a collective, but that this is less burdensome.

Passive target synchronization needs another mechanism entirely. Here the target process needs to have a background task (process, thread, daemon,...) running that listens for requests to lock the window. This can potentially be expensive.

9.8 Review questions

Find all the errors in this code.

```c
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

#define MASTER 0

int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;
  int r, p;
  MPI_Comm_rank(comm, &r);
  MPI_Comm_size(comm, &p);
  printf("Hello from %d\n", r);
  int result[1] = {0};
  //int assert = MPI_MODE_NOCHECK;
  int assert = 0;
  int one = 1;
  MPI_Win win_res;
  MPI_Win_allocate(1 * sizeof(MPI_INT), sizeof(MPI_INT), MPI_INFO_NULL, comm, &result[0],
     &win_res);
  MPI_Win_lock_all(assert, win_res);
  if (r == MASTER) {
    result[0] = 0;
    do{
      MPI_Fetch_and_op(&result, &result , MPI_INT, r, 0, MPI_NO_OP, win_res);
      printf("result: %d\n", result[0]);
    } while(result[0] != 4);
    printf("Master is done!\n");
  } else {
    MPI_Fetch_and_op(&one, &result, MPI_INT, 0, 0, MPI_SUM, win_res);
  }
  MPI_Win_unlock_all(win_res);
  MPI_Win_free(&win_res);
  MPI_Finalize();
  return 0;
```

9.9 Sources used in this chapter

9.9.1 Listing of code header

9.9.2 Listing of code examples/mpi/c/examples/putfencealloc.c

9.9.3 Listing of code examples/mpi/p/putfence.py

```
import numpy as np
import random # random.randint(1,N), random.random()
import sys

from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

intsize = np.dtype('int').itemsize
window_data = np.zeros(2,dtype=np.int)
win = MPI.Win.Create(window_data,intsize,comm=comm)

my_number = np.empty(1,dtype=np.int)
src = 0; tgt = nprocs-1
if procid==src:
    my_number[0] = 37
else:
    my_number[0] = 1
win.Fence()
if procid==src:
    # put data in the second element of the window
    win.Put(my_number,tgt,target=1)
win.Fence()

if procid==tgt:
    print("Window after put:",window_data)
```

9.9.4 Listing of code examples/mpi/c/putfence.c

```
#include <stdlib.h>
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>
```

```
int main(int argc,char **argv) {

#include "globalinit.c"

  MPI_Win the_window;
  int my_number=0, window_data[2], other = nprocs-1;
  if (procno==0)
    my_number = 37;

  MPI_Win_create
    (&window_data,2*sizeof(int),sizeof(int),
     MPI_INFO_NULL,comm,&the_window);
  MPI_Win_fence(0,the_window);
  if (procno==0) {
    MPI_Put
      ( /* data on origin: */   &my_number, 1,MPI_INT,
/* data on target: */   other,1,    1,MPI_INT,
the_window);
  }
  MPI_Win_fence(0,the_window);
  if (procno==other)
    printf("I got the following: %d\n",window_data[1]);
  MPI_Win_free(&the_window);

  MPI_Finalize();
  return 0;
}
```

9.9.5 Listing of code examples/mpi/f08/putfence.F90

```
Program PutFence

  use mpi_f08
  implicit none

  Type(MPI_Win) :: the_window
  integer :: window_elt_size
  integer(kind=MPI_ADDRESS_KIND) :: window_size
  integer(kind=MPI_ADDRESS_KIND) :: target_displacement
  integer :: my_number=0, window_data(2), other

  Type(MPI_Comm) :: comm;
  integer :: mytid,ntids,i,p,err;

  call MPI_Init()
  comm = MPI_COMM_WORLD
  call MPI_Comm_rank(comm,mytid)
  call MPI_Comm_size(comm,ntids)
  call MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN)

  other = ntids-1
  if (mytid.eq.0) my_number = 37
```

```
  call MPI_Sizeof(window_data,window_elt_size)
  window_size = 2*window_elt_size
  call MPI_Win_create(window_data,&
       window_size,window_elt_size,          & ! window size, unit size
       MPI_INFO_NULL,comm,the_window)
  call MPI_Win_fence(0,the_window)
  if (mytid.eq.0) then
#ifndef F90STYLE
     target_displacement = 1
     call MPI_Put( my_number, 1,MPI_INTEGER, &
          other,target_displacement, &
          1,MPI_INTEGER, &
          the_window)
#else
     call MPI_Put( my_number, 1,MPI_INTEGER, &
          other,1, &
          1,MPI_INTEGER, &
          the_window)
#endif
 endif
 call MPI_Win_fence(0,the_window)

 if (mytid.eq.other) then
      print *,"I got:",window_data(1+target_displacement)
   end if
   call MPI_Win_free(the_window)


   call MPI_Finalize(err);

 end Program PutFence
```

9.9.6 Listing of code examples/mpi/c/getfence.c

```
#include <stdlib.h>
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc,char **argv) {

#include "globalinit.c"

  {
    MPI_Win the_window;
    int my_number, other_number[2], other = nprocs-1;
    if (procno==other)
      other_number[1] = 27;
    MPI_Win_create(&other_number,2*sizeof(int),sizeof(int),
                   MPI_INFO_NULL,comm,&the_window);
    MPI_Win_fence(0,the_window);
```

```
    if (procno==0) {
      MPI_Get( /* data on origin: */   &my_number, 1,MPI_INT,
       /* data on target: */   other,1,    1,MPI_INT,
       the_window);
    }
    MPI_Win_fence(0,the_window);
    if (procno==0)
      printf("I got the following: %d\n",my_number);
    MPI_Win_free(&the_window);
  }

  MPI_Finalize();
  return 0;
}
```

9.9.7 Listing of code examples/mpi/p/getfence.py

```
import numpy as np
import random
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

other = nprocs-1-procid
mydata = random.random()

if procid==0 or procid==nprocs-1:
    win_mem = np.empty( 1,dtype=np.float64 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    win = MPI.Win.Create( None,comm=comm )

# put data on another process
win.Fence()
if procid==0 or procid==nprocs-1:
    putdata = np.empty( 1,dtype=np.float64 )
    putdata[0] = mydata
    print("[%d] putting %e" % (procid,mydata))
    win.Put( putdata,other )
win.Fence()

# see what you got
if procid==0 or procid==nprocs-1:
    print("[%d] getting %e" % (procid,win_mem[0]))

win.Free()
```

9.9.8 Listing of code examples/mpi/c/postwaitwin.c

```
#include <stdlib.h>
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc,char **argv) {

#include "globalinit.c"

  {
    MPI_Win the_window;
    MPI_Group all_group,two_group;
    int my_number = 37, other_number,
      twotids[2],origin,target;

    MPI_Win_create(&other_number,1,sizeof(int),
                   MPI_INFO_NULL,comm,&the_window);
    if (procno>0 && procno<nprocs-1) goto skip;

    origin = 0; target = nprocs-1;
    MPI_Comm_group(comm,&all_group);

    if (procno==origin) {
      MPI_Group_incl(all_group,1,&target,&two_group);
      // access
      MPI_Win_start(two_group,0,the_window);
      MPI_Put( /* data on origin: */   &my_number, 1,MPI_INT,
               /* data on target: */   target,0,   1,MPI_INT,
       the_window);
      MPI_Win_complete(the_window);
    }

    if (procno==target) {
      MPI_Group_incl(all_group,1,&origin,&two_group);
      // exposure
      MPI_Win_post(two_group,0,the_window);
      MPI_Win_wait(the_window);
    }
    if (procno==target)
      printf("Got the following: %d\n",other_number);

    MPI_Group_free(&all_group);
    MPI_Group_free(&two_group);
  skip:
    MPI_Win_free(&the_window);
  }

  MPI_Finalize();
  return 0;
}
```

9.9.9 Listing of code examples/mpi/c/fetchop.c

9.9.10 Listing of code examples/mpi/p/passive.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<2:
    print("C'mon, get real....")
    sys.exit(1)

repository = 0 # process that keeps the repository
ninputs = nprocs-1
checksum = nprocs*(nprocs-1)/2.

if procid==repository:
    # repository process creates a table of inputs
    # and associates it with the window
    win_mem = np.empty( ninputs,dtype=np.float32 )
    win = MPI.Win.Create( win_mem,comm=comm )
else:
    # everyone else has an empty window
    win = MPI.Win.Create( None,comm=comm )

# bookkeeping: which jobs did I process
myjobs = np.zeros( ninputs,dtype=np.int )

if procid!=repository:
    contribution = np.empty( 1,dtype=np.float32 )
    contribution[0] = 1.*procid
    table_element = np.empty( 1,dtype=np.float32 )
    win.Lock( repository,lock_type=MPI.LOCK_EXCLUSIVE )
    win.Fetch_and_op( contribution,table_element,repository,0,MPI.SUM)
    win.Unlock( repository )
    print(procid,"added its contribution to partial sum",table_element[0])

win.Free()

if procid==repository:
    if abs(win_mem[0]-checksum)>1.e-12:
        print("Incorrect result %e s/b %e" % (win_mem[0],checksum))
    print("finished")
```

9.9.11 Listing of code examples/mpi/c/countdownop.c

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <unistd.h>

#include <mpi.h>
#include "gather_sort_print.h"

int main(int argc,char **argv) {

  int nprocs,procno;
  MPI_Init(0,0);
  MPI_Comm comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_rank(comm,&procno);

  if (nprocs<2) {
    printf("Need at least 2 procs\n");
    MPI_Abort(comm,0);
  }

  // first set a unique random seed
  srand(procno*time(0));

  {
    /*
     * Create a window.
     * We only need a nonzero size on the last process,
     * which we label the `counter_process';
     * everyone else makes a window of size zero.
     */
    MPI_Win the_window;
    int counter_process = nprocs-1;
    int window_data,check_data;
    if (procno==counter_process) {
      window_data = 2*nprocs-1;
      check_data = window_data;
      MPI_Win_create(&window_data,sizeof(int),sizeof(int),
                     MPI_INFO_NULL,comm,&the_window);
    } else {
      MPI_Win_create(&window_data,0,sizeof(int),
                     MPI_INFO_NULL,comm,&the_window);
    }
    /*
     * Initialize the window
     * - PROCWRITES is approx the number of writes we want each process to do
     * - COLLISION  is approx how many processes will collide on a write
     */
#ifndef COLLISION
#define COLLISION 2
#endif
#ifndef PROCWRITES
#define PROCWRITES 40
#endif
    int counter_init = nprocs * PROCWRITES;
    MPI_Win_fence(0,the_window);
```

```
if (procno==counter_process)
  MPI_Put(&counter_init,1,MPI_INT,
          counter_process,0,1,MPI_INT,
          the_window);
MPI_Win_fence(0,the_window);

/*
 * Allocate an array (grossly over-dimensioned)
 * for the counter values that belong to me
 */
int *my_counter_values = (int*) malloc( counter_init * sizeof(int) );
if (!my_counter_values) {
  printf("[%d] could not allocate counter values\n",procno);
  MPI_Abort(comm,0);
}
int n_my_counter_values = 0;

/*
 * Loop:
 * - at random times update the counter on the counter process
 * - and read out the counter to see if we stop
 */
int total_decrement = 0;
int nsteps = PROCWRITES / COLLISION;
if (procno==0)
  printf("Doing %d steps, counter starting: %d\n    probably %d-way collision on each step\n",
         nsteps,counter_init,COLLISION);
for (int step=0; step<nsteps ; step++) {

  /*
   * Basic probability of a write is 1/P,
   * so each step only one proc will write.
   * Increase chance of collision by upping
   * the value of COLLISION.
   */
  float randomfraction = (rand() / (double)RAND_MAX);
  int i_am_available = randomfraction < ( COLLISION * 1./nprocs );

  /*
   * Exercise:
   * - atomically read and decrement the counter
   */
  MPI_Win_fence(0,the_window);
  int
    counter_value;
  if (i_am_available) {
    int
      decrement = -1;
    total_decrement++;
    MPI_Fetch_and_op
      ( /* operate with data from origin: */   &decrement,
        /* retrieve data from target:     */   &counter_value,
        MPI_INT, counter_process, 0, MPI_SUM,
```

```
            the_window);
#ifdef DEBUG
        printf("[%d] updating in step %d; retrieved %d\n",procno,step,counter_value);
#endif
      }
      MPI_Win_fence(0,the_window);
      if (i_am_available) {
        my_counter_values[n_my_counter_values++] = counter_value;
      }
    }

    /*
     * What counter values were actually obtained?
     */
    gather_sort_print( my_counter_values,n_my_counter_values, comm );

    /*
     * We do a correctness test by computing what the
     * window_data is supposed to be
     */
    {
      MPI_Win_fence(0,the_window);
      int counter_value;
      MPI_Get( /* origin data to set: */ &counter_value,1,MPI_INT,
               /* window data to get: */ counter_process,0,1,MPI_INT,
               the_window);
      MPI_Win_fence(0,the_window);
      MPI_Allreduce(MPI_IN_PLACE,&total_decrement,1,MPI_INT,MPI_SUM,comm);
      if (procno==counter_process) {
        if (counter_init-total_decrement==counter_value)
          printf("[%d] initial counter %d decreased by %d correctly giving %d\n",
                 procno,counter_init,total_decrement,counter_value);
        else
          printf("[%d] initial counter %d decreased by %d, giving %d s/b %d\n",
                 procno,counter_init,total_decrement,counter_value,counter_init-total_decrement);
      }
    }
    MPI_Win_free(&the_window);
  }

  MPI_Finalize();
  return 0;
}
```

9.9.12 Listing of code examples/mpi/c/countdownput.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include "time.h"

#include <mpi.h>
#include "gather_sort_print.h"
```

```
int main(int argc,char **argv) {

  int nprocs,procno;
  MPI_Init(0,0);
  MPI_Comm comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_rank(comm,&procno);

  if (nprocs<2) {
    printf("Need at least 2 procs\n");
    MPI_Abort(comm,0);
  }

  // first set a unique random seed
  srand(procno*time(0));

  {
    /*
     * Create a window.
     * We only need a nonzero size on the last process,
     * which we label the `counter_process';
     * everyone else makes a window of size zero.
     */
    MPI_Win the_window;
    int counter_process = nprocs-1;
    int window_data,check_data;
    if (procno==counter_process) {
      window_data = 2*nprocs-1;
      check_data = window_data;
      MPI_Win_create(&window_data,sizeof(int),sizeof(int),
                     MPI_INFO_NULL,comm,&the_window);
    } else {
      MPI_Win_create(&window_data,0,sizeof(int),
                     MPI_INFO_NULL,comm,&the_window);
    }
    /*
     * Initialize the window
     * - PROCWRITES is approx the number of writes we want each process to do
     * - COLLISION  is approx how many processes will collide on a write
     */
#ifndef COLLISION
#define COLLISION 1
#endif
#ifndef PROCWRITES
#define PROCWRITES 10
#endif
    int counter_init = nprocs * PROCWRITES;
    MPI_Win_fence(0,the_window);
    if (procno==counter_process)
      MPI_Put(&counter_init,1,MPI_INT,
              counter_process,0,1,MPI_INT,
              the_window);
```

```
    MPI_Win_fence(0,the_window);

    /*
     * Allocate an array (grossly over-dimensioned)
     * for the counter values that belong to me
     */
    int *my_counter_values = (int*) malloc( counter_init * sizeof(int) );
    if (!my_counter_values) {
      printf("[%d] could not allocate counter values\n",procno);
      MPI_Abort(comm,0);
    }
    int n_my_counter_values = 0;

    /*
     * Loop forever:
     * - at random times update the counter on the counter process
     * - and read out the counter to see if we stop
     */
    int total_decrement = 0;
    int nsteps = PROCWRITES / COLLISION;
    if (procno==0)
      printf("Doing %d steps, %d writes per proc,\n.. probably %d-way collision on each step\n",
             nsteps,PROCWRITES,COLLISION);
    for (int step=0; step<nsteps ; step++) {

      /*
       * Basic probability of a write is 1/P,
       * so each step only one proc will write.
       * Increase chance of collision by upping
       * the value of COLLISION.
       */
      float randomfraction = (rand() / (double)RAND_MAX);
      int i_am_available = randomfraction < ( COLLISION * .8/nprocs );

      /*
       * Exercise:
       * - decrement the counter by Get, compute new value, Put
       */
      MPI_Win_fence(0,the_window);
      int counter_value;
      MPI_Get( &counter_value,1,MPI_INT,
               counter_process,0,1,MPI_INT,
               the_window);
      MPI_Win_fence(0,the_window);
      if (i_am_available) {
#ifdef DEBUG
        printf("[%d] obtaining value %d in step %d\n",
               procno,counter_value,step);
#endif
        my_counter_values[ n_my_counter_values++ ] = counter_value;
        total_decrement++;
        int decrement = -1;
        counter_value += decrement;
```

```
        MPI_Put
          ( &counter_value,   1,MPI_INT,
            counter_process,0,1,MPI_INT,
            the_window);
      }
      MPI_Win_fence(0,the_window);
    }

    /*
     * What counter values were actually obtained?
     */
    gather_sort_print( my_counter_values,n_my_counter_values, comm );

    /*
     * We do a correctness test by computing what the
     * window_data is supposed to be
     */
    {
      MPI_Win_fence(0,the_window);
      int counter_value;
      MPI_Get( /* origin data to set: */ &counter_value,1,MPI_INT,
               /* window data to get: */ counter_process,0,1,MPI_INT,
               the_window);
      MPI_Win_fence(0,the_window);
      MPI_Allreduce(MPI_IN_PLACE,&total_decrement,1,MPI_INT,MPI_SUM,comm);
      if (procno==counter_process) {
        if (counter_init-total_decrement==counter_value)
          printf("[%d] initial counter %d decreased by %d correctly giving %d\n",
                 procno,counter_init,total_decrement,counter_value);
        else
          printf("[%d] initial counter %d decreased by %d, giving %d s/b %d\n",
                 procno,counter_init,total_decrement,counter_value,counter_init-total_decrement);
      }
    }
    MPI_Win_free(&the_window);
  }

  MPI_Finalize();
  return 0;
}
```

9.9.13 Listing of code examples/mpi/c/countdownacc.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

#include <mpi.h>
#include "gather_sort_print.h"

int main(int argc,char **argv) {

#include "globalinit.c"
```

```
  if (nprocs<2) {
    printf("Need at least 2 procs\n");
    MPI_Abort(comm,0);
  }

  // first set a unique random seed
  srand(procno*time(0));

  {
    /*
     * Create a window.
     * We only need a nonzero size on the last process,
     * which we label the `counter_process';
     * everyone else makes a window of size zero.
     */
    MPI_Win the_window;
    int counter_process = nprocs-1;
    int window_data,check_data;
    if (procno==counter_process) {
      window_data = 2*nprocs-1;
      check_data = window_data;
      MPI_Win_create(&window_data,sizeof(int),sizeof(int),
                     MPI_INFO_NULL,comm,&the_window);
    } else {
      MPI_Win_create(&window_data,0,sizeof(int),
                     MPI_INFO_NULL,comm,&the_window);
    }
    /*
     * Initialize the window
     * - PROCWRITES is approx the number of writes we want each process to do
     * - COLLISION  is approx how many processes will collide on a write
     */
#ifndef COLLISION
#define COLLISION 2
#endif
#ifndef PROCWRITES
#define PROCWRITES 40
#endif
    int counter_init = nprocs * PROCWRITES;
    MPI_Win_fence(0,the_window);
    if (procno==counter_process)
      MPI_Put(&counter_init,1,MPI_INT,
              counter_process,0,1,MPI_INT,
              the_window);
    MPI_Win_fence(0,the_window);

    /*
     * Allocate an array (grossly over-dimensioned)
     * for the counter values that belong to me
     */
    int *my_counter_values = (int*) malloc( counter_init * sizeof(int) );
    if (!my_counter_values) {
      printf("[%d] could not allocate counter values\n",procno);
```

```
      MPI_Abort(comm,0);
    }
    int n_my_counter_values = 0;

    /*
     * Loop:
     * - at random times update the counter on the counter process
     * - and read out the counter to see if we stop
     */
    int total_decrement = 0;
    int nsteps = PROCWRITES / COLLISION;
    if (procno==0)
      printf("Doing %d steps, counter starting: %d\n    probably %d-way collision on each step\n",
             nsteps,counter_init,COLLISION);
    for (int step=0; step<nsteps ; step++) {

      /*
       * Basic probability of a write is 1/P,
       * so each step only one proc will write.
       * Increase chance of collision by upping
       * the value of COLLISION.
       */
      float randomfraction = (rand() / (double)RAND_MAX);
      int i_am_available = randomfraction < ( COLLISION * 1./nprocs );

      /*
       * Exercise:
       * - decrement the counter by Get, compute new value, Accumulate
       */
      MPI_Win_fence(0,the_window);
      int counter_value;
      MPI_Get( &counter_value,1,MPI_INT,
               counter_process,0,1,MPI_INT,
               the_window);
      MPI_Win_fence(0,the_window);
      if (i_am_available) {
#ifdef DEBUG
        printf("[%d] updating in step %d\n",procno,step);
#endif
        my_counter_values[n_my_counter_values++] = counter_value;
        total_decrement++;
        int decrement = -1;
        MPI_Accumulate
          ( &decrement,       1,MPI_INT,
            counter_process,0,1,MPI_INT,
            MPI_SUM,
            the_window);
      }
      MPI_Win_fence(0,the_window);
    }

    /*
     * What counter values were actually obtained?
```

```
     */
    gather_sort_print( my_counter_values,n_my_counter_values, comm );

    /*
     * We do a correctness test by computing what the
     * window_data is supposed to be
     */
    {
      MPI_Win_fence(0,the_window);
      int counter_value;
      MPI_Get( /* origin data to set: */ &counter_value,1,MPI_INT,
               /* window data to get: */ counter_process,0,1,MPI_INT,
               the_window);
      MPI_Win_fence(0,the_window);
      MPI_Allreduce(MPI_IN_PLACE,&total_decrement,1,MPI_INT,MPI_SUM,comm);
      if (procno==counter_process) {
        if (counter_init-total_decrement==counter_value)
          printf("[%d] initial counter %d decreased by %d correctly giving %d\n",
                 procno,counter_init,total_decrement,counter_value);
        else
          printf("[%d] initial counter %d decreased by %d, giving %d s/b %d\n",
                 procno,counter_init,total_decrement,counter_value,counter_init-total_decrement);
      }
    }
    MPI_Win_free(&the_window);
  }

  MPI_Finalize();
  return 0;
}
```

9.9.14 Listing of code examples/mpi/c/windynamic.c

```
#include <stdlib.h>
#include <mpi.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc,char **argv) {

#include "globalinit.c"

  {
    MPI_Win the_window;
    int origin=0, data_proc = nprocs-1;
    int *retrieve=NULL,*window_buffer=NULL;

    MPI_Win_create_dynamic(MPI_INFO_NULL,comm,&the_window);
    if (procno==data_proc)
      window_buffer = (int*) malloc( 2*sizeof(int) );
      MPI_Win_attach(the_window,window_buffer,2*sizeof(int));

    if (procno==data_proc) {
```

```
      window_buffer[0] = 1;
      window_buffer[1] = 27;
    }
    if (procno==origin) {
      retrieve = (int*) malloc( sizeof(int) );
    }

    MPI_Aint data_address;
    if (procno==data_proc) {
      MPI_Get_address(window_buffer,&data_address);
    }
    MPI_Bcast(&data_address,1,MPI_LONG,data_proc,comm);

    MPI_Win_fence(0,the_window);
    if (procno==origin) {
      MPI_Aint disp = data_address+1*sizeof(int);
      MPI_Get( /* data on origin: */           retrieve, 1,MPI_INT,
       /* data on target: */ data_proc,disp,     1,MPI_INT,
       the_window);
    }
    MPI_Win_fence(0,the_window);

    if (procno==origin)
      printf("I got the following: %d\n",retrieve[0]);
    MPI_Win_free(&the_window);
  }

  MPI_Finalize();
  return 0;
}
```

Chapter 10

MPI topic: File I/O

This chapter discusses the I/O support of MPI, which is intended to alleviate the problems inherent in parallel file access. Let us first explore the issues. This story partly depends on what sort of parallel computer are you running on.

- On networks of workstations each node will have a separate drive with its own file system.
- On many clusters there will be a *shared file system* that acts as if every process can access every file.
- Cluster nodes may or may not have a private file system.

Based on this, the following strategies are possible, even before we start talking about MPI I/O.

- One process can collect all data with `MPI_Gather` and write it out. There are at least three things wrong wit this: it uses network bandwidth for the gather, it may require a large amount of memory on the root process, and centralized writing is a bottlenectk.
- Absent a shared file system, writing can be parallelized by letting every process create a unique file and merge these after the run. This makes the I/O symmetric, but collecting all the files is a bottleneck.
- Even with a with a shared file system this approach is possible, but it can put a lot of strain on the file system, and the post-processing can be a significant task.
- Using a shared file system, there is nothing against every process opening an existing file for reading, and using an individual file pointer to get its unique data.
- … but having every process open the same file for output is probably not a good idea. For instance, if two processes try to write at the end of the file, you may need to synchronize them, and synchronize the file system flushes.

For these reasons, MPI has a number of routines that make it possible to read and write a single file from a large number of processes, giving each well-defined locations where to access the data. In fact, MPI-IO uses MPI *derived datatype*s for both the source data (that is, in memory) and target data (that is, on disk). Thus, in one call that is collective on a communicator each process can address data that is not contiguous in memory, and place it in locations that are not contiguous on disc.

There are dedicated libraries for file I/O, such as *hdf5*, *netcdf*, or *silo*. However, these often add header inforation to a file that may not be understandable to post-processing applications. With MPI I/O you are in complete control of what goes to the file. (A useful tool for viewing your file is the unix utility `od`.)

Figure 10.1 `MPI_File_open`

Name	Param name	C type	F type	inout
`mpi_file_open (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`filename` *name of file to open*	`const char*`	`CHARACTER`	`in`
	`amode` *file access mode*	`int`	`INTEGER`	`in`
	`info` *info object*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`fh` *new file handle*	`MPI_File*`	`TYPE(MPI_File)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`File.Open(`		
	`cls`	
	`comm`	`Intracomm`
	`filename`	`str`
	`amode`	`int = Ellipsis`
	`info`	`Info = Ellipsis`
`) -> File`		

TACC note. Each node has a private `/tmp` file system (typically flash storage), to which you can write files. Considerations:

- Since these drives are separate from the shared file system, you don't have to worry about stress on the file servers.
- These temporary file systems are wiped after your job finishes, so you have to do the post-processing in your job script.
- The capacity of these local drives are fairly limited; see the userguide for exact numbers.

10.1 File handling

MPI has its own file handle: `MPI_File`.

You open a file with `MPI_File_open` (figure 10.1). This routine is collective, even if only certain processes will access the file with a read or write call. Similarly, `MPI_File_close` is collective.

Python note. Note the slightly unusual syntax for opening a file:

```
mpifile = MPI.File.Open(comm,filename,mode)
```

Even though the file is opened on a communicator, it is a class method for the `MPI.File` class, rather than for the communicator object. The latter is passed in as an argument.

File access modes:

Figure 10.2 `MPI_File_seek`

Name	Param name	C type	F type	inout
`mpi_file_seek (`				
	`fh` *file handle*	`MPI_File`	`TYPE(MPI_File)`	`in`
	`offset` *file offset*	`MPI_Offset`	`INTEGER(KIND=MPI_OFFSET_KIND)`	`in`
	`whence` *update mode*	`int`	`INTEGER`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`File.Seek(`		
	`self`	
	`offset`	`int`
	`whence`	`int = Ellipsis`
`) -> None`		

- `MPI_MODE_RDONLY`: read only,
- `MPI_MODE_RDWR`: reading and writing,
- `MPI_MODE_WRONLY`: write only,
- `MPI_MODE_CREATE`: create the file if it does not exist,
- `MPI_MODE_EXCL`: error if creating file that already exists,
- `MPI_MODE_DELETE_ON_CLOSE`: delete file on close,
- `MPI_MODE_UNIQUE_OPEN`: file will not be concurrently opened elsewhere,
- `MPI_MODE_SEQUENTIAL`: file will only be accessed sequentially,
- `MPI_MODE_APPEND`: set initial position of all file pointers to end of file.

These modes can be added or bitwise-or'ed.

You can delete a file with `MPI_File_delete`.

Buffers can be flushed with `MPI_File_sync`, which is a collective call.

10.2 File reading and writing

The basic file operations, in between the open and close calls, are the POSIX-like, noncollective, calls

- `MPI_File_seek` (figure 10.2). The *`whence`* parameter can be:
  - `MPI_SEEK_SET` The pointer is set to offset.
  - `MPI_SEEK_CUR` The pointer is set to the current pointer position plus offset.
  - `MPI_SEEK_END` The pointer is set to the end of the file plus offset.
- `MPI_File_write` (figure 10.3). This routine writes the specified data in the locations specified with the current file view. The number of items written is returned in the `MPI_Status` argument; all other fields of this argument are undefined. It can not be used if the file was opened with `MPI_MODE_SEQUENTIAL`.

Figure 10.3 `MPI_File_write`

Name	Param name	C type	F type	inout
`mpi_file_write (`				
	`fh` *file handle*	`MPI_File`	`TYPE(MPI_File)`	in
	`buf` *initial address of buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	in
	`count` *number of elements in buffer*	`int`	`INTEGER`	in
	`datatype` *datatype of each buffer element*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	in
	`status` *status object*	`MPI_Status*`	`TYPE(MPI_Status)`	out
`(opt)`	`ierror`		`INTEGER`	out
`)`				

Python:

Name	Param name	param type
`File.Write(`		
	`self`	
	`buf`	`BufSpec`
	`status`	`Optional[Status] = None`
`) -> None`		

- If all processes execute a write at the same logical time, it is better to use the collective call `MPI_File_write_all`.
- `MPI_File_read` (figure 10.4) This routine attempts to read the specified data from the locations specified in the current file view. The number of items read is returned in the `MPI_Status` argument; all other fields of this argument are undefined. It can not be used if the file was opened with `MPI_MODE_SEQUENTIAL`.
- If all processes execute a read at the same logical time, it is better to use the collective call `MPI_File_read_all` (figure 10.5).

For thread safety it is good to combine seek and read/write operations:

- `MPI_File_read_at`: combine read and seek. The collective variant is `MPI_File_read_at_all`.
- `MPI_File_write_at`: combine write and seekl The collective variant is `MPI_File_write_at_all`.

Writing to and reading from a parallel file is rather similar to sending a receiving:

- The process uses an elementary data type or a derived datatype to describe what elements in an array go to file, or are read from file.
- In the simplest case, your read or write that data to the file using an offset, or first having done a seek operation.
- But you can also set a 'file view' to describe explicitly what elements in the file will be involved.

10.2.1 Nonblocking read/write

Just like there are blocking and nonblocking sends, there are also nonblocking writes and reads: `MPI_File_iwrite` (figure 10.6), `MPI_File_iread` operations, and their collective versions `MPI_File_iwrite_all`, `MPI_File_iread_all` .

Figure 10.4 `MPI_File_read`

Name	Param name	C type	F type	inout
`mpi_file_read (`				
	`fh` *file handle*	`MPI_File`	`TYPE(MPI_File)`	`in`
	`buf` *initial address of buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`count` *number of elements in buffer*	`int`	`INTEGER`	`in`
	`datatype` *datatype of each buffer element*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`status` *status object*	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`File.Read(`		
	`self`	
	`buf`	`BufSpec`
	`status`	`Optional[Status] = None`
`) -> None`		

Figure 10.5 `MPI_File_read_all`

Name	Param name	C type	F type	inout
`mpi_file_read_all (`				
	`fh` *file handle*	`MPI_File`	`TYPE(MPI_File)`	`in`
	`buf` *initial address of buffer*	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	`count` *number of elements in buffer*	`int`	`INTEGER`	`in`
	`datatype` *datatype of each buffer element*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`status` *status object*	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`File.Read_all(`		
	`self`	
	`buf`	`BufSpec`
	`status`	`Optional[Status] = None`
`) -> None`		

Figure 10.6 `MPI_File_iwrite`

Name	Param name	C type	F type	inout
`mpi_file_iwrite (`				
	`fh` *file handle*	`MPI_File`	`TYPE(MPI_File)`	in
	`buf` *initial address of buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	in
	`count` *number of elements in buffer*	`int`	`INTEGER`	in
	`datatype` *datatype of each buffer element*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	in
	`request` *request object*	`MPI_Request*`	`TYPE(MPI_Request)`	out
`(opt)`	`ierror`		`INTEGER`	out
`)`				

Python:

Name	Param name	param type
`File.Iwrite(`		
	`self`	
	`buf`	`BufSpec`
`) -> Request`		

Also `MPI_File_iwrite_at_all`, `MPI_File_iread_at_all`.

These routines output an `MPI_Request` object, which can then be tested with `MPI_Wait` or `MPI_Test`.

Nonblocking collective I/O functions much like other nonblocking collectives (section sec:mpi3collect): the request is satisfied if all processes finish the collective.

There are also *split collective* routines that function like nonblocking collective I/O, but with the request/wait mechanism: `MPI_File_write_all_begin` / `MPI_File_write_all_end` (and similarly `MPI_File_read_all_begin` / `MPI_File_read_all_end`) where the second routine blocks until the collective write/read has been concluded.

Also `MPI_File_iread_shared`, `MPI_File_iwrite_shared`.

10.2.2 Individual file pointers, contiguous writes

After the collective open call, each process holds an *individual file pointer* each process can individually position the pointer somewhere in the shared file. Let's explore this modality.

The simplest way of writing a data to file is much like a send call: a buffer is specified with the usual count/datatype specification, and a target location in the file is given. The routine `MPI_File_write_at` (figure 10.7) gives this location in absolute terms with a parameter of type `MPI_Offset`, which counts bytes.

Exercise 10.1. Create a buffer of length `nwords=3` on each process, and write these buffers as a sequence to one file with `MPI_File_write_at`.
(There is a skeleton for this exercise under the name `blockwrite`.)

Instead of giving the position in the file explicitly, you can also use a `MPI_File_seek` call to position the file pointer, and write with `MPI_File_write` at the pointer location. The write call itself also *advances the file*

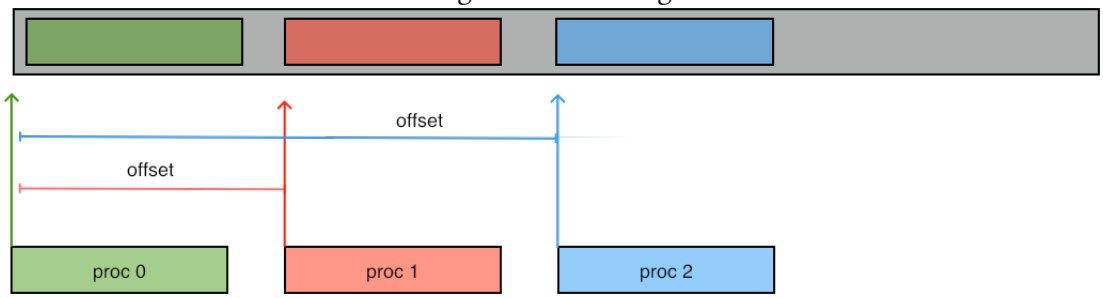
Figure 10.7 `MPI_File_write_at`

Name	Param name	C type	F type	inout
`mpi_file_write_at (`				
	`fh` *file handle*	`MPI_File`	`TYPE(MPI_File)`	`in`
	`offset` *file offset*	`MPI_Offset`	`INTEGER(KIND=MPI_OFFSET_KIND)`	`in`
	`buf` *initial address of buffer*	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	`count` *number of elements in buffer*	`int`	`INTEGER`	`in`
	`datatype` *datatype of each buffer element*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`status` *status object*	`MPI_Status*`	`TYPE(MPI_Status)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`File.Write_at(`		
	`self`	
	`offset`	`int`
	`buf`	`BufSpec`
	`status`	`Optional[Status] = None`
`) -> None`		

Figure 10.1: Writing at an offset



pointer so separate calls for writing contiguous elements need no seek calls with `MPI_SEEK_CUR`.

Exercise 10.2. Rewrite the code of exercise 10.1 to use a loop where each iteration writes only one item to file. Note that no explicit advance of the file pointer is needed.

Exercise 10.3. Construct a file with the consecutive integers $0, \ldots, WP$ where W some integer, and P the number of processes. Each process p writes the numbers $p, p+W, p+2W, \ldots$. Use a loop where each iteration

1. writes a single number with `MPI_File_write`, and
2. advanced the file pointer with `MPI_File_seek` with a *whence* parameter of `MPI_SEEK_CUR`.

Figure 10.8 `MPI_File_set_view`

Name	Param name	C type	F type	inout
`mpi_file_set_view (`				
	`fh` *file handle*	`MPI_File`	`TYPE(MPI_File)`	`in`
	`disp` *displacement*	`MPI_Offset`	`INTEGER(KIND=MPI_OFFSET_KIND)`	`in`
	`etype` *elementary datatype*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`filetype` *filetype*	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	`datarep` *data representation*	`const char*`	`CHARACTER`	`in`
	`info` *info object*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`File.Set_view(`		
	`self`	
	`disp`	`int = 0`
	`etype`	`Datatype = Ellipsis`
	`filetype`	`Optional[Datatype] = None`
	`datarep`	`str = native`
	`info`	`Info = Ellipsis`
`) -> None`		

10.2.3 File views

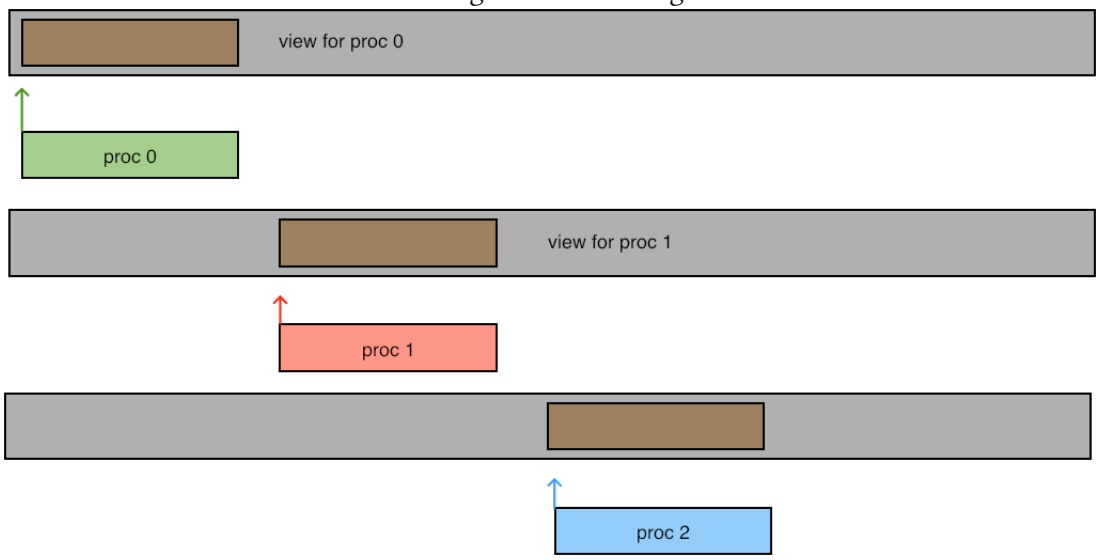
The previous mode of writing is enough for writing simple contiguous blocks in the file. However, you can also access noncontiguous areas in the file. For this you use `MPI_File_set_view` (figure 10.8). This call is collective, even if not all processes access the file.

- The `disp` displacement parameters is measured in bytes. It can differ between processes. On sequential files such as tapes or network streams it does not make sense to set a displacement; for those the `MPI_DISPLACEMENT_CURRENT` value can be used.
- The `etype` describes the data type of the file, it needs to be the same on all processes.
- The `filetype` describes how this process sees the file, so it can differ between processes.
- The `datarep` string can have the following values:
  - `native`: data on disk is represented in exactly the same format as in memory;
  - `internal`: data on disk is represented in whatever internal format is used by the MPI implementation;
  - `external`: data on disk is represented using XDR portable data formats.
- The `info` parameter is an `MPI_Info` object, or `MPI_INFO_NULL`. See section 14.1.1.3 for more. (See *T3PIO* [19] for a tool that assists in setting this object.)

Exercise 10.4.

(There is a skeleton for this exercise under the name `viewwrite`.) Write a file in the same way as in exercise 10.1, but now use `MPI_File_write` and use `MPI_File_set_view`

Figure 10.2: Writing at a view



to set a view that determines where the data is written.

You can get very creative effects by setting the view to a derived datatype.

> *Fortran note.* In Fortran you have to assure that the displacement parameter is of 'kind' `MPI_OFFSET_KIND`. In particular, you can not specify a literal zero '0' as the displacement; use `0_MPI_OFFSET_KIND` instead.

More: `MPI_File_set_size` `MPI_File_get_size` `MPI_File_preallocate` `MPI_File_get_view`

10.2.4 Shared file pointers

It is possible to have a file pointer that is shared (and therefore identical) between all processes of the communicator that was used to open the file. This file pointer is set with `MPI_File_seek_shared`. For reading and writing there are then two sets of routines:

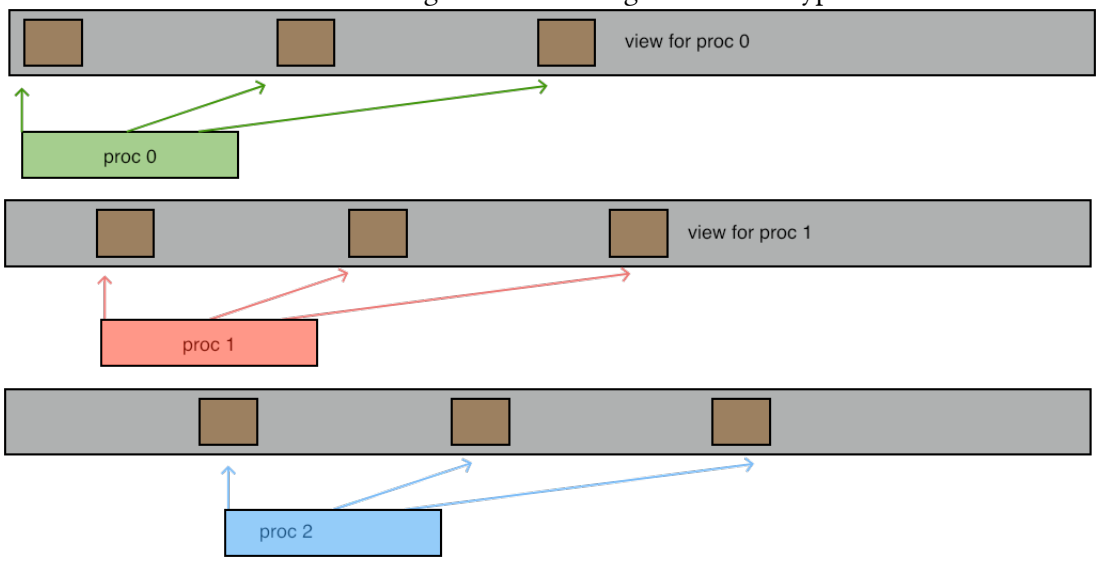
- Individual accesses are done with `MPI_File_read_shared` and `MPI_File_write_shared`. Nonblocking variants are `MPI_File_iread_shared` and `MPI_File_iwrite_shared`.
- Collective accesse are done with `MPI_File_read_ordered` and `MPI_File_write_ordered`, which execute the operations in order ascending by rank.

Shared file pointers require that the same view is used on all processes. Also, these operations are less efficient because of the need to maintain the shared pointer.

10.3 Consistency

It is possible for one process to read data previously writte by another process. For this it is of course necessary to impose a temporal order, for instance by using `MPI_Barrier`, or using a zero-byte send from the writing to the reading process.

Figure 10.3: Writing at a derived type



However, the file also needs to be declared *atomic*: `MPI_File_set_atomicity`.

10.4 Constants

`MPI_SEEK_SET` used to be called `SEEK_SET` which gave conflicts with the C++ library. This had to be circumvented with

```
make CPPFLAGS="-DMPICH_IGNORE_CXX_SEEK -DMPICH_SKIP_MPICXX"
```

and such.

10.5 Review questions

Exercise 10.5. T/F? After your *SLURM* job ends, you can copy from the login node the files you've written to `\tmp`.

Exercise 10.6. T/F? File views (`MPI_File_set_view`) are intended to

- write MPI derived types to file; without them you can only write contiguous buffers;
- prevent collisions in collective writes; they are not needed for individual writes.

Exercise 10.7. The sequence `MPI_File_seek_shared`, `MPI_File_read_shared` can be replaced by `MPI_File_seek`, `MPI_File_read` if you make what changes?

10.6 Sources used in this chapter

10.6.1 Listing of code header

Chapter 11

MPI topic: Topologies

A communicator describes a group of processes, but the structure of your computation may not be such that every process will communicate with every other process. For instance, in a computation that is mathematically defined on a Cartesian 2D grid, the processes themselves act as if they are two-dimensionally ordered and communicate with N/S/E/W neighbours. If MPI had this knowledge about your application, it could conceivably optimize for it, for instance by renumbering the ranks so that communicating processes are closer together physically in your cluster.

The mechanism to declare this structure of a computation to MPI is known as a *virtual topology*. The following types of topology are defined:

- `MPI_UNDEFINED`: this values holds for communicators where no topology has explicitly been specified.
- `MPI_CART`: this value holds for Cartesian toppologies, where processes act as if they are ordered in a multi-dimensional 'brick'; see section 11.1.
- `MPI_GRAPH`: this value describes the graph topology that was defined in MPI-1; section 11.2.4. It is unnecessarily burdensome, since each process needs to know the total graph, and should therefore be considered obsolete; the type `MPI_DIST_GRAPH` should be used instead.
- `MPI_DIST_GRAPH`: this value describes the distributed graph topology where each process only describes the edges in the process graph that touch itself; see section 11.2.

These values can be discovered with the routine `MPI_Topo_test` (figure 11.1).

11.1 Cartesian grid topology

A *Cartesian grid* is a structure, typically in 2 or 3 dimensions, of points that have two neighbours in each of the dimensions. Thus, if a Cartesian grid has sizes $K \times M \times N$, its points have coordinates (k, m, n) with $0 \leq k < K$ et cetera. Most points have six neighbours $(k \pm 1, m, n)$, $(k, m \pm 1, n)$, $(k, m, n \pm 1)$; the exception are the edge points. A grid where edge processors are connected through *wraparound connections* is called a *periodic grid*.

The most common use of Cartesian coordinates is to find the rank of process by referring to it in grid terms. For instance, one could ask 'what are my neighbours offset by $(1, 0, 0)$, $(-1, 0, 0)$, $(0, 1, 0)$ et cetera'.

Figure 11.1 `MPI_Topo_test`

Name	Param name	C type	F type	inout
`mpi_topo_test (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`status` *topology type of communicator comm*	`int*`	`INTEGER`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

While the Cartesian topology interface is fairly easy to use, as opposed to the more complicated general graph topology below, it is not actually sufficient for all Cartesian graph uses. Notably, in a so-called *star stencil*, such as the *nine-point stencil*, there are diagonal connections, which can not be described in a single step. Instead, it is necessary to take a separate step along each coordinate dimension. In higher dimensions this is of course fairly awkward.

Thus, even for Cartesian structures, it may be advisable to use the general graph topology interface.

11.1.1 Cartesian routines

The cartesian topology is specified by giving `MPI_Cart_create` the sizes of the processor grid along each axis, and whether the grid is periodic along that axis.

```
int MPI_Cart_create(
  MPI_Comm comm_old, int ndims, int *dims, int *periods,
  int reorder, MPI_Comm *comm_cart)
```

Each point in this new communicator has a coordinate and a rank. They can be queried with `MPI_Cart_coords` and `MPI_Cart_rank` respectively.

```
int MPI_Cart_coords(
  MPI_Comm comm, int rank, int maxdims,
  int *coords);
int MPI_Cart_rank(
  MPI_Comm comm, init *coords,
  int *rank);
```

Note that these routines can give the coordinates for any process, not just for the current process.

```
// cart.c
MPI_Comm comm2d;
ndim = 2; periodic[0] = periodic[1] = 0;
dimensions[0] = idim; dimensions[1] = jdim;
MPI_Cart_create(comm,ndim,dimensions,periodic,1,&comm2d);
MPI_Cart_coords(comm2d,procno,ndim,coord_2d);
MPI_Cart_rank(comm2d,coord_2d,&rank_2d);
printf("I am %d: (%d,%d); originally %d\n",rank_2d,coord_2d[0],coord_2d[1],procno);
```

For the full source of this example, see section 11.3.2

The `reorder` parameter to `MPI_Cart_create` indicates whether processes can have a rank in the new communicator that is different from in the old one.

Strangely enough you can only shift in one direction, you can not specify a shift vector.

```
int MPI_Cart_shift(MPI_Comm comm, int direction, int displ, int *source,
                   int *dest)
```

If you specify a processor outside the grid the result is MPI_PROC_NULL.

```
char mychar = 65+procno;
MPI_Cart_shift(comm2d,0,+1,&rank_2d,&rank_right);
MPI_Cart_shift(comm2d,0,-1,&rank_2d,&rank_left);
MPI_Cart_shift(comm2d,1,+1,&rank_2d,&rank_up);
MPI_Cart_shift(comm2d,1,-1,&rank_2d,&rank_down);
int irequest = 0; MPI_Request *requests = malloc(8*sizeof(MPI_Request));
MPI_Isend(&mychar,1,MPI_CHAR,rank_right, 0,comm, requests+irequest++);
MPI_Isend(&mychar,1,MPI_CHAR,rank_left,  0,comm, requests+irequest++);
MPI_Isend(&mychar,1,MPI_CHAR,rank_up,    0,comm, requests+irequest++);
MPI_Isend(&mychar,1,MPI_CHAR,rank_down,  0,comm, requests+irequest++);
MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_right, 0,comm, requests+irequest++);
MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_left,  0,comm, requests+irequest++);
MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_up,    0,comm, requests+irequest++);
MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_down,  0,comm, requests+irequest++);
```

For the full source of this example, see section 11.3.2

11.2 Distributed graph topology

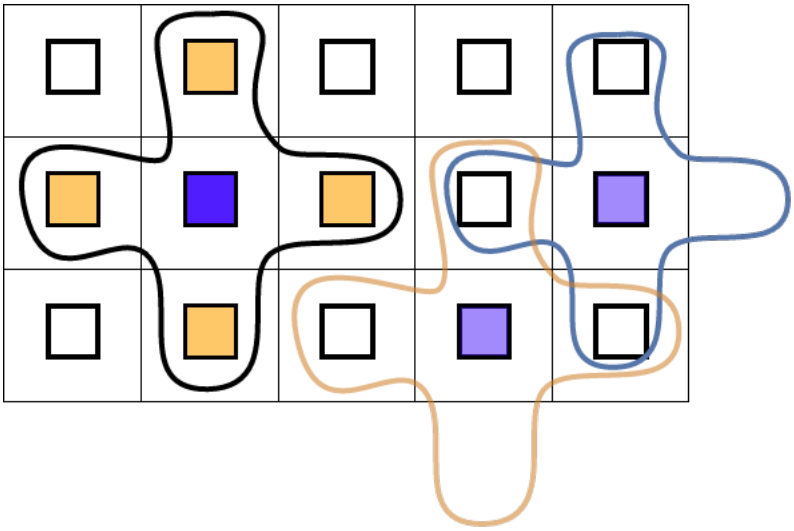


Figure 11.1: Illustration of a distributed graph topology where each node has four neighbours

In many calculations on a grid (using the term in its mathematical, Finite Element Method (FEM), sense), a grid point will collect information from grid points around it. Under a sensible distribution of the grid over processes, this means that each process will collect information from a number of neighbour processes.

The number of neighbours is dependent on that process. For instance, in a 2D grid (and assuming a five-point stencil for the computation) most processes communicate with four neighbours; processes on the edge with three, and processes in the corners with two.

Such a topology is illustrated in figure 11.1.

MPI's notion of *graph topology*, and the *neighbourhood collectives*, offer an elegant way of expressing such communication structures. There are various reasons for using graph topologies over the older, simpler methods.

- MPI is allowed to reorder the processes, so that network proximity in the cluster corresponds to proximity in the structure of the code.
- Ordinary collectives could not directly be used for graph problems, unless one would adopt a subcommunicator for each graph neighbourhood. However, scheduling would then lead to deadlock or serialization.
- The normal way of dealing with graph problems is through nonblocking communications. However, since the user indicates an explicit order in which they are posted, congestion at certain processes may occur.
- Collectives can pipeline data, while send/receive operations need to transfer their data in its entirety.
- Collectives can use spanning trees, while send/receive uses a direct connection.

Thus the minimal description of a process graph contains for each process:

- Degree: the number of neighbour processes; and
- the ranks of the processes to communicate with.

However, this ignores that communication is not always symmetric: maybe the processes you receive from are not the ones you send to. Worse, maybe only one side of this duality is easily described. Therefore, there are two routines:

- `MPI_Dist_graph_create_adjacent` assumes that a process knows both who it is sending it, and who will send to it. This is the most work for the programmer to specify, but it is ultimately the most efficient.
- `MPI_Dist_graph_create` specifies on each process only what it is the source for; that is, who this process will be sending to. Consequently, some amount of processing – including communication – is needed to build the converse information, the ranks that will be sending to a process.

11.2.1 Graph creation

There are two creation routines for process graphs. These routines are fairly general in that they allow any process to specify any part of the topology. In practice, of course, you will mostly let each process describe its own neighbour structure.

The routine `MPI_Dist_graph_create_adjacent` assumes that a process knows both who it is sending it, and who will send to it. This means that every edge in the communication graph is represented twice, so the memory footprint is double of what is strictly necessary. However, no communication is needed to build the graph.

The second creation routine, `MPI_Dist_graph_create` (figure 11.2), is probably easier to use, especially in

Figure 11.2 `MPI_Dist_graph_create`

Name	Param name	C type	F type	inout
`mpi_dist_graph_create (`				
	`comm_old` *input communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`n` *number of source nodes for which this process specifies edges*	`int`	`INTEGER`	`in`

	`sources`
	`length: n`
	array containing the n source nodes for which this process specifies edges
		`const int[]`	`INTEGER`	`in`

	`degrees`
	`length: n`
	array specifying the number of destinations for each source node in the source node array
		`const int[]`	`INTEGER`	`in`

	`destinations`
	`length: *`
	destination nodes for the source nodes in the source node array
		`const int[]`	`INTEGER`	`in`

	`weights`
	`length: *`
	weights for source to destination edges
		`const int[]`	`INTEGER`	`in`

	`info` *hints on optimization and interpretation of weights*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`reorder` *the ranks may be reordered (true) or not (false)*	`int`	`LOGICAL`	`in`
	`comm_dist_graph` *communicator with distributed graph topology added*	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

MPL:

```
dist_graph_communicator
   (const communicator &old_comm,
    const source_set &ss, const dest_set &ds, bool reorder=true)
where:
class dist_graph_communicator::source_set : private set< pair<int,int> >
class dist_graph_communicator::dest_set   : private set< pair<int,int> >
```

Python:

```
MPI.Comm.Create_dist_graph
    (self, sources, degrees, destinations, weights=None, Info info=INFO_NULL, bool reorder=False)
returns graph communicator
```

cases where the communication structure of your program is symmetric, meaning that a process sends to the same neighbours that it receives from. Now you specify on each process only what it is the source for; that is, who this process will be sending to.[1]. Consequently, some amount of processing – including communication – is needed to build the converse information, the ranks that will be sending to a process.

MPL note 55. The class `mpl::dist_graph_communicator` only has a constructor corresponding to `MPI_Dist_graph_create`.

End of MPL note

Figure 11.1 describes the common five-point stencil structure. If we let each process only describe itself, we get the following:

1. I disagree with this design decision. Specifying your sources is usually easier than specifying your destinations.

- `nsources`= 1 because the calling process describes on node in the graph: itself.
- `sources` is an array of length 1, containing the rank of the calling process.
- `degrees` is an array of length 1, containing the degree (probably: 4) of this process.
- `destinations` is an array of length the degree of this process, probably again 4. The elements of this array are the ranks of the neighbour nodes; strictly speaking the ones that this process will send to.
- `weights` is an array declaring the relative importance of the destinations. For an *unweighted graph* use `MPI_UNWEIGHTED`. In the case the graph is weighted, but the degree of a source is zero, you can pass an empty array as `MPI_WEIGHTS_EMPTY`.
- `reorder` (`int` in C, `LOGICAL` in Fortran) indicates whether MPI is allowed to shuffle processes to achieve greater locality.

The resulting communicator has all the processes of the original communicator, with the same ranks. In other words `MPI_Comm_size` and `MPI_Comm_rank` gives the same values on the graph communicator, as on the intra-communicator that it is constructed from. To get information about the grouping, use `MPI_Dist_graph_neighbors` and `MPI_Dist_graph_neighbors_count`.

Python note. Graph communicator creation is a method of the `Comm` class, and the graph communicator is a function return result:

```
graph_comm = oldcomm.Create_dist_graph(sources, degrees, destinations)
```

The weights, info, and reorder arguments have default values.

11.2.2 Neighbour collectives

We can now use the graph topology to perform a gather or allgather `MPI_Neighbor_allgather` (figure 11.3) that combines only the processes directly connected to the calling process.

The neighbour collectives have the same argument list as the regular collectives, but they apply to a graph communicator.

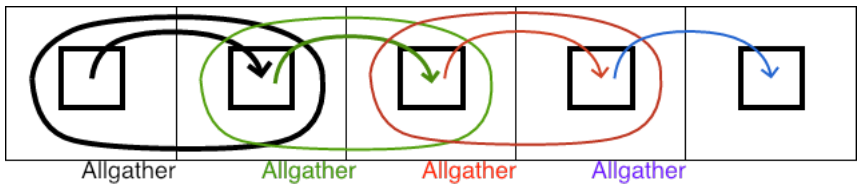


Figure 11.2: Solving the right-send exercise with neighbourhood collectives

Exercise 11.1. Revisit exercise 4.3 and solve it using `MPI_Dist_graph_create`. Use figure 11.2 for inspiration.
Use a degree value of 1.
(There is a skeleton for this exercise under the name `rightgraph`.)

The previous exercise can be done with a degree value of:

- 1, reflecting that each process communicates with just 1 other; or

Figure 11.3 `MPI_Neighbor_allgather`

Name	Param name	C type	F type	inout
`mpi_neighbor_allgather (`				
	`sendbuf`	`const void*`	`TYPE(*), DIMENSION(..)`	`in`
	starting address of send buffer			
	`sendcount`	`int`	`INTEGER`	`in`
	number of elements sent to each neighbor			
	`sendtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of send buffer elements			
	`recvbuf`	`void*`	`TYPE(*), DIMENSION(..)`	`out`
	starting address of receive buffer			
	`recvcount`	`int`	`INTEGER`	`in`
	number of elements received from each neighbor			
	`recvtype`	`MPI_Datatype`	`TYPE(MPI_Datatype)`	`in`
	data type of receive buffer elements			
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator with topology structure			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Topocomm.neighbor_allgather(`		
	`self`	
	`sendobj`	`Any`
`) -> List[Any]`		

- 2, reflecting that you really gather from two processes.

In the latter case, results do not wind up in the receive buffer in order of increasing process number as with a traditional gather. Rather, you need to use `MPI_Dist_graph_neighbors` to find their sequencing; see section 11.2.3.

Another neighbor collective is `MPI_Neighbor_alltoall`.

The vector variants are `MPI_Neighbor_allgatherv` and `MPI_Neighbor_alltoallv`.

There is a heterogenous (multiple datatypes) variant: `MPI_Neighbor_alltoallw`.

The list is: `MPI_Neighbor_allgather`, `MPI_Neighbor_allgatherv`, `MPI_Neighbor_alltoall`, `MPI_Neighbor_alltoallv`, `MPI_Neighbor_alltoallw`.

Nonblocking: `MPI_Ineighbor_allgather`, `MPI_Ineighbor_allgatherv`, `MPI_Ineighbor_alltoall`, `MPI_Ineighbor_alltoallv`, `MPI_Ineighbor_alltoallw`.

For unclear reasons there is no *`MPI_Neighbor_allreduce`*.

11.2.3 Query

There are two routines for querying the neighbors of a process: `MPI_Dist_graph_neighbors_count` and `MPI_Dist_graph_neighbors`.

While this information seems derivable from the graph construction, that is not entirely true for two reasons.

1. With the nonadjoint version `MPI_Dist_graph_create`, only outdegrees and destinations are specified; this call then supplies the indegrees and sources;
2. As observed above, the order in which data is placed in the receive buffer of a gather call is not determined by the create call, but can only be queried this way.

11.2.4 Graph topology (deprecated)

The original MPI-1 had a graph topology interface `MPI_Graph_create` which required each process to specify the full process graph. Since this is not scalable, it should be considered deprecated. Use the distributed graph topology (section 11.2) instead.

Other legacy routines: `MPI_Graph_neighbors`, `MPI_Graph_neighbors_count`, `MPI_Graph_get`, `MPI_Graphdims_get`.

11.3 Sources used in this chapter

11.3.1 Listing of code header

11.3.2 Listing of code examples/mpi/c/cart.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (nprocs<4) {
    printf("This program needs at least four processes\n");
    return -1;
  }

  //
  int idim,jdim;
  int ndim,periodic[2],dimensions[2],coord_2d[2],rank_2d;
  for (idim=(int)(sqrt(1.*nprocs)); idim>=2; idim--) {
    jdim = nprocs/idim;
    if (idim*jdim==nprocs) goto found;
  }
  printf("No prime numbers please\n"); return -1;
  MPI_Comm comm2d;
 found:

  ndim = 2; periodic[0] = periodic[1] = 0;
  dimensions[0] = idim; dimensions[1] = jdim;
  MPI_Cart_create(comm,ndim,dimensions,periodic,1,&comm2d);
  MPI_Cart_coords(comm2d,procno,ndim,coord_2d);
  MPI_Cart_rank(comm2d,coord_2d,&rank_2d);
  printf("I am %d: (%d,%d); originally %d\n",rank_2d,coord_2d[0],coord_2d[1],procno);

  int rank_left,rank_right,rank_up,rank_down;
  char indata[4]; int idata=0,sdata=0;
  for (int i=0; i<4; i++)
    indata[i] = 32;
  char mychar = 65+procno;
  MPI_Cart_shift(comm2d,0,+1,&rank_2d,&rank_right);
  MPI_Cart_shift(comm2d,0,-1,&rank_2d,&rank_left);
  MPI_Cart_shift(comm2d,1,+1,&rank_2d,&rank_up);
  MPI_Cart_shift(comm2d,1,-1,&rank_2d,&rank_down);
  int irequest = 0; MPI_Request *requests = malloc(8*sizeof(MPI_Request));
  MPI_Isend(&mychar,1,MPI_CHAR,rank_right, 0,comm, requests+irequest++);
  MPI_Isend(&mychar,1,MPI_CHAR,rank_left,  0,comm, requests+irequest++);
  MPI_Isend(&mychar,1,MPI_CHAR,rank_up,    0,comm, requests+irequest++);
```

```
  MPI_Isend(&mychar,1,MPI_CHAR,rank_down,  0,comm, requests+irequest++);
  MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_right, 0,comm, requests+irequest++);
  MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_left,  0,comm, requests+irequest++);
  MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_up,    0,comm, requests+irequest++);
  MPI_Irecv( indata+idata++, 1,MPI_CHAR, rank_down,  0,comm, requests+irequest++);
  MPI_Waitall(irequest,requests,MPI_STATUSES_IGNORE);
  printf("[%d] %s\n",procno,indata);
  /* for (int i=0; i<4; i++) */
  /*   sdata += indata[i]; */
  /* printf("[%d] %d,%d,%d,%d sum=%d\n",procno,indata[0],indata[1],indata[2],indata[3],sdata); */
  if (procno==0)
    printf("Finished\n");

  MPI_Finalize();
  return 0;
}
```

Chapter 12

MPI topic: Shared memory

Some programmers are under the impression that MPI would not be efficient on shared memory, since all operations are done through what looks like network calls. This is not correct: many MPI implementations have optimizations that detect shared memory and can exploit it, so that data is copied, rather than going through a communication layer. (Conversely, programming systems for shared memory such as *OpenMP* can actually have inefficiencies associated with thread handling.) The main inefficiency associated with using MPI on shared memory is then that processes can not actually share data.

The one-sided MPI calls (chapter 9) can also be used to emulate shared memory, in the sense that an origin process can access data from a target process without the target's active involvement. However, these calls do not distinguish between actually shared memory and one-sided access across the network.

In this chapter we will look at the ways MPI can interact with the presence of actual shared memory. (This functionality was added in the MPI-3 standard.) This relies on the `MPI_Win` windows concept, but otherwise uses direct access of other processes' memory.

12.1 Recognizing shared memory

MPI's one-sided routines take a very symmetric view of processes: each process can access the window of every other process (within a communicator). Of course, in practice there will be a difference in performance depending on whether the origin and target are actually on the same shared memory, or whether they can only communicate through the network. For this reason MPI makes it easy to group processes by shared memory domains using `MPI_Comm_split_type` (figure 12.1).

Here the `split_type` parameter has to be from the following (short) list:

- `MPI_COMM_TYPE_SHARED`: split the communicator into subcommunicators of processes sharing a memory area.

The following material is for the (unreleased) MPI-4 standard only

- `MPI_COMM_TYPE_HW_GUIDED` (MPI-4): split using an `info` value from `MPI_Get_hw_resource_types`.
- `MPI_COMM_TYPE_HW_UNGUIDED` (MPI-4): similar to `MPI_COMM_TYPE_HW_GUIDED`, but the resulting communicators should be a strict subset of the original communicator. On processes where this condition can not be fullfilled, `MPI_COMM_NULL` will be returned.

End of MPI-4 material

Figure 12.1 `MPI_Comm_split_type`

Name	Param name	C type	F type	inout
`mpi_comm_split_type (`				
	`comm` *communicator*	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	`split_type` *type of processes to be grouped together*	`int`	`INTEGER`	`in`
	`key` *control of rank assignment*	`int`	`INTEGER`	`in`
	`info` *info argument*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`newcomm` *new communicator*	`MPI_Comm*`	`TYPE(MPI_Comm)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Split_type(`		
	`self`	
	`split_type`	`int`
	`key`	`int = 0`
	`info`	`Info = Ellipsis`
`) -> Comm`		

MPL note 56. Similar to ordinary communicator splitting 54: `communicator::split_shared`.

End of MPL note

In the following example, `CORES_PER_NODE` is a platform-dependent constant:

```
// commsplittype.c
MPI_Info info;
MPI_Comm_split_type(MPI_COMM_WORLD,MPI_COMM_TYPE_SHARED,procno,info,&sharedcomm);
MPI_Comm_size(sharedcomm,&new_nprocs);
MPI_Comm_rank(sharedcomm,&new_procno);
```

For the full source of this example, see section 12.3.2

12.2 Shared memory for windows

Processes that exist on the same physical shared memory should be able to move data by copying, rather than through MPI send/receive calls – which of course will do a copy operation under the hood. In order to do such user-level copying:

1. We need to create a shared memory area with `MPI_Win_allocate_shared`, and
2. We need to get pointers to where a process' area is in this shared space; this is done with `MPI_Win_shared_query`.

Figure 12.2 `MPI_Win_allocate_shared`

Name	Param name	C type	F type	inout
`mpi_win_allocate_shared (`				
	`size`	`MPI_Aint`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`in`
	size of local window in bytes			
	`disp_unit`	`int`	`INTEGER`	`in`
	local unit size for displacements, in bytes			
	`info`	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	intra-communicator			
	`baseptr`	`void*`	`TYPE(C_PTR)`	`out`
	address of local allocated window segment			
	`win`	`MPI_Win*`	`TYPE(MPI_Win)`	`out`
	window object returned by the call			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Allocate_shared(`		
	`cls`	
	`size`	`int`
	`disp_unit`	`int = 1`
	`info`	`Info = Ellipsis`
	`comm`	`Intracomm = Ellipsis`
`) -> Win`		

12.2.1 Pointers to a shared window

The first step is to create a window (in the sense of one-sided MPI; section 9.1) on the processes on one node. Using the `MPI_Win_allocate_shared` (figure 12.2) call presumably will put the memory close to the *socket* on which the process runs.

```
// sharedbulk.c
MPI_Aint window_size; double *window_data; MPI_Win node_window;
if (onnode_procid==0)
  window_size = sizeof(double);
else window_size = 0;
MPI_Win_allocate_shared
  ( window_size,sizeof(double),MPI_INFO_NULL,
    nodecomm,
    &window_data,&node_window);
```

For the full source of this example, see section 12.3.3

The memory allocated by `MPI_Win_allocate_shared` is contiguous between the processes. This makes it possible to do address calculation. However, if a cluster node has a Non-Uniform Memory Access (NUMA) structure, for instance if two sockets have memory directly attached to each, this would increase latency for some processes. To prevent this, the key `alloc_shared_noncontig` can be set to `true` in the `MPI_Info` object.

The following material is for the (unreleased) MPI-4 standard only

In the contiguous case, the `mpi_minimum_memory_alignment` info argument (section 9.1.1) applies only to the

memory on the first process; in the noncontiguous case it applies to all.
End of MPI-4 material

```
// numa.c
MPI_Info window_info;
MPI_Info_create(&window_info);
  MPI_Info_set(window_info,"alloc_shared_noncontig","true");
MPI_Win_allocate_shared( window_size,sizeof(double),window_info,
                         nodecomm,
                         &window_data,&node_window);
MPI_Info_free(&window_info);
```

For the full source of this example, see section 12.3.4

Let's now consider a scenario where you spawn two MPI processes per node, and the node has 100G of memory. Using the above option to allow for noncontiguous window allocation, you hope that the windows of the two processes are placed 50G apart. However, if you print out the addresses, you will find that that they are placed considerably closer together. For a small windows that distance may be as little as 4K, the size of a *small page*.

The reason for this mismatch is that an address that you obtain with the ampersand operator in C is not a *physical address*, but a *virtual address*. The translation of where pages are placed in physical memory is determined by the *page table*.

12.2.2 Querying the shared structure

Even though the window created above is shared, that doesn't mean it's contiguous. Hence it is necessary to retrieve the pointer to the area of each process that you want to communicate with: `MPI_Win_shared_query` (figure 12.3).

```
MPI_Aint window_size0; int window_unit; double *win0_addr;
MPI_Win_shared_query( node_window,0,
                      &window_size0,&window_unit, &win0_addr );
```

For the full source of this example, see section 12.3.5

12.2.3 Heat equation example

As an example, which consider the 1D heat equation. On each process we create a local area of three point:

```
// sharedshared.c
MPI_Win_allocate_shared(3,sizeof(int),info,sharedcomm,&shared_baseptr,&shared_window);
```

For the full source of this example, see section 12.3.6

12.2.4 Shared bulk data

In applications such as *ray tracing*, there is a read-only large data object (the objects in the scene to be rendered) that is needed by all processes. In traditional MPI, this would need to be stored redundantly on each process, which leads to large memory demands. With MPI shared memory we can store the data

Figure 12.3 `MPI_Win_shared_query`

Name	Param name	C type	F type	inout
`mpi_win_shared_query (`				
	`win`	`MPI_Win`	`TYPE(MPI_Win)`	`in`
	shared memory window object			
	`rank`	`int`	`INTEGER`	`in`
	rank in the group of window win or `MPI_PROC_NULL`			
	`size`	`MPI_Aint*`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`out`
	size of the window segment			
	`disp_unit`	`int*`	`INTEGER`	`out`
	local unit size for displacements, in bytes			
	`baseptr`	`void*`	`TYPE(C_PTR)`	`out`
	address for load/store access to window segment			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Win.Shared_query(`		
	`self`	
	`rank`	`int`
`) -> Tuple[memory, int]`		

object once per node. Using as above `MPI_Comm_split_type` to find a communicator per NUMA domain, we store the object on process zero of this node communicator.

Exercise 12.1. Let the 'shared' data originate on process zero in `MPI_COMM_WORLD`. Then:

- create a communicator per shared memory domain;
- create a communicator for all the processes with number zero on their node;
- broadcast the shared data to the processes zero on each node.

(There is a skeleton for this exercise under the name `shareddata`.)

12.3 Sources used in this chapter

12.3.1 Listing of code header

12.3.2 Listing of code examples/mpi/c/commsplittype.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#ifndef CORES_PER_NODE
#define CORES_PER_NODE 16
#endif

int main(int argc,char **argv) {

#include "globalinit.c"

  if (nprocs<3) {
    printf("This program needs at least three processes\n");
    return -1;
  }

  if (procno==0)
    printf("There are %d ranks total\n",nprocs);

  int new_procno,new_nprocs;
  MPI_Comm sharedcomm;

  MPI_Info info;
  MPI_Comm_split_type(MPI_COMM_WORLD,MPI_COMM_TYPE_SHARED,procno,info,&sharedcomm);
  MPI_Comm_size(sharedcomm,&new_nprocs);
  MPI_Comm_rank(sharedcomm,&new_procno);

  ASSERT(new_procno<CORES_PER_NODE);

  if (new_procno==0) {
    char procname[MPI_MAX_PROCESSOR_NAME]; int namlen;
    MPI_Get_processor_name(procname,&namlen);
    printf("I am processor %d in a shared group of %d, running on %s\n",
   new_procno,new_nprocs,procname);
  }
  if (procno==0)
    printf("Finished\n");

  MPI_Finalize();
  return 0;
}
```

12.3.3 Listing of code examples/mpi/c/sharedbulk.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {
  MPI_Comm comm;
  int nprocs,procid;

  MPI_Init(&argc,&argv);
  comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_rank(comm,&procid);

  /*
   * Find the subcommunicator on the node,
   * and get the procid on the node.
   */
  MPI_Comm nodecomm; int onnode_procid;
  MPI_Comm_split_type
    (comm,MPI_COMM_TYPE_SHARED,procid,MPI_INFO_NULL,
     &nodecomm);
  MPI_Comm_rank(nodecomm,&onnode_procid);

  /*
   * Find the subcommunicators of
   * identical `onnode_procid' processes;
   * the procid on that communicator is the node ID
   */
  MPI_Comm crosscomm; int nodeid;
  MPI_Comm_split
    (comm,onnode_procid,procid,&crosscomm);
  MPI_Comm_rank(crosscomm,&nodeid);
  printf("[%d] %dx%d\n",procid,nodeid,onnode_procid);

  /*
   * Create data on global process zero,
   * and broadcast it to the zero processes on other nodes
   */
  double shared_data = 0;
  if (procid==0) shared_data = 3.14;
  if (onnode_procid==0)
    MPI_Bcast(&shared_data,1,MPI_DOUBLE,0,crosscomm);
  printf("[%d] Head nodes should have shared data: %e\n",procid,shared_data);

  /*
   * Create window on the node communicator;
   * it only has nonzero size on the first process
   */
  MPI_Aint window_size; double *window_data; MPI_Win node_window;
  if (onnode_procid==0)
```

```
    window_size = sizeof(double);
  else window_size = 0;
  MPI_Win_allocate_shared
    ( window_size,sizeof(double),MPI_INFO_NULL,
      nodecomm,
      &window_data,&node_window);

  /*
   * Put data on process zero of the node window
   * We use a Put call rather than a straight copy:
   * the Fence calls enforce coherence
   */
  MPI_Win_fence(0,node_window);
  if (onnode_procid==0) {
    MPI_Aint disp = 0;
    MPI_Put( &shared_data,1,MPI_DOUBLE,0,disp,1,MPI_DOUBLE,node_window);
  }
  MPI_Win_fence(0,node_window);

  /*
   * Now get on each process the address of the window of process zero.
   */
  MPI_Aint window_size0; int window_unit; double *win0_addr;
  MPI_Win_shared_query( node_window,0,
&window_size0,&window_unit, &win0_addr );

  /*
   * Check that we can indeed get at the data in the shared memory
   */
  printf("[%d,%d] data at shared window: %e\n",nodeid,onnode_procid,*win0_addr);

  /*
   * cleanup
   */
  MPI_Win_free(&node_window);
  MPI_Finalize();
  return 0;
}
```

12.3.4 Listing of code examples/mpi/c/numa.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <mpi.h>

int main(int argc,char **argv) {

#include "globalinit.c"

  /*
   * Find the subcommunicator on the node,
```

```
 * and get the procid on the node.
 */
MPI_Comm nodecomm;
int onnode_procno, onnode_nprocs;
MPI_Comm_split_type
  (comm,MPI_COMM_TYPE_SHARED,procno,MPI_INFO_NULL,
   &nodecomm);
MPI_Comm_size(nodecomm,&onnode_nprocs);
if (onnode_nprocs<2) {
  printf("This example needs at least two ranks per node\n");
  MPI_Abort(comm,0);
}
MPI_Comm_rank(nodecomm,&onnode_procno);

for (int strategy=0; strategy<2; strategy++) {
  /*
   * Create window on the node communicator;
   * one item on each process
   */
  MPI_Aint window_size; double *window_data; MPI_Win node_window;
  window_size = sizeof(double);
  MPI_Info window_info;
  MPI_Info_create(&window_info);
  if (strategy==0) {
    if (procno==0)
      printf("Strategy 0 : default behavior of shared window allocation\n");
    MPI_Info_set(window_info,"alloc_shared_noncontig","false");
  } else {
    if (procno==0)
      printf("Strategy 1 : allow non-contiguous shared window allocation\n");
    MPI_Info_set(window_info,"alloc_shared_noncontig","true");
  }
  MPI_Win_allocate_shared( window_size,sizeof(double),window_info,
                           nodecomm,
                           &window_data,&node_window);
  MPI_Info_free(&window_info);

  /*
   * Now process zero checks on window placement
   */
  if (onnode_procno==0) {
    MPI_Aint window_size0; int window0_unit; double *win0_addr;
    MPI_Win_shared_query( node_window,0,
                          &window_size0,&window0_unit, &win0_addr );
    size_t dist1,distp;
    for (int p=1; p<onnode_nprocs; p++) {
      MPI_Aint window_sizep; int windowp_unit; double *winp_addr;
      MPI_Win_shared_query( node_window,p,
                            &window_sizep,&windowp_unit, &winp_addr );
      distp = (size_t)winp_addr-(size_t)win0_addr;
      if (procno==0)
        printf("Distance %d to zero: %ld\n",p,(long)distp);
      if (p==1)
```

```
          dist1 = distp;
        else {
          if (distp%dist1!=0)
            printf("!!!! not a multiple of distance 0--1 !!!!\n");
        }
      }
    }
    MPI_Win_free(&node_window);
  }

  /*
   * cleanup
   */
  MPI_Finalize();
  return 0;
}
```

12.3.5 Listing of code code/mpi/shared.c

12.3.6 Listing of code examples/mpi/c/sharedshared.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

#ifndef CORES_PER_NODE
#define CORES_PER_NODE 16
#endif

int main(int argc,char **argv) {

#include "globalinit.c"

  if (nprocs<3) {
    printf("This program needs at least three processes\n");
    return -1;
  }

  if (procno==0)
    printf("There are %d ranks total\n",nprocs);

  int new_procno,new_nprocs;
  MPI_Comm sharedcomm;

  MPI_Info info;
  MPI_Comm_split_type(MPI_COMM_WORLD,MPI_COMM_TYPE_SHARED,procno,info,&sharedcomm);
  MPI_Comm_size(sharedcomm,&new_nprocs);
  MPI_Comm_rank(sharedcomm,&new_procno);
```

```
  ASSERT(new_procno<CORES_PER_NODE);

  if (new_nprocs!=nprocs) {
    printf("This example can only run on shared memory\n");
    MPI_Abort(comm,0);
  }

  MPI_Win shared_window; int *shared_baseptr;
  MPI_Win_allocate_shared(3,sizeof(int),info,sharedcomm,&shared_baseptr,&shared_window);

  {
    MPI_Aint check_size; int check_unit; int *check_baseptr;
    MPI_Win_shared_query
      (shared_window,new_procno,
       &check_size,&check_unit,&check_baseptr);
    printf("[%d;%d] size=%ld\n",procno,new_procno,check_size);
  }

  int *left_ptr,*right_ptr;
  int left_proc = new_procno>0 ? new_procno-1 : MPI_PROC_NULL,
    right_proc = new_procno<new_nprocs-1 ? new_procno+1 : MPI_PROC_NULL;
  MPI_Win_shared_query(shared_window,left_proc,NULL,NULL,&left_ptr);
  MPI_Win_shared_query(shared_window,right_proc,NULL,NULL,&right_ptr);

  if (procno==0)
    printf("Finished\n");

  MPI_Finalize();
  return 0;
}
```

Chapter 13

MPI topic: Tools interface

The following material is for the (unreleased) MPI-4 standard only

The tools interface requires a different initialization routine `MPI_T_init_thread`

```
int MPI_T_init_thread( int required,int *provided );
```

Likewise, there is `MPI_T_finalize`

```
int MPI_T_finalize();
```

These matching calls can be made multiple times, after MPI has already been initialized with `MPI_Init` or `MPI_Init_thread`.

Verbosity level is an integer parameter.

`MPI_T_VERBOSITY_{USER,TUNER,MPIDEV}_{BASIC,DETAIL,ALL}`

13.1 Control variables

We query how many control variables are available with `MPI_Cvar_get_num`:

```
int MPI_Cvar_get_num( int *number_of_cvars );
```

A description of the control variable can be obtained from `MPI_T_cvar_get_info`

```
int MPI_T_cvar_get_info( int cvar_num,
  char *name, int *name_length,
  int *verbosity, MPI_Datatype *type,MPI_T_enum *enumtype,
  char *description,int *description_length,
  int *bind,int *scope);
```

An invalid index leads to a function result of `MPI_T_ERR_INVALID_INDEX`. Any output parameter can be specified as `NULL` and MPI will not set this. The *bind* variable is an object type or `MPI_T_BIND_NO_OBJECT`. The *enumtype* variable is `MPI_T_ENUM_NULL` if the variable is not an enum type.

Conversely, given a variable name, its index can be retrieved with `MPI_T_cvar_get_index`:

```
int MPI_T_cvar_get_index(const char *name, int *cvar_index)
```

If the name can not be matched, the index is MPI_T_ERR_INVALID_NAME.

Accessing a control variable is done through a *control variable handle*.

```
int MPI_T_cvar_handle_alloc
   (int cvar_index, void *obj_handle,
    MPI_T_cvar_handle *handle, int *count)
```

The handle is freed with MPI_T_cvar_handle_free:

```
int MPI_T_cvar_handle_free(MPI_T_cvar_handle *handle)
```

Control variable access is done through MPI_T_cvar_read and MPI_T_cvar_write:

```
int MPI_T_cvar_read(MPI_T_cvar_handle handle, void* buf);
int MPI_T_cvar_write(MPI_T_cvar_handle handle, const void* buf);
```

13.2 Performance variables

Performance variables come in classes: MPI_T_PVAR_CLASS_STATE MPI_T_PVAR_CLASS_LEVEL MPI_T_PVAR_CLASS_SIZE MPI_T_PVAR_CLASS_PERCENTAGE MPI_T_PVAR_CLASS_HIGHWATERMARK MPI_T_PVAR_CLASS_LOWWATERMARK MPI_T_PVAR_CLASS_COUNTER MPI_T_PVAR_CLASS_AGGREGATE MPI_T_PVAR_CLASS_TIMER MPI_T_PVAR_CLASS_GENERIC

Query the number of performance variables with MPI_T_pvar_get_num:

```
int MPI_T_pvar_get_num(int *num_pvar);
```

Get information about each variable, by index, with MPI_T_pvar_get_info:

```
int MPI_T_pvar_get_info
   (int pvar_index, char *name, int *name_len,
    int *verbosity, int *var_class, MPI_Datatype *datatype,
    MPI_T_enum *enumtype, char *desc, int *desc_len, int *bind,
    int *readonly, int *continuous, int *atomic)
```

See general remarks about these in section 13.1.

- The *readonly* variable indicates that the variable can not be written.
- The *continuous* variable requires use of MPI_T_pvar_start and MPI_T_pvar_stop.

Given a name, the index can be retried with MPI_T_pvar_get_index:

```
int MPI_T_pvar_get_index(const char *name, int var_class, int *pvar_index)
```

Again, see section 13.1.

13.2.1 Performance experiment sessions

To prevent measurements from getting mixed up, they need to be done in *performance experiment sessions*, to be called 'sessions' in this chapter. However see section ??.

Create a session with `MPI_T_pvar_session_create`

```
int MPI_T_pvar_session_create(MPI_T_pvar_session *session)
```

and release it with `MPI_T_pvar_session_free`:

```
int MPI_T_pvar_session_free(MPI_T_pvar_session *session)
```

which sets the session variable to `MPI_T_PVAR_SESSION_NULL`.

We access a variable through a handle, associated with a certain session. The handle is created with `MPI_T_pvar_handle_alloc`:

```
int MPI_T_pvar_handle_alloc
   (MPI_T_pvar_session session, int pvar_index,
    void *obj_handle, MPI_T_pvar_handle *handle, int *count)
```

(If a routine takes both a session and handle argument, and the two are not associated, an error of `MPI_T_ERR_INVALID_HANDLE` is returned.)

Free the handle with `MPI_T_pvar_handle_free`:

```
int MPI_T_pvar_handle_free
   (MPI_T_pvar_session session,
    MPI_T_pvar_handle *handle)
```

which sets the variable to `MPI_T_PVAR_HANDLE_NULL`.

Continuous variables (see `MPI_T_pvar_get_info` above, which outputs this) can be started and stopped with `MPI_T_pvar_start` and `MPI_T_pvar_stop`:

```
int MPI_T_pvar_start(MPI_T_pvar_session session, MPI_T_pvar_handle handle);
int MPI_T_pvar_stop(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
```

Passing `MPI_T_PVAR_ALL_HANDLES` to the stop call attempts to stop all variables within the session. Failure to stop a variable returns `MPI_T_ERR_PVAR_NO_STARTSTOP`.

Variables can be read and written with `MPI_T_pvar_read` and `MPI_T_pvar_write`:

```
int MPI_T_pvar_read
   (MPI_T_pvar_session session, MPI_T_pvar_handle handle,
    void* buf)
int MPI_T_pvar_write
   (MPI_T_pvar_session session, MPI_T_pvar_handle handle,
    const void* buf)
```

If the variable can not be written (see the *readonly* parameter of `MPI_T_pvar_get_info`), `MPI_T_ERR_PVAR_NO_WRITE` is returned.

A special case of writing the variable is to reset it with

```
int MPI_T_pvar_reset(MPI_T_pvar_session session, MPI_T_pvar_handle handle)
```

The handle value of MPI_T_PVAR_ALL_HANDLES is allowed.

A call to MPI_T_pvar_readreset is an atomic combination of the read and reset calls:

```
int MPI_T_pvar_readreset
   (MPI_T_pvar_session session,MPI_T_pvar_handle handle,
    void* buf)
```

13.3 Categories of variables

Variables, both the control and performance kind, can be grouped into categories by the MPI implementation.

The number of categories is queried with MPI_T_category_get_num:

```
int MPI_T_category_get_num(int *num_cat)
```

and for each category the information is retrieved with MPI_T_category_get_info:

```
int MPI_T_category_get_info
   (int cat_index,
    char *name, int *name_len, char *desc, int *desc_len,
    int *num_cvars, int *num_pvars, int *num_categories)
```

For a given category name the index can be found with MPI_T_category_get_index:

```
int MPI_T_category_get_index(const char *name, int *cat_index)
```

The contents of a category are retrieved with MPI_T_category_get_cvars, MPI_T_category_get_pvars, MPI_T_category_get_cate
:

```
int MPI_T_category_get_cvars(int cat_index, int len, int indices[])
int MPI_T_category_get_pvars(int cat_index, int len, int indices[])
int MPI_T_category_get_categories(int cat_index, int len, int indices[])
```

These indices can subsequently be used in the calls MPI_T_cvar_get_info, MPI_T_pvar_get_info, MPI_T_category_get_info.

If categories change dynamically, this can be detected with MPI_T_category_changed

```
int MPI_T_category_changed(int *stamp)
```

End of MPI-4 material

13.4 Sources used in this chapter

13.4.1 Listing of code header

Chapter 14

MPI leftover topics

14.1 Contextual information, attributes, etc.

14.1.1 Info objects

Certain MPI routines can accept `MPI_Info` objects. These contain key-value pairs that can offer system or implementation dependent information.

Create an info object with `MPI_Info_create` (figure 14.1) and delete it with `MPI_Info_free` (figure 14.2).

Keys are then set with `MPI_Info_set` (figure 14.3), and they can be queried with `MPI_Info_get` (figure 14.4). Note that the output of the 'get' routine is not allocated: it is a buffer that is passed. The maximum length of a key is given by the parameter `MPI_MAX_INFO_KEY`. You can delete a key from an info object with `MPI_Info_delete` (figure 14.5).

There is a straightforward duplication of info objects: `MPI_Info_dup` (figure 14.6).

You can also query the number of keys in an info object with `MPI_Info_get_nkeys` (figure 14.7), after which the keys can be queried in succession with `MPI_Info_get_nthkey`

Info objects that are marked as 'In' or 'Inout' arguments are parsed before that routine returns. This means that in nonblocking routines they can be freed immediately, unlike, for instance, send buffers.

The following material is for the (unreleased) MPI-4 standard only

The routines `MPI_Info_get` and `MPI_Info_get_valuelen` are not robust with respect to the C language *null terminator*. Therefore, they are deprecated, and should be replaced with `MPI_Info_get_string`, which always returns a null-terminated string.

```
int MPI_Info_get_string
   (MPI_Info info, const char *key,
    int *buflen, char *value, int *flag)
```

End of MPI-4 material

14.1.1.1 Environment information

The object `MPI_INFO_ENV` is predefined, containing:

- `command` Name of program executed.

Figure 14.1 `MPI_Info_create`

Name	Param name	C type	F type	inout
`mpi_info_create (`				
	`info` *info object created*	`MPI_Info*`	`TYPE(MPI_Info)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Info.Create(`		
	`cls`	
`) -> Info`		

Figure 14.2 `MPI_Info_free`

Name	Param name	C type	F type	inout
`mpi_info_free (`				
	`info` *info object*	`MPI_Info*`	`TYPE(MPI_Info)`	`inout`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Info.Free(`		
	`self`	
`) -> None`		

Figure 14.3 `MPI_Info_set`

Name	Param name	C type	F type	inout
`mpi_info_set (`				
	`info` *info object*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`key` *key*	`const char*`	`CHARACTER`	`in`
	`value` *value*	`const char*`	`CHARACTER`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Info.Set(`		
	`self`	
	`key`	`str`
	`value`	`str`
`) -> None`		

Figure 14.4 `MPI_Info_get`

Name	Param name	C type	F type	inout
`mpi_info_get (`				
	`info` *info object*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`key` *key*	`const char*`	`CHARACTER`	`in`
	`valuelen` *length of value arg*	`int`	`INTEGER`	`in`
	`value`	`char*`	`CHARACTER`	`out`
	`length: valuelen` *value*			
	`flag` *true if key defined, false if not*	`int*`	`LOGICAL`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Info.get(`		
	`self`	
	`key`	`str`
	`default`	`Optional[str] = None`
`) -> Optional[str]`		

Figure 14.5 `MPI_Info_delete`

Name	Param name	C type	F type	inout
`mpi_info_delete (`				
	`info` *info object*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`key` *key*	`const char*`	`CHARACTER`	`in`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Info.Delete(`		
	`self`	
	`key`	`str`
`) -> None`		

Figure 14.6 `MPI_Info_dup`

Name	Param name	C type	F type	inout
`mpi_info_dup (`				
	`info` *info object*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`newinfo` *info object*	`MPI_Info*`	`TYPE(MPI_Info)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Info.Dup(`		
	`self`	
`) -> Info`		

Figure 14.7 `MPI_Info_get_nkeys`

Name	Param name	C type	F type	inout
`mpi_info_get_nkeys (`				
	`info` *info object*	`MPI_Info`	`TYPE(MPI_Info)`	`in`
	`nkeys` *number of defined keys*	`int*`	`INTEGER`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Info.Get_nkeys(`		
	`self`	
`) -> int`		

- `argv` Space separated arguments to command.
- `maxprocs` Maximum number of MPI processes to start.
- `soft` Allowed values for number of processors.
- `host` Hostname.
- `arch` Architecture name.
- `wdir` Working directory of the MPI process.
- `file` Value is the name of a file in which additional information is specified.
- `thread_level` Requested level of thread support, if requested before the program started execution.

Note that these are the requested values; the running program can for instance have lower thread support.

14.1.1.2 Communicator and window information

MPI has a built-in possibility of attaching information to *communicators* and *windows* using the calls `MPI_Comm_get_info` `MPI_Comm_set_info`, `MPI_Win_get_info`, `MPI_Win_set_info`.

Copying a communicator with `MPI_Comm_dup` does not cause the info to be copied; to propagate information to the copy there is `MPI_Comm_dup_with_info`.

14.1.1.3 File information

An `MPI_Info` object can be passed to the following file routines:

- `MPI_File_open`
- `MPI_File_set_view`
- `MPI_File_set_info`; collective.

The following keys are defined in the MPI-2 standard:

- *access_style*: A comma separated list of one or more of: `read_once`, `write_once`, `read_mostly`, `write_mostly`, `sequential`, `reverse_sequential`, `random`
- *collective_buffering*: true or false; enables or disables buffering on collective I/O operations
- *cb_block_size*: integer block size for collective buffering, in bytes
- *cb_buffer_size*: integer buffer size for collective buffering, in bytes
- *cb_nodes*: integer number of MPI processes used in collective buffering
- *chunked*: a comma separated list of integers describing the dimensions of a multidimensional array to be accessed using subarrays, starting with the most significant dimension (1st in C, last in Fortran)
- *chunked_item*: a comma separated list specifying the size of each array entry, in bytes
- *chunked_size*: a comma separated list specifying the size of the subarrays used in chunking
- *file_perm*: UNIX file permissions at time of creation, in octal
- *io_node_list*: a comma separated list of I/O nodes to use

The following material is for the (unreleased) MPI-4 standard only

- *mpi_minimum_memory_alignment*: aligment of allocated memory.

End of MPI-4 material

- *nb_proc*: integer number of processes expected to access a file simultaneously
- *num_io_nodes*: integer number of I/O nodes to use
- *striping_factor*: integer number of I/O nodes/devices a file should be striped across
- *striping_unit*: integer stripe size, in bytes

Additionally, file system-specific keys can exist.

14.1.2 Attributes

Some runtime (or installation dependendent) values are available as attributes through `MPI_Comm_get_attr` (figure 14.8). (The MPI-2 routine `MPI_Attr_get` is deprecated). The flag parameter has two functions:

- it returns whether the attributed was found;
- if on entry it was set to false, the value parameter is ignored and the routines only tests whether the key is present.

The return value parameter is subtle: while it is declared `void*`, it is actually the address of a `void*` pointer.

Figure 14.8 `MPI_Comm_get_attr`

Name	Param name	C type	F type	inout
`mpi_comm_get_attr (`				
	`comm`	`MPI_Comm`	`TYPE(MPI_Comm)`	`in`
	communicator to which the attribute is attached			
	`comm_keyval`	`int`	`INTEGER`	`in`
	key value			
	`attribute_val`	`void*`	`INTEGER(KIND=MPI_ADDRESS_KIND)`	`out`
	attribute value, unless flag = false			
	`flag`	`int*`	`LOGICAL`	`out`
	false if no attribute is associated with the key			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`Comm.Get_attr(`		
	`self`	
	`keyval`	`int`
`) -> Optional[Union[int, Any]]`		

```
// tags.c
int tag_upperbound;
void *v; int flag=1;
ierr = MPI_Comm_get_attr(comm,MPI_TAG_UB,&v,&flag);
tag_upperbound = *(int*)v;
```

For the full source of this example, see section 14.12.2

```
## tags.py
tag_upperbound = comm.Get_attr(MPI.TAG_UB)
if procid==0:
    print("Determined tag upperbound: {}".format(tag_upperbound))
```

For the full source of this example, see section 14.12.3

Attributes are:

- `MPI_TAG_UB` Upper bound for *tag value*. (The lower bound is zero.) Note that `MPI_TAG_UB` is the key, not the actual upper bound! This value has to be at least 32767.
- `MPI_HOST` Host process rank, if such exists, `MPI_PROC_NULL`, otherwise.
- `MPI_IO` rank of a node that has regular I/O facilities (possibly myrank). Nodes in the same communicator may return different values for this parameter.
- `MPI_WTIME_IS_GLOBAL` Boolean variable that indicates whether clocks are synchronized.

Also:

- `MPI_UNIVERSE_SIZE`: the total number of processes that can be created. This can be more than the size of `MPI_COMM_WORLD` if the host list is larger than the number of initially started processes. See section 8.1.
  Python: *mpi4py.MPI.UNIVERSE_SIZE*.
- `MPI_APPNUM`: if MPI is used in MPMD mode, or if `MPI_Comm_spawn_multiple` is used, this attribute reports the how-manieth program we are in.

Figure 14.9 `MPI_Get_version`

Name	Param name	C type	F type	inout
`mpi_get_version (`				
	`version` *version number*	`int*`	`INTEGER`	`out`
	`subversion` *subversion number*	`int*`	`INTEGER`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Get_version(`		
`) -> Tuple[int, int]`		

14.1.3 Processor name

You can query the *hostname* of a processor with `MPI_Get_processor_name`. This name need not be unique between different processor ranks.

You have to pass in the character storage: the character array must be at least `MPI_MAX_PROCESSOR_NAME` characters long. The actual length of the name is returned in the `resultlen` parameter.

14.1.4 Version information

For runtime determination, The *MPI version* is available through two parameters `MPI_VERSION` and `MPI_SUBVERSION` or the function `MPI_Get_version` (figure 14.9).

14.2 Error handling

Errors in normal programs can be tricky to deal with; errors in parallel programs can be even harder. This is because in addition to everything that can go wrong with a single executable (floating point errors, memory violation) you now get errors that come from faulty interaction between multiple executables.

A few examples of what can go wrong:

- MPI errors: an MPI routine can abort for various reasons, such as receiving much more data than its buffer can accomodate. Such errors, as well as the more common type mentioned above, typically cause your whole execution to abort. That is, if one incarnation of your executable aborts, the MPI runtime will kill all others.
- Deadlocks and other hanging executions: there are various scenarios where your processes individually do not abort, but are all waiting for each other. This can happen if two processes are both waiting for a message from each other, and this can be helped by using nonblocking calls. In another scenario, through an error in program logic, one process will be waiting for more messages (including nonblocking ones) than are sent to it.

14.2.1 Error codes

There are a bunch of error codes. These are all positive `int` values, while `MPI_SUCCESS` is zero. The maximum value of any built-in error code is `MPI_ERR_LASTCODE`. User-defined error codes are all larger than this.

- `MPI_ERR_ARG`: an argument was invalid that is not covered by another error code.
- `MPI_ERR_BUFFER` The buffer pointer is invalid; this typically means that you have supplied a null pointer.
- `MPI_ERR_COMM`: invalid communicator. A common error is to use a null communicator in a call.
- `MPI_ERR_INTERN` An internal error in MPI has been detected.
- `MPI_ERR_IN_STATUS` A functioning returning an array of statuses has at least one status where the `MPI_ERROR` field is set to other than `MPI_SUCCESS`. See section 4.3.2.3.
- `MPI_ERR_INFO`: invalid info object.
- `MPI_ERR_NO_MEM` is returned by `MPI_Alloc_mem` if memory is exhausted.
- `MPI_ERR_OTHER`: an error occurred; use `MPI_Error_string` to retrieve further information about this error.
- `MPI_ERR_PORT`: invalid port; this applies to `MPI_Comm_connect` and such.

The following material is for the (unreleased) MPI-4 standard only

- `MPI_ERR_PROC_ABORTED` is returned if a process tries to communicate with a process that has aborted.

End of MPI-4 material

- `MPI_ERR_SERVICE`: invalid service in `MPI_Unpublish_name`.

14.2.2 Error handling

The MPI library has a general mechanism for dealing with errors that it detects: one can specify an error handler, specific to MPI objects.

- Most commonly, an error handler is associated with a communicator: `MPI_Comm_set_errhandler` (and likewise it can be retrieved with `MPI_Comm_get_errhandler`);
- other possibilities are `MPI_File_set_errhandler`, `MPI_File_call_errhandler`,

The following material is for the (unreleased) MPI-4 standard only

`MPI_Session_set_errhandler`, `MPI_Session_call_errhandler`,

End of MPI-4 material

`MPI_Win_set_errhandler`, `MPI_Win_call_errhandler`.

Remark 15 *The routine `MPI_Errhandler_set` is deprecated, replaced by its MPI-2 variant `MPI_Comm_set_errhandler`*
.

Some handlers of type `MPI_Errhandler` are predefined, but you can define your own with `MPI_Errhandler_create` , to be freed later with `MPI_Errhandler_free`.

14.2.2.1 Abort

The default behaviour, where the full run is aborted, is equivalent to your code having the following call to

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_ARE_FATAL);
```

The handler MPI_ERRORS_ARE_FATAL, even though it is associated with a communicator, causes the whole application to abort.

The following material is for the (unreleased) MPI-4 standard only

The handler MPI_ERRORS_ABORT (MPI-4) aborts on the processes in the communicator for which it is specified.

End of MPI-4 material

14.2.2.2 Return

Another simple possibility is to specify

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_RETURN);
```

which causes the error code to be returned to the user. This gives you the opportunity to write code that handles the error return value.

14.2.2.3 Error printing

If the MPI_Errhandler value MPI_ERRORS_RETURN is used, you can compare the return code to MPI_SUCCESS and print out debugging information:

```
int ierr;
ierr = MPI_Something();
if (ierr!=MPI_SUCCESS) {
    // print out information about what your programming is doing
    MPI_Abort();
}
```

For instance,

```
Fatal error in MPI_Waitall:
See the MPI_ERROR field in MPI_Status for the error code
```

You could then retrieve the MPI_ERROR field of the status, and print out an error string with MPI_Error_string or maximal size MPI_MAX_ERROR_STRING:

```
MPI_Comm_set_errhandler(MPI_COMM_WORLD,MPI_ERRORS_RETURN);
ierr = MPI_Waitall(2*ntids-2,requests,status);
if (ierr!=0) {
   char errtxt[MPI_MAX_ERROR_STRING];
   for (int i=0; i<2*ntids-2; i++) {
     int err = status[i].MPI_ERROR;
     int len=MPI_MAX_ERROR_STRING;
     MPI_Error_string(err,errtxt,&len);
     printf("Waitall error: %d %s\n",err,errtxt);
   }
   MPI_Abort(MPI_COMM_WORLD,0);
}
```

Figure 14.10 `MPI_Comm_create_errhandler`

Name	Param name	C type	F type	inout
`mpi_comm_create_errhandler (`				
	`comm_errhandler_fn` *user defined error handling procedure*		`PROCEDURE`	`in`
	`errhandler`	`MPI_Errhandler*`	`TYPE(MPI_Errhandler)`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

One cases where errors can be handled is that of *MPI file I/O*: if an output file has the wrong permissions, code can possibly progress without writing data, or writing to a temporary file.

MPI operators (`MPI_Op`) do not return an error code. In case of an error they call `MPI_Abort`; if `MPI_ERRORS_RETURN` is the error handler, error codes may be silently ignored.

You can create your own error handler with `MPI_Comm_create_errhandler` (figure 14.10), which is then installed with `MPI_Comm_set_errhandler`. You can retrieve the error handler with `MPI_Comm_get_errhandler`.

MPL note 57. MPL does not allow for access to the wrapped communicators. However, for `MPI_COMM_WORLD`, the routine `MPI_Comm_set_errhandler` can be called directly.

End of MPL note

14.2.3 Defining your own MPI errors

You can define your own errors that behave like MPI errors. As an example, let's write a send routine that refuses to send zero-sized data.

The first step to defining a new error is to define an error class with `MPI_Add_error_class`:

```
int nonzero_class;
MPI_Add_error_class(&nonzero_class);
```

This error number is larger than `MPI_ERR_LASTCODE`, the upper bound on built-in error codes. The attribute `MPI_LASTUSEDCODE` records the last issued value.

Your new error code is then defined in this class with `MPI_Add_error_code`, and an error string can be added with `MPI_Add_error_string`:

```
int nonzero_code;
MPI_Add_error_code(nonzero_class,&nonzero_code);
MPI_Add_error_string(nonzero_code,"Attempting to send zero buffer");
```

You can then call an error handler with this code. For instance to have a wrapped send routine that will not send zero-sized messages:

```
// errorclass.c
int MyPI_Send( void *buffer,int n,MPI_Datatype type, int target,int tag,MPI_Comm comm) {
  if (n==0)
    MPI_Comm_call_errhandler( comm,nonzero_code );
   MPI_Ssend(buffer,n,type,target,tag,comm);
```

```
  return MPI_SUCCESS;
};
```

Here we used the default error handler associated with the communicator, but one can set a different one with `MPI_Comm_create_errhandler`.

We test our example:

```
for (int msgsize=1; msgsize>=0; msgsize--) {
  double buffer;
  if (procno==0) {
    printf("Trying to send buffer of length %d\n",msgsize);
    MyPI_Send(&buffer,msgsize,MPI_DOUBLE, 1,0,comm);
    printf(".. success\n");
  } else if (procno==1) {
    MPI_Recv (&buffer,msgsize,MPI_DOUBLE, 0,0,comm,MPI_STATUS_IGNORE);
  }
}
```

which gives:

```
Trying to send buffer of length 1
.. success
Trying to send buffer of length 0
Abort(1073742081) on node 0 (rank 0 in comm 0):
Fatal error in MPI_Comm_call_errhandler: Attempting to send zero buffer
```

14.3 Fortran issues

MPI is typically written in C, what if you program *Fortran*?

See section 6.2.2.1 for MPI types corresponding to *Fortran90 types*.

14.3.1 Assumed-shape arrays

Use of other than contiguous data, for instance `A(1:N:2)`, was a problem in MPI calls, especially non-blocking ones. In that case it was best to copy the data to a contiguous array. This has been fixed in MPI-3.

- Fortran routines have the same prototype as C routines except for the addition of an integer error parameter.
- The call for `MPI_Init` in Fortran does not have the commandline arguments; they need to be handled separately.
- The routine `MPI_Sizeof` is only available in Fortran, it provides the functionality of the C/C++ operator `sizeof`.

Figure 14.11 `MPI_Wtime`

Name	Param name	C type	F type	inout
`mpi_wtime (` `)`				

Python:

Name	Param name	param type
`MPI.Wtime(` `) -> float`		

14.4 Fault tolerance

Processors are not completely reliable, so it may happen that one 'breaks': for software or hardware reasons it becomes unresponsive. For an MPI program this means that it becomes impossible to send data to it, and any collective operation involving it will hang. Can we deal with this case? Yes, but it involves some programming.

First of all, one of the possible MPI error return codes (section 14.2) is `MPI_ERR_COMM`, which can be returned if a processor in the communicator is unavailable. You may want to catch this error, and add a 'replacement processor' to the program. For this, the `MPI_Comm_spawn` can be used (see 8.1 for details). But this requires a change of program design: the communicator containing the new process(es) is not part of the old `MPI_COMM_WORLD`, so it is better to set up your code as a collection of inter-communicators to begin with.

14.5 Performance, tools, and profiling

In most of this book we talk about functionality of the MPI library. There are cases where a problem can be solved in more than one way, and then we wonder which one is the most efficient. In this section we will explicitly address performance. We start with two sections on the mere act of measuring performance.

14.5.1 Timing

MPI has a *wall clock* timer: `MPI_Wtime` (figure 14.11) which gives the number of seconds from a certain point in the past. (Note the absence of the error parameter in the fortran call.)

```
double t;
t = MPI_Wtime();
for (int n=0; n<NEXPERIMENTS; n++) {
  // do something;
}
t = MPI_Wtime()-t; t /= NEXPERIMENTS;
```

The timer has a resolution of `MPI_Wtick` (figure 14.12).

MPL note 58. The timing routines `wtime` and `wtick` and `wtime_is_global` are environment methods:

```
double  mpl::environment::wtime ();
double  mpl::environment::wtick ();
bool mpl::environment::wtime_is_global ();
```

Figure 14.12 `MPI_Wtick`

Name	Param name	C type	F type	inout
`mpi_wtick (` `)`				

Python:

Name	Param name	param type
`MPI.Wtick(` `) -> float`		

End of MPL note

Timing in parallel is a tricky issue. For instance, most clusters do not have a central clock, so you can not relate start and stop times on one process to those on another. You can test for a global clock as follows`MPI_WTIME_IS_GLOBAL`:

```
int *v,flag;
MPI_Attr_get( comm, MPI_WTIME_IS_GLOBAL, &v, &flag );
if (mytid==0) printf("Time synchronized? %d->%d\n",flag,*v);
```

Normally you don't worry about the starting point for this timer: you call it before and after an event and subtract the values.

```
t = MPI_Wtime();
// something happens here
t = MPI_Wtime()-t;
```

If you execute this on a single processor you get fairly reliable timings, except that you would need to subtract the overhead for the timer. This is the usual way to measure timer overhead:

```
t = MPI_Wtime();
// absolutely nothing here
t = MPI_Wtime()-t;
```

14.5.1.1 Global timing

However, if you try to time a parallel application you will most likely get different times for each process, so you would have to take the average or maximum. Another solution is to synchronize the processors by using a *barrier*`MPI_Barrier`:

```
MPI_Barrier(comm)
t = MPI_Wtime();
// something happens here
MPI_Barrier(comm)
t = MPI_Wtime()-t;
```

Exercise 14.1. This scheme also has some overhead associated with it. How would you measure that?

14.5.1.2 Local timing

Now suppose you want to measure the time for a single send. It is not possible to start a clock on the sender and do the second measurement on the receiver, because the two clocks need not be synchronized. Usually a *ping-pong* is done:

```
if ( proc_source ) {
  MPI_Send( /* to target */ );
  MPI_Recv( /* from target */ );
else if ( proc_target ) {
  MPI_Recv( /* from source */ );
  MPI_Send( /* to source */ );
}
```

No matter what sort of timing you are doing, it is good to know the accuracy of your timer. The routine `MPI_Wtick` gives the smallest possible timer increment. If you find that your timing result is too close to this 'tick', you need to find a better timer (for CPU measurements there are cycle-accurate timers), or you need to increase your running time, for instance by increasing the amount of data.

14.5.2 Simple profiling

MPI allows you to write your own profiling interface. To make this possible, every routine `MPI_Something` calls a routine `PMPI_Something` that does the actual work. You can now write your `MPI_...` routine which calls `PMPI_...`, and inserting your own profiling calls. See figure 14.1.

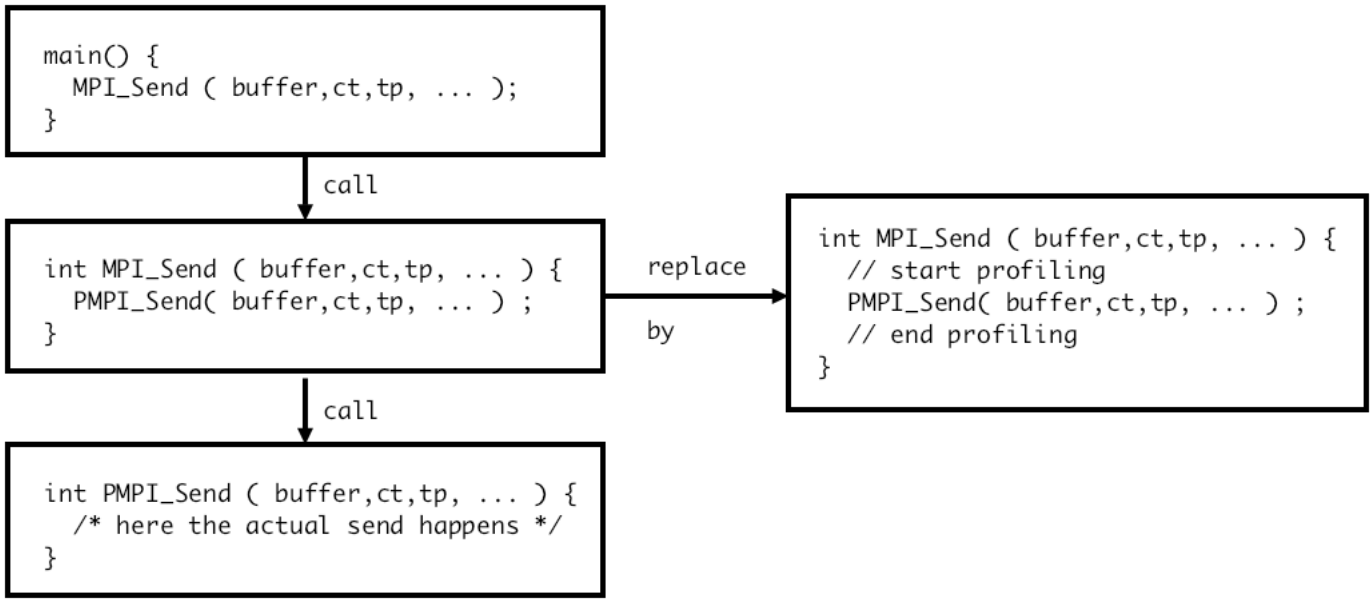


Figure 14.1: Calling hierarchy of MPI and PMPI routines

By default, the MPI routines are defined as *weak linker symbols* as a synonym of the PMPI ones. In the gcc case:

```
#pragma weak MPI_Send = PMPI_Send
```

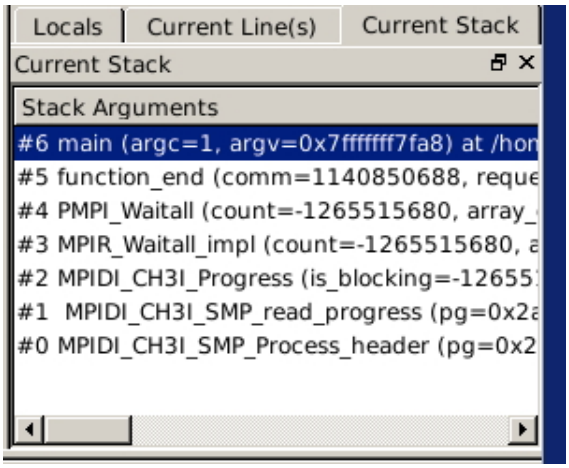


Figure 14.2: A stack trace, showing the PMPI calls.

As you can see in figure 14.2, normally only the PMPI routines show up in the stack trace.

14.5.3 Tools interface

Recent versions of MPI have a standardized way of reading out performance variables: the *tools interface* which improves on the old interface described in section 14.5.2.

The realization of the tools interface is installation-dependent, you first need to query how much of the tools interface is provided.

```
// mpit.c
MPI_Init_thread(&argc,&argv,MPI_THREAD_SINGLE,&tlevel);
MPI_T_init_thread(MPI_THREAD_SINGLE,&tlevel);
int npvar;
MPI_T_pvar_get_num(&npvar);
```

For the full source of this example, see section 14.12.4

```
int name_len=256,desc_len=256,
  verbosity,var_class,binding,isreadonly,iscontiguous,isatomic;
char var_name[256],description[256];
MPI_Datatype datatype; MPI_T_enum enumtype;
for (int pvar=0; pvar<npvar; pvar++) {
  MPI_T_pvar_get_info(pvar,var_name,&name_len,
                      &verbosity,&var_class,
                      &datatype,&enumtype,
                      description,&desc_len,
                      &binding,&isreadonly,&iscontiguous,&isatomic);
  if (procid==0)
    printf("pvar %d: %d/%s = %s\n",pvar,var_class,var_name,description);
}
```

For the full source of this example, see section 14.12.4

14.5.4 Programming for performance

We outline some issues pertaining to performance.

Eager limit Short blocking messages are handled by a simpler mechanism than longer. The limit on what is considered 'short' is known as the *eager limit* (section 4.1.2.2), and you could tune your code by increasing its value. However, note that a process may likely have a buffer accomodating eager sends for every single other process. This may eat into your available memory.

Blocking versus nonblocking The issue of *blocking versus nonblocking* communication is something of a red herring. While nonblocking communication allows *latency hiding*, we can not consider it an alternative to blocking sends, since replacing nonblocking by blocking calls will usually give *deadlock*.

Still, even if you use nonblocking communication for the mere avoidance of deadlock or serialization (section 4.1.2.3), bear in mind the possibility of overlap of communication and computation. This also brings us to our next point.

Looking at it the other way around, in a code with blocking sends you may get better performance from nonblocking, even if that is not structurally necessary.

Progress MPI is not magically active in the background, especially if the user code is doing scalar work that does not involve MPI. As sketched in section 5.4.1, there are various ways of ensuring that latency hiding actually happens.

Persistent sends If a communication between the same pair of processes, involving the same buffer, happens regularly, it is possible to set up a *persistent communication*. See section 5.1.

Buffering MPI uses internal buffers, and the copying from user data to these buffers may affect performance. For instance, derived types (section 6.3) can typically not be streamed straight through the network (this requires special hardware support [17]) so they are first copied. Somewhat surprisingly, we find that *buffered communication* (section 5.5) does not help. Perhaps MPI implementors have not optimized this mode since it is so rarely used.

This is issue is extensively investigated in [9].

Graph topology and neighborhood collectives Load balancing and communication minimization are important in irregular applications. There are dedicated programs for this (*ParMetis*, *Zoltan*), and libraries such as *PETSc* may offer convenient access to such capabilities.

In the declaration of a *graph topology* (section 11.2) MPI is allowed to reorder processes, which could be used to support such activities. It can also serve for better message sequencing when *neighbourhood collectives* are used.

Network issues In the discussion so far we have assumed that the network is a perfect conduit for data. However, there are issues of port design, in particular caused by *oversubscription* that adversely affect performance. While in an ideal world it may be possible to set up routine to avoid this, in the actual practice of a supercomputer cluster, *network contention* or *message collision* from different user jobs is hard to avoid.

Offloading and onloading There are different philosophies of *network card design*: *Mellanox*, being a network card manufacturer, believes in off-loading network activity to the Network Interface Card (NIC), while *Intel*, being a processor manufacturer, believes in 'on-loading' activity to the process. There are argument either way.

Either way, investigate the capabilities of your network.

14.5.5 MPIR

MPIR is the informally specified debugging interface for processes acquisition and message queue extraction.

14.6 Determinism

MPI processes are only synchronized to a certain extent, so you may wonder what guarantees there are that running a code twice will give the same result. You need to consider two cases: first of all, if the two runs are on different numbers of processors there are already numerical problems; see HPSC-3.5.5.

Let us then limit ourselves to two runs on the same set of processors. In that case, MPI is deterministic as long as you do not use wildcards such as `MPI_ANY_SOURCE`. Formally, MPI messages are 'nonovertaking': two messages between the same sender-receiver pair will arrive in sequence. Actually, they may not arrive in sequence: they are *matched* in sequence in the user program. If the second message is much smaller than the first, it may actually arrive earlier in the lower transport layer.

14.7 Subtleties with processor synchronization

Blocking communication involves a complicated dialog between the two processors involved. Processor one says 'I have this much data to send; do you have space for that?', to which processor two replies 'yes, I do; go ahead and send', upon which processor one does the actual send. This back-and-forth (technically known as a *handshake*) takes a certain amount of communication overhead. For this reason, network hardware will sometimes forgo the handshake for small messages, and just send them regardless, knowing that the other process has a small buffer for such occasions.

One strange side-effect of this strategy is that a code that should *deadlock* according to the MPI specification does not do so. In effect, you may be shielded from you own programming mistake! Of course, if you then run a larger problem, and the small message becomes larger than the threshold, the deadlock will

suddenly occur. So you find yourself in the situation that a bug only manifests itself on large problems, which are usually harder to debug. In this case, replacing every `MPI_Send` with a `MPI_Ssend` will force the handshake, even for small messages.

Conversely, you may sometimes wish to avoid the handshake on large messages. MPI as a solution for this: the `MPI_Rsend` ('ready send') routine sends its data immediately, but it needs the receiver to be ready for this. How can you guarantee that the receiving process is ready? You could for instance do the following (this uses nonblocking routines, which are explained below in section 4.2.1):

```
if ( receiving ) {
  MPI_Irecv()   // post nonblocking receive
  MPI_Barrier() // synchronize
else if ( sending ) {
  MPI_Barrier() // synchronize
  MPI_Rsend()   // send data fast
```

When the barrier is reached, the receive has been posted, so it is safe to do a ready send. However, global barriers are not a good idea. Instead you would just synchronize the two processes involved.

Exercise 14.2. Give pseudo-code for a scheme where you synchronize the two processes through the exchange of a blocking zero-size message.

14.8 Shell interaction

MPI programs are not run directly from the shell, but are started through an *ssh tunnel*. We briefly discuss ramifications of this.

14.8.1 Standard input

Letting MPI processes interact with the environment is not entirely straightforward. For instance, *shell input redirection* as in

```
mpirun -np 2 mpiprogram < someinput
```

may not work.

Instead, use a script `programscript` that has one parameter:

```
#!/bin/bash
mpirunprogram < $1
```

and run this in parallel:

```
mpirun -np 2 programscript someinput
```

14.8.2 Standard out and error

The *stdout* and *stderr* streams of an MPI process are returned through the ssh tunnel. Thus they can be caught as the *stdout/err of* mpirun.

```
// outerr.c
fprintf(stdout,"This goes to std out\n");
fprintf(stderr,"This goes to std err\n");
```

For the full source of this example, see section 14.12.5

14.8.3 Process status

The return code of `MPI_Abort` is returned as the *processes status of* mpirun. Running

```
// abort.c
if (procno==nprocs-1)
  MPI_Abort(comm,37);
```

For the full source of this example, see section 14.12.6

as

```
mpirun -np 4 ./abort ; \
echo "Return code from ${MPIRUN} is <<$$?>>"
```

gives

```
TACC:  Starting up job 3760534
TACC:  Starting parallel tasks...
application called MPI_Abort(MPI_COMM_WORLD, 37) - process 3
TACC:  MPI job exited with code: 37
TACC:  Shutdown complete. Exiting.
Return code from ibrun is <<37>>
```

14.8.4 Multiple program start

The sort of script of section 14.8.1 can also be used to implement *MPMD* runs: we let the script start one of a number of programs. For this, we use the fact that the MPI rank is known in the environment as `PMI_RANK`. Use a script `mpmdscript`:

```
#!/bin/bash
if [ ${PMI_RANK} -eq 0 ] ; then
  ./programmaster
else
  ./programworker
fi
```

which is then run in parallel:

```
mpirun -np 25 mpmdscript
```

14.9 The origin of one-sided communication in ShMem

The *Cray T3E* had a library called *shmem* which offered a type of shared memory. Rather than having a true global address space it worked by supporting variables that were guaranteed to be identical between

processors, and indeed, were guaranteed to occupy the same location in memory. Variables could be declared to be shared a 'symmetric' pragma or directive; their values could be retrieved or set by `shmem_get` and `shmem_put` calls.

14.10 Leftover topics

14.10.1 MPI constants

MPI has a number of built-in *constants*. These do not all behave the same.

- Some are *compile-time* constants. Examples are `MPI_VERSION` and `MPI_MAX_PROCESSOR_NAME`. Thus, they can be used in array size declarations, even before `MPI_Init`.
- Some *link-time* constants get their value by MPI initialization, such as `MPI_COMM_WORLD`. Such symbols, which include all predefined handles, can be used in initialization expressions.
- Some link-time symbols can not be used in initialization expressions, such as `MPI_BOTTOM` and `MPI_STATUS_IGNORE`.

For symbols, the binary realization is not defined. For instance, `MPI_COMM_WORLD` is of type `MPI_Comm`, but the implementation of that type is not specified.

See Annex A of the MPI-3.1 standard for full lists.

The following are the compile-time constants:

- `MPI_MAX_PROCESSOR_NAME`
- `MPI_MAX_LIBRARY_VERSION_STRING`
- `MPI_MAX_ERROR_STRING`
- `MPI_MAX_DATAREP_STRING`
- `MPI_MAX_INFO_KEY`
- `MPI_MAX_INFO_VAL`
- `MPI_MAX_OBJECT_NAME`
- `MPI_MAX_PORT_NAME`
- `MPI_VERSION`
- `MPI_SUBVERSION`
- `MPI_STATUS_SIZE` (Fortran only)
- `MPI_ADDRESS_KIND` (Fortran only)
- `MPI_COUNT_KIND` (Fortran only)
- `MPI_INTEGER_KIND` (Fortran only)
- `MPI_OFFSET_KIND` (Fortran only)
- `MPI_SUBARRAYS_SUPPORTED` (Fortran only)
- `MPI_ASYNC_PROTECTS_NONBLOCKING` (Fortran only)

The following are the link-time constants:

- `MPI_BOTTOM`
- `MPI_STATUS_IGNORE`
- `MPI_STATUSES_IGNORE`
- `MPI_ERRCODES_IGNORE`

- MPI_IN_PLACE
- MPI_ARGV_NULL
- MPI_ARGVS_NULL
- MPI_UNWEIGHTED
- MPI_WEIGHTS_EMPTY

Assorted constants:

- MPI_PROC_NULL
- MPI_ANY_SOURCE
- MPI_ANY_TAG
- MPI_UNDEFINED
- MPI_BSEND_OVERHEAD
- MPI_KEYVAL_INVALID
- MPI_LOCK_EXCLUSIVE
- MPI_LOCK_SHARED
- MPI_ROOT

(This section was inspired by http://blogs.cisco.com/performance/mpi-outside-of-c-and-fortran.)

14.10.2 Cancelling messages

In section 46 we showed a master-worker example where the master accepts in arbitrary order the messages from the workers. Here we will show a slightly more complicated example, where only the result of the first task to complete is needed. Thus, we issue an MPI_Recv with MPI_ANY_SOURCE as source. When a result comes, we broadcast its source to all processes. All the other workers then use this information to cancel their message with an MPI_Cancel operation.

```
// cancel.c
fprintf(stderr,"get set, go!\n");
if (procno==nprocs-1) {
  MPI_Status status;
  MPI_Recv(dummy,0,MPI_INT, MPI_ANY_SOURCE,0,comm,
                &status);
  first_tid = status.MPI_SOURCE;
  MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm);
  fprintf(stderr,"[%d] first msg came from %d\n",procno,first_tid);
} else {
  float randomfraction = (rand() / (double)RAND_MAX);
  int randomwait = (int) ( nprocs * randomfraction );
  MPI_Request request;
  fprintf(stderr,"[%d] waits for %e/%d=%d\n",
          procno,randomfraction,nprocs,randomwait);
  sleep(randomwait);
  MPI_Isend(dummy,0,MPI_INT, nprocs-1,0,comm,
                  &request);
  MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm
                );
  if (procno!=first_tid) {
    MPI_Cancel(&request);
    fprintf(stderr,"[%d] canceled\n",procno);
```

```
  }
}
```

For the full source of this example, see section 14.12.7

After the cancelling operation it is still necessary to call `MPI_Request_free`, `MPI_Wait`, or `MPI_Test` in order to free the request object.

The `MPI_Cancel` operation is local, so it can not be used for *nonblocking collectives* or one-sided transfers.

Remark 16 *As of MPI-3.2, cancelling a send is deprecated.*

14.10.3 Constants

MPI constants such as `MPI_COMM_WORLD` or `MPI_INT` are not necessarily statitally defined, such as by a `#define` statement: the best you can say is that they have a value after `MPI_Init` or `MPI_Init_thread`. That means you can not transfer a compiled MPI file between platforms, or even between compilers on one platform. However, a working MPI source on one MPI implementation will also work on another.

14.11 Literature

Online resources:

- MPI 1 Complete reference:
  http://www.netlib.org/utk/papers/mpi-book/mpi-book.html
- Official MPI documents:
  http://www.mpi-forum.org/docs/
- List of all MPI routines:
  http://www.mcs.anl.gov/research/projects/mpi/www/www3/

Tutorial books on MPI:

- Using MPI [11] by some of the original authors.

14.12 Sources used in this chapter

14.12.1 Listing of code header

14.12.2 Listing of code examples/mpi/c/tags.c

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc,char **argv) {

#ifndef FREQUENCY
#define FREQUENCY -1
#endif

  /*
   * Standard initialization
   */
  MPI_Comm comm = MPI_COMM_WORLD;
  int nprocs, procid;
  MPI_Init(&argc,&argv);
  MPI_Comm_set_errhandler(comm,MPI_ERRORS_RETURN);
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_rank(comm,&procid);
  int ierr;

  if (nprocs<2) {
    printf("This test needs at least 2 processes, not %d\n",nprocs);
    MPI_Abort(comm,0);
  }
  int sender = 0, receiver = nprocs-1;
  if (procid==0) {
    printf("Running on comm world of %d procs; communicating between %d--%d\n",
   nprocs,sender,receiver);
  }

  int tag_upperbound;
  void *v; int flag=1;
  ierr = MPI_Comm_get_attr(comm,MPI_TAG_UB,&v,&flag);
  tag_upperbound = *(int*)v;
  if (ierr!=MPI_SUCCESS) {
    printf("Error getting attribute: return code=%d\n",ierr);
    if (ierr==MPI_ERR_COMM)
      printf("invalid communicator\n");
    if (ierr==MPI_ERR_KEYVAL)
      printf("errorneous keyval\n");
    MPI_Abort(comm,0);
  }
  if (!flag) {
    printf("Could not get keyval\n");
    MPI_Abort(comm,0);
```

```
  } else {
    if (procid==sender)
      printf("Determined tag upperbound: %d\n",tag_upperbound);
  }


  MPI_Finalize();
  return 0;
}
```

14.12.3 Listing of code examples/mpi/p/tags.py

```
import numpy as np
import random # random.randint(1,N), random.random()
from mpi4py import MPI

comm = MPI.COMM_WORLD
procid = comm.Get_rank()
nprocs = comm.Get_size()
if nprocs<4:
    prin( "Need 4 procs at least")
    sys.exit(1)

tag_upperbound = comm.Get_attr(MPI.TAG_UB)
if procid==0:
    print("Determined tag upperbound: {}".format(tag_upperbound))
```

14.12.4 Listing of code code/mpi/c/mpit.c

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc,char **argv) {
  int procid,nprocs;

  int tlevel;
  MPI_Init_thread(&argc,&argv,MPI_THREAD_SINGLE,&tlevel);
  MPI_T_init_thread(MPI_THREAD_SINGLE,&tlevel);
  int npvar;
  MPI_T_pvar_get_num(&npvar);


  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&procid);

  if (procid==0)
    printf("#pvars: %d\n",npvar);
  int name_len=256,desc_len=256,
    verbosity,var_class,binding,isreadonly,iscontiguous,isatomic;
  char var_name[256],description[256];
  MPI_Datatype datatype; MPI_T_enum enumtype;
```

```
  for (int pvar=0; pvar<npvar; pvar++) {
    MPI_T_pvar_get_info(pvar,var_name,&name_len,
                        &verbosity,&var_class,
                        &datatype,&enumtype,
                        description,&desc_len,
                        &binding,&isreadonly,&iscontiguous,&isatomic);
    if (procid==0)
      printf("pvar %d: %d/%s = %s\n",pvar,var_class,var_name,description);
  }

  MPI_T_finalize();
  MPI_Finalize();

  return 0;
}
```

14.12.5 Listing of code examples/mpi/c/outerr.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

  fprintf(stdout,"This goes to std out\n");
  fprintf(stderr,"This goes to std err\n");

  return 0;
}
```

14.12.6 Listing of code examples/mpi/c/abort.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

#include "globalinit.c"

  if (procno==nprocs-1)
    MPI_Abort(comm,37);

  MPI_Finalize();
  return 0;
}
```

14.12.7 Listing of code examples/mpi/c/cancel.c

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "mpi.h"

int main(int argc,char **argv) {
  int first_tid,dummy[11];

#include "globalinit.c"

  // Initialize the random number generator
  srand((int)(procno*(double)RAND_MAX/nprocs));

  fprintf(stderr,"get set, go!\n");
  if (procno==nprocs-1) {
    MPI_Status status;
    MPI_Recv(dummy,0,MPI_INT, MPI_ANY_SOURCE,0,comm,
                  &status);
    first_tid = status.MPI_SOURCE;
    MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm);
    fprintf(stderr,"[%d] first msg came from %d\n",procno,first_tid);
  } else {
    float randomfraction = (rand() / (double)RAND_MAX);
    int randomwait = (int) ( nprocs * randomfraction );
    MPI_Request request;
    fprintf(stderr,"[%d] waits for %e/%d=%d\n",
   procno,randomfraction,nprocs,randomwait);
    sleep(randomwait);
    MPI_Isend(dummy,0,MPI_INT, nprocs-1,0,comm,
                   &request);
    MPI_Bcast(&first_tid,1,MPI_INT, nprocs-1,comm
                  );
    if (procno!=first_tid) {
      MPI_Cancel(&request);
      fprintf(stderr,"[%d] canceled\n",procno);
    }
  }

  MPI_Finalize();
  return 0;
}
```

PART II

OPENMP

Chapter 15

Getting started with OpenMP

This chapter explains the basic concepts of OpenMP, and helps you get started on running your first OpenMP program.

15.1 The OpenMP model

We start by establishing a mental picture of the hardware and software that OpenMP targets.

15.1.1 Target hardware

Modern computers have a multi-layered design. Maybe you have access to a cluster, and maybe you have learned how to use MPI to communicate between cluster nodes. OpenMP, the topic of this chapter, is concerned with a single *cluster node* or *motherboard*, and getting the most out of the available parallelism available there.

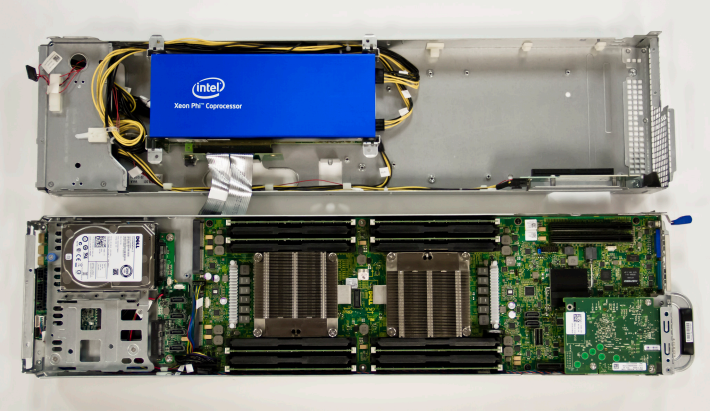


Figure 15.1: A node with two sockets and a co-processor

Figure 15.1 pictures a typical design of a node: within one enclosure you find two *sockets*: single processor chips. Your personal laptop of computer will probably have one socket, most supercomputers have nodes

with two or four sockets (the picture is of a *Stampede node* with two sockets)[1], although the recent *Intel Knight's Landing* is again a single-socket design.

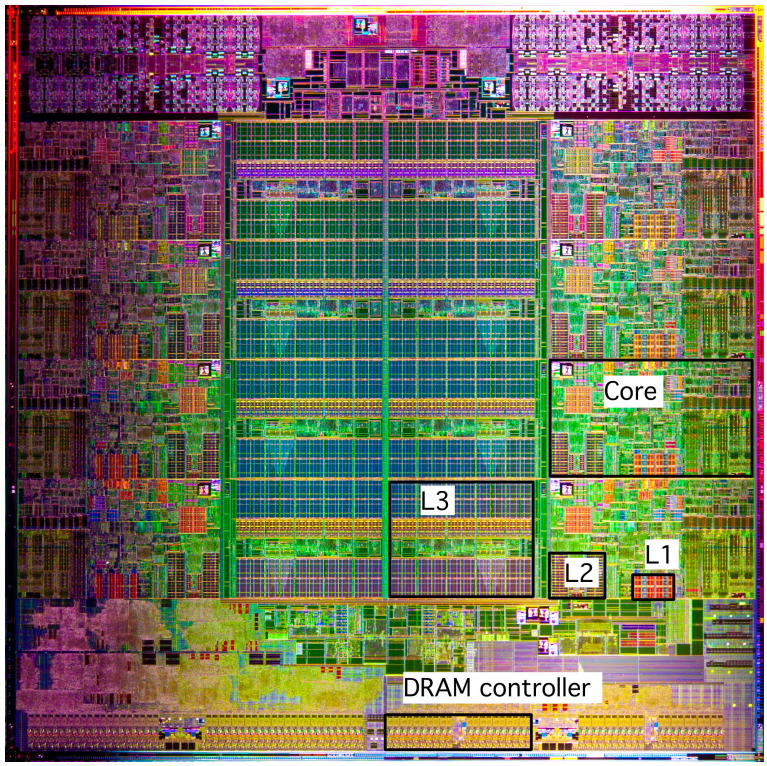


Figure 15.2: Structure of an Intel Sandybridge eight-core socket

To see where OpenMP operates we need to dig into the sockets. Figure 15.2 shows a picture of an *Intel Sandybridge* socket. You recognize a structure with eight *corescore*: independent processing units, that all have access to the same memory. (In figure 15.1 you saw four memory banks attached to each of the two sockets; all of the sixteen cores have access to all that memory.)

To summarize the structure of the architecture that OpenMP targets:

- A node has up to four sockets;
- each socket has up to 60 cores;
- each core is an independent processing unit, with access to all the memory on the node.

15.1.2 Target software

OpenMP is based on on two concepts: the use of *threads* and the *fork/join model* of parallelism. For now you can think of a thread as a sort of process: the computer executes a sequence of instructions. The fork/join model says that a thread can split itself ('fork') into a number of threads that are identical copies. At some point these copies go away and the original thread is left ('join'), but while the *team of threads* created by

1. In that picture you also see a co-processor: OpenMP is increasingly targeting those too.

the fork exists, you have parallelism available to you. The part of the execution between fork and join is known as a *parallel region*.

Figure 15.3 gives a simple picture of this: a thread forks into a team of threads, and these threads themselves can fork again.

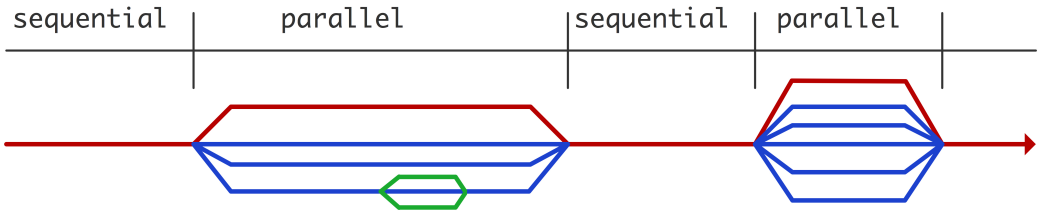


Figure 15.3: Thread creation and deletion during parallel execution

The threads that are forked are all copies of the *master thread*: they have access to all that was computed so far; this is their *shared data*. Of course, if the threads were completely identical the parallelism would be pointless, so they also have private data, and they can identify themselves: they know their thread number. This allows you to do meaningful parallel computations with threads.

This brings us to the third important concept: that of *work sharing* constructs. In a team of threads, initially there will be replicated execution; a work sharing construct divides available parallelism over the threads.

> So there you have it: OpenMP uses teams of threads, and inside a parallel region the work is distributed over the threads with a work sharing construct. Threads can access shared data, and they have some private data.

An important difference between OpenMP and MPI is that parallelism in OpenMP is dynamically activated by a thread spawning a team of threads. Furthermore, the number of threads used can differ between parallel regions, and threads can create threads recursively. This is known as as *dynamic mode*. By contrast, in an MPI program the number of running processes is (mostly) constant throughout the run, and determined by factors external to the program.

15.1.3 About threads and cores

OpenMP programming is typically done to take advantage of *multicore* processors. Thus, to get a good speedup you would typically let your number of threads be equal to the number of cores. However, there is nothing to prevent you from creating more threads: the operating system will use *time slicing* to let them all be executed. You just don't get a speedup beyond the number of actually available cores.

On some modern processors there are *hardware threads*, meaning that a core can actually let more than thread be executed, with some speedup over the single thread. To use such a processor efficiently you would let the number of OpenMP threads be 2× or 4× the number of cores, depending on the hardware.

15.1.4 About thread data

In most programming languages, visibility of data is governed by rules on the *scope of variables*: a variable is declared in a block, and it is then visible to any statement in that block and blocks with a *lexical scope*

contained in it, but not in surrounding blocks:

```
main () {
  // no variable `x' define here
  {
    int x = 5;
    if (somecondition) { x = 6; }
    printf("x=%e\n",x); // prints 5 or 6
  }
  printf("x=%e\n",x); // syntax error: `x' undefined
}
```

In C, you can redeclare a variable inside a nested scope:

```
{
  int x;
  if (something) {
    double x; // same name, different entity
  }
  x = ... // this refers to the integer again
}
```

Doing so makes the outer variable inaccessible.

Fortran has simpler rules, since it does not have blocks inside blocks.

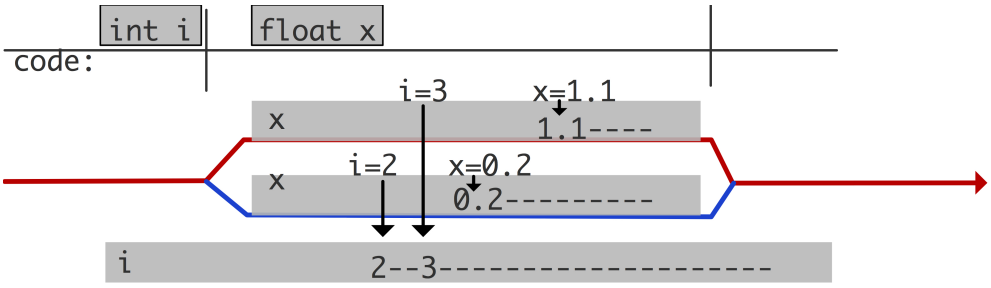


Figure 15.4: Locality of variables in threads

In OpenMP the situation is a bit more tricky because of the threads. When a team of threads is created they can all see the data of the master thread. However, they can also create data of their own. This is illustrated in figure 15.5. We will go into the details later.

15.2 Compiling and running an OpenMP program

15.2.1 Compiling

Your file or Fortran module needs to contain

```
#include "omp.h"
```

in C, and

```
use omp_lib
```

or

```
#include "omp_lib.h"
```

for Fortran.

OpenMP is handled by extensions to your regular compiler, typically by adding an option to your commandline:

```
# gcc
gcc -o foo foo.c -fopenmp
# Intel compiler
icc -o foo foo.c -qopenmp
```

If you have separate compile and link stages, you need that option in both.

When you use the openmp compiler option, a *cpp* variable `_OPENMP` will be defined. Thus, you can have conditional compilation by writing

```
#ifdef _OPENMP
   ...
#else
   ...
#endif
```

15.2.2 Running an OpenMP program

You run an OpenMP program by invoking it the regular way (for instance `./a.out`), but its behaviour is influenced by some *OpenMP environment variables*. The most important one is `OMP_NUM_THREADS`:

```
export OMP_NUM_THREADS=8
```

which sets the number of threads that a program will use. See section 26.1 for a list of all environment variables.

15.3 Your first OpenMP program

In this section you will see just enough of OpenMP to write a first program and to explore its behaviour. For this we need to introduce a couple of OpenMP language constructs. They will all be discussed in much greater detail in later chapters.

15.3.1 Directives

OpenMP is not magic, so you have to tell it when something can be done in parallel. This is mostly done through *directives*; additional specifications can be done through library calls.

In C/C++ the *pragma* mechanism is used: annotations for the benefit of the compiler that are otherwise not part of the language. This looks like:

```
#pragma omp somedirective clause(value,othervalue)
  parallel statement;

#pragma omp somedirective clause(value,othervalue)
 {
  parallel statement 1;
  parallel statement 2;
 }
```

with

- the `#pragma omp` *sentinel* to indicate that an OpenMP directive is coming;
- a directive, such as *parallel*;
- and possibly clauses with values.
- After the directive comes either a single statement or a block in *curly braces*.

Directives in C/C++ are case-sensitive. Directives can be broken over multiple lines by escaping the line end.

The sentinel in Fortran looks like a comment:

```
!$omp directive clause(value)
  statements
!$omp end directive
```

The difference with the C directive is that Fortran can not have a block, so there is an explicit *end-of directive* line.

If you break a directive over more than one line, all but the last line need to have a continuation character, and each line needs to have the sentinel:

```
!$OMP parallel do &
!%OMP   copyin(x),copyout(y)
```

The directives are case-insensitive. In *Fortran fixed-form source* files, `c$omp` and `*$omp` are allowed too.

15.3.2 Parallel regions

The simplest way to create parallelism in OpenMP is to use the `parallel` pragma. A block preceded by the `omp parallel` pragma is called a *parallel region*; it is executed by a newly created team of threads. This is an instance of the *Single Program Multiple Data (SPMD)* model: all threads execute (redundantly) the same segment of code.

```
#pragma omp parallel
{
  // this is executed by a team of threads
}
```

Exercise 15.1. Write a 'hello world' program, where the print statement is in a parallel region. Compile and run.
Run your program with different values of the environment variable `OMP_NUM_THREADS`. If you know how many cores your machine has, can you set the value higher?

We will go into much more detail in section 16.

15.3.3 An actual OpenMP program!

Let's start exploring how OpenMP handles parallelism, using the following functions:

- `omp_get_num_threads` reports how many threads are currently active, and
- `omp_get_thread_num` reports the number of the thread that makes the call.
- `omp_get_num_procs` reports the number of available cores.

Exercise 15.2. Take the hello world program of exercise 15.1 and insert the above functions, before, in, and after the parallel region. What are your observations?

Exercise 15.3. Extend the program from exercise 15.2. Make a complete program based on these lines:

```
int tsum=0;
#pragma omp parallel
  tsum += /* the thread number */
printf("Sum is %d\n",tsum);
```

Compile and run again. (In fact, run your program a number of times.) Do you see something unexpected? Can you think of an explanation?

15.3.4 Code and execution structure

Here are a couple of important concepts:

- An OpenMP directive is followed by an *structured block*; in C this is a single statement, a compound statement, or a block in braces; In Fortran it is delimited by the directive and its matching 'end' directive.
  A structured block can not be jumped into, so it can not start with a labeled statement, or contain a jump statement leaving the block.
- An OpenMP *construct* is the section of code starting with a directive and spanning the following structured block, plus in Fortran the end-directive. This is a lexical concept: it contains the statements directly enclosed, and not any subroutines called from them.
- A *region of code* is defined as all statements that are dynamically encountered while executing the code of an OpenMP construct. This is a dynamic concept: unlike a 'construct', it does include any subroutines that are called from the code in the structured block.

15.4 Thread data

In most programming languages, visibility of data is governed by rules on the *scope of variables*: a variable is declared in a block, and it is then visible to any statement in that block and blocks with a *lexical scope* contained in it, but not in surrounding blocks:

```
main () {
  // no variable `x' define here
  {
    int x = 5;
    if (somecondition) { x = 6; }
    printf("x=%e\n",x); // prints 5 or 6
  }
  printf("x=%e\n",x); // syntax error: `x' undefined
}
```

Fortran has simpler rules, since it does not have blocks inside blocks.

OpenMP has similar rules concerning data in parallel regions and other OpenMP constructs. First of all, data is visible in enclosed scopes:

```
main() {
  int x;
#pragma omp parallel
  {
     // you can use and set `x' here
  }
  printf("x=%e\n",x); // value depends on what
                      // happened in the parallel region
}
```

In C, you can redeclare a variable inside a nested scope:

```
{
  int x;
  if (something) {
    double x; // same name, different entity
  }
  x = ... // this refers to the integer again
}
```

Doing so makes the outer variable inaccessible.

OpenMP has a similar mechanism:

```
{
  int x;
#pragma omp parallel
  {
    double x;
  }
}
```

There is an important difference: each thread in the team gets its own instance of the enclosed variable.

This is illustrated in figure 15.5.

In addition to such scoped variables, which live on a *stack*, there are variables on the *heap*, typically created by a call to `malloc` (in C) or `new` (in C++). Rules for them are more complicated.

Summarizing the above, there are

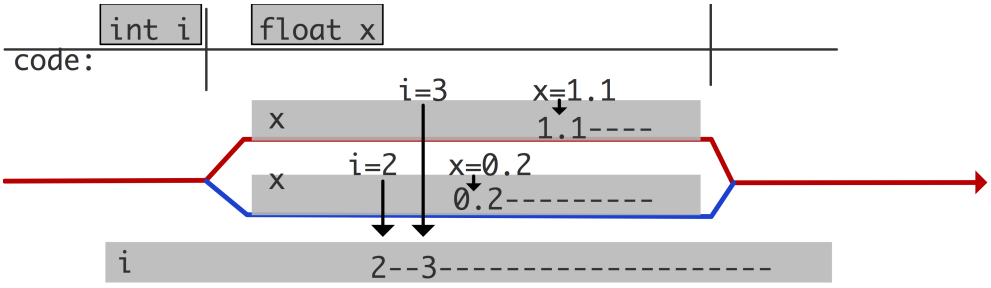


Figure 15.5: Locality of variables in threads

- *shared variables*, where each thread refers to the same data item, and
- *private variables*, where each thread has its own instance.

In addition to using scoping, OpenMP also uses options on the directives to control whether data is private or shared.

Many of the difficulties of parallel programming with OpenMP stem from the use of shared variables. For instance, if two threads update a shared variable, you not guarantee an order on the updates.

We will discuss all this in detail in section 19.

15.5 Creating parallelism

The *fork/join model* of OpenMP means that you need some way of indicating where an activity can be forked for independent execution. There are two ways of doing this:

1. You can declare a parallel region and split one thread into a whole team of threads. We will discuss this next in section 16. The division of the work over the threads is controlled by *work sharing construct* (section 18).
2. Alternatively, you can use tasks and indicating one parallel activity at a time. You will see this in section 22.

Note that OpenMP only indicates how much parallelism is present; whether independent activities are in fact executed in parallel is a runtime decision. The factors influencing this are discussed in section 15.5.

Declaring a parallel region tells OpenMP that a team of threads can be created. The actual size of the team depends on various factors (see section 26.1 for variables and functions mentioned in this section).

- The *environment variable* `OMP_NUM_THREADS` limits the number of threads that can be created.
- If you don't set this variable, you can also set this limit dynamically with the *library routine* `omp_set_num_threads`. This routine takes precedence over the aforementioned environment variable if both are specified.
- A limit on the number of threads can also be set as a clause on a parallel region.

If you specify a greater amount of parallelism than the hardware supports, the runtime system will probably ignore your specification and choose a lower value. To ask how much parallelism is actually used

in your parallel region, use `omp_get_num_threads`. To query these hardware limits, use `omp_get_num_procs` . You can query the maximum number of threads with `omp_get_max_threads`. This equals the value of `OMP_NUM_THREADS`, not the number of actually active threads in a parallel region.

```
// proccount.c
void nested_report() {
#pragma omp parallel
#pragma omp master
  printf("Parallel  : count %2d cores
    and %2d threads out of max %2d\n",
        omp_get_num_procs(),
    omp_get_num_threads(),
    omp_get_max_threads());
}

int main(int argc,char **argv) {

  printf("---------------- Parallelism
     report ----------------\n");

  printf("Sequential: count %2d cores
    and %2d threads out of max %2d\n",
        omp_get_num_procs(),
    omp_get_num_threads(),
    omp_get_max_threads());
#pragma omp parallel
#pragma omp master
  printf("Parallel  : count %2d cores
    and %2d threads out of max %2d\n",
        omp_get_num_procs(),
    omp_get_num_threads(),
    omp_get_max_threads());

#pragma omp parallel
#pragma omp master
  nested_report();
//codesnippet ompproccount

  return 0;
}
```

```
[c:48] for t in 1 2 4 8 16 ; do OMP_NUM_THREADS=$t ./proccount ; done
---------------- Parallelism report ----------------
Sequential: count  4 cores and  1 threads out of max  1
Parallel  : count  4 cores and  1 threads out of max  1
Parallel  : count  4 cores and  1 threads out of max  1
---------------- Parallelism report ----------------
Sequential: count  4 cores and  1 threads out of max  2
Parallel  : count  4 cores and  2 threads out of max  2
Parallel  : count  4 cores and  1 threads out of max  2
---------------- Parallelism report ----------------
Sequential: count  4 cores and  1 threads out of max  4
Parallel  : count  4 cores and  4 threads out of max  4
Parallel  : count  4 cores and  1 threads out of max  4
---------------- Parallelism report ----------------
Sequential: count  4 cores and  1 threads out of max  8
Parallel  : count  4 cores and  8 threads out of max  8
Parallel  : count  4 cores and  1 threads out of max  8
---------------- Parallelism report ----------------
Sequential: count  4 cores and  1 threads out of max 16
Parallel  : count  4 cores and 16 threads out of max 16
Parallel  : count  4 cores and  1 threads out of max 16
```

Another limit on the number of threads is imposed when you use nested parallel regions. This can arise if you have a parallel region in a subprogram which is sometimes called sequentially, sometimes in parallel. The variable `OMP_NESTED` controls whether the inner region will create a team of more than one thread.

15.6 Sources used in this chapter

15.6.1 Listing of code header

Chapter 16

OpenMP topic: Parallel regions

The simplest way to create parallelism in OpenMP is to use the `parallel` pragma. A block preceded by the `omp parallel` pragma is called a *parallel region*; it is executed by a newly created team of threads. This is an instance of the *SPMD* model: all threads execute the same segment of code.

```
#pragma omp parallel
{
  // this is executed by a team of threads
}
```

It would be pointless to have the block be executed identically by all threads. One way to get a meaningful parallel code is to use the function `omp_get_thread_num`, to find out which thread you are, and execute work that is individual to that thread. There is also a function `omp_get_num_threads` to find out the total number of threads. Both these functions give a number relative to the current team; recall from figure 15.3 that new teams can be created recursively.

For instance, if you program computes

```
result = f(x)+g(x)+h(x)
```

you could parallelize this as

```
double result,fresult,gresult,hresult;
#pragma omp parallel
{ int num = omp_get_thread_num();
  if (num==0)      fresult = f(x);
  else if (num==1) gresult = g(x);
  else if (num==2) hresult = h(x);
}
result = fresult + gresult + hresult;
```

The first thing we want to do is create a team of threads. This is done with a *parallel region*. Here is a very simple example:

```
// hello.c
#pragma omp parallel
  {
    int t = omp_get_thread_num();
    printf("Hello world from %d!\n",t);
  }
```

For the full source of this example, see section 16.3.2

or in Fortran

```
// hellocount.F90
!$omp parallel
  nthreads = omp_get_num_threads()
  mythread = omp_get_thread_num()
  write(*,'("Hello from",i3," out of",i3)') mythread,nthreads
!$omp end parallel
```

For the full source of this example, see section 16.3.3

or in C++

```
// hello.cxx
#pragma omp parallel
  {
    int t = omp_get_thread_num();
    stringstream proctext;
    proctext << "Hello world from " << t << endl;
    cerr << proctext.str();
  }
```

For the full source of this example, see section 16.3.4

(Note the use of `stringstream`: without that the output lines from the various threads may get mixed up.)

This code corresponds to the model we just discussed:

- Immediately preceding the parallel block, one thread will be executing the code. In the main program this is the *initial thread*.
- At the start of the block, a new *team of threads* is created, and the thread that was active before the block becomes the *master thread* of that team.
- After the block only the master thread is active.
- Inside the block there is team of threads: each thread in the team executes the body of the block, and it will have access to all variables of the surrounding environment. How many threads there are can be determined in a number of ways; we will get to that later.

Remark 17 *In future versions of OpenMP, the master thread will be called the* primary thread. *In 5.1 the master construct will be deprecated, and masked (with added functionality) will take its place. In 6.0 master will disappear from the Spec, including* `proc_bind` *master "variable" and combined master constructs (master taskloop, etc.)*

Exercise 16.1. Make a full program based on this fragment. Insert different print statements before, inside, and after the parallel region. Run this example. How many times is each print statement executed?

You see that the `parallel` directive

- Is preceded by a special marker: a `#pragma omp` for C/C++, and the `!$OMP` *sentinel* for Fortran;
- Is followed by a single statement or a block in C/C++, or followed by a block in Fortran which is delimited by an `!$omp end` directive.

Directives look like *cpp directives*, but they are actually handled by the compiler, not the preprocessor.

Exercise 16.2. Take the 'hello world' program above, and modify it so that you get multiple messages to you screen, saying

```
Hello from thread 0 out of 4!
Hello from thread 1 out of 4!
```

and so on. (The messages may very well appear out of sequence.)
What happens if you set your number of threads larger than the available cores on your computer?

Exercise 16.3. What happens if you call `omp_get_thread_num` and `omp_get_num_threads` outside a parallel region?

`omp_get_thread_limit`

`OMP_WAIT_POLICY` values: `ACTIVE,PASSIVE`

16.1 Nested parallelism

What happens if you call a function from inside a parallel region, and that function itself contains a parallel region?

```
int main() {
  ...
#pragma omp parallel
  {
  ...
  func(...)
  ...
  }
} // end of main
void func(...) {
#pragma omp parallel
  {
  ...
  }
}
```

By default, the nested parallel region will have only one thread. To allow nested thread creation, set `OMP_NESTED` (default: false):

```
OMP_NESTED=true
 or
omp_set_nested(1)
```

For more fine-grained control use the environment variable `OMP_MAX_ACTIVE_LEVELS` (default: 1) or the functions `omp_set_max_active_levels` and `omp_get_max_active_levels`:

```
OMP_MAX_ACTIVE_LEVELS=3
 or
void omp_set_max_active_levels(int);
int omp_get_max_active_levels(void);
```

Exercise 16.4. Test nested parallelism by writing an OpenMP program as follows:

1. Write a subprogram that contains a parallel region.
2. Write a main program with a parallel region; call the subprogram both inside and outside the parallel region.
3. Insert print statements
   (a) in the main program outside the parallel region,
   (b) in the parallel region in the main program,
   (c) in the subprogram outside the parallel region,
   (d) in the parallel region inside the subprogram.

Run your program and count how many print statements of each type you get.

Writing subprograms that are called in a parallel region illustrates the following point: directives are evaluation with respect to the *dynamic scope* of the parallel region, not just the lexical scope. In the following example:

```
#pragma omp parallel
{
  f();
}
void f() {
#pragma omp for
  for ( .... ) {
    ...
  }
}
```

the body of the function `f` falls in the dynamic scope of the parallel region, so the for loop will be parallelized.

If the function may be called both from inside and outside parallel regions, you can test which is the case with `omp_in_parallel`.

The amount of nested parallelism can be set:

```
OMP_NUM_THREADS=4,2
```

means that initially a parallel region will have four threads, and each thread can create two more threads.

```
OMP_MAX_ACTIVE_LEVELS=123

omp_set_max_active_levels( n )
n = omp_get_max_active_levels()

OMP_THREAD_LIMIT=123

n = omp_get_thread_limit()

omp_set_max_active_levels
omp_get_max_active_levels
omp_get_level
```

```
omp_get_active_level
omp_get_ancestor_thread_num

omp_get_team_size(level)
```

16.2 Cancel parallel construct

```
!$omp cancel construct [if (expr)]
```

where construct is *parallel*, sections, do or *taskgroup*

16.3 Sources used in this chapter

16.3.1 Listing of code header

16.3.2 Listing of code examples/omp/c/hello.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc,char **argv) {

#pragma omp parallel
  {
    int t = omp_get_thread_num();
    printf("Hello world from %d!\n",t);
  }

  return 0;
}
```

16.3.3 Listing of code examples/omp/f/hellocount.F90

```
  use omp_lib
  integer :: nthreads,mythread

!$omp parallel
  nthreads = omp_get_num_threads()
  mythread = omp_get_thread_num()
  write(*,'("Hello from",i3," out of",i3)') mythread,nthreads
!$omp end parallel

end Program Hello
```

16.3.4 Listing of code examples/omp/cxx/hello.cxx

```
#include <iostream>
using std::cerr;
using std::cout;
using std::endl;
#include <sstream>
using std::stringstream;

#include <omp.h>

int main(int argc,char **argv) {

#pragma omp parallel
  {
    int t = omp_get_thread_num();
```

```
    stringstream proctext;
    proctext << "Hello world from " << t << endl;
    cerr << proctext.str();
  }

  return 0;
}
```

Chapter 17

OpenMP topic: Loop parallelism

17.1 Loop parallelism

Loop parallelism is a very common type of parallelism in scientific codes, so OpenMP has an easy mechanism for it. OpenMP parallel loops are a first example of OpenMP 'worksharing' constructs (see section 18 for the full list): constructs that take an amount of work and distribute it over the available threads in a parallel region.

The parallel execution of a loop can be handled a number of different ways. For instance, you can create a parallel region around the loop, and adjust the loop bounds:

```
#pragma omp parallel
{
  int threadnum = omp_get_thread_num(),
    numthreads = omp_get_num_threads();
  int low = N*threadnum/numthreads,
    high = N*(threadnum+1)/numthreads;
  for (i=low; i<high; i++)
    // do something with i
}
```

A more natural option is to use the `parallel for` pragma:

```
#pragma omp parallel
#pragma omp for
for (i=0; i<N; i++) {
  // do something with i
}
```

This has several advantages. For one, you don't have to calculate the loop bounds for the threads yourself, but you can also tell OpenMP to assign the loop iterations according to different schedules (section 17.2).

Figure 17.1 shows the execution on four threads of

```
#pragma omp parallel
{
  code1();
#pragma omp for
  for (i=1; i<=4*N; i++) {
```

```
    code2();
  }
  code3();
}
```

The code before and after the loop is executed identically in each thread; the loop iterations are spread over the four threads.

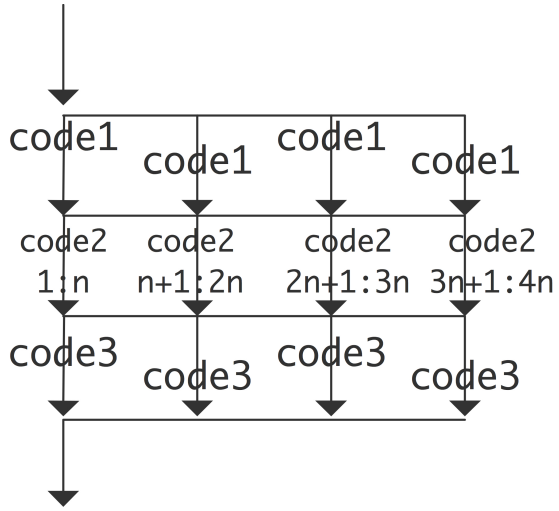


Figure 17.1: Execution of parallel code inside and outside a loop

Note that the `parallel do` and `parallel for` pragmas do not create a team of threads: they take the team of threads that is active, and divide the loop iterations over them.

This means that the `omp for` or `omp do` directive needs to be inside a parallel region. It is also possible to have a combined `omp parallel for` or `omp parallel do` directive.

If your parallel region only contains a loop, you can combine the pragmas for the parallel region and distribution of the loop iterations:

```
#pragma omp parallel for
  for (i=0; .....
```

Exercise 17.1. Compute π by *numerical integration*. We use the fact that π is the area of the unit circle, and we approximate this by computing the area of a quarter circle using *Riemann sums*.

- Let $f(x) = \sqrt{1-x^2}$ be the function that describes the quarter circle for $x = 0 \ldots 1$;
- Then we compute

$$\pi/4 \approx \sum_{i=0}^{N-1} \Delta x f(x_i) \qquad \text{where } x_i = i\Delta x \text{ and } \Delta x = 1/N$$

Write a program for this, and parallelize it using OpenMP parallel for directives.

1. Put a *parallel* directive around your loop. Does it still compute the right result? Does the time go down with the number of threads? (The answers should be no and no.)
2. Change the `parallel` to `parallel for` (or `parallel do`). Now is the result correct? Does execution speed up? (The answers should now be no and yes.)
3. Put a *critical* directive in front of the update. (Yes and very much no.)
4. Remove the *critical* and add a clause *reduction*`(+:quarterpi)` to the `for` directive. Now it should be correct and efficient.

Use different numbers of cores and compute the speedup you attain over the sequential computation. Is there a performance difference between the OpenMP code with 1 thread and the sequential code?

Remark 18 *In this exercise you may have seen the runtime go up a couple of times where you weren't expecting it. The issue here is* false sharing*; see HPSC-3.5.5 for more explanation.*

There are some restrictions on the loop: basically, OpenMP needs to be able to determine in advance how many iterations there will be.

- The loop can not contains `break`, `return`, `exit` statements, or `goto` to a label outside the loop.
- The `continue` (C) or `cycle` (F) statement is allowed.
- The index update has to be an increment (or decrement) by a fixed amount.
- The loop index variable is automatically private, and not changes to it inside the loop are allowed.

17.2 Loop schedules

Usually you will have many more iterations in a loop than there are threads. Thus, there are several ways you can assign your loop iterations to the threads. OpenMP lets you specify this with the *schedule* clause.

```
#pragma omp for schedule(....)
```

The first distinction we now have to make is between static and dynamic schedules. With static schedules, the iterations are assigned purely based on the number of iterations and the number of threads (and the *chunk* parameter; see later). In dynamic schedules, on the other hand, iterations are assigned to threads that are unoccupied. Dynamic schedules are a good idea if iterations take an unpredictable amount of time, so that *load balancing* is needed.

Figure 17.2 illustrates this: assume that each core gets assigned two (blocks of) iterations and these blocks take gradually less and less time. You see from the left picture that thread 1 gets two fairly long blocks, where as thread 4 gets two short blocks, thus finishing much earlier. (This phenomenon of threads having unequal amounts of work is known as *load imbalance*.) On the other hand, in the right figure thread 4 gets block 5, since it finishes the first set of blocks early. The effect is a perfect load balancing.

The default static schedule is to assign one consecutive block of iterations to each thread. If you want different sized blocks you can defined a `chunk` size:

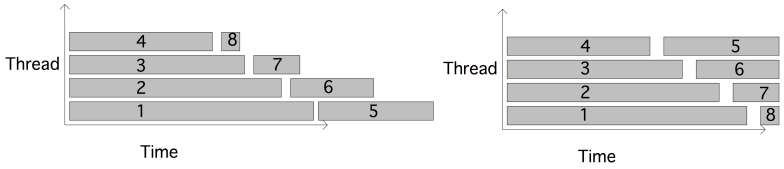


Figure 17.2: Illustration static round-robin scheduling versus dynamic

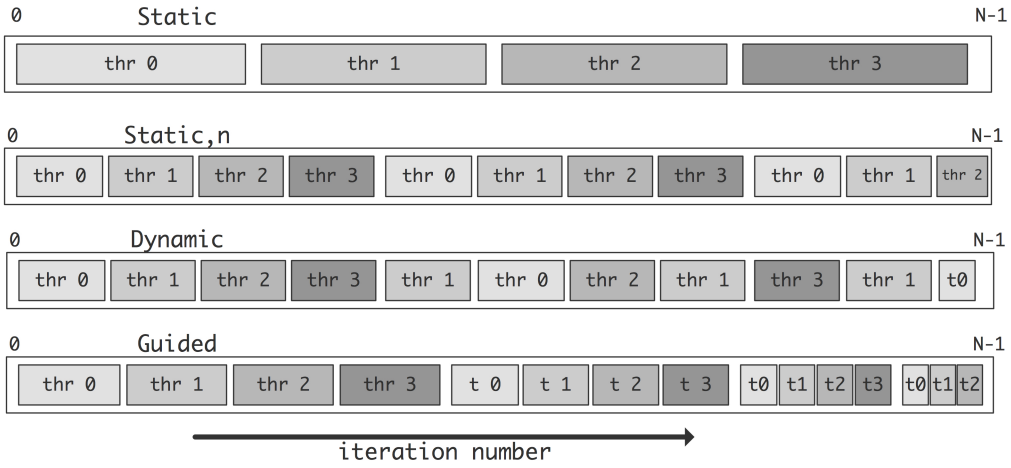


Figure 17.3: Illustration of the scheduling strategies of loop iterations

```
#pragma omp for schedule(static[,chunk])
```

(where the square brackets indicate an optional argument). With static scheduling, the compiler will split up the loop iterations at compile time, so, provided the iterations take roughly the same amount of time, this is the most efficient at runtime.

The choice of a chunk size is often a balance between the low overhead of having only a few chunks, versus the load balancing effect of having smaller chunks.

Exercise 17.2. Why is a chunk size of 1 typically a bad idea? (Hint: think about cache lines, and read HPSC-1.4.1.2.)

In dynamic scheduling OpenMP will put blocks of iterations (the default chunk size is 1) in a task queue, and the threads take one of these tasks whenever they are finished with the previous.

```
#pragma omp for schedule(static[,chunk])
```

While this schedule may give good load balancing if the iterations take very differing amounts of time to execute, it does carry runtime overhead for managing the queue of iteration tasks.

Finally, there is the `guided` schedule, which gradually decreases the chunk size. The thinking here is that large chunks carry the least overhead, but smaller chunks are better for load balancing. The various schedules are illustrated in figure 17.3.

If you don't want to decide on a schedule in your code, you can specify the `runtime` schedule. The actual schedule will then at runtime be read from the `OMP_SCHEDULE` environment variable. You can even just leave it to the runtime library by specifying `auto`

Exercise 17.3. We continue with exercise 17.1. We add 'adaptive integration': where needed, the program refines the step size[1]. This means that the iterations no longer take a predictable amount of time.

```
for (i=0; i<nsteps; i++) {
    double
    x = i*h,x2 = (i+1)*h,
    y = sqrt(1-x*x),y2 = sqrt
    (1-x2*x2),
    slope = (y-y2)/h;
    if (slope>15) slope = 15;
    int
    samples = 1+(int)slope,
    is;
    for (is=0; is<samples; is
    ++) {
        double
        hs = h/samples,
        xs = x+ is*hs,
        ys = sqrt(1-xs*xs);
        quarterpi += hs*ys;
        nsamples++;
    }
}
  pi = 4*quarterpi;
```

1. Use the `omp parallel for` construct to parallelize the loop. As in the previous lab, you may at first see an incorrect result. Use the *reduction* clause to fix this.
2. Your code should now see a decent speedup, using up to 8 cores. However, it is possible to get completely linear speedup. For this you need to adjust the schedule.
   Start by using *schedule*`(static,$n$)`. Experiment with values for n. When can you get a better speedup? Explain this.
3. Since this code is somewhat dynamic, try *schedule*`(dynamic)`. This will actually give a fairly bad result. Why? Use *schedule*`(dynamic,$n$)` instead, and experiment with values for n.
4. Finally, use *schedule*`(guided)`, where OpenMP uses a heuristic. What results does that give?

Exercise 17.4. Program the *LU factorization* algorithm without pivoting.

```
for k=1,n:
  A[k,k] = 1./A[k,k]
  for i=k+1,n:
    A[i,k] = A[i,k]/A[k,k]
    for j=k+1,n:
      A[i,j] = A[i,j] - A[i,k]*A[k,j]
```

1. Argue that it is not possible to parallelize the outer loop.
2. Argue that it is possible to parallelize both the i and j loops.
3. Parallelize the algorithm by focusing on the i loop. Why is the algorithm as given here best for a matrix on row-storage? What would you do if the matrix was on column storage?

1. It doesn't actually do this in a mathematically sophisticated way, so this code is more for the sake of the example.

4. Argue that with the default schedule, if a row is updated by one thread in one iteration, it may very well be updated by another thread in another. Can you find a way to schedule loop iterations so that this does not happen? What practical reason is there for doing so?

The schedule can be declared explicitly, set at runtime through the `OMP_SCHEDULE` environment variable, or left up to the runtime system by specifying `auto`. Especially in the last two cases you may want to enquire what schedule is currently being used with `omp_get_schedule`.

```
int omp_get_schedule(omp_sched_t * kind, int * modifier );
```

Its mirror call is `omp_set_schedule`, which sets the value that is used when schedule value `runtime` is used. It is in effect equivalent to setting the environment variable `OMP_SCHEDULE`.

```
void omp_set_schedule (omp_sched_t kind, int modifier);
```

Type	environment variable OMP_SCHEDULE=	clause schedule(...)	modifier default
static	static[,n]	static[,n]	$N/nthreads$
dynamic	dynamic[,n]	dynamic[,n]	1
guided	guided[,n]	guided[,n]	

Here are the various schedules you can set with the *schedule* clause:

affinity Set by using value *omp_sched_affinity*
auto The schedule is left up to the implementation. Set by using value *omp_sched_auto*
dynamic value: 2. The modifier parameter is the *chunk* size; default 1. Set by using value *omp_sched_dynamic*

guided Value: 3. The modifier parameter is the chunk size. Set by using value *omp_sched_guided*
runtime Use the value of the `OMP_SCHEDULE` environment variable. Set by using value *omp_sched_runtime*
static value: 1. The modifier parameter is the *chunk* size. Set by using value *omp_sched_static*

17.3 Reductions

So far we have focused on loops with independent iterations. Reductions are a common type of loop with dependencies. There is an extended discussion of reductions in section 20.

17.4 Collapsing nested loops

In general, the more work there is to divide over a number of threads, the more efficient the parallelization will be. In the context of parallel loops, it is possible to increase the amount of work by parallelizing all levels of loops instead of just the outer one.

Example: in

```
for ( i=0; i<N; i++ )
  for ( j=0; j<N; j++ )
    A[i][j] = B[i][j] + C[i][j]
```

all N^2 iterations are independent, but a regular `omp for` directive will only parallelize one level. The `collapse` clause will parallelize more than one level:

```
#pragma omp for collapse(2)
for ( i=0; i<N; i++ )
  for ( j=0; j<N; j++ )
    A[i][j] = B[i][j] + C[i][j]
```

It is only possible to collapse perfectly nested loops, that is, the loop body of the outer loop can consist only of the inner loop; there can be no statements before or after the inner loop in the loop body of the outer loop. That is, the two loops in

```
for (i=0; i<N; i++) {
  y[i] = 0.;
  for (j=0; j<N; j++)
    y[i] += A[i][j] * x[j]
  }
```

can not be collapsed.

Exercise 17.5. Can you rewrite the preceding code example so that it can be collapsed? Do timing tests to see if you can notice the improvement from collapsing.

17.5 Ordered iterations

Iterations in a parallel loop that are execution in parallel do not execute in lockstep. That means that in

```
#pragma omp parallel for
for ( ... i ... ) {
  ... f(i) ...
  printf("something with %d\n",i);
}
```

it is not true that all function evaluations happen more or less at the same time, followed by all print statements. The print statements can really happen in any order. The `ordered` clause coupled with the `ordered` directive can force execution in the right order:

```
#pragma omp parallel for ordered
for ( ... i ... ) {
  ... f(i) ...
#pragma omp ordered
  printf("something with %d\n",i);
}
```

Example code structure:

```
#pragma omp parallel for shared(y) ordered
for ( ... i ... ) {
  int x = f(i)
#pragma omp ordered
  y[i] += f(x)
  z[i] = g(y[i])
}
```

There is a limitation: each iteration can encounter only one *ordered* directive.

17.6 nowait

The implicit barrier at the end of a work sharing construct can be cancelled with a `nowait` clause. This has the effect that threads that are finished can continue with the next code in the parallel region:

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i<N; i++) { ... }
  // more parallel code
}
```

In the following example, threads that are finished with the first loop can start on the second. Note that this requires both loops to have the same schedule. We specify the static schedule here to have an identical scheduling of iterations over threads:

```
#pragma omp parallel
{
  x = local_computation()
#pragma omp for schedule(static) nowait
  for (i=0; i<N; i++) {
    x[i] = ...
  }
#pragma omp for schedule(static)
  for (i=0; i<N; i++) {
    y[i] = ... x[i] ...
  }
}
```

17.7 While loops

OpenMP can only handle 'for' loops: *while loops* can not be parallelized. So you have to find a way around that. While loops are for instance used to search through data:

```
while ( a[i]!=0 && i<imax ) {
 i++; }
// now i is the first index for which \n{a[i]} is zero.
```

We replace the while loop by a for loop that examines all locations:

```
result = -1;
#pragma omp parallel for
for (i=0; i<imax; i++) {
  if (a[i]!=0 && result<0) result = i;
}
```

Exercise 17.6. Show that this code has a race condition.

You can fix the race condition by making the condition into a critical section; section 21.2.1. In this particular example, with a very small amount of work per iteration, that is likely to be inefficient in this case (why?). A more efficient solution uses the `lastprivate` pragma:

```
result = -1;
#pragma omp parallel for lastprivate(result)
for (i=0; i<imax; i++) {
  if (a[i]!=0) result = i;
}
```

You have now solved a slightly different problem: the result variable contains the *last* location where `a[i]` is zero.

17.8 Sources used in this chapter

17.8.1 Listing of code header

Chapter 18

OpenMP topic: Work sharing

The declaration of a *parallel region* establishes a team of threads. This offers the possibility of parallelism, but to actually get meaningful parallel activity you need something more. OpenMP uses the concept of a *work sharing construct*: a way of dividing parallelizable work over a team of threads. The work sharing constructs are:

- `for` (for C) or `do` (for Fortran). The threads divide up the loop iterations among themselves; see 17.1.
- `sections` The threads divide a fixed number of sections between themselves; see section 18.1.
- `single` The section is executed by a single thread; section 18.2.
- `task` See section 22.
- `workshare` Can parallelize Fortran array syntax; section 18.3.

18.1 Sections

A parallel loop is an example of independent work units that are numbered. If you have a pre-determined number of independent work units, the `sections` is more appropriate. In a `sections` construct can be any number of `section` constructs. These need to be independent, and they can be execute by any available thread in the current team, including having multiple sections done by the same thread.

```
#pragma omp sections
{
#pragma omp section
  // one calculation
#pragma omp section
  // another calculation
}
```

This construct can be used to divide large blocks of independent work. Suppose that in the following line, both `f(x)` and `g(x)` are big calculations:

```
y = f(x) + g(x)
```

You could then write

```
double y1,y2;
#pragma omp sections
{
#pragma omp section
  y1 = f(x)
#pragma omp section
  y2 = g(x)
}
y = y1+y2;
```

Instead of using two temporaries, you could also use a critical section; see section 21.2.1. However, the best solution is have a `reduction` clause on the `sections` directive:

```
  y = f(x) + g(x)
```

You could then write

```
y = 0;
#pragma omp sections reduction(+:y)
{
#pragma omp section
  y += f(x)
#pragma omp section
  y += g(x)
}
```

18.2 Single/master

The `single` and `master` pragma limit the execution of a block to a single thread. This can for instance be used to print tracing information or doing *I/O* operations.

```
#pragma omp parallel
{
#pragma omp single
  printf("We are starting this section!\n");
  // parallel stuff
}
```

Another use of `single` is to perform initializations in a parallel region:

```
int a;
#pragma omp parallel
{
  #pragma omp single
    a = f(); // some computation
  #pragma omp sections
    // various different computations using a
}
```

The point of the single directive in this last example is that the computation needs to be done only once, because of the shared memory. Since it's a work sharing construct there is an *implicit barrier* after it, which guarantees that all threads have the correct value in their local memory (see section 24.3.

Exercise 18.1. What is the difference between this approach and how the same computation would be parallelized in MPI?

The `master` directive, also enforces execution on a single thread, specifically the master thread of the team, but it does not have the synchronization through the implicit barrier.

Exercise 18.2. Modify the above code to read:

```
int a;
#pragma omp parallel
{
  #pragma omp master
    a = f(); // some computation
  #pragma omp sections
    // various different computations using a
}
```

This code is no longer correct. Explain.

Above we motivated the `single` directive as a way of initializing shared variables. It is also possible to use `single` to initialize private variables. In that case you add the `copyprivate` clause. This is a good solution if setting the variable takes I/O.

Exercise 18.3. Give two other ways to initialize a private variable, with all threads receiving the same value. Can you give scenarios where each of the three strategies would be preferable?

18.3 Fortran array syntax parallelization

The `parallel do` directive is used to parallelize loops, and this applies to both C and Fortran. However, Fortran also has implied loops in its *array syntax*. To parallelize array syntax you can use the `workshare` directive.

The `workshare` directive exists only in Fortran. It can be used to parallelize the implied loops in *array syntax*, as well as *forall* loops.

18.4 Sources used in this chapter

18.4.1 Listing of code header

Chapter 19

OpenMP topic: Controlling thread data

In a parallel region there are two types of data: private and shared. In this sections we will see the various way you can control what category your data falls under; for private data items we also discuss how their values relate to shared data.

19.1 Shared data

In a parallel region, any data declared outside it will be shared: any thread using a variable `x` will access the same memory location associated with that variable.

Example:

```
  int x = 5;
#pragma omp parallel
  {
    x = x+1;
    printf("shared: x is %d\n",x);
  }
```

All threads increment the same variable, so after the loop it will have a value of five plus the number of threads; or maybe less because of the data races involved. See HPSC-2.6.1.5 for an explanation of the issues involved; see 21.2.1 for a solution in OpenMP.

Sometimes this global update is what you want; in other cases the variable is intended only for intermediate results in a computation. In that case there are various ways of creating data that is local to a thread, and therefore invisible to other threads.

19.2 Private data

In the C/C++ language it is possible to declare variables inside a *lexical scope*; roughly: inside curly braces. This concept extends to OpenMP parallel regions and directives: any variable declared in a block following an OpenMP directive will be local to the executing thread.

Example:

```
  int x = 5;
#pragma omp parallel
  {
    int x; x = 3;
    printf("local: x is %d\n",x);
  }
```

After the parallel region the outer variable `x` will still have the value 5: there is no *storage association* between the private variable and global one.

The Fortran language does not have this concept of scope, so you have to use a `private` clause:

```
!$OMP parallel private(x)
```

The `private` directive declares data to have a separate copy in the memory of each thread. Such private variables are initialized as they would be in a main program. Any computed value goes away at the end of the parallel region. (However, see below.) Thus, you should not rely on any initial value, or on the value of the outer variable after the region.

```
  int x = 5;
#pragma omp parallel private(x)
  {
    x = x+1; // dangerous
    printf("private: x is %d\n",x);
  }
  printf("after: x is %d\n",x); // also dangerous
```

Data that is declared private with the `private` directive is put on a separate *stack per thread*. The OpenMP standard does not dictate the size of these stacks, but beware of *stack overflow*. A typical default is a few megabyte; you can control it with the environment variable `OMP_STACKSIZE`. Its values can be literal or with suffixes:

```
123 456k 567K 678m 789M 246g 357G
```

A normal *Unix process* also has a stack, but this is independent of the OpenMP stacks for private data. You can query or set the Unix stack with `ulimit`:

```
[] ulimit -s
64000
[] ulimit -s 8192
[] ulimit -s
8192
```

The Unix stack can grow dynamically as space is needed. This does not hold for the OpenMP stacks: they are immediately allocated at their requested size. Thus it is important not too make them too large.

19.3 Data in dynamic scope

Functions that are called from a parallel region fall in the *dynamic scope* of that parallel region. The rules for variables in that function are as follows:

- Any variables locally defined to the function are private.
- `static` variables in C and `save` variables in Fortran are shared.
- The function arguments inherit their status from the calling environment.

19.4 Temporary variables in a loop

It is common to have a variable that is set and used in each loop iteration:

```
#pragma omp parallel for
for ( ... i ... ) {
  x = i*h;
  s = sin(x); c = cos(x);
  a[i] = s+c;
  b[i] = s-c;
}
```

By the above rules, the variables `x,s,c` are all shared variables. However, the values they receive in one iteration are not used in a next iteration, so they behave in fact like private variables to each iteration.

- In both C and Fortran you can declare these variables private in the parallel for directive.
- In C, you can also redefine the variables inside the loop.

Sometimes, even if you forget to declare these temporaries as private, the code may still give the correct output. That is because the compiler can sometimes eliminate them from the loop body, since it detects that their values are not otherwise used.

19.5 Default

- Loop variables in an `omp for` are private;
- Local variables in the parallel region are private.

You can alter this default behaviour with the `default` clause:

```
#pragma omp parallel default(shared) private(x)
{ ... }
#pragma omp parallel default(private) shared(matrix)
{ ... }
```

and if you want to play it safe:

```
#pragma omp parallel default(none) private(x) shared(matrix)
{ ... }
```

- The `shared` clause means that all variables from the outer scope are shared in the parallel region; any private variables need to be declared explicitly. This is the default behaviour.
- The `private` clause means that all outer variables become private in the parallel region. They are not initialized; see the next option. Any shared variables in the parallel region need to be declared explicitly. This value is not available in C.

- The `firstprivate` clause means all outer variables are private in the parallel region, and initialized with their outer value. Any shared variables need to be declared explicitly. This value is not available in C.
- The `none` option is good for debugging, because it forces you to specify for each variable in the parallel region whether it's private or shared. Also, if your code behaves differently in parallel from sequential there is probably a data race. Specifying the status of every variable is a good way to debug this.

19.6 Array data

The rules for arrays are slightly different from those for scalar data:

1. Statically allocated data, that is with a syntax like

```
int array[100];
integer,dimension(:) :: array(100}
```

   can be shared or private, depending on the clause you use.
2. Dynamically allocated data, that is, created with `malloc` or `allocate`, can only be shared.

Example of the first type: in

```
// alloc3.c
int array[nthreads];
{
  int t = 2;
  array += t;
  array[0] = t;
}
```

For the full source of this example, see section 19.9.2

each thread gets a private copy of the array, properly initialized.

On the other hand, in

```
// alloc1.c
int *array = (int*) malloc(nthreads*sizeof(int));
#pragma omp parallel firstprivate(array)
{
  int t = omp_get_thread_num();
  array += t;
  array[0] = t;
}
```

For the full source of this example, see section 19.9.3

each thread gets a private pointer, but all pointers point to the same object.

19.7 First and last private

Above, you saw that private variables are completely separate from any variables by the same name in the surrounding scope. However, there are two cases where you may want some *storage association* between a private variable and a global counterpart.

First of all, private variables are created with an undefined value. You can force their initialization with `firstprivate`.

```
  int t=2;
#pragma omp parallel firstprivate(t)
  {
    t += f( omp_get_thread_num() );
    g(t);
  }
```

The variable `t` behaves like a private variable, except that it is initialized to the outside value.

Secondly, you may want a private value to be preserved to the environment outside the parallel region. This really only makes sense in one case, where you preserve a private variable from the last iteration of a parallel loop, or the last section in an `sections` construct. This is done with `lastprivate`:

```
#pragma omp parallel for \
        lastprivate(tmp)
for (i=0; i<N; i+) {
  tmp = ......
  x[i] = .... tmp ....
}
..... tmp ....
```

19.8 Persistent data through `threadprivate`

Most data in OpenMP parallel regions is either inherited from the master thread and therefore shared, or temporary within the scope of the region and fully private. There is also a mechanism for *thread-private data*, which is not limited in lifetime to one parallel region. The `threadprivate` pragma is used to declare that each thread is to have a private copy of a variable:

```
#pragma omp threadprivate(var)
```

The variable needs be:

- a file or static variable in C,
- a static class member in C++, or
- a program variable or common block in Fortran.

19.8.1 Thread private initialization

If each thread needs a different value in its threadprivate variable, the initialization needs to happen in a parallel region.

In the following example a team of 7 threads is created, all of which set their thread-private variable. Later, this variable is read by a larger team: the variables that have not been set are undefined, though often simply zero:

```
// threadprivate.c
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

static int tp;

int main(int argc,char **argv) {

#pragma omp threadprivate(tp)

#pragma omp parallel num_threads(7)
  tp = omp_get_thread_num();

#pragma omp parallel num_threads(9)
  printf("Thread %d has %d\n",omp_get_thread_num(),tp);

  return 0;
}
```

For the full source of this example, see section 19.9.4

On the other hand, if the thread private data starts out identical in all threads, the `copyin` clause can be used:

```
#pragma omp threadprivate(private_var)

private_var = 1;
#pragma omp parallel copyin(private_var)
  private_var += omp_get_thread_num()
```

If one thread needs to set all thread private data to its value, the `copyprivate` clause can be used:

```
#pragma omp parallel
{
  ...
#pragma omp single copyprivate(private_var)
  private_var = read_data();
  ...
}
```

19.8.2 Thread private example

The typical application for thread-private variables is in *random number generator*s. A random number generator needs saved state, since it computes each next value from the current one. To have a parallel generator, each thread will create and initialize a private 'current value' variable. This will persist even when the execution is not in a parallel region; it gets updated only in a parallel region.

Exercise 19.1. Calculate the area of the *Mandelbrot set* by random sampling. Initialize the random number generator separately for each thread; then use a parallel loop to evaluate the points. Explore performance implications of the different loop scheduling strategies.

Fortran note. Named common blocks can be made thread-private with the syntax

```
!$OMP threadprivate( /blockname/ )
```

Threadprivate variables require `OMP_DYNAMIC` to be switched off.

19.9 Sources used in this chapter

19.9.1 Listing of code header

19.9.2 Listing of code examples/omp/c/alloc2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc,char **argv) {

  int nthreads;
#pragma omp parallel
#pragma omp master
  nthreads = omp_get_num_threads();

  int array[nthreads];
  for (int i=0; i<nthreads; i++)
    array[i] = 0;

#pragma omp parallel firstprivate(array)
  {
    int t = omp_get_thread_num();
    array[t] = t;
  }

  printf("Array result:\n");
  for (int i=0; i<nthreads; i++)
    printf("%d:%d, ",i,array[i]);
  printf("\n");

  return 0;
}
```

19.9.3 Listing of code examples/omp/c/alloc1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

int main(int argc,char **argv) {

  int nthreads;
#pragma omp parallel
#pragma omp master
  nthreads = omp_get_num_threads();

  int *array = (int*) malloc(nthreads*sizeof(int));
  for (int i=0; i<nthreads; i++)
    array[i] = 0;
```

```
#pragma omp parallel firstprivate(array)
  {
    int t = omp_get_thread_num();
    array += t;
    array[0] = t;
  }

  printf("Array result:\n");
  for (int i=0; i<nthreads; i++)
    printf("%d:%d, ",i,array[i]);
  printf("\n");

  return 0;
}
```

19.9.4 Listing of code examples/omp/c/threadprivate.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

static int tp;

int main(int argc,char **argv) {

#pragma omp threadprivate(tp)

#pragma omp parallel num_threads(7)
  tp = omp_get_thread_num();

#pragma omp parallel num_threads(9)
  printf("Thread %d has %d\n",omp_get_thread_num(),tp);

  return 0;
}
```

Chapter 20

OpenMP topic: Reductions

Parallel tasks often produce some quantity that needs to be summed or otherwise combined. In section 16 you saw an example, and it was stated that the solution given there was not very good.

The problem in that example was the *race condition* involving the `result` variable. The simplest solution is to eliminate the race condition by declaring a *critical section*:

```
double result = 0;
#pragma omp parallel
{
  double local_result;
  int num = omp_get_thread_num();
  if (num==0)      local_result = f(x);
  else if (num==1) local_result = g(x);
  else if (num==2) local_result = h(x);
#pragma omp critical
  result += local_result;
}
```

This is a good solution if the amount of serialization in the critical section is small compared to computing the functions f, g, h. On the other hand, you may not want to do that in a loop:

```
double result = 0;
#pragma omp parallel
{
  double local_result;
#pragma omp for
  for (i=0; i<N; i++) {
    local_result = f(x,i);
#pragma omp critical
    result += local_result;
  } // end of for loop
}
```

Exercise 20.1. Can you think of a small modification of this code, that still uses a critical section, that is more efficient? Time both codes.

The easiest way to effect a reduction is of course to use the `reduction` clause. Adding this to an `omp for` or an `omp sections` construct has the following effect:

- OpenMP will make a copy of the reduction variable per thread, initialized to the identity of the reduction operator, for instance 1 for multiplication.
- Each thread will then reduce into its local variable;
- At the end of the loop, the local results are combined, again using the reduction operator, into the global variable.

This is one of those cases where the parallel execution can have a slightly different value from the one that is computed sequentially, because floating point operations are not associative. See HPSC-3.5.5 for more explanation.

If your code can not be easily structure as a reduction, you can realize the above scheme by hand by 'duplicating' the global variable and gather the contributions later. This example presumes three threads, and gives each a location of their own to store the result computed on that thread:

```
double result,local_results[3];
#pragma omp parallel
{
  int num = omp_get_thread_num();
  if (num==0)      local_results[num] = f(x)
  else if (num==1) local_results[num] = g(x)
  else if (num==2) local_results[num] = h(x)
}
result = local_results[0]+local_results[1]+local_results[2]
```

While this code is correct, it may be inefficient because of a phenomemon called *false sharing*. Even though the threads write to separate variables, those variables are likely to be on the same *cacheline* (see HPSC-1.4.1.2 for an explanation). This means that the cores will be wasting a lot of time and bandwidth updating each other's copy of this cacheline.

False sharing can be prevent by giving each thread its own cacheline:

```
double result,local_results[3][8];
#pragma omp parallel
{
  int num = omp_get_thread_num();
  if (num==0)      local_results[num][1] = f(x)
// et cetera
}
```

A more elegant solution gives each thread a true local variable, and uses a critical section to sum these, at the very end:

```
double result = 0;
#pragma omp parallel
{
  double local_result;
  local_result = .....
#pragam omp critical
  result += local_result;
}
```

20.1 Built-in reduction operators

Arithmetic reductions: +, ∗, −, max, min

Logical operator reductions in C: `& && | || ^`

Logical operator reductions in Fortran: `.and. .or. .eqv. .neqv. .iand. .ior. .ieor.`

Exercise 20.2. The maximum and minimum reductions were not added to OpenMP until OpenMP-3.1. Write a parallel loop that computes the maximum and minimum values in an array. Discuss the various options. Do timings to evaluate the speedup that is attained and to find the best option.

20.2 Initial value for reductions

The treatment of initial values in reductions is slightly involved.

```
  x = init_x
#pragma omp parallel for reduction(min:x)
  for (int i=0; i<N; i++)
    x = min(x,data[i]);
```

Each thread does a partial reduction, but its initial value is not the user-supplied `init_x` value, but a value dependent on the operator. In the end, the partial results will then be combined with the user initial value. The initialization values are mostly self-evident, such as zero for addition and one for multiplication. For min and max they are respectively the maximal and minimal representable value of the result type.

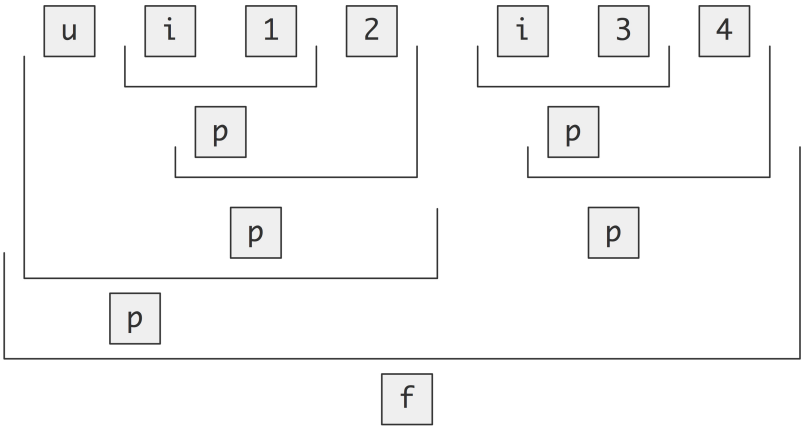


Figure 20.1: Reduction of four items on two threads, taking into account initial values.

Figure 20.1 illustrates this, where `1,2,3,4` are four data items, `i` is the OpenMP initialization, and `u` is the user initialization; each `p` stands for a partial reduction value. The figure is based on execution using two threads.

Exercise 20.3. Write a program to test the fact that the partial results are initialized to the unit of the reduction operator.

20.3 User-defined reductions

With *user-defined reductions*, the programmer specifies the function that does the elementwise comparison. This takes two steps.

1. You need a function of two arguments that returns the result of the comparison. You can do this yourself, but, especially with the C++ standard library, you can use functions such as `std::vector::insert`.
2. Specifying how this function operates on two variables *omp_out* and *omp_in*, corresponding to the partially reduced result and the new operand respectively. The new partial result should be left in `omp_out`.
3. Optionally, you can specify the value to which the reduction should be initialized.

This is the syntax of the definition of the reduction, which can then be used in multiple `reduction` clauses.

```
#pragma omp declare reduction
    ( identifier : typelist : combiner )
    [initializer(initializer-expression)]
```

where:

identifier is a name; this can be overloaded for different types, and redefined in inner scopes.
typelist is a list of types.
combiner is an expression that updates the internal variable *omp_out* as function of itself and *omp_in*.
initializer sets *omp_priv* to the identity of the reduction; this can be an expression or a brace initializer.

For instance, recreating the maximum reduction would look like this:

```
// ireduct.c
int mymax(int r,int n) {
// r is the already reduced value
// n is the new value
  int m;
  if (n>r) {
    m = n;
  } else {
    m = r;
  }
  return m;
}
#pragma omp declare reduction \
  (rwz:int:omp_out=mymax(omp_out,omp_in)) \
  initializer(omp_priv=INT_MIN)
  m = INT_MIN;
#pragma omp parallel for reduction(rwz:m)
  for (int idata=0; idata<ndata; idata++)
    m = mymax(m,data[idata]);
```

For the full source of this example, see section 20.5.2

Exercise 20.4. Write a reduction routine that operates on an array of nonnegative integers, finding the smallest nonzero one. If the array has size zero, or entirely consists of zeros, return `-1`.

Support for *C++ iterators*

```
#pragma omp declare reduction (merge : std::vector<int>
  : omp_out.insert(omp_out.end(), omp_in.begin(), omp_in.end()))
```

20.4 Reductions and floating-point math

The mechanisms that OpenMP uses to make a reduction parallel go against the strict rules for floating point expression evaluation in C; see HPSC-3.6.4. OpenMP ignores this issue: it is the programmer's job to ensure proper rounding behaviour.

20.5 Sources used in this chapter

20.5.1 Listing of code header

20.5.2 Listing of code examples/omp/c/ireduct.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <omp.h>

int mymax(int r,int n) {
  // r is the already reduced value
  // n is the new value
  int m;
  if (n>r) {
    m = n;
  } else {
    m = r;
  }
  printf("combine %d %d : %d\n",r,n,m);
  return m;
}

int main(int argc,char **argv) {

  int m,ndata = 4, data[4] = {2,-3,0,5};

#pragma omp declare reduction \
  (rwz:int:omp_out=mymax(omp_out,omp_in)) \
  initializer(omp_priv=INT_MIN)

  m = INT_MIN;
  for (int idata=0; idata<ndata; idata++)
    m = mymax(m,data[idata]);
  if (m!=5)
    printf("Sequential: wrong reduced value: %d, s/b %d\n",m,2);
  else
    printf("Sequential case succeeded\n");

  m = INT_MIN;
#pragma omp parallel for reduction(rwz:m)
  for (int idata=0; idata<ndata; idata++)
    m = mymax(m,data[idata]);

  if (m!=5)
    printf("Parallel: wrong reduced value: %d, s/b %d\n",m,2);
  else
    printf("Finished\n");

  return 0;
```

```
}
```

Chapter 21

OpenMP topic: Synchronization

In the constructs for declaring parallel regions above, you had little control over in what order threads executed the work they were assigned. This section will discuss *synchronization* constructs: ways of telling threads to bring a certain order to the sequence in which they do things.

- `critical`: a section of code can only be executed by one thread at a time; see 21.2.1.
- `atomic` *Atomic update* of a single memory location. Only certain specified syntax pattterns are supported. This was added in order to be able to use hardware support for atomic updates.
- `barrier`: section 21.1.
- `ordered`: section 17.5.
- locks: section 21.3.
- `flush`: section 24.3.
- `nowait`: section 17.6.

21.1 Barrier

A barrier defines a point in the code where all active threads will stop until all threads have arrived at that point. With this, you can guarantee that certain calculations are finished. For instance, in this code snippet, computation of `y` can not proceed until another thread has computed its value of `x`.

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
  y[mytid] = x[mytid]+x[mytid+1];
}
```

This can be guaranteed with a `barrier` pragma:

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  x[mytid] = some_calculation();
#pragma omp barrier
  y[mytid] = x[mytid]+x[mytid+1];
}
```

Apart from the barrier directive, which inserts an explicit barrier, OpenMP has *implicit barriers* after a load sharing construct. Thus the following code is well defined:

```
#pragma omp parallel
{
#pragma omp for
  for (int mytid=0; mytid<number_of_threads; mytid++)
    x[mytid] = some_calculation();
#pragma omp for
  for (int mytid=0; mytid<number_of_threads-1; mytid++)
    y[mytid] = x[mytid]+x[mytid+1];
}
```

You can also put each parallel loop in a parallel region of its own, but there is some overhead associated with creating and deleting the team of threads in between the regions.

21.1.1 Implicit barriers

At the end of a parallel region the team of threads is dissolved and only the master thread continues. Therefore, there is an *implicit barrier at the end of a parallel region*.

There is some *barrier behaviour* associated with `omp for` loops and other *worksharing constructs* (see section 18.3). For instance, there is an *implicit barrier* at the end of the loop. This barrier behaviour can be cancelled with the `nowait` clause.

You will often see the idiom

```
#pragma omp parallel
{
#pragma omp for nowait
  for (i=0; i<N; i++)
    a[i] = // some expression
#pragma omp for
  for (i=0; i<N; i++)
    b[i] = ..... a[i] .....
```

Here the `nowait` clause implies that threads can start on the second loop while other threads are still working on the first. Since the two loops use the same schedule here, an iteration that uses `a[i]` can indeed rely on it that that value has been computed.

21.2 Mutual exclusion

Sometimes it is necessary to let only one thread execute a piece of code. Such a piece of code is called a *critical section*, and OpenMP has several mechanisms for realizing this.

The most common use of critical sections is to update a variable. Since updating involves reading the old value, and writing back the new, this has the possibility for a *race condition*: another thread reads the current value before the first can update it; the second thread the updates to the wrong value.

Critical sections are an easy way to turn an existing code into a correct parallel code. However, there are disadvantages to this, and sometimes a more drastic rewrite is called for.

21.2.1 `critical` and `atomic`

There are two pragmas for critical sections: `critical` and `atomic`. Both denote *atomic operations* in a technical sense. The first one is general and can contain an arbitrary sequence of instructions; the second one is more limited but has performance advantages.

The typical application of a critical section is to update a variable:

```
#pragma omp parallel
{
  int mytid = omp_get_thread_num();
  double tmp = some_function(mytid);
#pragma omp critical
  sum += tmp;
}
```

Exercise 21.1. Consider a loop where each iteration updates a variable.

```
#pragma omp parallel for shared(result)
  for ( i ) {
      result += some_function_of(i);
  }
```

Discuss qualitatively the difference between:

- turning the update statement into a critical section, versus
- letting the threads accumulate into a private variable `tmp` as above, and summing these after the loop.

Do an Ahmdal-style quantitative analysis of the first case, assuming that you do n iterations on p threads, and each iteration has a critical section that takes a fraction f. Assume the number of iterations n is a multiple of the number of threads p. Also assume the default static distribution of loop iterations over the threads.

A `critical` section works by acquiring a lock, which carries a substantial overhead. Furthermore, if your code has multiple critical sections, they are all mutually exclusive: if a thread is in one critical section, the other ones are all blocked.

On the other hand, the syntax for `atomic` sections is limited to the update of a single memory location, but such sections are not exclusive and they can be more efficient, since they assume that there is a hardware mechanism for making them critical.

The problem with `critical` sections being mutually exclusive can be mitigated by naming them:

```
#pragma omp critical (optional_name_in_parens)
```

21.3 Locks

OpenMP also has the traditional mechanism of a *lock*. A lock is somewhat similar to a critical section: it guarantees that some instructions can only be performed by one process at a time. However, a critical

section is indeed about code; a lock is about data. With a lock you make sure that some data elements can only be touched by one process at a time.

One simple example of the use of locks is generation of a *histogram*. A histogram consists of a number of bins, that get updated depending on some data. Here is the basic structure of such a code:

```
int count[100];
float x = some_function();
int ix = (int)x;
if (ix>=100)
  error();
else
  count[ix]++;
```

It would be possible to guard the last line:

```
#pragma omp critical
  count[ix]++;
```

but that is unnecessarily restrictive. If there are enough bins in the histogram, and if the `some_function` takes enough time, there are unlikely to be conflicting writes. The solution then is to create an array of locks, with one lock for each `count` location.

Create/destroy:

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set and release:

```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

Since the set call is blocking, there is also

```
omp_test_lock();
```

Unsetting a lock needs to be done by the thread that set it.

Lock operations implicitly have a `flush`.

Exercise 21.2. In the following code, one process sets array A and then uses it to update B; the other process sets array B and then uses it to update A. Argue that this code can deadlock. How could you fix this?

```
#pragma omp parallel shared(a, b, nthreads, locka, lockb)
  #pragma omp sections nowait
    {
    #pragma omp section
      {
      omp_set_lock(&locka);
      for (i=0; i<N; i++)
        a[i] = ..
```

```
    omp_set_lock(&lockb);
    for (i=0; i<N; i++)
      b[i] = .. a[i] ..
    omp_unset_lock(&lockb);
    omp_unset_lock(&locka);
    }

  #pragma omp section
    {
    omp_set_lock(&lockb);
    for (i=0; i<N; i++)
      b[i] = ...

    omp_set_lock(&locka);
    for (i=0; i<N; i++)
      a[i] = .. b[i] ..
    omp_unset_lock(&locka);
    omp_unset_lock(&lockb);
    }
  } /* end of sections */
} /* end of parallel region */
```

21.3.1 Nested locks

A lock as explained above can not be locked if it is already locked. A *nested lock* can be locked multiple times by the same thread before being unlocked.

- `omp_init_nest_lock`
- `omp_destroy_nest_lock`
- `omp_set_nest_lock`
- `omp_unset_nest_lock`
- `omp_test_nest_lock`

lock|)

21.4 Example: Fibonacci computation

The *Fibonacci sequence* is recursively defined as

$$F(0) = 1, \qquad F(1) = 1, \qquad F(n) = F(n-1) + F(n-2) \text{ for } n \geq 2.$$

We start by sketching the basic single-threaded solution. The naive code looks like:

```
int main() {
  value = new int[nmax+1];
  value[0] = 1;
  value[1] = 1;
  fib(10);
}
```

```
int fib(int n) {
  int i, j, result;
  if (n>=2) {
    i=fib(n-1); j=fib(n-2);
    value[n] = i+j;
  }
  return value[n];
}
```

Howver, this is inefficienty, since most intermediate values will be computed more than once. We solve this by keeping track of which results are known:

```
  ...
  done = new int[nmax+1];
  for (i=0; i<=nmax; i++)
    done[i] = 0;
  done[0] = 1;
  done[1] = 1;
  ...
int fib(int n) {
  int i, j;
  if (!done[n]) {
    i = fib(n-1); j = fib(n-2);
    value[n] = i+j; done[n] = 1;
  }
  return value[n];
}
```

The OpenMP parallel solution calls for two different ideas. First of all, we parallelize the recursion by using tasks (section 22:

```
int fib(int n) {
  int i, j;
  if (n>=2) {
#pragma omp task shared(i) firstprivate(n)
    i=fib(n-1);
#pragma omp task shared(j) firstprivate(n)
    j=fib(n-2);
#pragma omp taskwait
    value[n] = i+j;
  }
  return value[n];
}
```

This computes the right solution, but, as in the naive single-threaded solution, it recomputes many of the intermediate values.

A naive addition of the `done` array leads to data races, and probably an incorrect solution:

```
int fib(int n) {
  int i, j, result;
  if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
```

```
    i=fib(n-1);
#pragma omp task shared(i) firstprivate(n)
    j=fib(n-2);
#pragma omp taskwait
    value[n] = i+j;
    done[n] = 1;
  }
  return value[n];
}
```

For instance, there is no guarantee that the `done` array is updated later than the `value` array, so a thread can think that `done[n-1]` is true, but `value[n-1]` does not have the right value yet.

One solution to this problem is to use a lock, and make sure that, for a given index `n`, the values `done[n]` and `value[n]` are never touched by more than one thread at a time:

```
int fib(int n)
{
  int i, j;
  omp_set_lock( &(dolock[n]) );
  if (!done[n]) {
#pragma omp task shared(i) firstprivate(n)
    i = fib(n-1);
#pragma omp task shared(j) firstprivate(n)
    j = fib(n-2);
#pragma omp taskwait
    value[n] = i+j;
    done[n] = 1;
  }
  omp_unset_lock( &(dolock[n]) );
  return value[n];
}
```

This solution is correct, optimally efficient in the sense that it does not recompute anything, and it uses tasks to obtain a parallel execution.

However, the efficiency of this solution is only up to a constant. A lock is still being set, even if a value is already computed and therefore will only be read. This can be solved with a complicated use of critical sections, but we will forego this.

21.5 Sources used in this chapter

21.5.1 Listing of code header

Chapter 22

OpenMP topic: Tasks

Tasks are a mechanism that OpenMP uses under the cover: if you specify something as being parallel, OpenMP will create a 'block of work': a section of code plus the data environment in which it occurred. This block is set aside for execution at some later point.

Let's look at a simple example using the `task` directive.

Code	Execution
`x = f();`	the variable `x` gets a value
`#pragma omp task`	a task is created with the current value of `x`
`{ y = g(x); }`	
`z = h();`	the variable `z` gets a value

The thread that executes this code segment creates a task, which will later be executed, probably by a different thread. The exact timing of the execution of the task is up to a *task scheduler*, which operates invisible to the user.

The task mechanism allows you to do things that are hard or impossible with the loop and section constructs. For instance, a *while loop* traversing a *linked list* can be implemented with tasks:

Code	Execution
`p = head_of_list();`	one thread traverses the list
`while (!end_of_list(p)) {`	
`#pragma omp task`	a task is created,
`process( p );`	one for each element
`p = next_element(p);`	the generating thread goes on without waiting
`}`	the tasks are executed while more are being generated.

The way tasks and threads interact is different from the worksharing constructs you've seen so far. Typically, one thread will generate the tasks, adding them to a queue, from which all threads can take and execute them. This leads to the following idiom:

```
#pragma omp parallel
#pragma omp single
{
  ...
#pragma omp task
  { ... }
```

```
  ...
}
```

1. A parallel region creates a team of threads;
2. a single thread then creates the tasks, adding them to a queue that belongs to the team,
3. and all the threads in that team (possibly including the one that generated the tasks)

With tasks it becomes possible to parallelize processes that did not fit the earlier OpenMP constructs. For instance, if a certain operation needs to be applied to all elements of a linked list, you can have one thread go down the list, generating a task for each element of the list.

Another concept that was hard to parallelize earlier is the 'while loop'. This does not fit the requirement for OpenMP parallel loops that the loop bound needs to be known before the loop executes.

Exercise 22.1. Use tasks to find the smallest factor of a large number (using $2999 \cdot 3001$ as test case): generate a task for each trial factor. Start with this code:

```
  int factor=0;
#pragma omp parallel
#pragma omp single
  for (int f=2; f<4000; f++) {
    { // see if `f' is a factor
      if (N%f==0) { // found factor!
        factor = f;
      }
    }
    if (factor>0)
      break;
  }
  if (factor>0)
    printf("Found a factor: %d\n",factor);
```

- Turn the factor finding block into a task.
- Run your program a number of times:
  `for i in `seq 1 1000` ; do ./taskfactor ; done | grep -v 2999`
  Does it find the wrong factor? Why? Try to fix this.
- Once a factor has been found, you should stop generating tasks. Let tasks that should not have been generated, meaning that they test a candidate larger than the factor found, print out a message.

22.1 Task data

Treatment of data in a task is somewhat subtle. The basic problem is that a task gets created at one time, and executed at another. Thus, if shared data is accessed, does the task see the value at creation time or at execution time? In fact, both possibilities make sense depending on the application, so we need to discuss the rules when which possibility applies.

The first rule is that shared data is shared in the task, but private data becomes `firstprivate`. To see the distinction, consider two code fragments. In the first example:

```
int count = 100;
#pragma omp parallel
#pragma omp single
{
  while (count>0) {
#pragma omp task
    {
      int countcopy = count;
      if (count==50) {
        sleep(1);
        printf("%d,%d\n",count,countcopy);
      } // end if
    }   // end task
    count--;
  }     // end while
}       // end single
```

the variable `count` is declared outside the parallel region and is therefore shared. When the print statement is executed, all tasks will have been generated, and so `count` will be zero. Thus, the output will likely be `0,50`.

In the second example:

```
#pragma omp parallel
#pragma omp single
{
  int count = 100;
  while (count>0) {
#pragma omp task
    {
      int countcopy = count;
      if (count==50) {
        sleep(1);
        printf("%d,%d\n",count,countcopy);
      } // end if
    }   // end task
    count--;
  }     // end while
}       // end single
```

the `count` variable is private to the thread creating the tasks, and so it will be `firstprivate` in the task, preserving the value that was current when the task was created.

22.2 Task synchronization

Even though the above segment looks like a linear set of statements, it is impossible to say when the code after the `task` directive will be executed. This means that the following code is incorrect:

```
  x = f();
#pragma omp task
  { y = g(x); }
  z = h(y);
```

Explanation: when the statement computing `z` is executed, the task computing y has only been scheduled; it has not necessarily been executed yet.

In order to have a guarantee that a task is finished, you need the `taskwait` directive. The following creates two tasks, which can be executed in parallel, and then waits for the results:

Code	Execution
`x = f();`	the variable `x` gets a value

`#pragma omp task`
`{ y1 = g1(x); }`
`#pragma omp task`
`{ y2 = g2(x); }`
	two tasks are created with the current value of `x`

`#pragma omp taskwait`	the thread waits until the tasks are finished
`z = h(y1)+h(y2);`	the variable `z` is computed using the task results

The `task` pragma is followed by a structured block. Each time the structured block is encountered, a new task is generated. On the other hand `taskwait` is a standalone directive; the code that follows is just code, it is not a structured block belonging to the directive.

Another aspect of the distinction between generating tasks and executing them: usually the tasks are generated by one thread, but executed by many threads. Thus, the typical idiom is:

```
#pragma omp parallel
#pragma omp single
{
  // code that generates tasks
}
```

This makes it possible to execute loops in parallel that do not have the right kind of iteration structure for a `omp parallel for`. As an example, you could traverse and process a linked list:

```
#pragma omp parallel
#pragma omp single
{
  while (!tail(p)) {
    p = p->next();
#pragma omp task
    process(p)
  }
#pragma omp taskwait
}
```

One task traverses the linked list creating an independent task for each element in the list. These tasks are then executed in parallel; their assignment to threads is done by the task scheduler.

You can indicate task dependencies in several ways:

1. Using the 'task wait' directive you can explicitly indicate the *join* of the *forked* tasks. The instruction after the wait directive will therefore be dependent on the spawned tasks.
2. The `taskgroup` directive, followed by a structured block, ensures completion of all tasks created in the block, even if recursively created.

3. Each OpenMP task can have a `depend` clause, indicating what *data dependency* of the task. By indicating what data is produced or absorbed by the tasks, the scheduler can construct the dependency graph for you.

Another mechanism for dealing with tasks is the `taskgroup`: a task group is a code block that can contain `task` directives; all these tasks need to be finished before any statement after the block is executed.

A task group is somewhat similar to having a `taskwait` directive after the block. The big difference is that that `taskwait` directive does not wait for tasks that are recursively generated, while a `taskgroup` does.

22.3 Task dependencies

It is possible to put a partial ordering on tasks through use of the `depend` clause. For example, in

```
#pragma omp task
  x = f()
#pragma omp task
  y = g(x)
```

it is conceivable that the second task is executed before the first, possibly leading to an incorrect result. This is remedied by specifying:

```
#pragma omp task depend(out:x)
  x = f()
#pragma omp task depend(in:x)
  y = g(x)
```

Exercise 22.2. Consider the following code:

```
for i in [1:N]:
    x[0,i] = some_function_of(i)
    x[i,0] = some_function_of(i)

for i in [1:N]:
    for j in [1:N]:
        x[i,j] = x[i-1,j]+x[i,j-1]
```

- Observe that the second loop nest is not amenable to OpenMP loop parallelism.
- Can you think of a way to realize the computation with OpenMP loop parallelism? Hint: you need to rewrite the code so that the same operations are done in a different order.
- Use tasks with dependencies to make this code parallel without any rewriting: the only change is to add OpenMP directives.

Tasks dependencies are used to indicated how two uses of one data item relate to each other. Since either use can be a read or a write, there are four types of dependencies.

RaW (Read after Write) The second task reads an item that the first task writes. The second task has to be executed after the first:

```
... omp task depend(OUT:x)
  foo(x)
... omp task depend( IN:x)
  foo(x)
```

WaR (Write after Read) The first task reads and item, and the second task overwrites it. The second task has to be executed second to prevent overwriting the initial value:

```
... omp task depend( IN:x)
  foo(x)
... omp task depend(OUT:x)
  foo(x)
```

WaW (Write after Write) Both tasks set the same variable. Since the variable can be used by an intermediate task, the two writes have to be executed in this order.

```
... omp task depend(OUT:x)
  foo(x)
... omp task depend(OUT:x)
  foo(x)
```

RaR (Read after Read) Both tasks read a variable. Since neither tasks has an 'out' declaration, they can run in either order.

```
... omp task depend(IN:x)
  foo(x)
... omp task depend(IN:x)
  foo(x)
```

22.4 More

22.4.1 Scheduling points

Normally, a task stays tied to the thread that first executes it. However, at a *task scheduling point* the thread may switch to the execution of another task created by the same team.

- There is a scheduling point after explicit task creation. This means that, in the above examples, the thread creating the tasks can also participate in executing them.
- There is a scheduling point at `taskwait` and `taskyield`.

On the other hand a task created with them `untied` clause on the task pragma is never tied to one thread. This means that after suspension at a scheduling point any thread can resume execution of the task. If you do this, beware that the value of a thread-id does not stay fixed. Also locks become a problem.

Example: if a thread is waiting for a lock, with a scheduling point it can suspend the task and work on another task.

```
while (!omp_test_lock(lock))
#pragma omp taskyield
  ;
```

22.4.2 Task cancelling

It is possible (in *OpenMP version 4*) to cancel tasks. This is useful when tasks are used to perform a search: the task that finds the result first can cancel any outstanding search tasks.

The directive `cancel` takes an argument of the surrounding construct (`parallel, for, sections, taskgroup`) in which the tasks are cancelled.

Exercise 22.3. Modify the prime finding example.

22.5 Examples

22.5.1 Fibonacci

As an example of the use of tasks, consider computing an array of Fibonacci values:

```
// taskgroup0.c
for (int i=2; i<N; i++)
  {
    fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
  }
```

For the full source of this example, see section 22.6.2

If you simply turn each calculation into a task, results will be unpredictable (confirm this!) since tasks can be executed in any sequence. To solve this, we put dependencies on the tasks:

```
// taskgroup2.c
  for (int i=2; i<N; i++)
#pragma omp task \
depend(out:fibo_values[i]) \
depend(in:fibo_values[i-1],fibo_values[i-2])
    {
      fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
    }
```

For the full source of this example, see section 22.6.3

22.5.2 Binomial coefficients

Exercise 22.4. An array of binomial coefficients can be computed as follows:

```
// binomial1.c
for (int row=1; row<=n; row++)
  for (int col=1; col<=row; col++)
    if (row==1 || col==1 || col==row)
        array[row][col] = 1;
    else
        array[row][col] = array[row-1][col-1] + array[row-1][col];
```

For the full source of this example, see section 22.6.4

Putting a single task group around the double loop, and use `depend` clauses to make the execution satisfy the proper dependencies.

22.5.3 Tree traversal

OpenMP tasks are a great way of handling trees.

22.5.3.1 Post-order traversal

In *post-order tree traversal* you visit the subtrees before visiting the root. This is the traversal that you use to find summary information about a tree, for instance the sum of all nodes, and the sums of nodes of all subtrees:

for *all children c* **do**
    compute the sum s_c

$s \leftarrow \sum_c s_c$

Another example is matrix factorization:

$$S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}$$

where the two inverses A_{11}^{-1}, A_{22}^{-1} can be computed indepedently and recursively.

22.5.3.2 Pre-order traversal

If a property needs to propagate from the root to all subtrees and nodes, you can use *pre-order tree traversal*:

Update node value s

for *all children c* **do**
    update c with the new value s

22.6 Sources used in this chapter

22.6.1 Listing of code header

22.6.2 Listing of code examples/omp/c/taskgroup0.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include <string.h>

int main(int argc,char **argv) {
  int N = 1000;
  if (argc>1) {
    if (!strcmp(argv[1],"-h")) {
      printf("usage: %s [nnn]\n",argv[0]);
      return 0;
    }
    N = atoi(argv[1]);
    if (N>99) {
      printf("Sorry, this overflows: setting N=99\n");
      N = 99;
    }
  }

  long int *fibo_values = (long int*)malloc(N*sizeof(long int));

  fibo_values[0] = 1; fibo_values[1] = 1;
  {
    for (int i=2; i<N; i++)
      {
        fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
      }
  }
  printf("F(%d) = %ld\n",N,fibo_values[N-1]);

  return 0;
}
```

22.6.3 Listing of code examples/omp/c/taskgroup2.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#include <string.h>

int main(int argc,char **argv) {
  int N = 1000;
  if (argc>1) {
    if (!strcmp(argv[1],"-h")) {
      printf("usage: %s [nnn]\n",argv[0]);
```

```
      return 0;
    }
    N = atoi(argv[1]);
    if (N>99) {
      printf("Sorry, this overflows: setting N=99\n");
      N = 99;
    }
  }

  long int *fibo_values = (long int*)malloc(N*sizeof(long int));

  fibo_values[0] = 1; fibo_values[1] = 1;
#pragma omp parallel
#pragma omp single
#pragma omp taskgroup
  {
    for (int i=2; i<N; i++)
#pragma omp task \
  depend(out:fibo_values[i]) \
  depend(in:fibo_values[i-1],fibo_values[i-2])
      {
        fibo_values[i] = fibo_values[i-1]+fibo_values[i-2];
      }
  }
  printf("F(%d) = %ld\n",N,fibo_values[N-1]);

  return 0;
}
```

22.6.4 Listing of code code/omp/c/binomial1.c

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

#define NMAX 30000/sizeof(int)
long int array[NMAX][NMAX];


long int coeff(int row,int col) {
  if (row==1) return 1;
  else if (col==1) return 1;
  else if (col==row) return 1;
  else return array[row-1][col-1]+array[row-1][col];
};

int main(int argc,char **argv) {

  int n;
  if (argc>1)
    n = atoi(argv[1]);
  else n = 100;
  if (n>NMAX) {
```

```
    printf("N too large: keep under %d\n",NMAX);
    return 0; }

  for (int row=1; row<=n; row++)
    for (int col=1; col<=row; col++)
      if (row==1 || col==1 || col==row)
array[row][col] = 1;
      else
array[row][col] = array[row-1][col-1] + array[row-1][col];
  //printf("set (%d,%d) = %ld\n",row,col,array[row][col]);

  for (int col=1; col<=n; col++) {
    long int binom = array[n][col];
    printf("%ld ",binom);
  } printf("\n");

  return 0;
}
```

Chapter 23

OpenMP topic: Affinity

23.1 OpenMP thread affinity control

The matter of thread affinity becomes important on *multi-socket nodes*; see the example in section 23.2.

Thread placement can be controlled with two environment variables:

- the environment variable `OMP_PROC_BIND` describes how threads are bound to *OpenMP places*; while
- the variable `OMP_PLACES` describes these places in terms of the available hardware.
- When you're experimenting with these variables it is a good idea to set `OMP_DISPLAY_ENV` to true, so that OpenMP will print out at runtime how it has interpreted your specification. The examples in the following sections will display this output.

23.1.1 Thread binding

The variable `OMP_PLACES` defines a series of places to which the threads are assigned.

Example: if you have two sockets and you define

```
OMP_PLACES=sockets
```

then

- thread 0 goes to socket 0,
- thread 1 goes to socket 1,
- thread 2 goes to socket 0 again,
- and so on.

On the other hand, if the two sockets have a total of sixteen cores and you define

```
OMP_PLACES=cores
OMP_PROC_BIND=close
```

then

- thread 0 goes to core 0, which is on socket 0,
- thread 1 goes to core 1, which is on socket 0,
- thread 2 goes to core 2, which is on socket 0,
- and so on, until thread 7 goes to core 7 on socket 0, and

- thread 8 goes to core 8, which is on socket 1,
- et cetera.

The value `OMP_PROC_BIND=close` means that the assignment goes successively through the available places. The variable `OMP_PROC_BIND` can also be set to `spread`, which spreads the threads over the places. With

```
OMP_PLACES=cores
OMP_PROC_BIND=spread
```

you find that

- thread 0 goes to core 0, which is on socket 0,
- thread 1 goes to core 8, which is on socket 1,
- thread 2 goes to core 1, which is on socket 0,
- thread 3 goes to core 9, which is on socket 1,
- and so on, until thread 14 goes to core 7 on socket 0, and
- thread 15 goes to core 15, which is on socket 1.

So you see that `OMP_PLACES=cores` and `OMP_PROC_BIND=spread` very similar to `OMP_PLACES=sockets`. The difference is that the latter choice does not bind a thread to a specific core, so the operating system can move threads about, and it can put more than one thread on the same core, even if there is another core still unused.

The value `OMP_PROC_BIND=master` puts the threads in the same place as the master of the team. This is convenient if you create teams recursively. In that case you would use the `proc_bind` clause rather than the environment variable, set to `spread` for the initial team, and to `master` for the recursively created team.

23.1.2 Effects of thread binding

Let's consider two example program. First we consider the program for computing π, which is purely compute-bound.

#threads	close/cores	spread/sockets	spread/cores
1	0.359	0.354	0.353
2	0.177	0.177	0.177
4	0.088	0.088	0.088
6	0.059	0.059	0.059
8	0.044	0.044	0.044
12	0.029	0.045	0.029
16	0.022	0.050	0.022

We see pretty much perfect speedup for the `OMP_PLACES=cores` strategy; with `OMP_PLACES=sockets` we probably get occasional collisions where two threads wind up on the same core.

Next we take a program for computing the time evolution of the *heat equation*:

$$t = 0, 1, 2, \ldots : \forall_i : x_i^{(t+1)} = 2x_i^{(t)} - x_{i-1}^{(t)} - x_{i+1}^{(t)}$$

This is a bandwidth-bound operation because the amount of computation per data item is low.

#threads	close/cores	spread/sockets	spread/cores
1	2.88	2.89	2.88
2	1.71	1.41	1.42
4	1.11	0.74	0.74
6	1.09	0.57	0.57
8	1.12	0.57	0.53
12	0.72	0.53	0.52
16	0.52	0.61	0.53

Again we see that `OMP_PLACES=sockets` gives worse performance for high core counts, probably because of threads winding up on the same core. The thing to observe in this example is that with 6 or 8 cores the `OMP_PROC_BIND=spread` strategy gives twice the performance of `OMP_PROC_BIND=close`.

The reason for this is that a single socket does not have enough bandwidth for all eight cores on the socket. Therefore, dividing the eight threads over two sockets gives each thread a higher available bandwidth than putting all threads on one socket.

23.1.3 Place definition

There are three predefined values for the `OMP_PLACES` variable: `sockets, cores, threads`. You have already seen the first two; the `threads` value becomes relevant on processors that have hardware threads. In that case, `OMP_PLACES=cores` does not tie a thread to a specific hardware thread, leading again to possible collisions as in the above example. Setting `OMP_PLACES=threads` ties each OpenMP thread to a specific hardware thread.

There is also a very general syntax for defining places that uses a

```
location:number:stride
```

syntax. Examples:

- `OMP_PLACES="{0:8:1},{8:8:1}"`
  is equivalent to `sockets` on a two-socket design with eight cores per socket: it defines two places, each having eight consecutive cores. The threads are then places alternating between the two places, but not further specified inside the place.
- The setting `cores` is equivalent to
  `OMP_PLACES="{0},{1},{2},...,{15}"`
- On a four-socket design, the specification
  `OMP_PLACES="{0:4:8}:4:1"`
  states that the place `0,8,16,24` needs to be repeated four times, with a stride of one. In other words, thread 0 winds up on core 0 of some socket, the thread 1 winds up on core 1 of some socket, et cetera.

23.1.4 Binding possibilities

Values for `OMP_PROC_BIND` are: `false, true, master, close, spread`.

- false: set no binding
- true: lock threads to a core
- master: collocate threads with the master thread
- close: place threads close to the master in the places list
- spread: spread out threads as much as possible

This effect can be made local by giving the `proc_bind` clause in the `parallel` directive.

A safe default setting is

```
export OMP_PROC_BIND=true
```

which prevents the operating system from *migrating a thread*. This prevents many scaling problems.

Good examples of *thread placement* on the *Intel Knight's Landing*: https://software.intel.com/en-us/articles/process-and-thread-affinity-for-intel-xeon-phi-processors-x200

As an example, consider a code where two threads write to a shared location.

```
// sharing.c
#pragma omp parallel
  { // not a parallel for: just a bunch of reps
    for (int j = 0; j < reps; j++) {
#pragma omp for schedule(static,1)
      for (int i = 0; i < N; i++){
#pragma omp atomic
        a++;
      }

    }
  }
```

For the full source of this example, see section 23.4.2

There is now a big difference in runtime depending on how close the threads are. We test this on a processor with both cores and hyperthreads. First we bind the OpenMP threads to the cores:

```
OMP_NUM_THREADS=2 OMP_PLACES=cores OMP_PROC_BIND=close ./sharing
run time = 4752.231836usec
sum = 80000000.0
```

Next we force the OpenMP threads to bind to hyperthreads inside one core:

```
OMP_PLACES=threads OMP_PROC_BIND=close ./sharing
run time = 941.970110usec
sum = 80000000.0
```

Of course in this example the inner loop is pretty much meaningless and parallelism does not speed up anything:

```
OMP_NUM_THREADS=1 OMP_PLACES=cores OMP_PROC_BIND=close ./sharing
run time = 806.669950usec
sum = 80000000.0
```

However, we see that the two-thread result is almost as fast, meaning that there is very little parallelization overhead.

23.2 First-touch

The affinity issue shows up in the *first-touch* phenomemon. Memory allocated with `malloc` and like routines is not actually allocated; that only happens when data is written to it. In light of this, consider the following OpenMP code:

```
double *x = (double*) malloc(N*sizeof(double));

for (i=0; i<N; i++)
  x[i] = 0;

#pragma omp parallel for
for (i=0; i<N; i++)
  .... something with x[i] ...
```

Since the initialization loop is not parallel it is executed by the master thread, making all the memory associated with the socket of that thread. Subsequent access by the other socket will then access data from memory not attached to that socket.

Exercise 23.1. Finish the following fragment and run it with first all the cores of one socket, then all cores of both sockets. (If you know how to do explicit placement, you can also try fewer cores.)

```
  for (int i=0; i<nlocal+2; i++)
    in[i] = 1.;
  for (int i=0; i<nlocal; i++)
    out[i] = 0.;

  for (int step=0; step<nsteps; step++) {
#pragma omp parallel for schedule(static)
    for (int i=0; i<nlocal; i++) {
      out[i] = ( in[i]+in[i+1]+in[i+2] )/3.;
    }
#pragma omp parallel for schedule(static)
    for (int i=0; i<nlocal; i++)
      in[i+1] = out[i];
    in[0] = 0; in[nlocal+1] = 1;
  }
```

Exercise 23.2. How do the OpenMP dynamic schedules relate to this?

C++ `valarray` does initialization, so it will allocate memory on thread 0.

You could move pages with `move_pages`.

By regarding affinity, in effect you are adopting an SPMD style of programming. You could make this explicit by having each thread allocate its part of the arrays separately, and storing a private pointer as `threadprivate` [18]. However, this makes it impossible for threads to access each other's parts of the distributed array, so this is only suitable for total *data parallel* or *embarrassingly parallel* applications.

23.3 Affinity control outside OpenMP

There are various utilities to control process and thread placement.

Process placement can be controlled on the Operating system level by `numactl` (the TACC utility `tacc_affinity` is a wrapper around this) on Linux (also `taskset`); Windows `start/affinity`.

Corresponding system calls: `pbing` on Solaris, `sched_setaffinity` on Linux, `SetThreadAffinityMask` on Windows.

Corresponding environment variables: `SUNW_MP_PROCBIND` on Solaris, `KMP_AFFINITY` on Intel.

The *Intel compiler* has an environment variable for affinity control:

```
export KMP_AFFINITY=verbose,scatter
```

values: `none,scatter,compact`

For *gcc*:

```
export GOMP_CPU_AFFINITY=0,8,1,9
```

For the *Sun compiler*:

```
SUNW_MP_PROCBIND
```

23.4 Sources used in this chapter

23.4.1 Listing of code header

23.4.2 Listing of code examples/omp/c/sharing.c

```
#include <stdio.h>
#include <omp.h>


int main() {

  int i,j;
  int reps = 1000;
  int N = 8*10000;

  double start, stop, delta;
  double a;

#pragma omp parallel
  a = 0;

  start = omp_get_wtime();
#pragma omp parallel
  { // not a parallel for: just a bunch of reps
    for (int j = 0; j < reps; j++) {
#pragma omp for schedule(static,1)
      for (int i = 0; i < N; i++){
#pragma omp atomic
a++;
      }

    }
  }
  stop  = omp_get_wtime();
  delta = ((double)(stop - start))/reps;
  printf("run time = %fusec\n", 1.0e6*delta);

  printf("sum = %.1f\n", a);

  return 0;
}
```

Chapter 24

OpenMP topic: Memory model

24.1 Thread synchronization

Let's do a *producer-consumer* model[1]. This can be implemented with sections, where one section, the producer, sets a flag when data is available, and the other, the consumer, waits until the flag is set.

```
#pragma omp parallel sections
{
  // the producer
  #pragma omp section
  {
    ... do some producing work ...
    flag = 1;
  }
  // the consumer
  #pragma omp section
  {
    while (flag==0) { }
    ... do some consuming work ...
  }
}
```

One reason this doesn't work, is that the compiler will see that the flag is never used in the producing section, and that is never changed in the consuming section, so it may optimize these statements, to the point of optimizing them away.

The producer then needs to do:

```
... do some producing work ...
#pragma omp flush
#pragma atomic write
  flag = 1;
#pragma omp flush(flag)
```

and the consumer does:

1. This example is from Intel's excellent OMP course by Tim Mattson

```
#pragma omp flush(flag)
while (flag==0) {
  #pragma omp flush(flag)
}
#pragma omp flush
```

This code strictly speaking has a *race condition* on the `flag` variable.

The solution is to make this an *atomic operation* and use an `atomic` pragma here: the producer has

```
#pragma atomic write
  flag = 1;
```

and the consumer:

```
while (1) {
  #pragma omp flush(flag)
  #pragma omp atomic read
    flag_read = flag
  if (flag_read==1) break;
}
```

24.2 Data races

OpenMP, being based on shared memory, has a potential for *race conditions*. These happen when two threads access the same data item. The problem with race conditions is that programmer convenience runs counter to efficient execution. For this reason, OpenMP simply does not allow some things that would be desirable.

For a simple example:

```
// race.c
#pragma omp parallel for shared(counter)
  for (int i=0; i<count; i++)
    counter++;
  printf("Counter should be %d, is %d\n",
         count,counter);
```

For the full source of this example, see section 24.4.2

The basic rule about multiple-thread access of a single data item is:

> Any memory location that is *written* by one thread, can not be *read* by another thread in the same parallel region, if no synchronization is done.

To start with that last clause: any workshare construct ends with an *implicit barrier*, so data written before that barrier can safely be read after it.

As an illustration of a possible problem:

```
c = d = 0;
#pragma omp sections
{
```

```
#pragma omp section
  { a = 1; c = b; }
#pragma omp section
  { b = 1; d = a; }
}
```

Under any reasonable interpretation of parallel execution, the possible values for `c,d` are $1,1$ $0,1$ or $1,0$. This is known as *sequential consistency*: the parallel outcome is consistent with a sequential execution that interleaves the parallel computations, respecting their local statement orderings. (See also HPSC-2.6.1.6.)

However, without synchronization, threads are allowed to maintain a value for a variable locally that is not the same as the stored value. In this example, that means that the thread executing the first section need not write its value of `a` to memory, and likewise `b` in the second thread, so $0,0$ is in fact a possible outcome.

In order to resolve multiple accesses:

1. Thread one reads the variable.
2. Thread one flushes the variable.
3. Thread two flushes the variable.
4. Thread two reads the variable.

24.3 Relaxed memory model

`flush`

- There is an implicit flush of all variables at the start and end of a *parallel region*.
- There is a flush at each barrier, whether explicit or implicit, such as at the end of a *work sharing*.
- At entry and exit of a *critical section*
- When a *lock* is set or unset.

24.4 Sources used in this chapter

24.4.1 Listing of code header

24.4.2 Listing of code examples/omp/c/race.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include <omp.h>

int main(int argc,char **argv) {

  int count = 100000;
  int counter = 0;

#pragma omp parallel for shared(counter)
  for (int i=0; i<count; i++)
    counter++;
  printf("Counter should be %d, is %d\n",
 count,counter);


  return 0;
}
```

Chapter 25

OpenMP topic: SIMD processing

You can declare a loop to be executable with *vector instructions* with `simd`

The `simd` pragma has the following clauses:

- `safelen(n)`: limits the number of iterations in a SIMD chunk. Presumably useful if you combine `parallel for simd`.
- `linear`: lists variables that have a linear relation to the iteration parameter.
- `aligned`: specifies alignment of variables.

If your SIMD loop includes a function call, you can declare that the function can be turned into vector instructions with `declare simd`

If a loop is both multi-threadable and vectorizable, you can combine directives as `pragma omp parallel for simd`.

Compilers can be made to report whether a loop was vectorized:

```
LOOP BEGIN at simdf.c(61,15)
   remark #15301: OpenMP SIMD LOOP WAS VECTORIZED
LOOP END
```

with such options as `-Qvec-report=3` for the Intel compiler.

Performance improvements of these directives need not be immediately obvious. In cases where the operation is bandwidth-limited, using `simd` parallelism may give the same or worse performance as thread parallelism.

The following function can be vectorized:

```
// tools.c
#pragma omp declare simd
double cs(double x1,double x2,double y1,double y2) {
  double
    inprod = x1*x2+y1*y2,
    xnorm = sqrt(x1*x1 + x2*x2),
    ynorm = sqrt(y1*y1 + y2*y2);
  return inprod / (xnorm*ynorm);
}
#pragma omp declare simd uniform(x1,x2,y1,y2) linear(i)
```

```
double csa(double *x1,double *x2,double *y1,double *y2, int i) {
  double
    inprod = x1[i]*x2[i]+y1[i]*y2[i],
    xnorm = sqrt(x1[i]*x1[i] + x2[i]*x2[i]),
    ynorm = sqrt(y1[i]*y1[i] + y2[i]*y2[i]);
  return inprod / (xnorm*ynorm);
}
```

For the full source of this example, see section 25.1.2

Compiling this the regular way

```
# parameter 1(x1): %xmm0
# parameter 2(x2): %xmm1
# parameter 3(y1): %xmm2
# parameter 4(y2): %xmm3

movaps    %xmm0, %xmm5    5 <- x1
movaps    %xmm2, %xmm4    4 <- y1
mulsd     %xmm1, %xmm5    5 <- 5 * x2 = x1 * x2
mulsd     %xmm3, %xmm4    4 <- 4 * y2 = y1 * y2
mulsd     %xmm0, %xmm0    0 <- 0 * 0 = x1 * x1
mulsd     %xmm1, %xmm1    1 <- 1 * 1 = x2 * x2
addsd     %xmm4, %xmm5    5 <- 5 + 4 = x1*x2 + y1*y2
mulsd     %xmm2, %xmm2    2 <- 2 * 2 = y1 * y1
mulsd     %xmm3, %xmm3    3 <- 3 * 3 = y2 * y2
addsd     %xmm1, %xmm0    0 <- 0 + 1 = x1*x1 + x2*x2
addsd     %xmm3, %xmm2    2 <- 2 + 3 = y1*y1 + y2*y2
sqrtsd    %xmm0, %xmm0    0 <- sqrt(0) = sqrt( x1*x1 + x2*x2 )
sqrtsd    %xmm2, %xmm2    2 <- sqrt(2) = sqrt( y1*y1 + y2*y2 )
```

which uses the scalar instruction `mulsd`: multiply scalar double precision.

With a `declare simd` directive:

```
movaps    %xmm0, %xmm7
movaps    %xmm2, %xmm4
mulpd     %xmm1, %xmm7
mulpd     %xmm3, %xmm4
```

which uses the vector instruction `mulpd`: multiply packed double precision, operating on 128-bit *SSE2 registers*.

Compiling for the *Intel Knight's Landing* gives more complicated code:

```
# parameter 1(x1): %xmm0
# parameter 2(x2): %xmm1
# parameter 3(y1): %xmm2
# parameter 4(y2): %xmm3

vmulpd    %xmm3, %xmm2, %xmm4                          4 <- y1*y2
```

```
vmulpd    %xmm1, %xmm1, %xmm5                              5 <- x1*x2
vbroadcastsd .L_2il0floatpacket.0(%rip), %zmm21
movl      $3, %eax                                         set accumulator EAX
vbroadcastsd .L_2il0floatpacket.5(%rip), %zmm24
kmovw     %eax, %k3                                        set mask k3
vmulpd    %xmm3, %xmm3, %xmm6                              6 <-y1*y1 (stall)
vfmadd231pd %xmm0, %xmm1, %xmm4                            4 <- 4 + x1*x2 (no reuse!)
vfmadd213pd %xmm5, %xmm0, %xmm0                            0 <- 0 + 0*5 = x1 + x1*(x1*x2)
vmovaps   %zmm21, %zmm18                                   #25.26 c7
vmovapd   %zmm0, %zmm3{%k3}{z}                             #25.26 c11
vfmadd213pd %xmm6, %xmm2, %xmm2                            #24.29 c13
vpcmpgtq  %zmm0, %zmm21, %k1{%k3}                          #25.26 c13
vscalefpd .L_2il0floatpacket.1(%rip){1to8}, %zmm0, %zmm3{%k1} #25.26 c15
vmovaps   %zmm4, %zmm26                                    #25.26 c15
vmovapd   %zmm2, %zmm7{%k3}{z}                             #25.26 c17
vpcmpgtq  %zmm2, %zmm21, %k2{%k3}                          #25.26 c17
vscalefpd .L_2il0floatpacket.1(%rip){1to8}, %zmm2, %zmm7{%k2} #25.26 c19
vrsqrt28pd %zmm3, %zmm16{%k3}{z}                           #25.26 c19
vpxorq    %zmm4, %zmm4, %zmm26{%k3}                        #25.26 c19
vrsqrt28pd %zmm7, %zmm20{%k3}{z}                           #25.26 c21
vmulpd    {rn-sae}, %zmm3, %zmm16, %zmm19{%k3}{z}          #25.26 c27 stall 2
vscalefpd .L_2il0floatpacket.2(%rip){1to8}, %zmm16, %zmm17{%k3}{z} #25.26 c27
vmulpd    {rn-sae}, %zmm7, %zmm20, %zmm23{%k3}{z}          #25.26 c29
vscalefpd .L_2il0floatpacket.2(%rip){1to8}, %zmm20, %zmm22{%k3}{z} #25.26 c29
vfnmadd231pd {rn-sae}, %zmm17, %zmm19, %zmm18{%k3}         #25.26 c33 stall 1
vfnmadd231pd {rn-sae}, %zmm22, %zmm23, %zmm21{%k3}         #25.26 c35
vfmadd231pd {rn-sae}, %zmm19, %zmm18, %zmm19{%k3}          #25.26 c39 stall 1
vfmadd231pd {rn-sae}, %zmm23, %zmm21, %zmm23{%k3}          #25.26 c41
vfmadd213pd {rn-sae}, %zmm17, %zmm17, %zmm18{%k3}          #25.26 c45 stall 1
vfnmadd231pd {rn-sae}, %zmm19, %zmm19, %zmm3{%k3}          #25.26 c47
vfmadd213pd {rn-sae}, %zmm22, %zmm22, %zmm21{%k3}          #25.26 c51 stall 1
vfnmadd231pd {rn-sae}, %zmm23, %zmm23, %zmm7{%k3}          #25.26 c53
vfmadd213pd %zmm19, %zmm18, %zmm3{%k3}                     #25.26 c57 stall 1
vfmadd213pd %zmm23, %zmm21, %zmm7{%k3}                     #25.26 c59
vscalefpd .L_2il0floatpacket.3(%rip){1to8}, %zmm3, %zmm3{%k1} #25.26 c63 stall 1
vscalefpd .L_2il0floatpacket.3(%rip){1to8}, %zmm7, %zmm7{%k2} #25.26 c65
vfixupimmpd $112, .L_2il0floatpacket.4(%rip){1to8}, %zmm0, %zmm3{%k3} #25.26 c65
vfixupimmpd $112, .L_2il0floatpacket.4(%rip){1to8}, %zmm2, %zmm7{%k3} #25.26 c67
vmulpd    %xmm7, %xmm3, %xmm0                              #25.26 c71
vmovaps   %zmm0, %zmm27                                    #25.26 c79
vmovaps   %zmm0, %zmm25                                    #25.26 c79
vrcp28pd  {sae}, %zmm0, %zmm27{%k3}                        #25.26 c81
vfnmadd213pd {rn-sae}, %zmm24, %zmm27, %zmm25{%k3}         #25.26 c89 stall 3
```

```
vfmadd213pd {rn-sae}, %zmm27, %zmm25, %zmm27{%k3}       #25.26 c95 stall 2
vcmppd    $8, %zmm26, %zmm27, %k1{%k3}                  #25.26 c101 stall 2
vmulpd    %zmm27, %zmm4, %zmm1{%k3}{z}                  #25.26 c101
kortestw  %k1, %k1                                      #25.26 c103
je        ..B1.3        # Prob 25%                      #25.26 c105
vdivpd    %zmm0, %zmm4, %zmm1{%k1}                      #25.26 c3 stall 1
vmovaps   %xmm1, %xmm0                                  #25.26 c77
ret                                                     #25.26 c79
```

```
#pragma omp declare simd uniform(op1) linear(k) notinbranch
  double SqrtMul(double *op1, double op2, int k) {
    return (sqrt(op1[k]) * sqrt(op2));
  }
```

25.1 Sources used in this chapter

25.1.1 Listing of code header

25.1.2 Listing of code code/omp/c/simd/tools.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#pragma omp declare simd
double cs(double x1,double x2,double y1,double y2) {
  double
    inprod = x1*x2+y1*y2,
    xnorm = sqrt(x1*x1 + x2*x2),
    ynorm = sqrt(y1*y1 + y2*y2);
  return inprod / (xnorm*ynorm);
}

#pragma omp declare simd uniform(x1,x2,y1,y2) linear(i)
double csa(double *x1,double *x2,double *y1,double *y2, int i) {
  double
    inprod = x1[i]*x2[i]+y1[i]*y2[i],
    xnorm = sqrt(x1[i]*x1[i] + x2[i]*x2[i]),
    ynorm = sqrt(y1[i]*y1[i] + y2[i]*y2[i]);
  return inprod / (xnorm*ynorm);
}
```

Chapter 26

OpenMP remaining topics

26.1 Runtime functions and internal control variables

OpenMP has a number of settings that can be set through *environment variables*, and both queried and set through *library routines*. These settings are called *Internal Control Variables (ICVs)*: an OpenMP implementation behaves as if there is an internal variable storing this setting.

The runtime functions are:

- Counting threads and cores: `omp_set_num_threads`, `omp_get_num_threads`, `omp_get_max_threads`, `omp_get_num_procs` ; see section 15.5.
- Querying the current thread: `omp_get_thread_num`, `omp_in_parallel`
- `omp_set_dynamic`
- `omp_get_dynamic`
- `omp_set_nested`
- `omp_get_nested`
- `omp_get_wtime`
- `omp_get_wtick`
- `omp_set_schedule`
- `omp_get_schedule`
- `omp_set_max_active_levels`
- `omp_get_max_active_levels`
- `omp_get_thread_limit`
- `omp_get_level`
- `omp_get_active_level`
- `omp_get_ancestor_thread_num`
- `omp_get_team_size`

Here are the OpenMP *environment variables*:

- `OMP_CANCELLATION` Set whether cancellation is activated
- `OMP_DISPLAY_ENV` Show OpenMP version and environment variables
- `OMP_DEFAULT_DEVICE` Set the device used in target regions
- `OMP_DYNAMIC` Dynamic adjustment of threads
- `OMP_MAX_ACTIVE_LEVELS` Set the maximum number of nested parallel regions

- `OMP_MAX_TASK_PRIORITY` Set the maximum task priority value
- `OMP_NESTED` Nested parallel regions
- `OMP_NUM_THREADS` Specifies the number of threads to use
- `OMP_PROC_BIND` Whether theads may be moved between CPUs
- `OMP_PLACES` Specifies on which CPUs the theads should be placed
- `OMP_STACKSIZE` Set default thread stack size
- `OMP_SCHEDULE` How threads are scheduled
- `OMP_THREAD_LIMIT` Set the maximum number of threads
- `OMP_WAIT_POLICY` How waiting threads are handled; ICV `wait-policy-var`. Values: `ACTIVE` for keeping threads spinning, `PASSIVE` for possibly yielding the processor when threads are waiting.

There are 4 ICVs that behave as if each thread has its own copy of them. The default is implementation-defined unless otherwise noted.

- It may be possible to adjust dynamically the number of threads for a parallel region. Variable: `OMP_DYNAMIC`; routines: `omp_set_dynamic`, `omp_get_dynamic`.
- If a code contains *nested parallel regions*, the inner regions may create new teams, or they may be executed by the single thread that encounters them. Variable: `OMP_NESTED`; routines `omp_set_nested`, `omp_get_nested`. Allowed values are `TRUE` and `FALSE`; the default is false.
- The number of threads used for an encountered parallel region can be controlled. Variable: `OMP_NUM_THREADS`; routines `omp_set_num_threads`, `omp_get_max_threads`.
- The schedule for a parallel loop can be set. Variable: `OMP_SCHEDULE`; routines `omp_set_schedule`, `omp_get_schedule`.

Nonobvious syntax:

```
export OMP_SCHEDULE="static,100"
```

Other settings:

- `omp_get_num_threads`: query the number of threads active at the current place in the code; this can be lower than what was set with `omp_set_num_threads`. For a meaningful answer, this should be done in a parallel region.
- `omp_get_thread_num`
- `omp_in_parallel`: test if you are in a parallel region (see for instance section 16).
- `omp_get_num_procs`: query the physical number of cores available.

Other environment variables:

- `OMP_STACKSIZE` controls the amount of space that is allocated as per-thread *stack*; the space for private variables.
- `OMP_WAIT_POLICY` determines the behaviour of threads that wait, for instance for *critical section*:
  - `ACTIVE` puts the thread in a *spin-lock*, where it actively checks whether it can continue;
  - `PASSIVE` puts the thread to sleep until the Operating System (OS) wakes it up.

  The 'active' strategy uses CPU while the thread is waiting; on the other hand, activating it after the wait is instantaneous. With the 'passive' strategy, the thread does not use any CPU while waiting, but activating it again is expensive. Thus, the passive strategy only makes sense if threads will be waiting for a (relatively) long time.

- `OMP_PROC_BIND` with values `TRUE` and `FALSE` can bind threads to a processor. On the one hand, doing so can minimize data movement; on the other hand, it may increase load imbalance.

26.2 Timing

OpenMP has a wall clock timer routine `omp_get_wtime`

```
double omp_get_wtime(void);
```

The starting point is arbitrary and is different for each program run; however, in one run it is identical for all threads. This timer has a resolution given by `omp_get_wtick`.

Exercise 26.1. Use the timing routines to demonstrate speedup from using multiple threads.

- Write a code segment that takes a measurable amount of time, that is, it should take a multiple of the tick time.
- Write a parallel loop and measure the speedup. You can for instance do this

```
for (int use_threads=1; use_threads<=nthreads; use_threads++) {
#pragma omp parallel for num_threads(use_threads)
    for (int i=0; i<nthreads; i++) {
        .....
    }
    if (use_threads==1)
      time1 = tend-tstart;
    else // compute speedup
```

- In order to prevent the compiler from optimizing your loop away, let the body compute a result and use a reduction to preserve these results.

26.3 Thread safety

With OpenMP it is relatively easy to take existing code and make it parallel by introducing parallel sections. If you're careful to declare the appropriate variables shared and private, this may work fine. However, your code may include calls to library routines that include a *race condition*; such code is said not to be *thread-safe*.

For example a routine

```
static int isave;
int next_one() {
 int i = isave;
 isave += 1;
 return i;
}

...
for ( .... ) {
  int ivalue = next_one();
}
```

has a clear race condition, as the iterations of the loop may get different `next_one` values, as they are supposed to, or not. This can be solved by using an `critical` pragma for the `next_one` call; another solution is to use an `threadprivate` declaration for `isave`. This is for instance the right solution if the `next_one` routine implements a *random number generator*.

26.4 Performance and tuning

The performance of an OpenMP code can be influenced by the following.

Amdahl effects Your code needs to have enough parts that are parallel (see HPSC-2.2.3). Sequential parts may be sped up by having them executed redundantly on each thread, since that keeps data locally.

Dynamism Creating a thread team takes time. In practice, a team is not created and deleted for each parallel region, but creating teams of different sizes, or recursize thread creation, may introduce overhead.

Load imbalance Even if your program is parallel, you need to worry about load balance. In the case of a parallel loop you can set the `schedule` clause to *`dynamic`*, which evens out the work, but may cause increased communication.

Communication Cache coherence causes communication. Threads should, as much as possible, refer to their own data.

- Threads are likely to read from each other's data. That is largely unavoidable.
- Threads writing to each other's data should be avoided: it may require synchronization, and it causes coherence traffic.
- If threads can migrate, data that was local at one time is no longer local after migration.
- Reading data from one socket that was allocated on another socket is inefficient; see section 23.2.

Affinity Both data and execution threads can be bound to a specific locale to some extent. Using local data is more efficient than remote data, so you want to use local data, and minimize the extent to which data or execution can move.

- See the above points about phenomena that cause communication.
- Section 23.1.1 describes how you can specify the binding of threads to places. There can, but does not need, to be an effect on affinity. For instance, if an OpenMP thread can migrate between hardware threads, cached data will stay local. Leaving an OpenMP thread completely free to migrate can be advantageous for load balancing, but you should only do that if data affinity is of lesser importance.
- Static loop schedules have a higher chance of using data that has affinity with the place of execution, but they are worse for load balancing. On the other hand, the `nowait` clause can aleviate some of the problems with static loop schedules.

Binding You can choose to put OpenMP threads close together or to spread them apart. Having them close together makes sense if they use lots of shared data. Spreading them apart may increase bandwidth. (See the examples in section 23.1.2.)

Synchronization Barriers are a form of synchronization. They are expensive by themselves, and they expose load imbalance. Implicit barriers happen at the end of worksharing constructs; they can be removed with *`nowait`*.

Critical sections imply a loss of parallelism, but they are also slow as they are realized through *operating system* functions. These are often quite costly, taking many thousands of cycles. Critical sections should be used only if the parallel work far outweighs it.

26.5 Accelerators

In OpenMP 4.0 there is support for offloading work to an *accelerator* or *co-processor*:

```
#pragma omp target [clauses]
```

with clauses such as

- `data`: place data
- `update`: make data consistent between host and device

26.6 Sources used in this chapter

26.6.1 Listing of code header

Chapter 27

OpenMP Review

27.1 Concepts review

27.1.1 Basic concepts

- process / thread / thread team
- threads / cores / tasks
- directives / library functions / environment variables

27.1.2 Parallel regions

execution by a team

27.1.3 Work sharing

- loop / sections / single / workshare
- implied barrier
- loop scheduling, reduction
- sections
- single vs master
- (F) workshare

27.1.4 Data scope

- shared vs private, C vs F
- loop variables and reduction variables
- default declaration
- firstprivate, lastprivate

27.1.5 Synchronization

- barriers, implied and explicit
- nowait
- critical sections
- locks, difference with critical

27.1.6 Tasks

- generation vs execution
- dependencies

27.2 Review questions

27.2.1 Directives

What do the following program output?

```
int main() {
  printf("procs %d\n",
    omp_get_num_procs());
  printf("threads %d\n",
    omp_get_num_threads());
  printf("num %d\n",
    omp_get_thread_num());
  return 0;
}
```

```
int main() {
#pragma omp parallel
  {
  printf("procs %d\n",
    omp_get_num_procs());
  printf("threads %d\n",
    omp_get_num_threads());
  printf("num %d\n",
    omp_get_thread_num());
  }
  return 0;
}
```

```
Program main
  use omp_lib
  print *,"Procs:",&
    omp_get_num_procs()
  print *,"Threads:",&
    omp_get_num_threads()
  print *,"Num:",&
    omp_get_thread_num()
End Program
```

```
Program main
  use omp_lib
!$OMP parallel
  print *,"Procs:",&
    omp_get_num_procs()
  print *,"Threads:",&
    omp_get_num_threads()
  print *,"Num:",&
    omp_get_thread_num()
!$OMP end parallel
End Program
```

27.2.2 Parallelism

Can the following loops be parallelized? If so, how? (Assume that all arrays are already filled in, and that there are no out-of-bounds errors.)

```
// variant #1
for (i=0; i<N; i++) {
  x[i] = a[i]+b[i+1];
  a[i] = 2*x[i] + c[i+1];
}
```

```
// variant #2
for (i=0; i<N; i++) {
  x[i] = a[i]+b[i+1];
  a[i] = 2*x[i+1] + c[i+1];
}
```

```
// variant #3
for (i=1; i<N; i++) {
  x[i] = a[i]+b[i+1];
  a[i] = 2*x[i-1] + c[i+1];
}
```

```
// variant #4
for (i=1; i<N; i++) {
  x[i] = a[i]+b[i+1];
  a[i+1] = 2*x[i-1] + c[i+1];
}
```

```
! variant #1
do i=1,N
  x(i) = a(i)+b(i+1)
  a(i) = 2*x(i) + c(i+1)
end do
```

```
! variant #2
do i=1,N
  x(i) = a(i)+b(i+1)
  a(i) = 2*x(i+1) + c(i+1)
end do
```

```
! variant #3
do i=2,N
  x(i) = a(i)+b(i+1)
  a(i) = 2*x(i-1) + c(i+1)
end do
```

```
! variant #3
do i=2,N
  x(i) = a(i)+b(i+1)
  a(i+1) = 2*x(i-1) + c(i+1)
end do
```

27.2.3 Data and synchronization

27.2.3.1

What is the output of the following fragments? Assume that there are four threads.

```
// variant #1
int nt;
#pragma omp parallel
  {
  nt = omp_get_thread_num();
  printf("thread number: %d\n",nt);
  }
```

```
// variant #2
int nt;
#pragma omp parallel private(nt)
  {
  nt = omp_get_thread_num();
  printf("thread number: %d\n",nt);
  }
```

```
// variant #3
int nt;
#pragma omp parallel
  {
#pragma omp single
    {
    nt = omp_get_thread_num();
    printf("thread number: %d\n",nt);
    }
  }
```

```
! variant #1
  integer nt
!$OMP parallel
  nt = omp_get_thread_num()
  print *,"thread number:",nt
!$OMP end parallel
```

```
! variant #2
  integer nt
!$OMP parallel private(nt)
  nt = omp_get_thread_num()
  print *,"thread number:",nt
!$OMP end parallel
```

```
! variant #3
  integer nt
!$OMP parallel
```

```
// variant #4
int nt;
#pragma omp parallel
  {
#pragma omp master
    {
    nt = omp_get_thread_num();
    printf("thread number: %d\n",nt);
    }
  }
```

```
// variant #5
int nt;
#pragma omp parallel
  {
#pragma omp critical
    {
    nt = omp_get_thread_num();
    printf("thread number: %d\n",nt);
    }
  }
```

```
!$OMP single
    nt = omp_get_thread_num()
    print *,"thread number:",nt
!$OMP end single
!$OMP end parallel
```

```
! variant #4
  integer nt
!$OMP parallel
!$OMP master
    nt = omp_get_thread_num()
    print *,"thread number:",nt
!$OMP end master
!$OMP end parallel
```

```
! variant #5
  integer nt
!$OMP parallel
!$OMP critical
    nt = omp_get_thread_num()
    print *,"thread number:",nt
!$OMP end critical
!$OMP end parallel
```

27.2.3.2

The following is an attempt to parallelize a serial code. Assume that all variables and arrays are defined. What errors and potential problems do you see in this code? How would you fix them?

```
#pragma omp parallel
{
  x = f();
  #pragma omp for
  for (i=0; i<N; i++)
    y[i] = g(x,i);
  z = h(y);
}
```

```
!$OMP parallel
  x = f()
!$OMP do
  do i=1,N
    y(i) = g(x,i)
  end do
!$OMP end do
  z = h(y)
!$OMP end parallel
```

27.2.3.3

Assume two threads. What does the following program output?

```
int a;
#pragma omp parallel private(a) {
  ...
  a = 0;
  #pragma omp for
  for (int i = 0; i < 10; i++)
  {
    #pragma omp atomic
    a++; }
  #pragma omp single
    printf("a=%e\n",a);
}
```

27.2.4 Reductions

27.2.4.1

Is the following code correct? Is it efficient? If not, can you improve it?

```
#pragma omp parallel shared(r)
{
  int x;
  x = f(omp_get_thread_num());
#pragma omp critical
  r += f(x);
}
```

27.2.4.2

Compare two fragments:

```
// variant 1
#pragma omp parallel reduction(+:s)
#pragma omp for
  for (i=0; i<N; i++)
    s += f(i);
```

```
// variant 2
#pragma omp parallel
#pragma omp for reduction(+:s)
  for (i=0; i<N; i++)
    s += f(i);
```

```
! variant 1
!$OMP parallel reduction(+:s)
!$OMP do
  do i=1,N
    s += f(i);
  end do
```

```
!$OMP end do
!$OMP end parallel
```

```
! variant 2
!$OMP parallel
!$OMP do reduction(+:s)
  do i=1,N
    s += f(i);
  end do
!$OMP end do
!$OMP end parallel
```

Do they compute the same thing?

27.2.5 Barriers

Are the following two code fragments well defined?

```
#pragma omp parallel
{
#pragma omp for
for (mytid=0; mytid<nthreads; mytid++)
  x[mytid] = some_calculation();
#pragma omp for
for (mytid=0; mytid<nthreads-1; mytid
    ++)
  y[mytid] = x[mytid]+x[mytid+1];
}
```

```
#pragma omp parallel
{
#pragma omp for
for (mytid=0; mytid<nthreads; mytid++)
  x[mytid] = some_calculation();
#pragma omp for nowait
for (mytid=0; mytid<nthreads-1; mytid
    ++)
  y[mytid] = x[mytid]+x[mytid+1];
}
```

27.2.6 Data scope

The following program is supposed to initialize as many rows of the array as there are threads.

```
int main() {
  int i,icount,iarray[100][100];
  icount = -1;
#pragma omp parallel private(i)
  {
#pragma omp critical
    { icount++; }
    for (i=0; i<100; i++)
      iarray[icount][i] = 1;
  }
  return 0;
}
```

```
Program main
  integer :: i,icount,iarray(100,100)
  icount = 0
!$OMP parallel private(i)
!$OMP critical
    icount = icount + 1
!$OMP end critical
    do i=1,100
      iarray(icount,i) = 1
    end do
!$OMP end parallel
End program
```

Describe the behaviour of the program, with argumentation,

- as given;
- if you add a clause `private(icount)` to the *parallel* directive;
- if you add a clause *firstprivate*`(icount)`.

What do you think of this solution:

```
#pragma omp parallel private(i) shared
    (icount)
  {
#pragma omp critical
    { icount++;
      for (i=0; i<100; i++)
        iarray[icount][i] = 1;
    }
  }
  return 0;
}
```

```
!$OMP parallel private(i) shared(
    icount)
!$OMP critical
    icount = icount+1
    do i=1,100
      iarray(icount,i) = 1
    end do
!$OMP critical
!$OMP end parallel
```

27.2.7 Tasks

Fix two things in the following example:

```
#pragma omp parallel
#pragma omp single
{
  int x,y,z;
#pragma omp task
  x = f();
#pragma omp task
  y = g();
#pragma omp task
  z = h();
  printf("sum=%d\n",x+y+z);
}
```

```
  integer :: x,y,z
!$OMP parallel
!$OMP single

!$OMP task
  x = f()
!$OMP end task

!$OMP task
  y = g()
!$OMP end task

!$OMP task
  z = h()
!$OMP end task

  print *,"sum=",x+y+z
!$OMP end single
!$OMP end parallel
```

27.2.8 Scheduling

Compare these two fragments. Do they compute the same result? What can you say about their efficiency?

```
#pragma omp parallel
#pragma omp single
  {
    for (i=0; i<N; i++) {
    #pragma omp task
      x[i] = f(i)
    }
    #pragma omp taskwait
  }
```

```
#pragma omp parallel
#pragma omp for schedule(dynamic)
  {
    for (i=0; i<N; i++) {
      x[i] = f(i)
    }
  }
```

How would you make the second loop more efficient? Can you do something similar for the first loop?

27.3 Sources used in this chapter

27.3.1 Listing of code header

PART III

PETSC

Chapter 28

PETSc basics

28.1 What is PETSc and why?

PETSc is a library with a great many uses, but for now let's say that it's primarily a library for dealing with the sort of linear algebra that comes from discretized Partial Diffential Equations (PDEs). On a single processor, the basics of such computations can be coded out by a grad student during a semester course in numerical analysis, but on large scale issues get much more complicated and a library becomes indispensible.

PETSc's prime justification is then that it helps you realize scientific computations at large scales, meaning large problem sizes on large numbers of processors.

There are two points to emphasize here:

- Linear algebra with dense matrices is relatively simple to formulate. For sparse matrices the amount of logistics in dealing with nonzero patterns increases greatly. PETSc does most of that for you.
- Linear algebra on a single processor, even a multicore one, is managable; distributed memory parallelism is much harder, and distributed memory sparse linear algebra operations are doubly so. Using PETSc will save you many, many, Many! hours of coding over developing everything yourself from scratch.

Remark 19 *The PETSc library has hundreds of routines. In this chapter and the next few we will only touch on a basic subset of these. The full list of man pages can be found at* `https://www.mcs.anl.gov/petsc/petsc-current/docs/manualpages/singleindex.html`*. Each man page comes with links to related routines, as well as (usually) example codes for that routine.*

28.1.1 What is in PETSc?

The routines in PETSc (of which there are hundreds) can roughly be divided in these classes:

- Basic linear algebra tools: dense and sparse matrices, both sequential and parallel, their construction and simple operations.
- Solvers for linear systems, and to a lesser extent nonlinear systems; also time-stepping methods.
- Profiling and tracing: after a successful run, timing for various routines can be given. In case of failure, there are traceback and memory tracing facilities.

28.1.2 Programming model

PETSc, being based on MPI, uses the SPMD programming model (section 2.1), where all processes execute the same executable. Even more than in regular MPI codes, this makes sense here, since most PETSc objects are collectively created on some communicator, often `MPI_COMM_WORLD`. With the object-oriented design (section 28.1.3) this means that a PETSc program almost looks like a sequential program.

```
MatMult(A,x,y);      // y <- Ax
VecCopy(y,res);      // r <- y
VecAXPY(res,-1.,b);  // r <- r - b
```

This is sometimes called *sequential semantics*.

28.1.3 Design philosophy

PETSc has an object-oriented design, even though it is written in C. There are classes of objects, such `Mat` for matrices and `Vec` for Vectors, but there is also the `KSP` (for "Krylov SPace solver") class of linear system solvers, and `PetscViewer` for outputting matrices and vectors to screen or file.

Part of the object-oriented design is the polymorphism of objects: after you have created a `Mat` matrix as sparse or dense, all methods such as MatMult (for the matrix-vector product) take the same arguments: the matrix, and an input and output vector.

This design where the programmer manipulates a 'handle' also means that the internal of the object, the actual storage of the elements, is hidden from the programmer. This hiding goes so far that even filling in elements is not done directly but through function calls:

```
VecSetValue(i,j,v,mode)
MatSetValue(i,j,v,mode)
MatSetValues(ni,is,nj,js,v,mode)
```

28.1.4 Language support

28.1.4.1 C/C++

PETSc is implemented in C, so there is a natural interface to C. There is no separate C++ interface.

28.1.4.2 Fortran

A *Fortran90* interface exists. The *Fortran77* interface is only of interest for historical reasons.

To use Fortran, include both a module and a cpp header file:

```
#include "petsc/finclude/petscXXX.h"
use petscXXX
```

(here `XXX` stands for one of the PETSc types, but including `petsc.h` and using `use petsc` gives inclusion of the whole library.)

Variables can be declared with their type (`Vec`, `Mat`, `KSP` et cetera), but internally they are Fortran `Type` objects so they can be declared as such.

Example:

```
#include "petsc/finclude/petscvec.h"
use petscvec
Vec b
type(tVec) x
```

The output arguments of many query routines are optional in PETSc. While in C a generic `NULL` can be passed, Fortran has type-specific nulls, such as `PETSC_NULL_INTEGER`, `PETSC_NULL_OBJECT`.

28.1.4.3 Python

A *python* interface was written by Lisandro Dalcin, and requires separate installation, based on already defined `PETSC_DIR` and `PETSC_ARCH` variables. This can be downloaded at https://bitbucket.org/petsc/petsc4py/src/master/, with documentation at https://www.mcs.anl.gov/petsc/petsc4py-current/docs/.

28.1.5 Documentation

PETSc comes with a manual in pdf form and web pages with the documentation for every routine. The starting point is the web page https://www.mcs.anl.gov/petsc/documentation/index.html.

There is also a mailing list with excellent support for questions and bug reports.

> *TACC note.* For questions specific to using PETSc on TACC resources, submit tickets to the *TACC* or *XSEDE portal*.

28.2 Basics of running a PETSc program

28.2.1 Compilation

A PETSc compilation needs a number of include and library paths, probably too many to specify interactively. The easiest solution is to create a makefile:

```
include ${PETSC_DIR}/lib/petsc/conf/variables
include ${PETSC_DIR}/lib/petsc/conf/rules
program : program.o
        ${CLINKER} -o $@ $^ ${PETSC_LIB}
```

The two include lines provide the compilation rule and the library variable.

If you want to write your own compile rule, there is an example makefile `$PETSC_DIR/share/petsc/Makefile.user` you can take for inspiration. Invoked without arguments it prints out the relevant variables:

```
[c:246] make -f ! $PETSC_DIR/share/petsc/Makefile.user
CC=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpicc
CXX=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpicxx
FC=/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/bin/mpif90
CFLAGS=-Wall -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas -fstack-protector -Qunu
CXXFLAGS=-Wall -Wwrite-strings -Wno-strict-aliasing -Wno-unknown-pragmas -fstack-protector -fv
FFLAGS=-m64 -g
CPPFLAGS=-I/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/include -I/Users/eij
LDFLAGS=-L/Users/eijkhout/Installation/petsc/petsc-3.13/macx-clang-debug/lib -Wl,-rpath,/Users
LDLIBS=-lpetsc -lm
```

You can use these rules:

```
% : %.F90
        $(LINK.F) -o $@ $^ $(LDLIBS)
%.o: %.F90
        $(COMPILE.F) $(OUTPUT_OPTION) $<
% : %.cxx
        $(LINK.cc) -o $@ $^ $(LDLIBS)
%.o: %.cxx
        $(COMPILE.cc) $(OUTPUT_OPTION) $<

## example link rule:
# app : a.o b.o c.o
#       $(LINK.F) -o $@ $^ $(LDLIBS)
```

(The `PETSC_CC_INCLUDES` variable contains all paths for compilation of C programs; correspondingly there is `PETSC_FC_INCLUDES` for Fortran source.)

If don't want to include those configuration files, you can find out the include options by:

```
cd $PETSC_DIR
make getincludedirs
```

and copying the results into your compilation script.

The build process assumes that variables `PETSC_DIR` and `PETSC_ARCH` have been set. These depend on your local installation. Usually there will be one installation with debug settings and one with production settings. Develop your code with the former: it will do memory and bound checking. Then recompile and run your code with the optimized production installation.

TACC note. On TACC clusters, a petsc installation is loaded by commands such as

```
module load petsc/3.11
```

Use `module avail petsc` to see what configurations exist. The basic versions are

```
# development
module load petsc/3.11-debug
# production
module load petsc/3.11
```

Figure 28.1 `PetscInitialize`

```
C:
PetscErrorCode  PetscInitialize
   (int *argc,char ***args,const char file[],const char help[])

Input Parameters:
argc - count of number of command line arguments
args - the command line arguments
file - [optional] PETSc database file.
help - [optional] Help message to print, use NULL for no message

Fortran:
call PetscInitialize(file,ierr)

Input parameters:
ierr - error return code
file - [optional] PETSc database file,
    use PETSC_NULL_CHARACTER to not check for code specific file.
```

Other installations are real versus complex, or 64bit integers instead of the default 32. The command

```
module spider petsc
```

tells you all the available petsc versions. The listed modules have a naming convention such as `petsc/3.11-i64debug` where the 3.11 is the PETSc release (minor patches are not included in this version; TACC aims to install only the latest patch, but generally several versions are available), and `i64debug` describes the debug version of the installation with 64bit integers.

28.2.2 Running

PETSc programs use MPI for parallelism, so they are started like any other MPI program:

```
mpirun -np 5 -machinefile mf \
    your_petsc_program option1 option2 option3
```

TACC note. On TACC clusters, use `ibrun`.

28.2.3 Initialization and finalization

PETSc has an call that initializes both PETSc and MPI, so normally you would replace `MPI_Init` by `PetscInitialize` (figure 28.1). Unlike with MPI, you do not want to use a `NULL` value for the `argc,argv` arguments, since PETSc makes extensive use of commandline options; see section 34.3.

```
// init.c
ierr = PetscInitialize(&argc,&argv,(char*)0,help); CHKERRQ(ierr);
int flag;
MPI_Initialized(&flag);
if (flag)
  printf("MPI was initialized by PETSc\n");
else
  printf("MPI not yet initialized\n");
```

For the full source of this example, see section 28.4.2

There are two further arguments to `PetscInitialize`:

1. the name of an options database file; and
2. a help string, that is displayed if you run your program with the `-h` option.

Fortran note.

- The Fortran version has no arguments for commandline options; it also doesn't take a help string.
- If no help string is passed, give `PETSC_NULL_CHARACTER` as argument.
- If your main program is in C, but some of your PETSc calls are in Fortran files, it is necessary to call `PetscInitializeFortran` after `PetscInitialize`.

```
// init.F90
call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
CHKERRA(ierr)
call MPI_Initialized(flag,ierr)
CHKERRA(ierr)
if (flag) then
   print *,"MPI was initialized by PETSc"
```

For the full source of this example, see section 28.4.3

Python note. The following works if you don't need commandline options.

```
from petsc4py import PETSc
```

To pass commandline arguments to PETSc, do:

```
import sys
from petsc4py import init
init(sys.argv)
from petsc4py import PETSc
```

After initialization, you can use `MPI_COMM_WORLD` or `PETSC_COMM_WORLD` (which is created by `MPI_Comm_dup` and used internally by PETSc):

```
MPI_Comm comm = PETSC_COMM_WORLD;
MPI_Comm_rank(comm,&mytid);
MPI_Comm_size(comm,&ntids);
```

Python note.

```
comm = PETSc.COMM_WORLD
nprocs = comm.getSize(self)
procno = comm.getRank(self)
```

The corresponding call to replace `MPI_Finalize` is `PetscFinalize`. You can elegantly capture and return the error code by the idiom

```
return PetscFinalize();
```

at the end of your main program.

28.3 PETSc installation

PETSc has a large number of installation options. These can roughly be divided into:

1. Options to describe the environment in which PETSc is being installed, such as the names of the compilers or the location of the MPI library;
2. Options to specify the type of PETSc installation: real versus complex, 32 versus 64-bit integers, et cetera;
3. Options to specify additional packages to download.

For an existing installation, you can find the options used, and other aspects of the build history, in the `configure.log` / `make.log` files:

```
$PETSC_DIR/$PETSC_ARCH/lib/petsc/conf/configure.log
$PETSC_DIR/$PETSC_ARCH/lib/petsc/conf/make.log
```

28.3.1 Debug

For any set of options, you will typically make two installations: one with `-with-debugging=yes` and once `no`. See section 34.1.1 for more detail.

28.3.2 Environment options

Compilers, compiler options, MPI.

While it is possible to specify *-download_mpich*, this should only be done on machines that you are certain do not already have an MPI library, such as your personal laptop. Supercomputer clusters are likely to have an optimized MPI library, and letting PETSc download its own will lead to degraded performance.

28.3.3 Variants

- Scalars: the option *-with-scalar-type* has values `real`, `complex`; *-with-precision* has values `single`, `double`, `__float128`, `__fp16`.

28.3.4 External packages

PETSc can extend its functionality through external packages such as *mumps*, *Hypre*, *fftw*. These can be specified in two ways:

1. Referring to an installation already on your system:
   `--with-hdf5-include=${TACC_HDF5_INC}`
   `--with-hf5_lib=${TACC_HDF5_LIB}`
2. By letting petsc download and install them itself:
   `--with-parmetis=1 --download-parmetis=1`

Remark 20 *There are two packages that PETSc is capable of downloading and install, but that you may want to avoid:*

- `fblaslapack`*: this gives you BLAS/LAPACK through the Fortran 'reference implementation'. If you have an optimized version, such as Intel's* mkl *available, this will give much higher performance.*
- `mpich`*: this installs a MPI implementation, which may be required for your laptop. However, supercomputer clusters will already have an MPI implementation that uses the high-speed network. PETSc's downloaded version does not do that. Again, finding and using the already installed software may greatly improve your performance.*

28.4 Sources used in this chapter

28.4.1 Listing of code header

28.4.2 Listing of code examples/petsc/c/init.c

```
#include <stdlib.h>
#include <stdio.h>

#include <petscsys.h>

int main(int argc,char **argv)
{
  PetscErrorCode ierr;

  char help[] = "\nInit example.\n\n";
  ierr = PetscInitialize(&argc,&argv,(char*)0,help); CHKERRQ(ierr);
  int flag;
  MPI_Initialized(&flag);
  if (flag)
    printf("MPI was initialized by PETSc\n");
  else
    printf("MPI not yet initialized\n");

  return PetscFinalize();
}
```

28.4.3 Listing of code examples/petsc/f/init.F90

```
Program PetscInit

#include <petsc/finclude/petscsys.h>
  use petsc
  implicit none

  logical :: flag
  PetscErrorCode ierr;
  character*80 :: help = "\nInit example.\n\n";
  call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
  CHKERRA(ierr)
  call MPI_Initialized(flag,ierr)
  CHKERRA(ierr)
  if (flag) then
     print *,"MPI was initialized by PETSc"
  else
     print *,"MPI not yet initialized"
  end if
  call PetscFinalize(ierr); CHKERRQ(ierr);

End Program PetscInit
```

Chapter 29

PETSc objects

29.1 Distributed objects

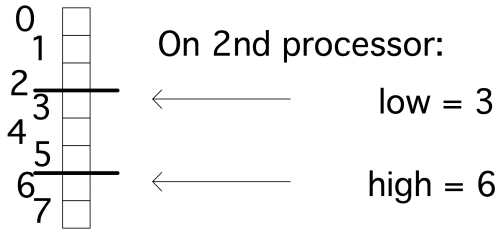
PETSc is based on the SPMD model, and all its objects act like they exist in parallel, spread out over all the processes. Therefore, prior to discussing specific objects in detail, we briefly discuss how PETSc treats distributed objects.

For a matrix or vector you need to specify the size. This can be done two ways:

- you specify the global size and PETSc distributes the object over the processes, or
- you specify on each process the local size

If you specify both the global size and the local sizes, PETSc will check for consistency.

For example, if you have a vector of N components, or a matrix of N rows, and you have P processes, each process will receive N/P components or rows if P divides evenly in N. If P does not divide evenly, the excess is spread over the processes.



The way the distribution is done is by contiguous blocks: with 10 processes and 1000 components in a vector, process 0 gets the range $0 \cdots 99$, process 1 gets $1 \cdots 199$, et cetera. This simple scheme suffices for many cases, but PETSc has facilities for more sophisticated load balancing.

29.1.1 Support for distributions

Once an object has been created and distributed, you do not need to remember the size or the distribution yourself: you can query these with calls such as `VecGetSize`, `VecGetLocalSize`.

The corresponding matrix routines `MatGetSize`, `MatGetLocalSize` give both information for the distributions in i and j direction, which can be independent. Since a matrix is distributed by rows, `MatGetOwnershipRange` only gives a row range.

Figure 29.1 **PetscSplitOwnership**

```
PetscSplitOwnership

Synopsis
#include "petscsys.h"
PetscErrorCode  PetscSplitOwnership
   (MPI_Comm comm,PetscInt *n,PetscInt *N)

Collective (if n or N is PETSC_DECIDE)

Input Parameters
comm - MPI communicator that shares the object being divided
n - local length (or PETSC_DECIDE to have it set)
N - global length (or PETSC_DECIDE)
```

```
// split.c
N = 100; n = PETSC_DECIDE;
PetscSplitOwnership(comm,&n,&N);
PetscPrintf(comm,"Global %d, local %d\n",N,n);

N = PETSC_DECIDE; n = 10;
PetscSplitOwnership(comm,&n,&N);
PetscPrintf(comm,"Global %d, local %d\n",N,n);
```

For the full source of this example, see section 29.8.2

While PETSc objects are implemented using local memory on each process, conceptually they act like global objects, with a global indexing scheme. Thus, each process can query which elements out of the global object are stored locally. For vectors, the relevant routine is `VecGetOwnershipRange`, which returns two parameters, *low* and *high*, respectively the first element index stored, and one-more-than-the-last index stored.

This gives the idiom:

```
VecGetOwnershipRange(myvector,&low,&high);
for (int myidx=low; myidx<high; myidx++)
  // do something at index myidx
```

These conversions between local and global size can also be done explicitly, using the `PetscSplitOwnership` (figure 29.1) routine. This routine takes two parameter, for the local and global size, and whichever one is initialized to `PETSC_DECIDE` gets computed from the other.

29.2 Scalars

Unlike programming languages that explicitly distinguish between single and double precision numbers, PETSc has only a single scalar type: `PetscScalar`. The precision of this is determined at installation time. In fact, a `PetscScalar` can even be a complex number if the installation specified that the scalar type is complex.

Even in applications that use complex numbers there can be quantities that are real: for instance, the norm of a complex vector is a real number. For that reason, PETSc also has the type `PetscReal`. There is also an explicit `PetscComplex`.

Furthermore, there is

```
#define PETSC_BINARY_INT_SIZE    (32/8)
#define PETSC_BINARY_FLOAT_SIZE  (32/8)
#define PETSC_BINARY_CHAR_SIZE   (8/8)
#define PETSC_BINARY_SHORT_SIZE  (16/8)
#define PETSC_BINARY_DOUBLE_SIZE (64/8)
#define PETSC_BINARY_SCALAR_SIZE sizeof(PetscScalar)
```

29.2.1 Integers

Integers in PETSc are likewise of a size determined at installation time: `PetscInt` can be 32 or 64 bits. Furthermore, there is a `PetscErrorCode` type for catching the return code of PETSc routines.

For compatibility with other packages there are two more integer types:

- `PetscBLASInt` is the integer type used by the *Basic Linear Algebra Subprograms (BLAS) / Linear Algebra Package (LAPACK)* library. This is 32-bits if the `-download-blas-lapack` option is used, but it can be 64-bit if *MKL* is used. The routine `PetscBLASIntCast` casts a `PetscInt` to `PetscBLASInt`, or returns `PETSC_ERR_ARG_OUTOFRANGE` if it is too large.
- `PetscMPIInt` is the integer type of the MPI library, which is always 32-bits. The routine `PetscMPIIntCast` casts a `PetscInt` to `PetscMPIInt`, or returns `PETSC_ERR_ARG_OUTOFRANGE` if it is too large.

Many external packages do not support 64-bit integers.

29.2.2 Complex

Numbers of type `PetscComplex` have a precision matching `PetscReal`.

Form a complex number using `PETSC_i`:

```
PetscComplex x = 1.0 + 2.0 * PETSC_i;
```

The real and imaginary part can be extract with the functions `PetscRealPart` and `PetscImaginaryPart` which return a `PetscReal`.

There are also routines `VecRealPart` and `VecImaginaryPart` that replace a vector with its real or imaginary part respectively. Likewise `MatRealPart` and `MatImaginaryPart`.

29.2.3 MPI Scalars

For MPI calls, `MPIU_REAL` is the MPI type corresponding to the current `PetscReal`.

For MPI calls, `MPIU_SCALAR` is the MPI type corresponding to the current `PetscScalar`.

For MPI calls, `MPIU_COMPLEX` is the MPI type corresponding to the current `PetscComplex`.

Figure 29.2 `VecCreate`

```
C:
PetscErrorCode VecCreate(MPI_Comm comm,Vec *v);

F:
VecCreate( comm,v,ierr )
MPI_Comm :: comm
Vec      :: v
PetscErrorCode :: ierr

Python:
vec = PETSc.Vec()
vec.create()
# or:
vec = PETSc.Vec().create()
```

Figure 29.3 `VecDestroy`

```
Synopsis
#include "petscvec.h"
PetscErrorCode  VecDestroy(Vec *v)

Collective on Vec

Input Parameters:
v -the vector
```

29.2.4 Booleans

There is a `PetscBool` datatype with values `PETSC_TRUE` and `PETSC_FALSE`.

29.3 Vec: Vectors

Vectors are objects with a linear index. The elements of a vector are floating point numbers or complex numbers (see section 29.2), but not integers: for that see section 29.5.1.

29.3.1 Vector construction

Constructing a vector takes a number of steps. First of all, the vector object needs to be created on a communicator with `VecCreate` (figure 29.2)

Python note. In python, `PETSc.Vec()` creates an object with null handle, so a subsequent `create()` call is needed. In C and Fortran, the vector type is a keyword; in Python it is a member of `PETSc.Vec.Type`.

The corresponding routine `VecDestroy` (figure 29.3) deallocates data and zeros the pointer.

The vector type needs to be set with `VecSetType` (figure 29.4).

The most common vector types are:

- `VECSEQ` for sequential vectors, that is, living on a single process; This is typically created on the `MPI_COMM_SELF` or `PETSC_COMM_SELF` communicator.

Figure 29.4 **VecSetType**

```
Synopsis:
#include "petscvec.h"
PetscErrorCode VecSetType(Vec vec, VecType method)

Collective on Vec

Input Parameters:
vec- The vector object
method- The name of the vector type

Options Database Key
-vec_type <type> -Sets the vector type; use -help for a list of available types
```

Figure 29.5 **VecSetSizes**

```
C:
#include "petscvec.h"
PetscErrorCode  VecSetSizes(Vec v, PetscInt n, PetscInt N)
Collective on Vec

Input Parameters
v :the vector
n : the local size (or PETSC_DECIDE to have it set)
N : the global size (or PETSC_DECIDE)

Python:
PETSc.Vec.setSizes(self, size, bsize=None)
size is a tuple of local/global
```

- `VECMPI` for a vector distributed over the communicator. This is typically created on the `MPI_COMM_WORLD` or `PETSC_COMM_WORLD` communicator, or one derived from it.

Once you have created one vector, you can make more like it by `VecDuplicate`,

```
VecDuplicate(Vec old,Vec *new);
```

or `VecDuplicateVecs`

```
VecDuplicateVecs(Vec old,PetscInt n,Vec **new);
```

for multiple vectors. For the latter, there is a joint destroy call `VecDestroyVecs`:

```
VecDestroyVecs(PetscInt n,Vec **vecs);
```

(which is different in Fortran).

29.3.2 Vector layout

Next in the creation process the vector size is set with `VecSetSizes` (figure 29.5). Since a vector is typically distributed, this involves the global size and the sizes on the processors. Setting both is redundant, so it is possible to specify one and let the other be computed by the library. This is indicated by setting it to `PETSC_DECIDE` (`PETSc.DECIDE` in python).

Figure 29.6 **VecGetSize**

```
VecGetSize / VecGetLocalSize

C:
#include "petscvec.h"
PetscErrorCode  VecGetSize(Vec x,PetscInt *gsize)
PetscErrorCode  VecGetLocalSize(Vec x,PetscInt *lsize)

Input Parameter
x -the vector

Output Parameters
gsize - the global length of the vector
lsize - the local length of the vector

Python:
PETSc.Vec.getLocalSize(self)
PETSc.Vec.getSize(self)
PETSc.Vec.getSizes(self)
```

Figure 29.7 **VecGetOwnershipRange**

```
#include "petscvec.h"
PetscErrorCode  VecGetOwnershipRange(Vec x,PetscInt *low,PetscInt *high)

Input parameter:
x - the vector

Output parameters:
low  - the first local element, pass in NULL if not interested
high - one more than the last local element, pass in NULL if not interested

Fortran note:
use PETSC_NULL_INTEGER for NULL.
```

The size is queried with `VecGetSize` (figure 29.6) for the global size and `VecGetLocalSize` (figure 29.6) for the local size.

Each processor gets a contiguous part of the vector. Use `VecGetOwnershipRange` (figure 29.7) to query the first index on this process, and the first one of the next process.

In general it is best to let PETSc take care of memory management of matrix and vector objects, including allocating and freeing the memory. However, in cases where PETSc interfaces to other applications it maybe desirable to create a `Vec` object from an already allocated array: `VecCreateSeqWithArray` and `VecCreateMPIWithArray`.

```
VecCreateSeqWithArray
   (MPI_Comm comm,PetscInt bs,
    PetscInt n,PetscScalar *array,Vec *V);
VecCreateMPIWithArray
   (MPI_Comm comm,PetscInt bs,
    PetscInt n,PetscInt N,PetscScalar *array,Vec *vv);
```

As you will see in section 29.4.1, you can also create vectors based on the layout of a matrix, using

Figure 29.8 **VecView**

```
C:
#include "petscvec.h"
PetscErrorCode  VecView(Vec vec,PetscViewer viewer)

for ascii output use:
PETSC_VIEWER_STDOUT_WORLD

Python:
PETSc.Vec.view(self, Viewer viewer=None)

ascii output is default or use:
PETSc.Viewer.STDOUT(type cls, comm=None)
```

MatCreateVecs.

29.3.3 Vector operations

There are many routines operating on vectors that you need to write scientific applications. Examples are: norms, vector addition (including BLAS-type 'AXPY' routines), pointwise scaling, inner products. A large number of such operations are available in PETSc through single function calls to VecXYZ routines.

For debugging purpoases, the VecView (figure 29.8) routine can be used to display vectors on screen as ascii output,

```
// fftsine.c
ierr = VecView(signal,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
ierr = MatMult(transform,signal,frequencies); CHKERRQ(ierr);
ierr = VecView(frequencies,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
```

For the full source of this example, see section 29.8.3

but the routine call also use more general PetscViewer objects, for instance to dump a vector to file.

Here are a couple of representative vector routines:

```
PetscReal lambda;
ierr = VecNorm(y,NORM_2,&lambda); CHKERRQ(ierr);
ierr = VecScale(y,1./lambda); CHKERRQ(ierr);
```

Exercise 29.1. Use the routines VecDot (figure 29.9), VecScale (figure 29.10) and VecNorm (figure 29.11) to compute the inner product of vectors x,y, scale the vector x, and check its norm:

$$p \leftarrow x^t y$$
$$x \leftarrow x/p$$
$$n \leftarrow \|x\|_2$$

29.3.3.1 Split collectives

MPI is capable (in principle) of 'overlapping computation and communication', or *latency hiding*. PETSc supports this by splitting norms and inner products into two phases.

Figure 29.9 `VecDot`

```
Synopsis:
#include "petscvec.h"
PetscErrorCode VecDot(Vec x,Vec y,PetscScalar *val)

Collective on Vec

Input Parameters:
x, y - the vectors

Output Parameter:
val - the dot product
```

Figure 29.10 `VecScale`

```
Synopsis:
#include "petscvec.h"
PetscErrorCode VecScale(Vec x, PetscScalar alpha)

Not collective on Vec

Input Parameters:
x     - the vector
alpha - the scalar

Output Parameter:
x - the scaled vector
```

- Start inner product / norm with `VecDotBegin` / `VecNormBegin`;
- Conclude inner product / norm with `VecDotEnd` / `VecNormEnd`;

Even if you achieve no overlap, it is possible to use these calls to combine a number of 'collectives': do the `Begin` calls of one inner product and one norm; then do (in the same sequence) the `End` calls. This means that only a single reduction is performed on a two-word package, rather than two separate reductions on a single word.

29.3.4 Vector elements

Setting elements of a traditional array is simple. Setting elements of a distributed array is harder. First of all, `VecSet` sets the vector to a constant value:

```
ierr = VecSet(x,1.); CHKERRQ(ierr);
```

In the general case, setting elements in a PETSc vector is done through a function `VecSetValue` (figure 29.12) for setting elements that uses global numbering; any process can set any elements in the vector. There is also a routine `VecSetValues` (figure 29.13) for setting multiple elements. This is mostly useful for setting dense subblocks of a block matrix.

We illustrate both routines by setting a single element with `VecSetValue`, and two elements with `VecSetValues`. In the latter case we need an array of length two for both the indices and values. The indices need not be successive.

```
i = 1; v = 3.14;
```

Figure 29.11 **VecNorm**

```
C:
#include "petscvec.h"
PetscErrorCode  VecNorm(Vec x,NormType type,PetscReal *val)
where type is
    NORM_1, NORM_2, NORM_FROBENIUS, NORM_INFINITY

Python:
PETSc.Vec.norm(self, norm_type=None)

where norm is variable in PETSc.NormType:
    NORM_1, NORM_2, NORM_FROBENIUS, NORM_INFINITY or
    N1, N2, FRB, INF
```

Figure 29.12 **VecSetValue**

```
Synopsis
#include <petscvec.h>
PetscErrorCode VecSetValue
   (Vec v,PetscInt row,PetscScalar value,InsertMode mode);

Not Collective

Input Parameters
v- the vector
row- the row location of the entry
value- the value to insert
mode- either INSERT_VALUES or ADD_VALUES
```

Figure 29.13 **VecSetValues**

```
Synopsis
#include "petscvec.h"
PetscErrorCode  VecSetValues
   (Vec x,PetscInt ni,const PetscInt
    ix[],const PetscScalar y[],InsertMode iora)

Not Collective

Input Parameters:
x - vector to insert in
ni - number of elements to add
ix - indices where to add
y - array of values
iora - either INSERT_VALUES or ADD_VALUES, where
    ADD_VALUES adds values to any existing entries, and
    INSERT_VALUES replaces existing entries with new values
```

Figure 29.14 **VecAssemblyBegin**

```
#include "petscvec.h"
PetscErrorCode  VecAssemblyBegin(Vec vec)
PetscErrorCode  VecAssemblyEnd(Vec vec)

Collective on Vec

Input Parameter
vec -the vector
```

```
VecSetValue(x,i,v,INSERT_VALUES);
ii[0] = 1; ii[1] = 2; vv[0] = 2.7; vv[1] = 3.1;
VecSetValues(x,2,ii,vv,INSERT_VALUES);
```

```
call VecSetValue(x,i,v,INSERT_VALUES,ierr)
ii(1) = 1; ii(2) = 2; vv(1) = 2.7; vv(2) = 3.1
call VecSetValues(x,2,ii,vv,INSERT_VALUES,ierr)
```

Using `VecSetValue` for specifying a local vector element corresponds to simple insertion in the local array. However, an element that belongs to another process needs to be transferred. This done in two calls: `VecAssemblyBegin` (figure 29.14) and `VecAssemblyEnd`.

```
if (myrank==0) then
   do vecidx=0,globalsize-1
      vecelt = vecidx
      call VecSetValue(vector,vecidx,vecelt,INSERT_VALUES,ierr)
   end do
end if
call VecAssemblyBegin(vector,ierr)
call VecAssemblyEnd(vector,ierr)
```

For the full source of this example, see section 29.8.4

(If you know the MPI library, you'll recognize that the first call corresponds to posting nonblocking send and receive calls; the second then contains the wait calls. Thus, the existence of these separate calls make *latency hiding* possible.)

```
VecAssemblyBegin(myvec);
// do work that does not need the vector myvec
VecAssemblyEnd(myvec);
```

Elements can either be inserted with `INSERT_VALUES`, or added with `ADD_VALUES` in the `VecSetValue` / `VecSetValues` call. You can not immediately mix these modes; to do so you need to call `VecAssemblyBegin` / `VecAssemblyEnd` in between add/insert phases.

29.3.4.1 Explicit element access

Since the vector routines cover a large repertoire of operations, you hardly ever need to access the actual elements. Should you still need those elements, you can use `VecGetArray` (figure 29.15) for general access or `VecGetArrayRead` (figure 29.15) for read-only.

Figure 29.15 **VecGetArray**

```
C:
#include "petscvec.h"
PetscErrorCode VecGetArray(Vec x,PetscScalar **a)
PetscErrorCode VecGetArrayRead(Vec x,const PetscScalar **a)

Input Parameter
x : the vector

Output Parameter
a : location to put pointer to the array

PetscErrorCode VecRestoreArray(Vec x,PetscScalar **a)
PetscErrorCode VecRestoreArrayRead(Vec x,const PetscScalar **a)

Input Parameters
x : the vector
a : location of pointer to array obtained from VecGetArray()

Fortran90:
#include <petsc/finclude/petscvec.h>
use petscvec
VecGetArrayF90(Vec x,{Scalar, pointer :: xx_v(:)},integer ierr)
(there is a Fortran77 version)
VecRestoreArrayF90(Vec x,{Scalar, pointer :: xx_v(:)},integer ierr)

Python:
PETSc.Vec.getArray(self, readonly=False)
?? PETSc.Vec.resetArray(self, force=False)
```

Figure 29.16 **VecRestoreArray**

```
C:
#include "petscvec.h"
PetscErrorCode VecRestoreArray(Vec x,PetscScalar **a)

Logically Collective on Vec

Input Parameters:
x- the vector
a- location of pointer to array obtained from VecGetArray()

Fortran90:
#include <petsc/finclude/petscvec.h>
use petscvec
VecRestoreArrayF90(Vec x,{Scalar, pointer :: xx_v(:)},integer ierr)

Input Parameters:
x- vector
xx_v- the Fortran90 pointer to the array
```

Figure 29.17 **VecPlaceArray**

```
Replace the storage of a vector by another array
Synopsis

#include "petscvec.h"
PetscErrorCode  VecPlaceArray(Vec vec,const PetscScalar array[])
PetscErrorCode  VecReplaceArray(Vec vec,const PetscScalar array[])

Input Parameters
vec   - the vector
array - the array
```

PETSc insists that you properly release this pointer again with `VecRestoreArray` (figure 29.16) or `VecRestoreArrayRead` (figure 29.16).

Note that in a distributed running context you can only get the array of local elements. Accessing the elements from another process requires explicit communication; see section 29.5.2.

```
PetscScalar *in_array,*out_array;
VecGetArrayRead(in,&in_array);
VecGetArray(out,&out_array);
VecGetLocalSize(in,&localsize);
for (int i=0; i<localsize; i++)
  out_array[i] = 2*in_array[i];
VecRestoreArrayRead(in,&in_array);
VecRestoreArray(out,&out_array);
```

There are some variants to the `VecGetArray` operation:

- `VecReplaceArray` (figure 29.17) frees the memory of the `Vec` object, and replaces it with a different array. That latter array needs to be allocated with `PetscMalloc`.
- `VecPlaceArray` (figure 29.17) also installs a new array in the vector, but it keeps the original array; this can be restored with `VecResetArray`.

Putting the array of one vector into another has a common application, where you have a distributed vector, but want to apply PETSc operations to its local section as if it were a sequential vector. In that case you would create a sequential vector, and `VecPlaceArray` the contents of the distributed vector into it.

Fortran note. There are routines such as `VecGetArrayF90` (with corresponding `VecRestoreArrayF90`) that return a (Fortran) pointer to a one-dimensional array.

```
// vecset.F90
Vec            :: vector
PetscScalar,dimension(:),pointer :: elements
call VecGetArrayF90(vector,elements,ierr)
write (msg,10) myrank,elements(1)
10 format("First element on process",i3,":",f7.4,"\n")
call PetscSynchronizedPrintf(comm,msg,ierr)
call PetscSynchronizedFlush(comm,PETSC_STDOUT,ierr)
call VecRestoreArrayF90(vector,elements,ierr)
```

For the full source of this example, see section 29.8.4

Figure 29.18 `MatCreate`

```
C:
PetscErrorCode MatCreate(MPI_Comm comm,Mat *v);

Python:
mat = PETSc.Mat()
mat.create()
# or:
mat = PETSc.Mat().create()
```

29.3.5 File I/O

As mentioned above, `VecView` can be used for displaying a vector on the terminal screen. However, viewers are actually much more general. As explained in section 34.2.2, they can also be used to export vector data, for instance to file.

The converse operation, to load a vector that was exported in this manner, is *VecLoad*.

Since these operations are each other's inverses, usually you don't need to know the file format. But just in case:

```
PetscInt    VEC_FILE_CLASSID
PetscInt    number of rows
PetscScalar *values of all entries
```

That is, the file starts with a magic number, then the number of vector elements, and subsequently all scalar values.

29.4 Mat: Matrices

PETSc matrices come in a number of types, sparse and dense being the most important ones. Another possibility is to have the matrix in operation form, where only the action $y \leftarrow Ax$ is defined.

29.4.1 Matrix creation

Creating a matrix also starts by specifying a communicator on which the matrix lives collectively: `MatCreate` (figure 29.18)

Set the matrix type with `MatSetType` (figure 29.19). The main choices are between sequential versus distributed and dense versus sparse, giving types: `MATMPIDENSE`, `MATMPIAIJ`, `MATSEQDENSE`, `MATSEQAIJ`.

Distributed matrices are partitioned by block rows: each process stores a *block row*, that is, a contiguous set of matrix rows. It stores all elements in that block row. In order for a matrix-vector product to be executable, both the input and output vector need to be partitioned conforming to the matrix.

While for dense matrices the block row scheme is not scalable, for matrices from PDEs it makes sense. There, a subdivision by matrix blocks would lead to many empty blocks.

Figure 29.19 **MatSetType**

```
#include "petscmat.h"
PetscErrorCode  MatSetType(Mat mat, MatType matype)

Collective on Mat

Input Parameters:
mat- the matrix object
matype- matrix type

Options Database Key
-mat_type <method> -Sets the type; use -help for a list of available methods (for instance, seqaij)
```

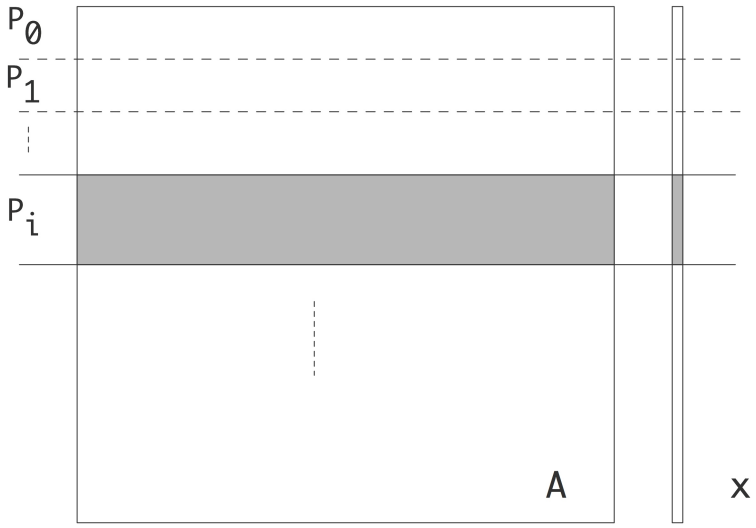


Figure 29.1: Matrix partitioning by block rows

Just as with vectors, there is a local and global size; except that that now applies to rows and columns. Set sizes with `MatSetSizes` (figure 29.20) and subsequently query them with `MatSizes` (figure 29.21). The concept of local column size is tricky: since a process stores a full block row you may expect the local column size to be the full matrix size, but that is not true. The exact definition will be discussed later, but for square matrices it is a safe strategy to let the local row and column size to be equal.

Instead of querying a matrix size and creating vectors accordingly, the routine `MatCreateVecs` (figure 29.22) can be used. (Sometimes this is even required; see section 29.4.8.)

29.4.2 Nonzero structure

In case of a dense matrix, once you have specified the size and the number of MPI processes, it is simple to determine how much space PETSc needs to allocate for the matrix. For a sparse matrix this is more complicated, since the matrix can be anywhere between completely empty and completely filled in. It would be possible to have a dynamic approach where, as elements are specified, the space grows; however, repeated allocations and re-allocations are inefficient. For this reason PETSc puts a small burden on the

Figure 29.20 **MatSetSizes**

```
C:
#include "petscmat.h"
PetscErrorCode MatSetSizes(Mat A,
    PetscInt m, PetscInt n, PetscInt M, PetscInt N)

Input Parameters
A : the matrix
m : number of local rows (or PETSC_DECIDE)
n : number of local columns (or PETSC_DECIDE)
M : number of global rows (or PETSC_DETERMINE)
N : number of global columns (or PETSC_DETERMINE)

Python:
PETSc.Mat.setSizes(self, size, bsize=None)
where 'size' is a tuple of 2 global sizes
or a tuple of 2 local/global pairs
```

Figure 29.21 **MatSizes**

```
C:
#include "petscmat.h"
PetscErrorCode MatGetSize(Mat mat,PetscInt *m,PetscInt *n)
PetscErrorCode MatGetLocalSize(Mat mat,PetscInt *m,PetscInt *n)

Python:
PETSc.Mat.getSize(self) # tuple of global sizes
PETSc.Mat.getLocalSize(self) # tuple of local sizes
PETSc.Mat.getSizes(self) # tuple of local/global size tuples
```

Figure 29.22 **MatCreateVecs**

```
Synopsis
Get vector(s) compatible with the matrix, i.e. with the same parallel layout

#include "petscmat.h"
PetscErrorCode MatCreateVecs(Mat mat,Vec *right,Vec *left)

Collective on Mat

Input Parameter
mat - the matrix

Output Parameter;
right  - (optional) vector that the matrix can be multiplied against
left  - (optional) vector that the matrix vector product can be stored in
```

Figure 29.23 MatSeqAIJSetPreallocation

```
#include "petscmat.h"
PetscErrorCode  MatSeqAIJSetPreallocation
  (Mat B,PetscInt nz,const PetscInt nnz[])
PetscErrorCode  MatMPIAIJSetPreallocation
  (Mat B,PetscInt d_nz,const PetscInt d_nnz[],
   PetscInt o_nz,const PetscInt o_nnz[])

Input Parameters

B - the matrix
nz/d_nz/o_nz - number of nonzeros per row in matrix or
    diagonal/off-diagonal portion of local submatrix
nnz/d_nnz/o_nnz - array containing the number of nonzeros in the various rows of
    the sequential matrix / diagonal / offdiagonal part of the local submatrix
    or NULL (PETSC_NULL_INTEGER in Fortran) if nz/d_nz/o_nz is used.

Python:
PETSc.Mat.setPreallocationNNZ(self, [nnz_d,nnz_o] )
PETSc.Mat.setPreallocationCSR(self, csr)
PETSc.Mat.setPreallocationDense(self, array)
```

programmer: you need to specify a bound on how many elements the matrix will contain.

We explain this by looking at some cases. First we consider a matrix that only lives on a single process. You would then use `MatSeqAIJSetPreallocation` (figure 29.23). In the case of a tridiagonal matrix you would specify that each row has three elements:

```
MatSeqAIJSetPreallocation(A,3, NULL);
```

If the matrix is less regular you can use the third argument to give an array of explicit row lengths:

```
int *rowlengths;
// allocate, and then:
for (int row=0; row<nrows; row++)
  rowlengths[row] = // calculation of row length
MatSeqAIJSetPreallocation(A,NULL,rowlengths);
```

In case of a distributed matrix you need to specify this bound with respect to the block structure of the matrix. As illustrated in figure 29.2, a matrix has a diagonal part and an off-diagonal part. The diagonal part describes the matrix elements that couple elements of the input and output vector that live on this process. The off-diagonal part contains the matrix elements that are multiplied with elements not on this process, in order to compute elements that do live on this process.

The preallocation specification now has separate parameters for these diagonal and off-diagonal parts: with `MatMPIAIJSetPreallocation` (figure 29.23). you specify for both either a global upper bound on the number of nonzeros, or a detailed listing of row lengths. For the matrix of the *Laplace equation*, this specification would seem to be:

```
MatMPIAIJSetPreallocation(A, 3, NULL, 2, NULL);
```

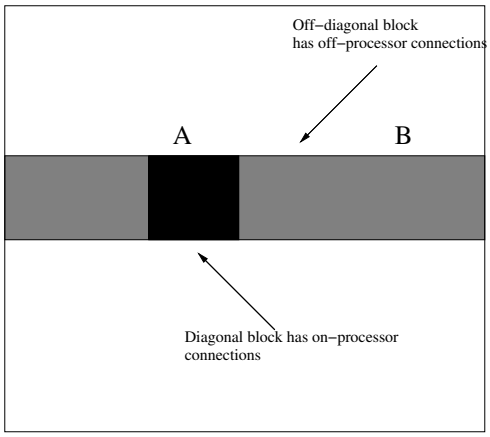


Figure 29.2: The diagonal and off-diagonal parts of a matrix

Figure 29.24 `MatSetValue`

```
C:
#include <petscmat.h>
PetscErrorCode MatSetValue(
    Mat m,PetscInt row,PetscInt col,PetscScalar value,InsertMode mode)

Input Parameters
m : the matrix
row : the row location of the entry
col : the column location of the entry
value : the value to insert
mode : either INSERT_VALUES or ADD_VALUES

Python:
PETSc.Mat.setValue(self, row, col, value, addv=None)
also supported:
A[row,col] = value
```

However, this is only correct if the block structure from the parallel division equals that from the lines in the domain. In general it may be necessary to use values that are an overestimate. It is then possible to contract the storage by copying the matrix.

Specifying bounds on the number of nonzeros is often enough, and not too wasteful. However, if many rows have fewer nonzeros than these bounds, a lot of space is wasted. In that case you can replace the NULL arguments by an array that lists for each row the number of nonzeros in that row.

29.4.3 Matrix elements

You can set a single matrix element with `MatSetValue` (figure 29.24) or a block of them, where you supply a set of i and j indices, using `MatSetValues`.

After setting matrix elements, the matrix needs to be assembled. This is where PETSc moves matrix elements to the right processor, if they were specified elsewhere. As with vectors this takes two calls:

Figure 29.25 **MatAssemblyBegin**

```
C:
#include "petscmat.h"
PetscErrorCode MatAssemblyBegin(Mat mat,MatAssemblyType type)
PetscErrorCode MatAssemblyEnd(Mat mat,MatAssemblyType type)

Input Parameters
mat- the matrix
type- type of assembly, either MAT_FLUSH_ASSEMBLY
    or MAT_FINAL_ASSEMBLY

Python:
assemble(self, assembly=None)
assemblyBegin(self, assembly=None)
assemblyEnd(self, assembly=None)

there is a class PETSc.Mat.AssemblyType:
FINAL = FINAL_ASSEMBLY = 0
FLUSH = FLUSH_ASSEMBLY = 1
```

`MatAssemblyBegin` (figure 29.25) and `MatAssemblyEnd` (figure 29.25) which can be used to achieve *latency hiding*.

Elements can either be inserted (`INSERT_VALUES`) or added (`ADD_VALUES`). You can not immediately mix these modes; to do so you need to call `MatAssemblyBegin` / `MatAssemblyEnd` with a value of `MAT_FLUSH_ASSEMBLY`.

PETSc sparse matrices are very flexible: you can create them empty and then start adding elements. However, this is very inefficient in execution since the OS needs to reallocate the matrix every time it grows a little. Therefore, PETSc has calls for the user to indicate how many elements the matrix will ultimately contain.

```
MatSetOption(A, MAT_NEW_NONZERO_ALLOCATION_ERR, PETSC_FALSE)
```

29.4.3.1 Element access

If you absolutely need access to the matrix elements, there are routines such as `MatGetRow` (figure 29.26). With this, any process can request, using global row numbering, the contents of a row that it owns. (Requesting elements that are not local requires the different mechanism of taking submatrices; section 29.4.5.)

Since PETSc is geared towards *sparse matrices*, this returns not only the element values, but also the column numbers, as well as the mere number of stored columns. If any of these three return values are not needed, they can be unrequested by setting the parameter passed to *NULL*.

PETSc insists that you properly release the row again with `MatRestoreRow` (figure 29.26).

It is also possible to retrieve the full Compressed Row Storage (CRS) contents of the local matrix with *MatDenseGetArray*, *MatDenseRestoreArray*, *MatSeqAIJGetArray*, *MatSeqAIJRestoreArray*. (Routines `MatGetArray` / `MatRestoreArray` are deprecated.)

Figure 29.26 `MatGetRow`

```
Synopsis:

#include "petscmat.h"
PetscErrorCode MatGetRow
   (Mat mat,PetscInt row,
    PetscInt *ncols,const PetscInt *cols[],const PetscScalar *vals[])
PetscErrorCode MatRestoreRow
   (Mat mat,PetscInt row,
    PetscInt *ncols,const PetscInt *cols[],const PetscScalar *vals[])

Input Parameters:
mat - the matrix
row - the row to get

Output Parameters
ncols - if not NULL, the number of nonzeros in the row
cols - if not NULL, the column numbers
vals - if not NULL, the values
```

Figure 29.27 `MatMult`

```
Synopsis
#include "petscmat.h"
PetscErrorCode MatMult(Mat mat,Vec x,Vec y)
PetscErrorCode MatMultTranspose(Mat mat,Vec x,Vec y)

Neighbor-wise Collective on Mat

Input Parameters
mat - the matrix
x - the vector to be multiplied

Output Parameters
y - the result
```

29.4.4 Matrix operations

29.4.4.1 Matrix-vector operations

In the typical application of PETSc, solving large sparse linear systems of equations with iterative methods, matrix-vector operations are most important. Foremost there is the matrix-vector product `MatMult` (figure 29.27) and the transpose product `MatMultTranspose` (figure 29.27). (In the complex case, the transpose product is not the Hermitian matrix product; for that use `MatMultHermitianTranspose`.)

For the BLAS gemv semantics $y \leftarrow \alpha Ax + \beta y$, `MatMultAdd` (figure 29.28) computes $z \leftarrow Ax + y$.

29.4.4.2 Matrix-matrix operations

There is a number of matrix-matrix routines such as *`MatMatMult`*.

29.4.5 Submatrices

Given a parallel matrix, there are two routines for extracting submatrices:

Figure 29.28 MatMultAdd

```
Synopsis
#include "petscmat.h"
PetscErrorCode MatMultAdd(Mat mat,Vec x,Vec y,Vec z)

Neighbor-wise Collective on Mat

Input Parameters
mat  - the matrix
x, y  - the vectors

Output Parameters
z -the result

Notes
The vectors x and z cannot be the same.
```

- MatCreateSubMatrix creates a single parallel submatrix.
- MatCreateSubMatrices creates a sequential submatrix on each process.

29.4.6 Shell matrices

In many scientific applications, a matrix stands for some operator, and we are not intrinsically interested in the matrix elements, but only in the action of the matrix on a vector. In fact, under certain circumstances it is more convenient to implement a routine that computes the matrix action than to construct the matrix explicitly.

Maybe surprisingly, solving a linear system of equations can be handled this way. The reason is that PETSc's iterative solvers (section 32.1) only need the matrix-times-vector (and perhaps the matrix-transpose-times-vector) product.

PETSc supports this mode of working. The routine MatCreateShell (figure 29.29) declares the argument to be a matrix given in operator form.

29.4.6.1 Shell operations

The next step is then to add the custom multiplication routine, which will be invoked by MatMult: MatShellSetOperation (figure 29.30)

The routine that implements the actual product should have the same prototype as MatMult, accepting a matrix and two vectors. The key to realizing your own product routine lies in the 'context' argument to the create routine. With MatShellSetContext (figure 29.31) you pass a pointer to some structure that contains all contextual information you need. In your multiplication routine you then retrieve this with MatShellGetContext (figure 29.32).

What operation is specified is determined by a keyword *MATOP_<OP>* where *OP* is the name of the matrix routine, minus the Mat part, in all caps.

```
MatCreate(comm,&A);
MatSetSizes(A,localsize,localsize,matrix_size,matrix_size);
```

Figure 29.29 **MatCreateShell**

```
#include "petscmat.h"
PetscErrorCode  MatCreateShell
   (MPI_Comm comm,
    PetscInt m,PetscInt n,PetscInt M,PetscInt N,
    void *ctx,Mat *A)

Collective

Input Parameters:
comm- MPI communicator
m- number of local rows (must be given)
n- number of local columns (must be given)
M- number of global rows (may be PETSC_DETERMINE)
N- number of global columns (may be PETSC_DETERMINE)
ctx- pointer to data needed by the shell matrix routines

Output Parameter:
A -the matrix
```

Figure 29.30 **MatShellSetOperation**

```
#include "petscmat.h"
PetscErrorCode MatShellSetOperation
   (Mat mat,MatOperation op,void (*g)(void))

Logically Collective on Mat

Input Parameters:
mat- the shell matrix
op- the name of the operation
g- the function that provides the operation.
```

Figure 29.31 **MatShellSetContext**

```
Synopsis
#include "petscmat.h"
PetscErrorCode  MatShellSetContext(Mat mat,void *ctx)

Input Parameters
mat - the shell matrix
ctx - the context
```

Figure 29.32 **MatShellGetContext**

```
#include "petscmat.h"
PetscErrorCode  MatShellGetContext(Mat mat,void *ctx)

Not Collective

Input Parameter:
mat -the matrix, should have been created with MatCreateShell()

Output Parameter:
ctx -the user provided context
```

```
MatSetType(A,MATSHELL);
MatSetFromOptions(A);
MatShellSetOperation(A,MATOP_MULT,(void*)&mymatmult);
MatShellSetContext(A,(void*)Diag);
MatSetUp(A);
```

(The call to `MatSetSizes` needs to come before `MatSetType`.)

29.4.6.2 Shell context

Setting the context means passing a pointer (really: an address) to some allocated structure

```
struct matrix_data mystruct;
MatShellSetContext( A, &mystruct );
```

The routine prototype has this argument as a `void*` but it's not necessary to cast it to that. Getting the context means that a pointer to your structure needs to be set

```
struct matrix_data *mystruct;
MatShellGetContext( A, &mystruct );
```

Somewhat confusingly, the Get routine also has a `void*` argument, even though it's really a pointer variable.

29.4.7 Multi-component matrices

For multi-component physics problems there are essentially two ways of storing the linear system

1. Grouping the physics equations together, or
2. grouping the domain nodes together.

In both cases this corresponds to a block matrix, but for a problem of N nodes and 3 equations, the respective structures are:

1. 3×3 blocks of size N, versus
2. $N \times N$ blocks of size 3.

The first case can be pictured as

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$$

and while it looks natural, there is a computational problem with it. Preconditioners for such problems often look like

$$\begin{pmatrix} A_{00} & & \\ & A_{11} & \\ & & A_{22} \end{pmatrix} \quad \text{or} \quad \begin{pmatrix} A_{00} & & \\ A_{10} & A_{11} & \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$$

With the block-row partitioning of PETSc's matrices, this means at most a 50% efficiency for the preconditioner solve.

It is better to use the second scheme, which requires the `MATMPIBIJ` format, and use so-called *field-split preconditioners*; see section **??**.

29.4.8 Fourier transform

The *Fast Fourier Transform (FFT)* can be considered a matrix-vector multiplication. PETSc supports this by letting you create a matrix with `MatCreateFFT`. This requires that you add an FFT library, such as *fftw*, at configuration time; see section 28.3.4.

FFT libraries may use padding, so vectors should be created with `MatCreateVecsFFTW`, not with an independent `VecSetSizes`.

29.5 Index sets and Vector Scatters

In the PDE type of applications that PETSc was originally intended for, vector data can only be real or complex: there are no vector of integers. On the other hand, integers are used for indexing into vector, for instance for gathering boundary elements into a *halo region*, or for doing the *data transpose* of an *FFT* operation.

To support this, PETSc has the following object types:

- An `IS` object describes a set of integer indices;
- a `VecScatter` object describes the correspondence between a group of indices in an input vector and a group of indices in an output vector.

29.5.1 IS: index sets

An `IS` object contains a set of `PetscInt` values. It can be created with

- `ISCreate` for creating an empty set;
- `ISCreateStride` for a strided set;
- `ISCreateBlock` for a set of contiguous blocks, placed at an explicitly given list of starting indices.
- `ISCreateGeneral` for an explicitly given list of indices.

For example, to describe odd and even indices (on two processes):

```
// oddeven.c
IS oddeven;
if (procid==0) {
  ierr = ISCreateStride(comm,Nglobal/2,0,2,&oddeven); CHKERRQ(ierr);
} else {
  ierr = ISCreateStride(comm,Nglobal/2,1,2,&oddeven); CHKERRQ(ierr);
}
```

For the full source of this example, see section 29.8.5

After this, there are various query and set operations on index sets.

You can read out the indices of a set by `ISGetIndices` and `ISRestoreIndices`.

Figure 29.33 `VecScatterCreate`

```
Synopsis
Creates a vector scatter context. Collective on Vec

#include "petscvec.h"
PetscErrorCode VecScatterCreate(Vec xin,IS ix,Vec yin,IS iy,VecScatter *newctx)

Input Parameters:
xin : a vector that defines the layout of vectors from which we scatter
yin : a vector that defines the layout of vectors to which we scatter
ix : the indices of xin to scatter (if NULL scatters all values)
iy : the indices of yin to hold results (if NULL fills entire vector yin)

Output Parameter
newctx : location to store the new scatter context
```

29.5.2 VecScatter: all-to-all operations

A `VecScatter` object is a generalization of an all-to-all operation. However, unlike MPI `MPI_Alltoall`, which formulates everything in terms of local buffers, a `VecScatter` is more implicit in only describing indices in the input and output vectors.

The `VecScatterCreate` (figure 29.33) call has as arguments:

- An input vector. From this, the parallel layout is used; any vector being scattered from should have this same layout.
- An `IS` object describing what indices are being scattered; if the whole vector is rearranged, *NULL* (Fortran: `PETSC_NULL_IS`) can be given.
- An output vector. From this, the parallel layout is used; any vector being scattered into should have this same layout.
- An `IS` object describing what indices are being scattered into; if the whole vector is a target, *NULL* can be given.

As a simple example, the odd/even sets defined above can be used to move all components with even index to process zero, and the ones with odd index to process one:

```
VecScatter separate;
ierr = VecScatterCreate
  (in,oddeven,out,NULL,&separate); CHKERRQ(ierr);
ierr = VecScatterBegin
  (separate,in,out,INSERT_VALUES,SCATTER_FORWARD); CHKERRQ(ierr);
ierr = VecScatterEnd
  (separate,in,out,INSERT_VALUES,SCATTER_FORWARD); CHKERRQ(ierr);
```

For the full source of this example, see section 29.8.5

Note that the index set is applied to the input vector, since it describes the components to be moved. The output vector uses *NULL* since these components are placed in sequence.

Exercise 29.2. Modify this example so that the components are still separated odd/even, but now placed in descending order on each process.

Exercise 29.3. Can you extend this example so that process p receives all indices that are multiples of p? Is your solution correct if *Nglobal* is not a multiple of *nprocs*?

29.5.2.1 More VecScatter modes

There is an added complication, in that a `VecScatter` can have both sequential and parallel input or output vectors. Scattering onto process zero is also a popular option.

29.6 AO: Application Orderings

PETSc's decision to partition a matrix by contiguous block rows may be a limitation in the sense an application can have a natural ordering that is different. For such cases the `AO` type can translate between the two schemes.

29.7 Partitionings

By default, PETSc uses partitioning of matrices and vectors based on consecutive blocks of variables. In regular cases that is not a bad strategy. However, for some matrices a permutation and re-division can be advantageous. For instance, one could look at the *adjacency graph*, and minimize the number of *edge cuts* or the sum of the *edge weights*.

This functionality is not native to PETSc, but can be provided by *graph partitioning packages* such as *ParMetis* or *Zoltan*. The basic object is the `MatPartitioning`, with routines for

- Create and destroy: `MatPartitioningCreate`, `MatPartitioningDestroy`;
- Setting the type `MatPartitioningSetType` to an explicit partitioner, or something generated as the dual or a refinement of the current matrix;
- Apply with `MatPartitioningApply`, giving a distribued `IS` object, which can then be used in `MatCreateSubMatrix` to repartition.

Illustrative example:

```
MatPartitioning part;
MatPartitioningCreate(comm,&part);
MatPartitioningSetType(part,MATPARTITIONINGPARMETIS);
MatPartitioningApply(part,&is);
/* get new global number of each old global number */
ISPartitioningToNumbering(is,&isn);
ISBuildTwoSided(is,NULL,&isrows);
MatCreateSubMatrix(A,isrows,isrows,MAT_INITIAL_MATRIX,&perA);
```

Other scenario:

```
MatPartitioningSetAdjacency(part,A);
MatPartitioningSetType(part,MATPARTITIONINGHIERARCH);
MatPartitioningHierarchicalSetNcoarseparts(part,2);
MatPartitioningHierarchicalSetNfineparts(part,2);
```

29.8 Sources used in this chapter

29.8.1 Listing of code header

29.8.2 Listing of code examples/petsc/c/split.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

static char help[] = "\nOwership example.\n\n";

#include <petscvec.h>

int main(int argc,char **argv)
{
  Vec            x,y;              /* vectors */
  int nprocs,procid;
  PetscErrorCode ierr;

  PetscInitialize(&argc,&argv,(char*)0,help);
  MPI_Comm comm = PETSC_COMM_WORLD;
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_rank(comm,&procid);

  PetscInt N,n;
  N = 100; n = PETSC_DECIDE;
  PetscSplitOwnership(comm,&n,&N);
  PetscPrintf(comm,"Global %d, local %d\n",N,n);

  N = PETSC_DECIDE; n = 10;
  PetscSplitOwnership(comm,&n,&N);
  PetscPrintf(comm,"Global %d, local %d\n",N,n);

  return PetscFinalize();
}
```

29.8.3 Listing of code examples/petsc/c/fftsine.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

static char help[] = "\nFFT example.\n\n";

#include <petscmat.h>

int main(int argc,char **argv)
{
  Vec            x,y;              /* vectors */
  int nprocs,procid;
```

```
  PetscErrorCode ierr;

  PetscInitialize(&argc,&argv,(char*)0,help);
  MPI_Comm comm = PETSC_COMM_WORLD;
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_rank(comm,&procid);

  PetscInt Nlocal = 10, Nglobal = Nlocal*nprocs;
  PetscPrintf(comm,"FFT examples on N=%d n=%d\n",Nglobal,Nlocal);

  Mat transform;
  int dimensionality=1;
  PetscInt dimensions[dimensionality]; dimensions[0] = Nglobal;
  PetscPrintf(comm,"Creating fft D=%d, dim=%d\n",dimensionality,dimensions[0]);
  ierr = MatCreateFFT(comm,dimensionality,dimensions,MATFFTW,&transform); CHKERRQ(ierr);
  {
    PetscInt fft_i,fft_j;
    ierr = MatGetSize(transform,&fft_i,&fft_j); CHKERRQ(ierr);
    PetscPrintf(comm,"FFT global size %d x %d\n",fft_i,fft_j);
  }
  Vec signal,frequencies;
  ierr = MatCreateVecsFFTW(transform,&frequencies,&signal,PETSC_NULL); CHKERRQ(ierr);
  ierr = PetscObjectSetName((PetscObject)signal,"signal"); CHKERRQ(ierr);
  ierr = PetscObjectSetName((PetscObject)frequencies,"frequencies"); CHKERRQ(ierr);
  ierr = VecAssemblyBegin(signal); CHKERRQ(ierr);
  ierr = VecAssemblyEnd(signal); CHKERRQ(ierr);
  {
    PetscInt nlocal,nglobal;
    ierr = VecGetLocalSize(signal,&nlocal); CHKERRQ(ierr);
    ierr = VecGetSize(signal,&nglobal); CHKERRQ(ierr);
    ierr = PetscPrintf(comm,"Signal local=%d global=%d\n",nlocal,nglobal); CHKERRQ(ierr);
  }

  PetscInt myfirst,mylast;
  ierr = VecGetOwnershipRange(signal,&myfirst,&mylast); CHKERRQ(ierr);
  printf("Setting %d -- %d\n",myfirst,mylast);
  //  for (PetscInt vecindex=myfirst; vecindex<mylast; vecindex++) {
  for (PetscInt vecindex=0; vecindex<Nglobal; vecindex++) {
    PetscScalar
      pi = 4. * atan(1.0),
      h = 1./Nglobal,
      phi = 2* pi * (vecindex+1) * h,
      puresine = cos( phi )
#if defined(PETSC_USE_COMPLEX)
          + PETSC_i * sin(phi)
#endif
      ;
    ierr = VecSetValue(signal,vecindex,puresine,INSERT_VALUES); CHKERRQ(ierr);
  }
  ierr = VecAssemblyBegin(signal); CHKERRQ(ierr);
  ierr = VecAssemblyEnd(signal); CHKERRQ(ierr);

  ierr = VecView(signal,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
```

```
  ierr = MatMult(transform,signal,frequencies); CHKERRQ(ierr);
  ierr = VecView(frequencies,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);

  ierr = MatDestroy(&transform); CHKERRQ(ierr);
  ierr = VecDestroy(&signal); CHKERRQ(ierr);
  ierr = VecDestroy(&frequencies); CHKERRQ(ierr);

  return PetscFinalize();
}
```

29.8.4 Listing of code examples/petsc/f/vecset.F90

```
Program VecSetF90

#include <petsc/finclude/petsc.h>
  use petsc
  implicit none

  Vec            :: vector
  PetscScalar,dimension(:),pointer :: elements
  PetscErrorCode :: ierr
  PetscInt       :: globalsize
  integer        :: myrank,vecidx,comm
  PetscScalar    :: vecelt
  character*80   :: msg

  call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
  CHKERRA(ierr)
  comm = MPI_COMM_WORLD
  call MPI_Comm_rank(comm,myrank,ierr)

  call VecCreate(comm,vector,ierr)
  call VecSetType(vector,VECMPI,ierr)
  call VecSetSizes(vector,2,PETSC_DECIDE,ierr)
  call VecGetSize(vector,globalsize,ierr)

  if (myrank==0) then
     do vecidx=0,globalsize-1
        vecelt = vecidx
        call VecSetValue(vector,vecidx,vecelt,INSERT_VALUES,ierr)
     end do
  end if
  call VecAssemblyBegin(vector,ierr)
  call VecAssemblyEnd(vector,ierr)
  call VecView(vector,PETSC_VIEWER_STDOUT_WORLD,ierr)

  call VecGetArrayF90(vector,elements,ierr)
  write (msg,10) myrank,elements(1)
10 format("First element on process",i3,":",f7.4,"\n")
  call PetscSynchronizedPrintf(comm,msg,ierr)
  call PetscSynchronizedFlush(comm,PETSC_STDOUT,ierr)
  call VecRestoreArrayF90(vector,elements,ierr)
```

```
  call VecDestroy(vector,ierr)
  call PetscFinalize(ierr); CHKERRQ(ierr);

End Program VecSetF90
```

29.8.5 Listing of code examples/petsc/c/oddeven.c

```
#include "petscksp.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
  PetscErrorCode ierr;
  MPI_Comm comm;

  PetscFunctionBegin;
  PetscInitialize(&argc,&args,0,0);

  comm = MPI_COMM_WORLD;

  int nprocs,procid;
  MPI_Comm_rank(comm,&procid);
  MPI_Comm_size(comm,&nprocs);
  if (nprocs!=2) {
    PetscPrintf(comm,"This example only works on 2 processes, not %d\n",nprocs);
    PetscFunctionReturn(-1); }

  PetscInt Nglobal = 2*nprocs;
  {
    PetscInt x=1;
    ierr = PetscOptionsGetInt(PETSC_NULL,PETSC_NULL,"-x",&x,NULL); CHKERRQ(ierr);
    Nglobal *= x;
  }

  Vec in,out;
  ierr = VecCreate(comm,&in); CHKERRQ(ierr);
  ierr = VecSetType(in,VECMPI); CHKERRQ(ierr);
  ierr = VecSetSizes(in,PETSC_DECIDE,Nglobal); CHKERRQ(ierr);
  ierr = VecDuplicate(in,&out); CHKERRQ(ierr);

  {
    PetscInt myfirst,mylast;
    ierr = VecGetOwnershipRange(in,&myfirst,&mylast); CHKERRQ(ierr);
    for (PetscInt index=myfirst; index<mylast; index++) {
      PetscScalar v = index;
      ierr = VecSetValue(in,index,v,INSERT_VALUES); CHKERRQ(ierr);
    }
    ierr = VecAssemblyBegin(in); CHKERRQ(ierr);
    ierr = VecAssemblyEnd(in); CHKERRQ(ierr);
  }

  IS oddeven;
```

```
  if (procid==0) {
    ierr = ISCreateStride(comm,Nglobal/2,0,2,&oddeven); CHKERRQ(ierr);
  } else {
    ierr = ISCreateStride(comm,Nglobal/2,1,2,&oddeven); CHKERRQ(ierr);
  }
  ISView(oddeven,0);

  VecScatter separate;
  ierr = VecScatterCreate
    (in,oddeven,out,NULL,&separate); CHKERRQ(ierr);
  ierr = VecScatterBegin
    (separate,in,out,INSERT_VALUES,SCATTER_FORWARD); CHKERRQ(ierr);
  ierr = VecScatterEnd
    (separate,in,out,INSERT_VALUES,SCATTER_FORWARD); CHKERRQ(ierr);

  ierr = ISDestroy(&oddeven); CHKERRQ(ierr);
  ierr = VecScatterDestroy(&separate); CHKERRQ(ierr);

  ierr = VecView(in,0); CHKERRQ(ierr);
  ierr = VecView(out,0); CHKERRQ(ierr);

  ierr = VecDestroy(&in); CHKERRQ(ierr);
  ierr = VecDestroy(&out); CHKERRQ(ierr);

  return PetscFinalize();
}
```

Chapter 30

Grid support

PETSc's DM objects raise the abstraction level from the linear algebra problem to the physics problem: they allow for a more direct expression of operators in terms of their domain of definition. In this section we look at the DMDA 'distributed array' objects, which correspond to problems defined on Cartesian grids. Distributed arrays make it easier to construct the coefficient matrix of an operator that is defined as a *stencil* on a 1/2/3-dimensional *Cartesian grid*.

The main creation routine exists in three variants that mostly differ their number of parameters. For instance, DMDACreate2d has parameters along the *x*,*y* axes. However, DMDACreate1d has no parameter for the stencil type, since in 1D those are all the same, or for the process distribution.

30.1 Grid definition

A two-dimensional grid is created with DMDACreate2d (figure 30.1)

```
DMDACreate2d( communicator,
  x_boundary,y_boundary,
  stenciltype,
  gridx,gridy, procx,procy, dof,width,
  partitionx,partitiony,
  grid);
```

- Boundary type is a value of type DMBoundaryType. Values are:
  - DM_BOUNDARY_NONE
  - DM_BOUNDARY_GHOSTED,
  - DM_BOUNDARY_PERIODIC,
- The stencil type is of type DMStencilType, with values
  - DM_STENCIL_BOX,
  - DM_STENCIL_STAR.

  (See figure 30.1.)
- The *gridx*,*gridy* values are the global grid size. This can easily be set with commandline options `-da_grid_x/y/z`.
- The *procx*,*procy* variables are an explicit specification of the processor grid. Failing this specification, PETSc will try to find a distribution similar to the domain grid.

Figure 30.1 DMDACreate2d

```
#include "petscdmda.h"
PetscErrorCode  DMDACreate2d(MPI_Comm comm,
    DMBoundaryType bx,DMBoundaryType by,DMDAStencilType stencil_type,
    PetscInt M,PetscInt N,PetscInt m,PetscInt n,PetscInt dof,
    PetscInt s,const PetscInt lx[],const PetscInt ly[],
    DM *da)

Input Parameters

comm - MPI communicator
bx,by - type of ghost nodes: DM_BOUNDARY_NONE, DM_BOUNDARY_GHOSTED, DM_BOUNDARY_PERIODIC.
stencil_type - stencil type: DMDA_STENCIL_BOX or DMDA_STENCIL_STAR.
M,N - global dimension in each direction of
m,n - corresponding number of processors in each dimension (or PETSC_DECIDE)
dof - number of degrees of freedom per node
s - stencil width
lx, ly - arrays containing the number of
    nodes in each cell along the x and y coordinates, or NULL.

Output Parameter

da  -the resulting distributed array object
```

Figure 30.2 DMDAGetLocalInfo

```
#include "petscdmda.h"
PetscErrorCode  DMDAGetLocalInfo(DM da,DMDALocalInfo *info)
```

- *dof* indicates the number of 'degrees of freedom', where 1 corresponds to a scalar problem.
- *width* indicates the extent of the stencil: 1 for a 5-point stencil or more general a 2nd order stencil for 2nd order PDEs, 2 for 2nd order discretizations of a 4th order PDE, et cetera.
- *partitionx*,*partitiony* are arrays giving explicit partitionings of the grid over the processors, or PETSC_NULL for default distributions.

After you define a DM object, each process has a contiguous subdomain out of the total grid. You can query its size and location with DMDAGetCorners, or query that and all other information with DMDAGetLocalInfo (figure 30.2), which returns an DMDALocalInfo (figure 30.4) structure.

(A DMDALocalInfo (figure 30.4) struct is the same for 1/2/3 dimensions, so certain fields may not be applicable to your specific PDE.)

Using the fields in this structure, each process can now iterate over its own subdomain. For instance, the 'top left' corner of the owned subdomain is at `xs,ys` and the number of points is `xm,ym` (see figure 30.2), so we can iterate over the subdomain as:

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
  for (int i=info.xs; i<info.xs+info.xm; i++) {
    // actions on point i,j
  }
}
```

On each point of the domain, we describe the stencil at that point. First of all, we now have the information to compute the x, y coordinates of the domain points:

Figure 30.4 DMDALocalInfo

```
typedef struct {
  PetscInt         dim,dof,sw;
  PetscInt         mx,my,mz;    /* global number of grid points in each direction */
  PetscInt         xs,ys,zs;    /* starting point of this processor, excluding ghosts */
  PetscInt         xm,ym,zm;    /* number of grid points on this processor, excluding ghosts */
  PetscInt         gxs,gys,gzs;    /* starting point of this processor including ghosts */
  PetscInt         gxm,gym,gzm;    /* number of grid points on this processor including ghosts */
  DMBoundaryType   bx,by,bz; /* type of ghost nodes at boundary */
  DMDAStencilType  st;
  DM               da;
} DMDALocalInfo;

Fortran Notes - This should be declared as

   DMDALocalInfo :: info(DMDA_LOCAL_INFO_SIZE)

and the entries accessed via

   info(DMDA_LOCAL_INFO_DIM)
   info(DMDA_LOCAL_INFO_DOF) etc.

   The entries bx,by,bz, st, and da are not accessible from Fortran.
```

Figure 30.4 DMDALocalInfo

```
typedef struct {
  PetscInt         dim,dof,sw;
  PetscInt         mx,my,mz;    /* global number of grid points in each direction */
  PetscInt         xs,ys,zs;    /* starting point of this processor, excluding ghosts */
  PetscInt         xm,ym,zm;    /* number of grid points on this processor, excluding ghosts */
  PetscInt         gxs,gys,gzs;    /* starting point of this processor including ghosts */
  PetscInt         gxm,gym,gzm;    /* number of grid points on this processor including ghosts */
  DMBoundaryType   bx,by,bz; /* type of ghost nodes at boundary */
  DMDAStencilType  st;
  DM               da;
} DMDALocalInfo;

Fortran Notes - This should be declared as

   DMDALocalInfo :: info(DMDA_LOCAL_INFO_SIZE)

and the entries accessed via

   info(DMDA_LOCAL_INFO_DIM)
   info(DMDA_LOCAL_INFO_DOF) etc.

   The entries bx,by,bz, st, and da are not accessible from Fortran.
```

```
PetscReal
  hx = 1. / ( info.mx-1 ),
  hy = 1. / ( info.my-1 );
for (int j=info.ys; j<info.ys+info.ym; j++) {
  for (int i=info.xs; i<info.xs+info.xm; i++) {
    PetscReal x = i*hx, y = j*hy;
    ...
  }
}
```

30.2 Constructing a vector on a grid

A `DMDA` object is a description of a grid, so we now need to concern how to construct a linear system defined on that grid.

We start with vectors: we need a solution vector and a right-hand side. Here we have two options:

1. we can build a vector from scratch that has the right structure; or
2. we can use the fact that a grid object has a vector that can be extracted.

30.2.1 Create confirming vector

If we create a vector with `VecCreate` and `VecSetSizes`, it is easy to get the global size right, but the default partitioning will probably not be conformal to the grid distribution. Also, getting the indexing scheme right is not trivial.

First of all, the local size needs to be set explicitly, using information from the `DMDALocalInfo` object:

```
// dmrhs.c
Vec xy;
ierr = VecCreate(comm,&xy); CHKERRQ(ierr);
ierr = VecSetType(xy,VECMPI); CHKERRQ(ierr);
PetscInt nlocal = info.xm*info.ym, nglobal = info.mx*info.my;
ierr = VecSetSizes(xy,nlocal,nglobal); CHKERRQ(ierr);
```

After this, you don't use `VecSetValues`, but set elements directly in the raw array, obtained by *DMDAVecGetArray*:

```
PetscReal **xyarray;
DMDAVecGetArray(grid,xy,&xyarray); CHKERRQ(ierr);
for (int j=info.ys; j<info.ys+info.ym; j++) {
  for (int i=info.xs; i<info.xs+info.xm; i++) {
    PetscReal x = i*hx, y = j*hy;
    xyarray[j][i] = x*y;
  }
}
DMDAVecRestoreArray(grid,xy,&xyarray); CHKERRQ(ierr);
```

30.2.2 Extract vector from DMDA

30.2.3 Refinement

The routine *DMDASetRefinementFactor* can be activated with the options *-da_refine* or separately *-da_refine_x*/y/z for the directions.

30.3 Constructing a matrix on a grid

```
for (int j=info.ys; j<info.ys+info.ym; j++) {
  for (int i=info.xs; i<info.xs+info.xm; i++) {
    PetscReal x = i*hx, y = j*hy;
    ...
    // set the row, col, v values
    ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);CHKERRQ(ierr);
  }
}
```

Next, we express matrix row/column coordinates in terms of domain coordinates. The row number corresponds to the (i, j) pair:

```
MatStencil  row;
row.i = i; row.j = j;
```

For a 5-point stencil we need five column numbers, as well as five element values:

```
MatStencil col[5];
PetscScalar v[5];
PetscInt    ncols = 0;
/**** diagonal element ****/
col[ncols].i = i; col[ncols].j = j;
v[ncols++] = 4.;
```

The other 'legs' of the stencil need to be set conditionally: the connection to $(i-1, j)$ is missing on the top row of the domain, and the connection to $(i, j-1)$ is missing on the left column.

```
/* if not top row */
if (i>0) {
    col[ncols].j = j;   col[ncols].i = i-1;
    v[ncols++] = -1.;
}
/* if not left column */
if (j>0) {
    col[ncols].j = j-1; col[ncols].i = i;
    v[ncols++] = -1.;
}
```

Ditto for the connections to $(i+1, j)$ and $(i, j+1)$.

30.4 Vectors of a distributed array

A distributed array is similar to a distributed vector, so there are routines of extracting the values of the array in the form of a vector. This can be done in two ways: of ways. (The routines here actually pertain to the more general DM 'Data Management' object, but we will for now discuss them in the context of DMDA.)

1. You can create a 'global' vector, defined on the same communicator as the array, and which is disjointly partitioned in the same manner. This is done with DMCreateGlobalVector:

```
PetscErrorCode DMCreateGlobalVector(DM dm,Vec *vec)
```

2. You can create a 'local' vector, which is sequential and defined on PETSC_COMM_SELF, that has not only the points local to the process, but also the 'halo' region with the extent specified in the definition of the DMDACreate call. For this, use DMCreateLocalVector:

```
PetscErrorCode DMCreateLocalVector(DM dm,Vec *vec)
```

Values can be moved between local and global vectors by:

- DMGlobalToLocal: this establishes a local vector, including ghost/halo points from a disjointly distributed global vector. (For overlapping communication and computation, use DMGlobalToLocalBegin and DMGlobalToLocalEnd.)
- DMLocalToGlobal: this copies the disjoint parts of a local vector back into a global vector. (For overlapping communication and computation use DMLocalToGlobalBegin and DMLocalToGlobalEnd .)

30.5 Matrices of a distributed array

Once you have a grid, can create its associated matrix:

```
DMSetUp(grid);
DMCreateMatrix(grid,&A)
```

With this subdomain information you can then start to create the coefficient matrix:

```
DM grid;
PetscInt i_first,j_first,i_local,j_local;
DMDAGetCorners(grid,&i_first,&j_first,NULL,&i_local,&j_local,NULL);
for ( PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
  for ( PetscInt j_index=j_first; j_index<j_first+j_local; j_index++) {
  // construct coefficients for domain point (i_index,j_index)
  }
}
```

Note that indexing here is in terms of the grid, not in terms of the matrix.

For a simple example, consider 1-dimensional smoothing. From DMDAGetCorners we need only the parameters in i-direction:

```
// grid1d.c
PetscInt i_first,i_local;
ierr = DMDAGetCorners(grid,&i_first,NULL,NULL,&i_local,NULL,NULL);CHKERRQ(ierr);
for (PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
```

For the full source of this example, see section 30.6.2

We then use a single loop to set elements for the local range in i-direction:

```
MatStencil  row = {0},col[3] = {{0}};
PetscScalar v[3];
PetscInt    ncols = 0;
row.i = i_index;
col[ncols].i = i_index; v[ncols] = 2.;
ncols++;
if (i_index>0)          { col[ncols].i = i_index-1; v[ncols] = 1.; ncols++; }
if (i_index<i_global-1) { col[ncols].i = i_index+1; v[ncols] = 1.; ncols++; }
ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);CHKERRQ(ierr);
```

For the full source of this example, see section 30.6.2

30.6 Sources used in this chapter

30.6.1 Listing of code header

30.6.2 Listing of code examples/petsc/c/grid1d.c

```
#include <stdlib.h>
#include <stdio.h>
#include "petsc.h"
#include "petscdmda.h"

int main(int argc,char **argv) {

  PetscErrorCode ierr;
  ierr = PetscInitialize(&argc,&argv,0,0); CHKERRQ(ierr);

  MPI_Comm comm = MPI_COMM_WORLD;
  int nprocs,procno;
  MPI_Comm_size(comm,&nprocs);
  MPI_Comm_rank(comm,&procno);

  PetscInt i_global = 10*nprocs;

  /*
   * Create a 2d grid and a matrix on that grid.
   */
  DM grid;
  ierr = DMDACreate1d            // IN:
    ( comm,                      // collective on this communicator
      DM_BOUNDARY_NONE,          // no periodicity and such
      i_global,                  // global size 100x100; can be changed with options
      1,                         // degree of freedom per node
      1,                         // stencil width
      NULL,                      // arrays of local sizes in each direction
      &grid                      // OUT: resulting object
      ); CHKERRQ(ierr);
  ierr = DMSetUp(grid); CHKERRQ(ierr);

  Mat A;
  ierr = DMCreateMatrix(grid,&A); CHKERRQ(ierr);

  /*
   * Print out how the grid is distributed over processors
   */
  PetscInt i_first,i_local;
  ierr = DMDAGetCorners(grid,&i_first,NULL,NULL,&i_local,NULL,NULL);CHKERRQ(ierr);
  /* ierr = PetscSynchronizedPrintf */
  /*   (comm, */
  /*    "[%d] Local part = %d-%d x %d-%d\n", */
  /*    procno,info.xs,info.xs+info.xm,info.ys,info.ys+info.ym); CHKERRQ(ierr); */
  /* ierr = PetscSynchronizedFlush(comm,stdout); CHKERRQ(ierr); */
```

```
/*
 * Fill in the elements of the matrix
 */
for (PetscInt i_index=i_first; i_index<i_first+i_local; i_index++) {
  MatStencil  row = {0},col[3] = {{0}};
  PetscScalar v[3];
  PetscInt    ncols = 0;
  row.i = i_index;
  col[ncols].i = i_index; v[ncols] = 2.;
  ncols++;
  if (i_index>0)          { col[ncols].i = i_index-1; v[ncols] = 1.; ncols++; }
  if (i_index<i_global-1) { col[ncols].i = i_index+1; v[ncols] = 1.; ncols++; }
  ierr = MatSetValuesStencil(A,1,&row,ncols,col,v,INSERT_VALUES);CHKERRQ(ierr);
}

ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);

/*
 * Create vectors on the grid
 */
Vec x,y;
ierr = DMCreateGlobalVector(grid,&x); CHKERRQ(ierr);
ierr = VecDuplicate(x,&y); CHKERRQ(ierr);

/*
 * Set vector values: first locally, then global
 */
PetscReal one = 1.;
{
  Vec xlocal;
  ierr = DMCreateLocalVector(grid,&xlocal); CHKERRQ(ierr);
  ierr = VecSet(xlocal,one); CHKERRQ(ierr);
  ierr = DMLocalToGlobalBegin(grid,xlocal,INSERT_VALUES,x); CHKERRQ(ierr);
  ierr = DMLocalToGlobalEnd(grid,xlocal,INSERT_VALUES,x); CHKERRQ(ierr);
  ierr = VecDestroy(&xlocal); CHKERRQ(ierr);
}

/*
 * Solve a linear system on the grid
 */
KSP solver;
ierr = KSPCreate(comm,&solver); CHKERRQ(ierr);
ierr = KSPSetType(solver,KSPBCGS); CHKERRQ(ierr);
ierr = KSPSetOperators(solver,A,A); CHKERRQ(ierr);
ierr = KSPSetFromOptions(solver); CHKERRQ(ierr);
ierr = KSPSolve(solver,x,y); CHKERRQ(ierr);

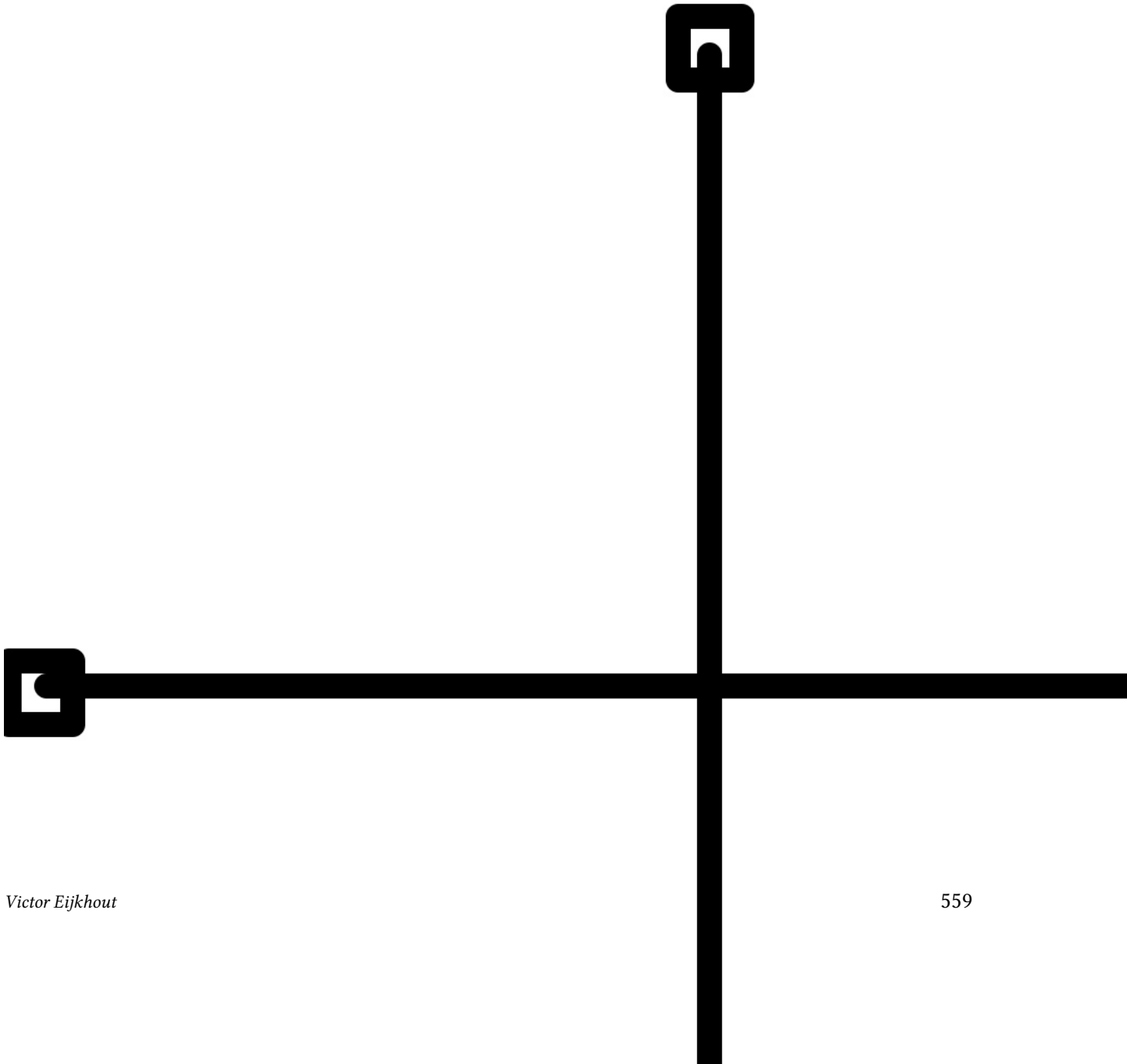
/*
 * Report on success of the solver, or lack thereof
 */
{
  PetscInt its; KSPConvergedReason reason;
```

```
    ierr = KSPGetConvergedReason(solver,&reason);
    ierr = KSPGetIterationNumber(solver,&its); CHKERRQ(ierr);
    if (reason<0) {
      PetscPrintf(comm,"Failure to converge after %d iterations; reason %s\n",
  its,KSPConvergedReasons[reason]);
    } else {
      PetscPrintf(comm,"Number of iterations to convergence: %d\n",its);
    }
  }

  /*
   * Clean up
   */
  ierr = KSPDestroy(&solver); CHKERRQ(ierr);
  ierr = VecDestroy(&x); CHKERRQ(ierr);
  ierr = VecDestroy(&y); CHKERRQ(ierr);
  ierr = MatDestroy(&A); CHKERRQ(ierr);
  ierr = DMDestroy(&grid); CHKERRQ(ierr);

  return PetscFinalize();
}
```

Star stencil



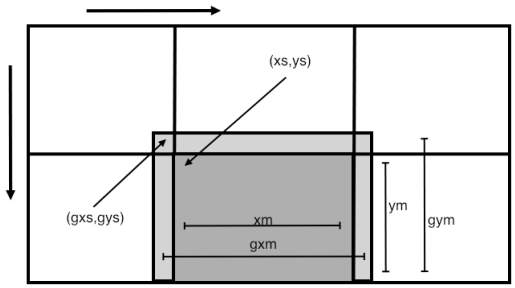


Figure 30.2: Illustration of various fields of the DMDALocalInfo structure

Chapter 31

Finite Elements support

```
PetscDSSetJacobian
Set the pointwise Jacobian function for given test and basis fields
Synopsis

#include "petscds.h"
PetscErrorCode PetscDSSetJacobian(PetscDS prob, PetscInt f, PetscInt g,
    void (*g0)(PetscInt dim, PetscInt Nf, PetscInt NfAux,
        const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar u[], const
    PetscScalar u_t[], const PetscScalar u_x[],
        const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar a[], const
    PetscScalar a_t[], const PetscScalar a_x[],
        PetscReal t, PetscReal u_tShift, const PetscReal x[], PetscInt numConstants,
    const PetscScalar constants[], PetscScalar g0[]),
    void (*g1)(PetscInt dim, PetscInt Nf, PetscInt NfAux,
        const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar u[], const
    PetscScalar u_t[], const PetscScalar u_x[],
        const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar a[], const
    PetscScalar a_t[], const PetscScalar a_x[],
        PetscReal t, PetscReal u_tShift, const PetscReal x[], PetscInt numConstants,
    const PetscScalar constants[], PetscScalar g1[]),
    void (*g2)(PetscInt dim, PetscInt Nf, PetscInt NfAux,
        const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar u[], const
    PetscScalar u_t[], const PetscScalar u_x[],
        const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar a[], const
    PetscScalar a_t[], const PetscScalar a_x[],
        PetscReal t, PetscReal u_tShift, const PetscReal x[], PetscInt numConstants,
    const PetscScalar constants[], PetscScalar g2[]),
    void (*g3)(PetscInt dim, PetscInt Nf, PetscInt NfAux,
        const PetscInt uOff[], const PetscInt uOff_x[], const PetscScalar u[], const
    PetscScalar u_t[], const PetscScalar u_x[],
        const PetscInt aOff[], const PetscInt aOff_x[], const PetscScalar a[], const
    PetscScalar a_t[], const PetscScalar a_x[],
        PetscReal t, PetscReal u_tShift, const PetscReal x[], PetscInt numConstants,
    const PetscScalar constants[], PetscScalar g3[])
)
```

$$\int_\Omega \phi g_0(u, u_t, \nabla u, x, t)\psi + \phi \vec{g}_1(u, u_t, \nabla u, x, t)\nabla\psi + \nabla\phi \cdot \vec{g}_2(u, u_t, \nabla u, x, t)\psi + \nabla\phi \cdot \overleftrightarrow{g}_3(u, u_t, \nabla u, x, t) \cdot \nabla\psi$$

31.1 Sources used in this chapter

31.1.1 Listing of code header

Chapter 32

PETSc solvers

Probably the most important activity in PETSc is solving a linear system. This is done through a solver object: an object of the class `KSP`. (This stands for Krylov SPace solver.) The solution routine `KSPSolve` takes a matrix and a right-hand-side and gives a solution; however, before you can call this some amount of setup is needed.

There two very different ways of solving a linear system: through a direct method, essentially a variant of Gaussian elimination; or through an iterative method that makes successive approximations to the solution. In PETSc there are only iterative methods. We will show how to achieve direct methods later. The default linear system solver in PETSc is fully parallel, and will work on many linear systems, but there are many settings and customizations to tailor the solver to your specific problem.

32.1 KSP: linear system solvers

32.1.1 Math background

Many scientific applications boil down to the solution of a system of linear equations at some point:

$$?_x : \ Ax = b$$

The elementary textbook way of solving this is through an *LU factorization*, also known as *Gaussian elimination*:

$$LU \leftarrow A, \qquad Lz = b, \qquad Ux = z.$$

While PETSc has support for this, its basic design is geared towards so-called iterative solution methods. Instead of directly computing the solution to the system, they compute a sequence of approximations that, with luck, converges to the true solution:

while not converged
$x_{i+1} \leftarrow f(x_i)$

The interesting thing about iterative methods is that the iterative step only involves the *matrix-vector product*:

Figure 32.1 `KSPCreate`

```
C:
PetscErrorCode KSPCreate(MPI_Comm comm,KSP *v);

Python:
ksp = PETSc.KSP()
ksp.create()
# or:
ksp = PETSc.KSP().create()
```

while not converged
$r_i = Ax_i - b$
$x_{i+1} \leftarrow f(r_i)$

This *residual* is also crucial in determining whether to stop the iteration: since we (clearly) can not measure the distance to the true solution, we use the size of the residual as a proxy measurement.

The remaining point to know is that iterative methods feature a *preconditioner*. Mathematically this is equivalent to transforming the linear system to

$$M^{-1}Ax = M^{-1}b$$

so conceivably we could iterate on the transformed matrix and right-hand side. However, in practice we apply the preconditioner in each iteration:

while not converged
$r_i = Ax_i - b$
$z_i = M^{-1}r_i$
$x_{i+1} \leftarrow f(z_i)$

In this schematic presentation we have left the nature of the $f()$ update function unspecified. Here, many possibilities exist; the primary choice here is of the iterative method type, such as 'conjugate gradients', 'generalized minimum residual', or 'bi-conjugate gradients stabilized'. (We will go into direct solvers in section 32.2.)

32.1.2 Solver objects

First we create a KSP object, which contains the coefficient matrix, and various parameters such as the desired accuracy, as well as method specific parameters: `KSPCreate` (figure 32.1).

After this, the basic scenario is:

```
Vec rhs,sol;
KSP solver;
KSPCreate(comm,&solver);
KSPSetOperators(solver,A,A);
KSPSetFromOptions(solver);
KSPSolve(solver,rhs,sol);
KSPDestroy(&solver);
```

Figure 32.2 KSPSetTolerances

```
#include "petscksp.h"
PetscErrorCode  KSPSetTolerances
   (KSP ksp,PetscReal rtol,PetscReal abstol,PetscReal dtol,PetscInt maxits)

Logically Collective on ksp

Input Parameters:
ksp- the Krylov subspace context
rtol- the relative convergence tolerance, relative decrease in the
(possibly preconditioned) residual norm
abstol- the absolute convergence tolerance absolute size of the
(possibly preconditioned) residual norm
dtol- the divergence tolerance, amount (possibly preconditioned)
residual norm can increase before KSPConvergedDefault() concludes that
the method is diverging
maxits- maximum number of iterations to use

Options Database Keys
-ksp_atol <abstol>- Sets abstol
-ksp_rtol <rtol>- Sets rtol
-ksp_divtol <dtol>- Sets dtol
-ksp_max_it <maxits>- Sets maxits
```

using various default settings. The vectors and the matrix have to be conformly partitioned. The `KSPSetOperators` call takes two operators: one is the actual coefficient matrix, and the second the one that the preconditioner is derived from. In some cases it makes sense to specify a different matrix here. The call `KSPSetFromOptions` can cover almost all of the settings discussed next.

KSP objects have many options to control them, so it is convenient to call `KSPView` (or use the commandline option *-ksp_view*) to get a listing of all the settings.

32.1.3 Tolerances

Since neither solution nor solution speed is guaranteed, an iterative solver is subject to some tolerances:

- a relative tolerance for when the residual has been reduced enough;
- an absolute tolerance for when the residual is objectively small;
- a divergence tolerance that stops the iteration if the residual grows by too much; and
- a bound on the number of iterations, regardless any progress the process may still be making.

These tolerances are set with `KSPSetTolerances` (figure 32.2), or options *-ksp_atol*, *-ksp_rtol*, *-ksp_divtol*, *-ksp_max_it*. Specify to `PETSC_DEFAULT` to leave a value unaltered.

In the next section we will see how you can determine which of these tolerances caused the solver to stop.

32.1.4 Why did my solver stop? Did it work?

On return of the `KSPSolve` routine there is no guarantee that the system was successfully solved. Therefore, you need to invoke `KSPGetConvergedReason` (figure 32.3) to get a `KSPConvergedReason` parameter that indicates what state the solver stopped in:

Figure 32.3 KSPGetConvergedReason

```
C:
PetscErrorCode KSPGetConvergedReason
   (KSP ksp,KSPConvergedReason *reason)
Not Collective

Input Parameter
ksp -the KSP context

Output Parameter
reason -negative value indicates diverged, positive value converged,
see KSPConvergedReason

Python:
r = KSP.getConvergedReason(self)
where r in PETSc.KSP.ConvergedReason
```

- The iteration can have successfully converged; this corresponds to *reason*> 0;
- the iteration can have diverged, or otherwise failed: *reason*< 0;
- or the iteration may have stopped at the maximum number of iterations while still making progress; *reason*$= 0$.

For more detail, `KSPConvergenceReasonView` (before version 3.14: `KSPReasonView`) can print out the reason in readable form; for instance

```
KSPConvergenceReasonView(solver,PETSC_VIEWER_STDOUT_WORLD);
// before 3.14:
KSPReasonView(solver,PETSC_VIEWER_STDOUT_WORLD);
```

(This can also be activated with the *-ksp_converged_reason* commandline option.)

In case of successful convergence, you can use `KSPGetIterationNumber` to report how many iterations were taken.

The following snippet analyzes the status of a `KSP` object that has stopped iterating:

```
// shellvector.c
PetscInt its; KSPConvergedReason reason;
Vec Res; PetscReal norm;
ierr = KSPGetConvergedReason(Solve,&reason); CHKERRQ(ierr);
ierr = KSPReasonView(Solve,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
if (reason<0) {
  PetscPrintf(comm,"Failure to converge: reason=%d\n",reason);
} else {
  ierr = KSPGetIterationNumber(Solve,&its); CHKERRQ(ierr);
  PetscPrintf(comm,"Number of iterations: %d\n",its);
}
```

For the full source of this example, see section 32.4.2

32.1.5 Choice of iterator

There are many iterative methods, and it may take a few function calls to fully specify them. The basic routine is `KSPSetType` (figure 32.4), or use the option *-ksp_type*.

Figure 32.4 KSPSetType

```
#include "petscksp.h"
PetscErrorCode  KSPSetType(KSP ksp, KSPType type)

Logically Collective on ksp

Input Parameters:
ksp : the Krylov space context
type : a known method
```

Figure 32.5 KSPMatSolve

```
PetscErrorCode KSPMatSolve(KSP ksp, Mat B, Mat X)

Input Parameters
ksp - iterative context
B - block of right-hand sides

Output Parameter
X - block of solutions
```

Here are some values (the full list is in *petscksp.h*:

- KSPCG: only for symmetric positive definite systems. It has a cost of both work and storage that is constant in the number of iterations.
  There are variants such as KSPPIPECG that are mathematically equivalent, but possibly higher performing at large scale.
- KSPGMRES: a minimization method that works for nonsymmetric and indefinite systems. However, to satisfy this theoretical property it needs to store the full residual history to orthogonalize each compute residual to, implying that storage is linear, and work quadratic, in the number of iterations. For this reason, GMRES is always used in a truncated variant, that regularly restarts the orthogonalization. The restart length can be set with the routine KSPGMRESSetRestart or the option *-ksp_gmres_restart*.
- KSPBCGS: a quasi-minimization method; uses less memory than GMRES.

Depending on the iterative method, there can be several routines to tune its workings. Especially if you're still experimenting with what method to choose, it may be more convenient to specify these choices through commandline options, rather than explicitly coded routines. In that case, a single call to KSPSetFromOptions is enough to incorporate those.

32.1.6 Multiple right-hand sides

For the case of multiple right-hand sides, use *KSPMatSolve* (figure 32.5).

32.1.7 Preconditioners

Another part of an iterative solver is the *preconditioner*. The mathematical background of this is given in section 32.1.1. The preconditioner acts to make the coefficient matrix better conditioned, which will improve the convergence speed; it can even be that without a suitable preconditioner a solver will not converge at all.

32.1.7.1 Background

The mathematical requirement that the preconditioner M satisfy $M \approx A$ can take two forms:

1. We form an explicit approximation to A^{-1}; this is known as a *sparse approximate inverse*.
2. We form an operator M (often given in factored or other implicit) form, such that $M \approx A$, and solving a system $Mx = y$ for x can be done relatively quickly.

In deciding on a preconditioner, we now have to balance the following factors.

1. What is the cost of constructing the preconditioner? This should not be more than the gain in solution time of the iterative method.
2. What is the cost per iteration of applying the preconditioner? There is clearly no point in using a preconditioner that decreases the number of iterations by a certain amount, but increases the cost per iteration much more.
3. Many preconditioners have parameter settings that make these considerations even more complicated: low parameter values may give a preconditioner that is cheaply to apply but does not improve convergence much, while large parameter values make the application more costly but decrease the number of iterations.

32.1.7.2 Usage

Unlike most of the other PETSc object types, a `PC` object is typically not explicitly created. Instead, it is created as part of the `KSP` object, and can be retrieved from it.

```
PC prec;
KSPGetPC(solver,&prec);
PCSetType(prec,PCILU);
```

Beyond setting the type of the preconditioner, there are various type-specific routines for setting various parameters. Some of these can get quite tedious, and it is more convenient to set them through commandline options.

32.1.7.3 Types

Method	PCType	Options Database Name
Jacobi	PCJACOBI	jacobi
Block Jacobi	PCBJACOBI	bjacobi
SOR (and SSOR)	PCSOR	sor
SOR with Eisenstat trick	PCEISENSTAT	eisenstat
Incomplete Cholesky	PCICC	icc
Incomplete LU	PCILU	ilu
Additive Schwarz	PCASM	asm
Generalized Additive Schwarz	PCGASM	gasm
Algebraic Multigrid	PCGAMG	gamg
Balancing Domain Decomposition by Constraints Linear solver	PCBDDC	bddc
Use iterative method	PCKSP	ksp
Combination of preconditioners	PCCOMPOSITE	composite
LU	PCLU	lu
Cholesky	PCCHOLESKY	cholesky
No preconditioning	PCNONE	none
Shell for user-defined PC	PCSHELL	shell

Here are some of the available preconditioner types.

The *Hypre* package (which needs to be installed during configuration time) contains itself several preconditioners. In your code, you can set the preconditioner to `PCHYPRE`, and use `PCHYPRESetType` to one of: euclid, pilut, parasails, boomeramg, ams, ads. However, since these preconditioners themselves have options, it is usually more convenient to use commandline options:

```
-pc_type hypre -pc_hypre_type xxxx
```

32.1.7.3.1 Sparse approximate inverses

The inverse of a sparse matrix (at least, those from PDEs) is typically dense. Therefore, we aim to construct a *sparse approximate inverse*.

PETSc offers two such preconditioners, both of which require an external package.

- `PCSPAI`. This is a preconditioner that can only be used in single-processor runs, or as local solver in a block preconditioner; section 32.1.7.3.3.
- As part of the `PCHYPRE` package, the parallel variant *parasails* is available.
  ```
  -pc_type hypre -pc_hypre_type parasails
  ```

32.1.7.3.2 Incomplete factorizations

The *LU* factorization of a matrix stemming from PDEs problems has several practical problems:

- It takes (considerably) more storage space than the coefficient matrix, and
- it correspondingly takes more time to apply.

For instance, for a three-dimensional PDE in N variables, the coefficient matrix can take storage space $7N$, while the *LU* factorization takes $O(N^{5/3})$.

For this reason, often incompletely *LU* factorizations are popular.

- PETSc has natively a PCILU type, but this can only be used sequentially. This may sound like a limitation, but in parallel it can still be used as the subdomain solver in a block methods; section 32.1.7.3.3.
- As part of *Hypre*, *pilut* is a parallel ILU.

There are many options for the ILU type, such as *PCFactorSetLevels* (option *-pc_factor_levels*), which sets the number of levels of fill-in allowed.

32.1.7.3.3 Block methods Certain preconditioners seem almost intrinsically sequential. For instance, an ILU solution is sequential between the variables. There is a modest amount of parallelism, but that is hard to explore.

Taking a step back, one of the problems with parallel preconditioners lies in the cross-process connections in the matrix. If only those were not present, we could solve the linear system on each process independently. Well, since a preconditioner is an approximate solution to begin with, ignoring those connections only introduces an extra degree of approxomaticity.

There are two preconditioners that operate on this notion:

- PCBJACOBI: block Jacobi. Here each process solves locally the system consisting of the matrix coefficients that couple the local variables. In effect, each process solves an independent system on a subdomain.
  The next question is then what solver is used on the subdomains. Here any preconditioner can be used, in particular the ones that only existed in a sequential version. Specifying all this in code gets tedious, and it is usually easier to specify such a complicated solver through commandline options:
  ```
  -pc_type jacobi -sub_ksp_type preonly \
      -sub_pc_type ilu -sub_pc_factor_levels 1
  ```
  (Note that this also talks about a `sub_ksp`: the subdomain solver is in fact a KSP object. By setting its type to *preonly* we state that the solver should consist of solely applying its preconditioner.)
  The block Jacobi preconditioner can asympotically only speed up the system solution by a factor relating to the number of subdomains, but in practice it can be quite valuable.
- PCASM: additive Schwarz method. Here each process solves locally a slightly larger system, based on the local variables, and one (or a few) levels of connections to neighboring processes. In effect, the processes solve system on overlapping subdomains. This preconditioner can asympotically reduce the number of iterations to $O(1)$, but that requires exact solutions on the subdomains, and in practice it may not happen anyway.

Figure 32.1 illustrates these preconditioners both in matrix and subdomain terms.

32.1.7.3.4 Multigrid preconditioners

- There is a native Algebraic MultiGrid (AMG) type: PCGAMG;
- the external packages *Hypre* and *ML* have AMG methods.
- There is a general **MG!** (**MG!**) type: PCMG.

32.1.7.3.5 Field split preconditioners For background refer to section 29.4.7.

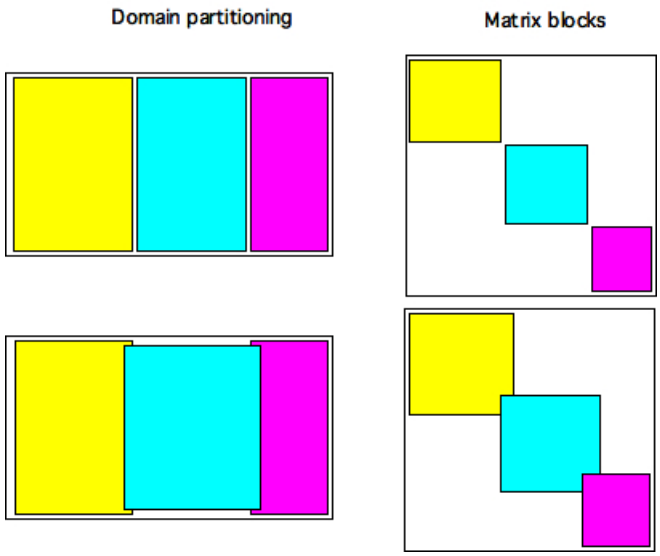


Figure 32.1: Illustration of block Jacobi and Additive Schwarz preconditioners

32.1.8 Customization: monitoring and convergence tests

PETSc solvers can do various *callback*s to user functions.

32.1.8.0.1 Shell preconditioners You already saw that, in an iterative methods, the coefficient matrix can be given operationally as a *shell matrix*; section 29.4.6. Similarly, the preconditioner matrix can be specified operationally by specifying type `PCSHELL`.

This needs specification of the application routine through `PCShellSetApply`:

```
PCShellSetApply(PC pc,PetscErrorCode (*apply)(PC,Vec,Vec));
```

and probably specification of a context pointer through `PCShellSetContext`:

```
PCShellSetContext(PC pc,void *ctx);
```

The application function then retrieves this context with `PCShellGetContext`:

```
PCShellGetContext(PC pc,void **ctx);
```

If the shell preconditioner requires setup, a routine for this can be specified with `PCShellSetSetUp`:

```
PCShellSetSetUp(PC pc,PetscErrorCode (*setup)(PC));
```

32.1.8.0.2 Combining preconditioners It is possible to combine preconditioners with `PCCOMPOSITE`

```
PCSetType(pc,PCCOMPOSITE);
PCCompositeAddPC(pc,type1);
PCCompositeAddPC(pc,type2);
```

By default, the preconditioners are applied additively; for multiplicative application

```
PCCompositeSetType(PC pc,PCCompositeType PC_COMPOSITE_MULTIPLICATIVE);
```

32.1.8.1 Convergence tests

For instance, you can set your own convergence test with `KSPSetConvergenceTest`.

```
KSPSetConvergenceTest
   (KSP ksp,
    PetscErrorCode (*test)(
        KSP ksp,PetscInt it,PetscReal rnorm,
        KSPConvergedReason *reason,void *ctx),
    void *ctx,PetscErrorCode (*destroy)(void *ctx));
```

This routines accepts

- the custom stopping test function,
- a 'context' void pointer to pass information to the tester, and
- optionally a custom destructor for the context information.

By default, PETSc behaves as if this function has been called with `KSPConvergedDefault` as argument.

32.1.8.2 Convergence monitoring

There is also a callback for monitoring each iteration. It can be set with `KSPMonitorSet`.

```
KSPMonitorSet
   (KSP ksp,
    PetscErrorCode (*mon)(
        KSP ksp,PetscInt it,PetscReal rnorm,void *ctx),
    void *ctx,PetscErrorCode (*mondestroy)(void**));
```

By default no monitor is set, meaning that the iteration process runs without output. The option `-ksp_monitor` activates printing a norm of the residual. This corresponds to setting `KSPMonitorDefault` as the monitor.

This actually outputs the 'preconditined norm' of the residual, which is not the L2 norm, but the square root of $r^t M^{-1} r$, a quantity that is computed in the course of the iteration process. Specifying `KSPMonitorTrueResidualNorm` (with corresponding option `-ksp_monitor_true_residual`) as the monitor prints the actual norm $\sqrt{r^t r}$. However, to compute this involves extra computation, since this quantity is not normally computed.

32.1.8.3 Auxiliary routines

`KSPGetSolution` `KSPGetRhs` `KSPBuildSolution` `KSPBuildResidual`

```
KSPGetSolution(KSP ksp,Vec *x);
KSPGetRhs(KSP ksp,Vec *rhs);
KSPBuildSolution(KSP ksp,Vec w,Vec *v);
KSPBuildResidual(KSP ksp,Vec t,Vec w,Vec *v);
```

Figure 32.6 **KSPSetFromOptions**

```
Synopsis

#include "petscksp.h"
PetscErrorCode  KSPSetFromOptions(KSP ksp)

Collective on ksp

Input Parameters
ksp - the Krylov space context
```

32.2 Direct solvers

PETSc has some support for direct solvers, that is, variants of LU decomposition. In a sequential context, the `PCLU` preconditioner can be use for this: a direct solver is equivalent to an iterative method that stops after one preconditioner application. This can be forced by specifying a KSP type of `KSPPREONLY`.

Distributed direct solvers are more complicated. PETSc does not have this implemented in its basic code, but it becomes available by configuring PETSc with the *scalapack* library.

You need to specify which package provides the LU factorization:

```
PCFactorSetMatSolverType(pc, <solvertype> )
```

where solvertype can be mumps, superlu, umfpack, or a number of others. Note that availability of these packages depends on how PETSc was installed on your system.

32.3 Control through command line options

From the above you may get the impression that there are lots of calls to be made to set up a PETSc linear system and solver. And what if you want to experiment with different solvers, does that mean that you have to edit a whole bunch of code? Fortunately, there is an easier way to do things. If you call the routine `KSPSetFromOptions` (figure 32.6) with the *solver* as argument, PETSc will look at your command line options and take those into account in defining the solver. Thus, you can either omit setting options in your source code, or use this as a way of quickly experimenting with different possibilities. Example:

```
myprogram -ksp_max_it 200 \
    -ksp_type gmres -ksp_type_gmres_restart 20  \
    -pc_type ilu -pc_type_ilu_levels 3
```

32.4 Sources used in this chapter

32.4.1 Listing of code header

32.4.2 Listing of code code/petsc/c

Chapter 33

PETSC nonlinear solvers

33.1 Nonlinear systems

Nonlinear system solving means finding the zero of a general nonlinear function, that is:

$$\underset{x}{?} : \ f(x) = 0$$

with $f : \ \mathbb{R}^n - \mathbb{R}^n$. In the special case of a linear function,

$$f(x) = Ax - b,$$

we solve this by any of the methods in chapter 32.

The general case can be solved by a number of methods, foremost *Newton's method*, which iterates

$$x_{n+1} = x_n - F(x_n)^{-1} f(x_n)$$

where F is the *Hessian* $F_{ij} = \partial f_i / \partial x_j$.

You see that you need to specify two functions that are dependent on your specific problem: the objective function itself, and its Hessian.

33.1.1 Basic setup

The PETSc nonlinear solver object is of type `SNES`: 'simple nonlinear equation solver'. As with linear solvers, we create this solver on a communicator, set its type, incorporate options, and call the solution routine `SNESSolve` (figure 33.1):

```
Vec value_vector,solution_vector;
/* vector creation code missing */
SNES solver;
SNESCreate( comm,&solver );
SNESSetFunction( solver,value_vector,formfunction, NULL );
SNESSetFromOptions( solver );
SNESSolve( solver,NULL,solution_vector );
```

The function has the type

Figure 33.1 **SNESSolve**

```
#include "petscsnes.h"
PetscErrorCode  SNESSolve(SNES snes,Vec b,Vec x)

Collective on SNES

Input Parameters
snes - the SNES context
b    - the constant part of the equation F(x) = b, or NULL to use zero.
x    - the solution vector.
```

```
PetscErrorCode formfunction(SNES,Vec,Vec,void*)
```

where the parameters are:

- the solver object, so that you can access to its internal parameters
- the x value at which to evaluate the function
- the result vector $f(x)$ for the given input
- a context pointer for further application-specific information.

Example:

```
PetscErrorCode evaluation_function( SNES solver,Vec x,Vec fx, void *ctx ) {
  const PetscReal *x_array;
  PetscReal *fx_array;
  VecGetArrayRead(fx,&fx_array);
  VecGetArray(x,&x_array);
  for (int i=0; i<localsize; i++)
    fx_array[i] = pointfunction( x_array[i] );
  VecRestoreArrayRead(fx,&fx_array);
  VecRestoreArray(x,&x_array);
};
```

Comparing the above to the introductory description you see that the Hessian is not specified here. An analytic Hessian can be dispensed with if you instruct PETSc to approximate it by finite differences:

$$H(x)y \approx \frac{f(x+hy) - f(x)}{h}$$

with h some finite diference. The commandline option `-snes_fd` forces the use of this finite difference approximation. However, it may lead to a large number of function evaluations. The option `-snes_fd_color` applies a coloring to the variables, leading to a drastic reduction in the number of function evaluations.

If you can form the analytic Jacobian / Hessian, you can specify it with *SNESSetJacobian* (figure 33.2), where the Jacobian is a function of type *SNESJacobianFunction* (figure 33.3).

Specifying the Jacobian:

```
Mat J;
ierr = MatCreate(comm,&J); CHKERRQ(ierr);
ierr = MatSetType(J,MATSEQDENSE); CHKERRQ(ierr);
ierr = MatSetSizes(J,n,n,N,N); CHKERRQ(ierr);
ierr = MatSetUp(J); CHKERRQ(ierr);
ierr = SNESSetJacobian(solver,J,J,&Jacobian,NULL); CHKERRQ(ierr);
```

Figure 33.2 **SNESSetJacobian**

```
#include "petscsnes.h"
PetscErrorCode  SNESSetJacobian(SNES snes,Mat Amat,Mat Pmat,PetscErrorCode (*J)(SNES,Vec,Mat,Mat,void*),void *ctx)

Logically Collective on SNES

Input Parameters
snes - the SNES context
Amat - the matrix that defines the (approximate) Jacobian
Pmat - the matrix to be used in constructing the preconditioner, usually the same as Amat.
J    - Jacobian evaluation routine (if NULL then SNES retains any previously set value)
ctx  - [optional] user-defined context for private data for the Jacobian evaluation routine
```

Figure 33.3 **SNESJacobianFunction**

```
#include "petscsnes.h"
PetscErrorCode SNESJacobianFunction(SNES snes,Vec x,Mat Amat,Mat Pmat,void *ctx);

Collective on snes

Input Parameters
x   - input vector, the Jacobian is to be computed at this value
ctx - [optional] user-defined Jacobian context

Output Parameters
Amat - the matrix that defines the (approximate) Jacobian
Pmat - the matrix to be used in constructing the preconditioner, usually the same as Amat.
```

33.2 Time-stepping

For cases

$$u_t = G(t,u)$$

call *TSSetRHSFunction*.

```
#include "petscts.h"
PetscErrorCode TSSetRHSFunction
   (TS ts,Vec r,
    PetscErrorCode (*f)(TS,PetscReal,Vec,Vec,void*),
    void *ctx);
```

For implicit cases

$$F(t,u,u_t) = 0$$

call *TSSetIFunction*

```
#include "petscts.h"
PetscErrorCode TSSetIFunction
   (TS ts,Vec r,TSIFunction f,void *ctx)
```

33.3 Sources used in this chapter

33.3.1 Listing of code header

Chapter 34

PETSc tools

34.1 Error checking and debugging

34.1.1 Debug mode

During installation (see section 28.3), there is an option of turning on debug mode. An installation with debug turned on:

- Does more runtime checks on numerics, or array indices;
- Does a memory analysis when you insert the `CHKMEMQ` macro (section 34.1.3);
- Has the macro *PETSC_USE_DEBUG* set to 1.

34.1.2 Error codes

PETSc performs a good amount of runtime error checking. Some of this is for internal consistency, but it can also detect certain mathematical errors. To facilitate error reporting, the following scheme is used.

1. Every PETSc routine is a function returning a parameter of type `PetscErrorCode`.
2. For a good traceback, surround the executable part of any subprogram with `PetscFunctionBegin` and `PetscFunctionReturn`, where the latter has the return value as parameter.
3. Calling the macro `CHKERRQ` on the error code will cause an error to be printed and the current routine to be terminated. Recursively this gives a traceback of where the error occurred.

```
PetscErrorCode ierr;
ierr = AnyPetscRoutine( arguments ); CHKERRQ(ierr);
```

4. You can effect your own error return by using `SETERRQ` (figure 34.1) `SETERRQ1` (figure 34.1), `SETERRQ2` (figure 34.1).

Fortran note. In the main program, use `CHKERRA` and `SETERRA`. Also beware that these error 'commands' are macros, and after expansion may interfere with *Fortran line length*, so they should only be used in `.F90` files.

Example. We write a routine that sets an error:

```
// backtrace.c
PetscErrorCode this_function_bombs() {
  PetscFunctionBegin;
  SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this");
```

Figure 34.1 **SETERRQ**

```
#include <petscsys.h>
PetscErrorCode SETERRQ (MPI_Comm comm,PetscErrorCode ierr,char *message)
PetscErrorCode SETERRQ1(MPI_Comm comm,PetscErrorCode ierr,char *formatmessage,arg1)
PetscErrorCode SETERRQ2(MPI_Comm comm,PetscErrorCode ierr,char *formatmessage,arg1,arg2)
PetscErrorCode SETERRQ3(MPI_Comm comm,PetscErrorCode ierr,char *formatmessage,arg1,arg2,arg3)

Input Parameters:
comm - A communicator, so that the error can be collective
ierr - nonzero error code, see the list of standard error codes in include/petscerror.h
message - error message in the printf format
arg1,arg2,arg3 - argument (for example an integer, string or double)
```

```
  PetscFunctionReturn(0);
}
```

For the full source of this example, see section 34.6.2

Running this gives, in process zero, the output

```
[0]PETSC ERROR: We cannot go on like this
[0]PETSC ERROR: See https://www.mcs.anl.gov/petsc/documentation/faq.html for trouble shooting.
[0]PETSC ERROR: Petsc Release Version 3.12.2, Nov, 22, 2019
[0]PETSC ERROR: backtrace on a [computer name]
[0]PETSC ERROR: Configure options [all options]
[0]PETSC ERROR: #1 this_function_bombs() line 20 in backtrace.c
[0]PETSC ERROR: #2 main() line 30 in backtrace.c
```

Fortran note. In Fortran the backtrace is not quite as elegant.

```
// backtrace.F90
Subroutine this_function_bombs(ierr)
  implicit none
  integer,intent(out) :: ierr

  SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this")
  ierr = -1

end Subroutine this_function_bombs
```

For the full source of this example, see section 34.6.3

```
[0]PETSC ERROR: ----- Error Message --------------------------------
[0]PETSC ERROR: We cannot go on like this
[....]
[0]PETSC ERROR: #1 User provided function() line 0 in User file
```

Remark 21 *In this example, the use of `PETSC_COMM_SELF` indicates that this error is individually generated on a process; use `PETSC_COMM_WORLD` only if the same error would be detected everywhere.*

Exercise 34.1. Look up the definition of `SETERRQ1`. Write a routine to compute square roots that is used as follows:

```
x = 1.5; ierr = square_root(x,&rootx); CHKERRQ(ierr);
PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",x,rootx);
x = -2.6; ierr = square_root(x,&rootx); CHKERRQ(ierr);
PetscPrintf(PETSC_COMM_WORLD,"Root of %f is %f\n",x,rootx);
```

This should give as output:

```
Root of 1.500000 is 1.224745
[0]PETSC ERROR: ----- Error Message --------------------------------------------
[0]PETSC ERROR: Cannot compute the root of -2.600000
[...]
[0]PETSC ERROR: #1 square_root() line 23 in root.c
[0]PETSC ERROR: #2 main() line 39 in root.c
```

34.1.3 Memory corruption

PETSc has its own memory management (section 34.5) and this facilitates finding memory corruption errors. The macro `CHKMEMQ` (`CHKMEMA` in void functions) checks all memory that was allocated by PETSc, either internally or throug the allocation routines, for corruption. Sprinkling this macro through your code can detect memory problems before they lead to a *segfault*.

This testing is only done if the commandline argument `-malloc_debug` (`-malloc_test` in debug mode) is supplied, so it carries no overhead for production runs.

34.1.3.1 Valgrind

Valgrind is rather verbose in its output. To limit the number of processs that run under valgrind:

```
mpiexec -n 3 valgrind --track-origins=yes ./app -args : -n 5 ./app -args
```

34.2 Program output

PETSc has as variety of mechanisms to export or visualize program data. We will consider a few possibilities here.

34.2.1 Screen I/O

Printing screen output in parallel is tricky. If two processes execute a print statement at more or less the same time there is no guarantee as to in what order they may appear on screen. (Even attempts to have them print one after the other may not result in the right ordering.) Furthermore, lines from multi-line print actions on two processes may wind up on the screen interleaved.

Figure 34.2 **PetscPrintf**

```
C:
PetscErrorCode  PetscPrintf(MPI_Comm comm,const char format[],...)

Fortran:
PetscPrintf(MPI_Comm, character(*), PetscErrorCode ierr)

Python:
PETSc.Sys.Print(type cls, *args, **kwargs)
kwargs:
comm : communicator object
```

Figure 34.3 **PetscSynchronizedPrintf**

```
C:
PetscErrorCode  PetscSynchronizedPrintf(
    MPI_Comm comm,const char format[],...)

Fortran:
PetscSynchronizedPrintf(MPI_Comm, character(*), PetscErrorCode ierr)

python:
PETSc.Sys.syncPrint(type cls, *args, **kargs)
kwargs:
comm : communicator object
flush : if True, do synchronizedFlush
other keyword args as for python3 print function
```

34.2.1.1 printf replacements

PETSc has two routines that fix this problem. First of all, often the information printed is the same on all processes, so it is enough if only one process, for instance process 0, prints it. This is done with `PetscPrintf` (figure 34.2).

If all processes need to print, you can use `PetscSynchronizedPrintf` (figure 34.3) that forces the output to appear in process order.

To make sure that output is properly flushed from all system buffers use `PetscSynchronizedFlush` (figure 34.4) where for ordinary screen output you would use `stdout` for the file.

> *Fortran note.* The Fortran calls are only wrappers around C routines, so you can use `\n` newline characters in the Fortran string argument to `PetscPrintf`.
>
> The file to flush is typically `PETSC_STDOUT`.

Python note. Since the print routines use the python `print` call, they automatically include the trailing newline. You don't have to specify it as in the C calls.

34.2.1.2 scanf replacement

Using `scanf` in Petsc is tricky, since integers and real numbers can be of different sizes, depending on the installation. Instead, use `PetscViewerRead` (figure 34.5), which operates in terms of `PetscDataType`.

Figure 34.4 **PetscSynchronizedFlush**

```
C:
PetscErrorCode  PetscSynchronizedFlush(MPI_Comm comm,FILE *fd)
fd : output file pointer, needs to be valid on process zero

Fortran:
PetscSynchronizedFlush(comm,fd,err)
Integer :: comm
fd is usually PETSC_STDOUT
PetscErrorCode :: err

python:
PETSc.Sys.syncFlush(type cls, comm=None)
```

Figure 34.5 **PetscViewerRead**

```
Synopsis

#include "petscviewer.h"
PetscErrorCode  PetscViewerRead(PetscViewer viewer, void *data, PetscInt num, PetscInt *count, PetscDataType dtype

Collective

Input Parameters
viewer  - The viewer
data  - Location to write the data
num  - Number of items of data to read
datatype  - Type of data to read

Output Parameters
count -number of items of data actually read, or NULL
```

34.2.2 Exporting internal data structures

In order to export PETSc matrix or vector data structures there is a `PetscViewer` object type. This is a quite general concept of viewing: it encompasses ascii output to screen, binary dump to file, or communication to a running Matlab process. Calls such as `MatView` or `KSPView` accept a `PetscViewer` argument.

In cases where this makes sense, there is also an inverse 'load' operation. See section 29.3.5 for vectors.

Some viewers are predefined, such as `PETSC_VIEWER_STDOUT_WORLD` for ascii rendering to standard out. (In C, specifying zero or `NULL` also uses this default viewer; for Fortran use `PETSC_NULL_VIEWER`.)

34.2.2.1 Viewer types

For activities such as dumping to file you first need create the viewer with `PetscViewerCreate` and set its type with `PetscViewerSetType`.

```
PetscViewerCreate(comm,&viewer);
PetscViewerSetType(viewer,PETSCVIEWERBINARY);
```

Popular types include `PETSCVIEWERASCII`, `PETSCVIEWERBINARY`, `PETSCVIEWERSTRING`, `PETSCVIEWERDRAW`, `PETSCVIEWERSOCKET`, `PETSCVIEWERHDF5`, `PETSCVIEWERVTK`; the full list can be found in `include/petscviewer.h`.

34.2.2.2 *Viewer formats*

Viewers can take further format specifications by using `PetscViewerPushFormat`:

```
PetscViewerPushFormat
   (PETSC_VIEWER_STDOUT_WORLD,
    PETSC_VIEWER_ASCII_INFO_DETAIL);
```

and afterwards a corresponding `PetscViewerPopFormat`

34.2.2.3 *Commandline option for viewers*

Petsc objects viewers can be activated by calls such as `MatView`, but often it is more convenient to do this through commandline options, such as *-mat_view*, *-vec_view*, or *-ksp_view*. By default, these output to `stdout` in `ascii` form, but this can be controlled by further option values:

```
program -mat_view binary:matrix.dat
```

where `binary` forces a binary dump (`ascii` is the default) and a file name is explicitly given.

If a viewer needs to be triggered at a specific location, calls such as `VecViewFromOptions` can be used. These routines all have a similar calling sequence:

```
#include "petscsys.h"
PetscErrorCode  PetscObjectViewFromOptions(PetscObject obj,PetscObject bobj,const char
    optionname[])
PetscErrorCode  VecViewFromOptions(Vec A,PetscObject obj,const char name[])
```

`AOViewFromOptions`, `DMViewFromOptions`, `ISViewFromOptions`, `ISLocalToGlobalMappingViewFromOptions`, `KSPConvergedReasonViewFromOptions`, `KSPViewFromOptions`, `MatPartitioningViewFromOptions`, `MatCoarsenViewFromOptions`, `MatViewFromOptions`, `PetscObjectViewFromOptions`, `PetscPartitionerViewFromOptions`, `PetscDrawViewFromOptions`, `PetscRandomViewFromOptions`, `PetscDualSpaceViewFromOptions`, `PetscSFViewFromOptions`, `PetscFEViewFromOptions`, `PetscFVViewFromOptions`, `PetscSectionViewFromOptions`, `PCViewFromOptions`, `PetscSpaceViewFromOptions`, `PFViewFromOptions`, `PetscLimiterViewFromOptions`, `PetscLogViewFromOptions`, `PetscDSViewFromOptions`, `PetscViewerViewFromOptions`, `SNESConvergedReasonViewFromOptions`, `SNESViewFromOptions`, `TSTrajectoryViewFromOptions`, `TSViewFromOptions`, `TaoLineSearchViewFromOptions`, `TaoViewFromOptions`, `VecViewFromOptions`, `VecScatterViewFromOptions`,

34.2.2.4 *Naming objects*

A helpful facility for viewing is to name an object: that name will then be displayed when the object is viewed.

```
Vec i_local;
ierr = VecCreate(comm,&i_local); CHKERRQ(ierr);
ierr = PetscObjectSetName((PetscObject)i_local,"space local"); CHKERRQ(ierr);
```

giving:

```
Vec Object: space local 4 MPI processes
  type: mpi
Process [0]
[ ... et cetera ... ]
```

34.3 Commandline options

PETSc has as large number of commandline options, most of which we will discuss later. For now we only mention `-log_summary` which will print out profile of the time taken in various routines. For these options to be parsed, it is necessary to pass `argc,argv` to the `PetscInitialize` call.

34.3.1 Adding your own options

You can add custom commandline options to your program. Various routines such as `PetscOptionsGetInt` scan the commandline for options and set parameters accordingly. For instance,

```
// ksp.c
PetscBool flag;
int domain_size = 100;
ierr = PetscOptionsGetInt
  (NULL,PETSC_NULL,"-n",&domain_size,&flag); CHKERRQ(ierr);
PetscPrintf(comm,"Using domain size %d\n",domain_size);
```

For the full source of this example, see section 34.6.4

declares the existence of an option `-n` to be followed by an integer.

Now executing

```
mpiexec yourprogram -n 5
```

will

1. set the `flag` to true, and
2. set the parameter `domain_size` to the value on the commandline.

Omitting the `-n` option will leave the default value of `domain_size` unaltered.

Python note. In Python, do not specify the initial hyphen of an option name. Also, the functions such as `getInt` do not return the boolean flag; if you need to test for the existence of the commandline option, use:

```
hasn = PETSc.Options().hasName("n")
```

There is a related mechanism using `PetscOptionsBegin` / `PetscOptionsEnd`:

```
// optionsbegin.c
ierr = PetscOptionsBegin(comm,NULL,"Parameters",NULL); CHKERRQ(ierr);
ierr = PetscOptionsInt("-i","i value",__FILE__,i_value,&i_value,&i_flag); CHKERRQ(ierr);
ierr = PetscOptionsInt("-j","j value",__FILE__,j_value,&j_value,&j_flag); CHKERRQ(ierr);
ierr = PetscOptionsEnd(); CHKERRQ(ierr);
if (i_flag)
  PetscPrintf(comm,"Option `-i' was used\n");
if (j_flag)
  PetscPrintf(comm,"Option `-j' was used\n");
```

For the full source of this example, see section 34.6.5

The selling point for this approach is that running your code with

```
mpiexec yourprogram -help
```

will display these options as a block. Together with a ton of other options, unfortunately.

34.3.2 Options prefix

In many cases, your code will have only one `KSP` solver object, so specifying *-ksp_view* or *-ksp_monitor* will display / trace that one. However, you may have multiple solvers, or nested solvers. You may then not want to display all of them.

As an example of the nest solver case, consider the case of a *block jacobi preconditioner*, where the block is itself solved with an iterative method. You can trace that one with *--sub_ksp_monitor*.

The `sub_` is an *option prefix*, and you can defined your own with `KSPSetOptionsPrefix`. (There are similar routines for other PETSc object types.)

Example:

```
KSPCreate(comm,&time_solver);
KSPCreate(comm,&space_solver);
KSPSetOptionsPrefix(time_solver,"time_");
KSPSetOptionsPrefix(space_solver,"space_");
```

You can then use options `-time_ksp_monitor` and such. Note that the prefix does not have a leading dash, but it does have the trailing underscore.

Similar routines: `MatSetOptionsPrefix`, `PCSetOptionsPrefix`, `PetscObjectSetOptionsPrefix`, `PetscViewerSetOptionsPrefix`, `SNESSetOptionsPrefix`, `TSSetOptionsPrefix`, `VecSetOptionsPrefix`, and some more obscure ones.

34.3.3 Where to specify options

Commandline options can obviously go on the commandline. However, there are more places where they can be specified.

Options can be specified programmatically with `PetscOptionsSetValue`:

```
PetscOptionsSetValue( NULL, // for global options
  "-some_option","value_as_string");
```

Options can be specified in a file `.petscrc` in the user's home directory.

Finally, an environment variable `PETSC_OPTIONS` can be set.

The `rc` file is processed first, then the environment variable, then any commandline arguments. This parsing is done in `PetscInitialize`, so any values from `PetscOptionsSetValue` override this.

Figure 34.6 `PetscTime`

```
Synopsis
Returns the CPU time in seconds used by the process.

#include "petscsys.h"
#include "petsctime.h"
PetscErrorCode PetscGetCPUTime(PetscLogDouble *t)
PetscErrorCode PetscTime(PetscLogDouble *v)
```

Figure 34.7 `PetscMalloc1`

```
Synopsis
Allocates an array of memory aligned to PETSC_MEMALIGN

C:
#include <petscsys.h>
PetscErrorCode PetscMalloc1(size_t m1,type **r1)

Input Parameter:
m1 - number of elements to allocate (may be zero)

Output Parameter:
r1 - memory allocated
```

34.4 Timing and profiling

PETSc has a number of timing routines that make it unnecessary to use system routines such as `getrusage` or MPI routines such as `MPI_Wtime`. The main (wall clock) timer is `PetscTime` (figure 34.6). Note the return type of `PetscLogDouble` which can have a different precision from `PetscReal`.

The routine `PetscGetCPUTime` is less useful, since it measures only time spent in computation, and ignores things such as communication.

34.4.1 Logging

Petsc does a lot of logging on its own operations. Additionally, you can introduce your own routines into this log.

The simplest way to display statistics is to run with an option *-log_view*. This takes an optional file name argument:

```
mpiexec -n 10 yourprogram -log_view :statistics.txt
```

The corresponding routine is `PetscLogView`.

34.5 Memory management

Allocate the memory for a given pointer: `PetscNew`, allocate arbitrary memory with `PetscMalloc`, allocate a number of objects with `PetscMalloc1` (figure 34.7) (this does not zero the memory allocated, use `PetscCalloc1` to obtain memory that has been zeroed); use `PetscFree` (figure 34.8) to free.

Figure 34.8 **PetscFree**

```
Synopsis
Frees memory, not collective

C:
#include <petscsys.h>
PetscErrorCode PetscFree(void *memory)

Input Parameter:
memory - memory to free (the pointer is ALWAYS set to NULL upon sucess)
```

```
PetscInt *idxs;
PetscMalloc1(10,&idxs);
// better than:
// PetscMalloc(10*sizeof(PetscInt),&idxs);
for (PetscInt i=0; i<10; i++)
  idxs[i] = f(i);
PetscFree(idxs);
```

Allocated memory is aligned to `PETSC_MEMALIGN`.

The state of memory allocation can be written to file or standard out with `PetscMallocDump`. The commandline option *-malloc_dump* outputs all not-freed memory during `PetscFinalize`.

34.6 Sources used in this chapter

34.6.1 Listing of code header

34.6.2 Listing of code examples/petsc/c/backtrace.c

```
#include <stdlib.h>
#include <stdio.h>

#include <petsc.h>

PetscErrorCode this_function_bombs() {
  PetscFunctionBegin;
  SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this");
  PetscFunctionReturn(0);
}

int main(int argc,char **argv)
{
  PetscErrorCode ierr;

  char help[] = "\nInit example.\n\n";
  ierr = PetscInitialize(&argc,&argv,(char*)0,help); CHKERRQ(ierr);
  ierr = this_function_bombs(); CHKERRQ(ierr);
  return PetscFinalize();
}
```

34.6.3 Listing of code examples/petsc/f/backtrace.F90

```
Program BackTrace

#include <petsc/finclude/petsc.h>
  use petsc
  implicit none

  PetscErrorCode :: ierr

  call PetscInitialize(PETSC_NULL_CHARACTER,ierr)
  CHKERRA(ierr)
  call this_function_bombs(ierr)
  CHKERRA(ierr)
  call PetscFinalize(ierr)
  CHKERRA(ierr)

contains

  Subroutine this_function_bombs(ierr)
    implicit none
    integer,intent(out) :: ierr

    SETERRQ(PETSC_COMM_SELF,1,"We cannot go on like this")
```

```
    ierr = -1

  end Subroutine this_function_bombs

End Program BackTrace
```

34.6.4 Listing of code examples/petsc/c/kspcg.c

```
#include "petscksp.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
  PetscErrorCode ierr;
  MPI_Comm comm;
  KSP Solver;
  Mat A;
  Vec Rhs,Sol;
  PetscScalar one = 1.0;

  PetscFunctionBegin;
  PetscInitialize(&argc,&args,0,0);

  comm = PETSC_COMM_SELF;

  /*
   * Read the domain size, and square it to get the matrix size
  */
  PetscBool flag;
  int matrix_size = 100;
  ierr = PetscOptionsGetInt
    (NULL,PETSC_NULL,"-n",&matrix_size,&flag); CHKERRQ(ierr);
  PetscPrintf(comm,"Using matrix size %d\n",matrix_size);

  /*
   * Create the five-point laplacian matrix
  */
  ierr = MatCreate(comm,&A); CHKERRQ(ierr);
  ierr = MatSetType(A,MATSEQAIJ); CHKERRQ(ierr);
  ierr = MatSetSizes(A,matrix_size,matrix_size,matrix_size,matrix_size); CHKERRQ(ierr);
  ierr = MatSeqAIJSetPreallocation(A,3,PETSC_NULL); CHKERRQ(ierr);
  ierr = MatCreateVecs(A,&Rhs,PETSC_NULL); CHKERRQ(ierr);
  for (int i=0; i<matrix_size; i++) {
    PetscScalar
      h = 1./(matrix_size+1), pi = 3.1415926,
      sin1 = i * pi * h, sin2 = 2 * i * pi * h, sin3 = 3 * i * pi * h,
      coefs[3] = {-1,2,-1};
    PetscInt cols[3] = {i-1,i,i+1};
    ierr = VecSetValue(Rhs,i,sin1 + .5 * sin2 + .3 * sin3, INSERT_VALUES); CHKERRQ(ierr);
    if (i==0) {
      ierr = MatSetValues(A,1,&i,2,cols+1,coefs+1,INSERT_VALUES); CHKERRQ(ierr);
    } else if (i==matrix_size-1) {
```

```
    ierr = MatSetValues(A,1,&i,2,cols,coefs,INSERT_VALUES); CHKERRQ(ierr);
  } else {
    ierr = MatSetValues(A,1,&i,2,cols,coefs,INSERT_VALUES); CHKERRQ(ierr);
  }
}
ierr = MatAssemblyBegin(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
ierr = MatAssemblyEnd(A,MAT_FINAL_ASSEMBLY); CHKERRQ(ierr);
//MatView(A,PETSC_VIEWER_STDOUT_WORLD);

/*
 * Create right hand side and solution vectors
 */
ierr = VecDuplicate(Rhs,&Sol); CHKERRQ(ierr);
ierr = VecSet(Rhs,one); CHKERRQ(ierr);

/*
 * Create iterative method and preconditioner
 */
ierr = KSPCreate(comm,&Solver);
ierr = KSPSetOperators(Solver,A,A); CHKERRQ(ierr);
ierr = KSPSetType(Solver,KSPCG); CHKERRQ(ierr);
{
  PC Prec;
  ierr = KSPGetPC(Solver,&Prec); CHKERRQ(ierr);
  ierr = PCSetType(Prec,PCNONE); CHKERRQ(ierr);
}

/*
 * Incorporate any commandline options for the KSP
 */
ierr = KSPSetFromOptions(Solver); CHKERRQ(ierr);

/*
 * Solve the system and analyze the outcome
 */
ierr = KSPSolve(Solver,Rhs,Sol); CHKERRQ(ierr);
{
  PetscInt its; KSPConvergedReason reason;
  ierr = KSPGetConvergedReason(Solver,&reason);
  ierr = KSPGetIterationNumber(Solver,&its); CHKERRQ(ierr);
  if (reason<0) {
    PetscPrintf(comm,"Failure to converge after %d iterations; reason %s\n",
its,KSPConvergedReasons[reason]);
  } else {
    PetscPrintf(comm,"Number of iterations to convergence: %d\n",its);
  }
}

ierr = MatDestroy(&A); CHKERRQ(ierr);
ierr = KSPDestroy(&Solver); CHKERRQ(ierr);
ierr = VecDestroy(&Rhs); CHKERRQ(ierr);
ierr = VecDestroy(&Sol); CHKERRQ(ierr);
```

```
  return PetscFinalize();
}
```

34.6.5 Listing of code examples/petsc/c/optionsbegin.c

```
#include "petsc.h"

#undef __FUNCT__
#define __FUNCT__ "main"
int main(int argc,char **args)
{
  PetscErrorCode ierr;
  MPI_Comm comm;

  PetscFunctionBegin;
  PetscInitialize(&argc,&args,0,0);
  comm = MPI_COMM_WORLD;

  PetscInt i_value = 1, j_value = 1;
  PetscBool i_flag = 0, j_flag = 0;

  ierr = PetscOptionsBegin(comm,NULL,"Parameters",NULL); CHKERRQ(ierr);
  ierr = PetscOptionsInt("-i","i value",__FILE__,i_value,&i_value,&i_flag); CHKERRQ(ierr);
  ierr = PetscOptionsInt("-j","j value",__FILE__,j_value,&j_value,&j_flag); CHKERRQ(ierr);
  ierr = PetscOptionsEnd(); CHKERRQ(ierr);
  if (i_flag)
    PetscPrintf(comm,"Option `-i' was used\n");
  if (j_flag)
    PetscPrintf(comm,"Option `-j' was used\n");

  PetscPrintf(comm,"i=%d, j=%d\n",i_value,j_value);

  return PetscFinalize();
}
```

Chapter 35

PETSc topics

35.1 Communicators

PETSc has a 'world' communicator, which by default equals `MPI_COMM_WORLD`. If you want to run PETSc on a subset of processes, you can assign a subcommunicator to the variable `PETSC_COMM_WORLD` in between the calls to `MPI_Init` and `PetscInitialize`. Petsc communicators are of type *PetscComm*.

35.2 Sources used in this chapter

35.2.1 Listing of code header

PART IV

OTHER PROGRAMMING MODELS

Chapter 36

Co-array Fortran

This chapter explains the basic concepts of Co-array Fortran (CAF), and helps you get started on running your first program.

36.1 History and design

https://en.wikipedia.org/wiki/Coarray_Fortran

36.2 Compiling and running

CAF is built on the same SPMD design as MPI. Where MPI talks about processes or ranks, CAF calls the running instances of your program *image*s.

The Intel compiler uses the flag `-coarray=xxx` with values `single`, `shared`, `distributed` gpu.

It is possible to bake the number of 'images' into the executable, but by default this is not done, and it is determined at runtime by the variable `FOR_COARRAY_NUM_IMAGES`.

CAF can not be mixed with OpenMP.

36.3 Basics

Co-arrays are defined by giving them, in addition to the `Dimension`, a `Codimension`

```
Complex,codimension(*) :: number
Integer,dimension(:,:,:),codimension[-1:1,*] :: grid
```

This means we are respectively declaring an array with a single number on each image, or a three-dimensional grid spread over a two-dimensional processor grid.

Traditional-like syntax can also be used:

```
Complex :: number[*]
Integer :: grid(10,20,30)[-1:1,*]
```

Unlike Message Passing Interface (MPI), which normally only supports a linear process numbering, CAF allows for multi-dimensional process grids. The last dimension is always specified as `*`, meaning it is determined at runtime.

36.3.1 Image identification

As in other models, in CAF one can ask how many images/processes there are, and what the number of the current one is, with `num_images` and `this_image` respectively.

```
// hello.F90
write(*,*) "Hello from image ", this_image(), &
     "out of ", num_images()," total images"
```

For the full source of this example, see section 36.4.2

If you call `this_image` with a co-array as argument, it will return the image index, as a tuple of `cosubscripts`, rather than a linear index. Given such a set of subscripts, `image_index` will return the linear index.

The functions `lcobound` and `ucobound` give the lower and upper bound on the image subscripts, as a linear index, or a tuple if called with a co-array variable.

36.3.2 Remote operations

The appeal of CAF is that moving data between images looks (almost) like an ordinary copy operation:

```
real :: x(2)[*]
integer :: p
p = this_image()
x(1)[ p+1 ] = x(2)[ p ]
```

Exchanging grid boundaries is elegantly done with array syntax:

```
Real,Dimension( 0:N+1,0:N+1 )[*] :: grid
grid( N+1,: )[p] = grid( 0,: )[p+1]
grid(   0,: )[p] = grid( N,: )[p-1]
```

36.3.3 Synchronization

The fortran standard forbids *race conditions*:

> If a variable is defined on an image in a segment, it shall not be referenced, defined or become undefined in a segment on another image unless the segments are ordered.

That is, you should not cause them to happen. The language and runtime are certainly not going to help yu with that.

Well, a little. After remote updates you can synchronize images with the `sync` call. The easiest variant is a global synchronization:

```
sync all
```

Compare this to a wait call after MPI nonblocking calls.

More fine-grained, one can synchronize with specific images:

```
sync images( (/ p-1,p,p+1 /) )
```

While remote operations in CAF are nicely one-sided, synchronization is not: if image p issues a call

```
sync(q)
```

then q also needs to issue a mirroring call to synchronize with p.

As an illustration, the following code is not a correct implementation of a *ping-pong*:

```
// pingpong.F90
sync all
if (procid==1) then
   number[procid+1] = number[procid]
else if (procid==2) then
   number[procid-1] = 2*number[procid]
end if
sync all
```

We can solve this with a global synchronization:

```
sync all
if (procid==1) &
     number[procid+1] = number[procid]
sync all
if (procid==2) &
     number[procid-1] = 2*number[procid]
sync all
```

or a local one:

```
if (procid==1) &
     number[procid+1] = number[procid]
if (procid<=2) sync images( (/1,2/) )
if (procid==2) &
     number[procid-1] = 2*number[procid]
if (procid<=2) sync images( (/2,1/) )
```

Note that the local sync call is done on both images involved.

Example of how you would synchronize a collective:

```
if ( this_image() .eq. 1 ) sync images( * )
if ( this_image() .ne. 1 ) sync images( 1 )
```

Here image 1 synchronizes with all others, but the others don't synchronize with each other.

```
if (procid==1) then
   sync images( (/procid+1/) )
else if (procid==nprocs) then
   sync images( (/procid-1/) )
```

```
else
   sync images( (/procid-1,procid+1/) )
end if
```

For the full source of this example, see section 36.4.3

36.3.4 Collectives

Collectives are not part of CAF as of the 2008 Fortran standard.

36.4 Sources used in this chapter

36.4.1 Listing of code header

36.4.2 Listing of code examples/caf/f08/hello.F90

```
program hello_image

  write(*,*) "Hello from image ", this_image(), &
       "out of ", num_images()," total images"

end program hello_image
```

36.4.3 Listing of code examples/caf/f08/rightcopy.F90

```
program RightCopy

  integer,dimension(2),codimension[*] :: numbers
  integer :: procid,nprocs

  procid = this_image()
  nprocs = num_images()
  numbers(:)[procid] = procid
  if (procid<nprocs) then
     numbers(1)[procid+1] = procid
  end if
  if (procid==1) then
     sync images( (/procid+1/) )
  else if (procid==nprocs) then
     sync images( (/procid-1/) )
  else
     sync images( (/procid-1,procid+1/) )
  end if

  write(*,*) "After shift,",procid," has",numbers(:)[procid]

end program RightCopy
```

Chapter 37

Sycl, OneAPI, DPC++

This chapter explains the basic concepts of Sycl/Dpc++, and helps you get started on running your first program.

- *SYCL* is a C++-based language for portable parallel programming.
- *Data Parallel C++ (DPCPP)* is Intel's extension of Sycl.
- *OneAPI* is Intel's compiler suite, which contains the DPCPP compiler.

Intel DPC++ extension. The various Intel extensions are listed here: https://spec.oneapi.com/versions/latest/elements/dpcpp/source/index.html#extensions-table

37.1 Logistics

Headers:

```
#include <CL/sycl.hpp>
```

You can now include namespace, but with care! If you use

```
using namespace cl;
```

you have to prefix all SYCL class with `sycl::`, which is a bit of a bother. However, if you use

```
using namespace cl::sycl;
```

you run into the fact that SYCL has its own versions of many Standard Template Library (STL) commands, and so you will get name collisions. The most obvious example is that the `cl::sycl` name space has its own versions of `cout` and `endl`. Therefore you have to use explicitly `std::cout` and `std::end`. Using the wrong I/O will cause tons of inscrutable error messages. Additionally, SYCL has its own version of `free`, and of several math routines.

Intel DPC++ extension.

```
    using namespace sycl;
```

37.2 Platforms and devices

Since DPCPP is cross-platform, we first need to discovers the devices.

First we list the platforms:

```
// devices.cxx
std::vector<sycl::platform> platforms = sycl::platform::get_platforms();
for (const auto &plat : platforms) {
// get_info is a template. So we pass the type as an `arguments`.
  std::cout << "Platform: "
            << plat.get_info<sycl::info::platform::name>() << " "
            << plat.get_info<sycl::info::platform::vendor>() << " "
            << plat.get_info<sycl::info::platform::version>() << std::endl;
```

For the full source of this example, see section 37.11.2

Then for each platform we list the devices:

```
std::vector<sycl::device> devices = plat.get_devices();
for (const auto &dev : devices) {
  std::cout << "-- Device: "
            << dev.get_info<sycl::info::device::name>()
            << (dev.is_host() ? ": is the host" : "")
            << (dev.is_cpu() ? ": is a cpu" : "")
            << (dev.is_gpu() ? ": is a gpu" : "")
            << std::endl;
```

For the full source of this example, see section 37.11.2

You can query what type of device you are dealing with by `is_host`, `is_cpu`, `is_gpu`.

37.3 Queues

The execution mechanism of SYCL is the *queue*: a sequence of actions that will be executed on a selected device. The only user action is submitting actions to a queue; the queue is executed at the end of the scope where it is declared.

Queue execution is asynchronous with host code.

37.3.1 Device selectors

You need to select a device on which to execute the queue. A single queue can only dispatch to a single device.

A queue is coupled to one specific device, so it can not spread work over multiple devices. You can find a default device for the queue with

```
sycl::queue myqueue;
```

The following example explicitly assigns the queue to the CPU device using the `sycl::cpu_selector`.

```
// cpuname.cxx
sycl::queue myqueue( sycl::cpu_selector{} );
```

For the full source of this example, see section 37.11.3

The `sycl::host_selector` bypasses any devices and make the code run on the host.

It is good for your sanity to print the name of the device you are running on:

```
// devname.cxx
std::cout << myqueue.get_device().get_info<sycl::info::device::name>()
          << std::endl;
```

For the full source of this example, see section 37.11.4

If you try to select a device that is not available, a `sycl::runtime_error` exception will be thrown.

Intel DPC++ extension.

```
    #include "CL/sycl/intel/fpga_extensions.hpp"
    fpga_selector
```

37.3.2 Queue execution

It seems that queue kernels will also be executed when only they go out of scope, but not the queue:

```
cpu_selector selector;
queue q(selector);
{
  q.submit( /* some kernel */ );
} // here the kernel executes
```

37.3.3 Kernel ordering

Kernels are not necessarily executed in the order in which they are submitted. You can enforce this by specifying an *in-order queue*:

```
sycl::queue myqueue{property::queue::inorder()};
```

37.4 Kernels

One kernel per submit.

```
myqueue.submit( [&] ( handler &commandgroup ) {
    commandgroup.parallel_for<uniquename>
      ( range<1>{N},
        [=] ( id<1> idx ) { ... idx }
      )
    } );
```

Note that the lambda in the kernel captures by value. Capturing by reference makes no sense, since the kernel is executed on a device.

```
cgh.single_task(
  [=]() {
    // kernel function is executed EXACTLY once on a SINGLE work-item
});
```

The `submit` call results in an event object:

```
auto myevent = myqueue.submit( /* stuff */ );
```

This can be used for two purposes:

1. It becomes possible to wait for this specific event:

   ```
   myevent.wait();
   ```

2. It can be used to indicate kernel dependencies:

   ```
   myqueue.submit( [=] (handler &h) {
       h.depends_on(myevent);
       /* stuff */
       } );
   ```

37.5 Parallel operations

37.5.1 Loops

```
cgh.parallel_for(
  range<3>(1024,1024,1024),
  // using 3D in this example
  [=](id<3> myID) {
    // kernel function is executed on an n-dimensional range (NDrange)
});

cgh.parallel_for(
  nd_range<3>( {1024,1024,1024},{16,16,16} ),
  // using 3D in this example
  [=](nd_item<3> myID) {
    // kernel function is executed on an n-dimensional range (NDrange)
});

cgh.parallel_for_work_group(
  range<2>(1024,1024),
  // using 2D in this example
  [=](group<2> myGroup) {
    // kernel function is executed once per work-group
});

grp.parallel_for_work_item(
  range<1>(1024),
  // using 1D in this example
```

```
  [=](h_item<1> myItem) {
    // kernel function is executed once per work-item
});
```

37.5.1.1 Loop bounds: ranges

SYCL adopts the modern C++ philosophy that one does not iterate over by explicitly enumerating indices, but by indicating their range. This is realized by the `range` class, which is templated over the number of space dimensions.

```
sycl::range<2> matrix{10,10};
```

Some compilers are sensitive to the type of the integer arguments:

```
sycl::range<1> array{ static_cast<size_t>(size)} ;
```

37.5.1.2 Loop indices

Kernels such as `parallel_for` expects two arguments:

- a `range` over which to index; and
- a lambda of one argument: an index.

There are several ways of indexing. The `id<nd>` class of multi-dimensional indices.

```
myHandle.parallel_for<class uniqueID>
   ( mySize,
     [=]( id<1> index ) {
       float x = index.get(0) * h;
       deviceAccessorA[index] *= 2.;
     }
   )
```

```
cgh.parallel_for<class foo>(
    range<1>{D*D*D},
    [=](id<1> item) {
        xx[ item[0] ] = 2 * item[0] + 1;
    }
)
```

While the C++ vectors remain one-dimensional, DPCPP allows you to make multi-dimensional buffers:

```
std::vector<int> y(D*D*D);
buffer<int,1> y_buf(y.data(), range<1>(D*D*D));
cgh.parallel_for<class foo2D>
   (range<2>{D,D*D},
    [=](id<2> item) {
        yy[ item[0] + D*item[1] ] = 2;
    }
   );
```

Intel DPC++ extension. There is an implicit conversion from the one-dimensional `sycl::id<1>` to `size_t`, so

```
[=](sycl::id<1> i) {
   data[i] = i;
}
```

is legal, which in SYCL requires

```
data[i[0]] = i[0];
```

37.5.1.3 *Multi-dimensional indexing*

```
// stencil2d.cxx
sycl::range<2> stencil_range(N, M);
sycl::range<2> alloc_range(N + 2, M + 2);
std::vector<float>
  input(alloc_range.size()),
  output(alloc_range.size());
  sycl::buffer<float, 2> input_buf(input.data(), alloc_range);
  sycl::buffer<float, 2> output_buf(output.data(), alloc_range);
```

```
constexpr size_t B = 4;
sycl::range<2> local_range(B, B);
sycl::range<2> tile_range = local_range + sycl::range<2>(2, 2); // Includes boundary
    cells
auto tile = sycl::local_accessor<float, 2>(tile_range, h);
```

We first copy global data into an array local to the work group:

```
sycl::id<2> offset(1, 1);
h.parallel_for
  ( sycl::nd_range<2>(stencil_range, local_range, offset),
    [=] ( sycl::nd_item<2> it ) {
// Load this tile into work-group local memory
    sycl::id<2>    lid    = it.get_local_id();
    sycl::range<2> lrange = it.get_local_range();
    for   (int ti = lid[0]; ti < B + 2; ti += lrange[0]) {
      for (int tj = lid[1]; tj < B + 2; tj += lrange[1]) {
        int gi = ti + B * it.get_group(0);
        int gj = tj + B * it.get_group(1);
        tile[ti][tj] = input[gi][gj];
      }
    }
```

Global coordinates in the input are computed from the `nd_item`'s coordinate and group:

```
[=] ( sycl::nd_item<2> it ) {
for   (int ti ... ) {
  for (int tj ... ) {
    int gi = ti + B * it.get_group(0);
    int gj = tj + B * it.get_group(1);
    ... = input[gi][gj];
```

Local coordinates in the tile, including boundary, I DON'T QUITE GET THIS YET.

```
[=] ( sycl::nd_item<2> it ) {
sycl::id<2>    lid    = it.get_local_id();
sycl::range<2> lrange = it.get_local_range();
for   (int ti = lid[0]; ti < B + 2; ti += lrange[0]) {
  for (int tj = lid[1]; tj < B + 2; tj += lrange[1]) {
    tile[ti][tj] = ..
```

37.5.2 Task dependencies

Each `submit` call can be said to correspond to a 'task'. Since it returns a token, it becomes possible to specify task dependencies by refering to a token as a dependency in a later specified task.

```
queue myQueue;
auto myTokA = myQueue.submit
   ( [&](handler& h) {
       h.parallel_for<class taskA>(...);
     }
   );
auto myTokB = myQueue.submit
   ( [&](handler& h) {
       h.depends_on(myTokA);
       h.parallel_for<class taskB>(...);
     }
   );
```

37.5.3 Race conditions

Sycl has the same problems with race conditions that other shared memory system have:

```
// sum1d.cxx
auto array_accessor =
  array_buffer.get_access<sycl::access::mode::read>(h);
auto scalar_accessor =
  scalar_buffer.get_access<sycl::access::mode::read_write>(h);
h.parallel_for<class uniqueID>
  ( array_range,
    [=](sycl::id<1> index)
    {
      scalar_accessor[0] += array_accessor[index];
    }
    ); // end of parallel for
```

To get this working correctly would need either a reduction primitive or atomics on the accumulator. The 2020 proposed standard has improved atomics.

```
// reduct1d.cxx
auto array_accessor =
  array_buffer.get_access<sycl::access::mode::read>(h);
auto scalar_accessor = sycl::reducer
  (scalar_buffer.get_access<sycl::access::mode::read_write>(h), std::plus<float>());
h.parallel_for<class uniqueID>
```

```
  ( array_range,
    [=](sycl::id<1> index)
    {
      scalar_accessor += array_accessor[index];
    }
    ); // end of parallel for
```

37.5.4 Reductions

Reduction operations were added in the the SYCL 2020 Provisional Standard, meaning that they are not yet finalized.

Here is one approach, which works in *hipsycl*:

```
// reductscalar.cxx
auto reduce_to_sum =
  sycl::reduction( sum_array, static_cast<float>(0.), std::plus<float>() );
myqueue.parallel_for// parallel_for<reduction_kernel<T,BinaryOp,__LINE__>>
  ( array_range,    // sycl::range<1>(input_size),
    reduce_to_sum,  // sycl::reduction(output, identity, op),
    [=] (sycl::id<1> idx, auto& reducer) { // type of reducer is impl-dependent, so use
    auto
    reducer.combine(shared_array[idx[0]]); //(input[idx[0]]);
//reducer += shared_array[idx[0]]; // see line 216: add_reducer += input0[idx[0]];
  } ).wait();
```

Here a `sycl::reduction` object is created from the target data and the reduction operator. This is then passed to the `parallel_for` and its `combine` method is called.

37.6 Memory access

Memory treatment in SYCL is a little complicated, because is (at the very least) host memory and device memory, which are not necessarily coherent.

There are also three mechanisms:

- Unified Shared Memory, based on ordinary C/C++ 'star'-pointers.
- Buffers, using the `buffer` class; this needs the `accessor` class to access the data.
- Images.

Table 37.1: Memory types and treatments

Location	allocation	copy
Host	`malloc`	`queue::memcpy`
	`malloc_host`	coherent host/device
Device	`malloc_device`	`queue::memcpy`
Shared	`malloc_shared`	coherent host/device

37.6.1 Unified shared memory

Memory allocated with `malloc_host` is visible on the host:

```
// outshared.cxx
floattype
  *host_float = (floattype*)malloc_host( sizeof(floattype),ctx ),
  *shar_float = (floattype*)malloc_shared( sizeof(floattype),dev,ctx );
    cgh.single_task
        ( [=] () {
          shar_float[0] = 2 * host_float[0];
          sout << "Device sets " << shar_float[0] << sycl::endl;
        } );
```

For the full source of this example, see section 37.11.5

Device memory is allocated with `malloc_device`, passing the queue as parameter:

```
// reductimpl.cxx
floattype
  *host_float = (floattype*)malloc( sizeof(floattype) ),
  *devc_float = (floattype*)malloc_device( sizeof(floattype),dev,ctx );
   [&](sycl::handler &cgh) {
     cgh.memcpy(devc_float,host_float,sizeof(floattype));
   }
```

Note the corresponding `free` call that also has the queue as parameter.

Note that you need to be in a parallel task. The following gives a segmentation error:

```
  [&](sycl::handler &cgh) {
    shar_float[0] = host_float[0];
  }
```

Ordinary memory, for instance from `malloc`, has to be copied in a kernel:

```
   [&](sycl::handler &cgh) {
     cgh.memcpy(devc_float,host_float,sizeof(floattype));
   }
   [&](sycl::handler &cgh) {
     sycl::stream sout(1024, 256, cgh);
     cgh.single_task
         (
          [=] () {
            sout << "Number " << devc_float[0] << sycl::endl;
          }
          );
   } // end of submitted lambda
free(host_float);
sycl::free(devc_float,myqueue);
```

For the full source of this example, see section 37.11.6

37.6.2 Buffers and accessors

Arrays need to be declared in a way such that they can be access from any device.

```
// forloop.cxx
std::vector<int> myArray(SIZE);
  range<1> mySize{myArray.size()};
  buffer<int, 1> bufferA(myArray.data(), myArray.size());
```

For the full source of this example, see section 37.11.7

Inside the kernel, the array is then unpacked from the buffer:

```
myqueue.submit( [&] (handler &h) {
        auto deviceAccessorA =
          bufferA.get_access<access::mode::read_write>(h);
```

For the full source of this example, see section 37.11.7

However, the `get_access` function results in a `sycl::accessor`, not a pointer to a simple type. The precise type is templated and complicated, so this is a good place to use `auto`.

Accessors can have a mode associated: `sycl::access::mode::read` `sycl::access::mode::write`

Intel DPC++ extension.

```
    array<floattype,1> leftsum{0.};
#ifdef __INTEL_CLANG_COMPILER
    sycl::buffer leftbuf(leftsum);
#else
    sycl::range<1> scalar{1};
    sycl::buffer<floattype,1> leftbuf(leftsum.data(),scalar);
```

Intel DPC++ extension. there are modes

```
// standard
sycl::accessor acc = buffer.get_access<sycl::access::mode:write>(h);
// dpcpp extension
sycl::accessor acc( buffer,h,sycl::read_only );
sycl::accessor acc( buffer,h,sycl::write_only );
```

37.6.3 Querying

The function `get_range` can query the size of either a buffer or an accessor:

```
// range2.cxx
sycl::buffer<int, 2>
  a_buf(a.data(), sycl::range<2>(N, M)),
  b_buf(b.data(), sycl::range<2>(N, M)),
  c_buf(c.data(), sycl::range<2>(N, M));

sycl::range<2>
  a_range = a_buf.get_range(),
  b_range = b_buf.get_range();

if (a_range==b_range) {
```

For the full source of this example, see section 37.11.8

```
sycl::accessor c = c_buf.get_access<sycl::access::mode::write>(h);

sycl::range<2> c_range = c.get_range();
if (a_range==c_range) {
  h.parallel_for
    ( a_range,
      [=]( sycl::id<2> idx ) {
      c[idx] = a[idx] + b[idx];
    } );
```

For the full source of this example, see section 37.11.8

37.7 Parallel output

There is a `sycl::cout` and `sycl::endl`.

```
// hello.cxx
[&](sycl::handler &cgh) {
  sycl::stream sout(1024, 256, cgh);
  cgh.parallel_for<class hello_world>
        (
         sycl::range<1>(global_range), [=](sycl::id<1> idx) {
           sout << "Hello, World: World rank " << idx << sycl::endl;
         }); // End of the kernel function
}
```

For the full source of this example, see section 37.11.9

Since the end of a queue does not flush stdout, it may be necessary to call `sycl::queue::wait`

```
myQueue.wait();
```

37.8 DPCPP extensions

Intel has made some extensions to SYCL:

- Unified Shared Memory,
- Ordered queues.

37.9 Intel devcloud notes

`qsub -I` for interactive session.

`gdb-oneapi` for debugging.

https://community.intel.com/t5/Intel-oneAPI-Toolkits/ct-p/oneapi for support.

37.10 Examples

37.10.1 Kernels in a loop

The following idiom works:

```
sycl::event last_event = queue.submit( [&] (sycl::handler &h) {
for (int iteration=0; iteration<N; iteration++) {
  last_event = queue.submit( [&] (sycl::handler &h) {
      h.depends_on(last_event);
```

For the full source of this example, see section 37.11.10

37.10.2 Stencil computations

The problem with stencil computations is that only interior points are updated. Translated to SYCL: we need to iterate over a subrange of the range over which the buffer is defined. First let us define these ranges:

```
// jacobi1d.cxx
sycl::range<1> unknowns(N);
sycl::range<1> with_boundary(N + 2);
std::vector<float>
  old_values(with_boundary.size(),0.),
  new_values(with_boundary.size(),0.);
old_values.back() = 1.; new_values.back() = 1.;
```

For the full source of this example, see section 37.11.10

Note the boundary value 1. on the right boundary.

Restricting the iteration to the interior points is done through the `offset` parameter of the `parallel_for`:

```
sycl::id<1> offset(1);
h.parallel_for
    ( unknowns, offset,
      [=] (sycl::id<1> idx) {
          int i = idx[0];
          float self = old_array[i];
          float left = old_array[i - 1];
          float righ = old_array[i + 1];
          new_array[i] = (self + left + righ) / 3.0f;
    } );
```

For the full source of this example, see section 37.11.10

37.11 Sources used in this chapter

37.11.1 Listing of code header

37.11.2 Listing of code code/dpcpp/cxx/devices.cxx

```
#include <CL/sycl.hpp>
#include <vector>

namespace sycl = cl::sycl;

int main() {

  std::cout << "List Platforms and Devices" << std::endl;
  std::vector<sycl::platform> platforms = sycl::platform::get_platforms();
  for (const auto &plat : platforms) {
    // get_info is a template. So we pass the type as an `arguments`.
    std::cout << "Platform: "
              << plat.get_info<sycl::info::platform::name>() << " "
              << plat.get_info<sycl::info::platform::vendor>() << " "
              << plat.get_info<sycl::info::platform::version>() << std::endl;

    std::vector<sycl::device> devices = plat.get_devices();
    for (const auto &dev : devices) {
      std::cout << "-- Device: "
                << dev.get_info<sycl::info::device::name>()
                << (dev.is_host() ? ": is the host" : "")
                << (dev.is_cpu() ? ": is a cpu" : "")
                << (dev.is_gpu() ? ": is a gpu" : "")
                << std::endl;
      // sycl::info::device::device_type exist, but do not overload the <<
      // operator
    }
  }
}
```

37.11.3 Listing of code code/dpcpp/cxx/cpuname.cxx

```
#include <CL/sycl.hpp>
#include <iostream>
#include <array>
#include <cstdio>

using namespace cl;

int main() {

  sycl::queue myqueue( sycl::cpu_selector{} );

  std::cout << myqueue.get_device().get_info<sycl::info::device::name>() << std::endl;
```

```
  return 0;
}
```

37.11.4 Listing of code code/dpcpp/cxx/devname.cxx

```
#include <CL/sycl.hpp>
#include <iostream>
#include <array>
#include <cstdio>

using namespace cl;

int main() {

#if 0
  sycl::cpu_selector selector;
  sycl::queue myqueue(selector);
#else
  sycl::queue myqueue;
#endif

  std::cout << myqueue.get_device().get_info<sycl::info::device::name>()
            << std::endl;

  return 0;
}
```

37.11.5 Listing of code code/dpcpp/cxx/outshared.cxx

```
#include <CL/sycl.hpp>
#include <vector>

namespace sycl = cl::sycl;
using floattype = float;

int main(int argc, char **argv) {

  sycl::queue myqueue;
  std::cout << "Hello example running on "
            << myqueue.get_device().get_info<sycl::info::device::name>()
            << std::endl;

  auto ctx = myqueue.get_context();
  auto dev = myqueue.get_device();
  floattype
    *host_float = (floattype*)malloc_host( sizeof(floattype),ctx ),
    *shar_float = (floattype*)malloc_shared( sizeof(floattype),dev,ctx );
  host_float[0] = 3.14;

  myqueue.submit
    ( [&](sycl::handler &cgh) {
      // WRONG shar_float[0] = host_float[0];
```

```
      sycl::stream sout(1024, 256, cgh);
      cgh.single_task
( [=] () {
  shar_float[0] = 2 * host_float[0];
  sout << "Device sets " << shar_float[0] << sycl::endl;
} );
    } ); // end of lambda submission
  myqueue.wait();

  std::cout << "Host receives " << shar_float[0] << std::endl;

  return 0;
}
```

37.11.6 Listing of code code/dpcpp/cxx/outdevice.cxx

```
#include <CL/sycl.hpp>
#include <vector>

namespace sycl = cl::sycl;
using floattype = float;

int main(int argc, char **argv) {

  sycl::queue myqueue;
  std::cout << "Hello example running on "
            << myqueue.get_device().get_info<sycl::info::device::name>()
            << std::endl;

  auto ctx = myqueue.get_context();
  auto dev = myqueue.get_device();
  floattype
    *host_float = (floattype*)malloc( sizeof(floattype) ),
    *devc_float = (floattype*)malloc_device( sizeof(floattype),myqueue);
  host_float[0] = 3.14;

  myqueue.submit
    (
     [&](sycl::handler &cgh) {
       cgh.memcpy(devc_float,host_float,sizeof(floattype));
     }
     );
  myqueue.wait();

  myqueue.submit
    (
     [&](sycl::handler &cgh) {
       sycl::stream sout(1024, 256, cgh);
       cgh.single_task
 (
  [=] () {
    sout << "Number " << devc_float[0] << sycl::endl;
  }
```

```
  );
     } // end of submitted lambda
     );
  myqueue.wait();

  free(host_float);
  sycl::free(devc_float,myqueue);

  return 0;
}
```

37.11.7 Listing of code code/dpcpp/cxx/forloop.cxx

```
#include <CL/sycl.hpp>
#include <iostream>
#include <array>
#include <vector>
#include <cstdio>

#define SIZE 1024
using namespace cl::sycl;

int main() {
  std::vector<int> myArray(SIZE);
  for (int i = 0; i<SIZE; ++i)
    myArray[i] = i;

  printf("Value at start: myArray[42] is %d.\n",myArray[42]);
  {
    cpu_selector selector;
    queue myqueue(selector);

    range<1> mySize{myArray.size()};
    buffer<int, 1> bufferA(myArray.data(), myArray.size());

    myqueue.submit( [&] (handler &h) {
auto deviceAccessorA =
  bufferA.get_access<access::mode::read_write>(h);
h.parallel_for<class uniqueID>
  ( mySize,
    [=](id<1> index) {
    deviceAccessorA[index] *= 2; }
    );
      }
      );
    printf("Value after submit: myArray[42] is %d.\n",myArray[42]);
    auto hostAccessorA =
      bufferA.get_access<access::mode::read>();
    printf("Value through host access: myArray[42] is %d.\n",myArray[42]);
  }
  printf("Value at finish: myArray[42] is %d.\n",myArray[42]);

  return 0;
```

```
}

/*
      ( [&](handler &myHandle) {
auto deviceAccessorA =
  bufferA.get_access<access::mode::read_write>(myHandle);
myHandle
*/
```

37.11.8 Listing of code code/dpcpp/cxx/range2.cxx

```
#include <CL/sycl.hpp>
#include <algorithm>
#include <iostream>
namespace sycl = cl::sycl;

int main() {
  // Set up queue on any available device
  sycl::queue Q;

  // Initialize input and output memory on the host
  constexpr size_t N = 256;
  constexpr size_t M = 256;
  std::vector<int> a(N * M), b(N * M), c(N * M);
  std::fill(a.begin(), a.end(), 1);
  std::fill(b.begin(), b.end(), 2);
  std::fill(c.begin(), c.end(), 0);

  {
    // Create buffers associated with inputs and output
    sycl::buffer<int, 2>
      a_buf(a.data(), sycl::range<2>(N, M)),
      b_buf(b.data(), sycl::range<2>(N, M)),
      c_buf(c.data(), sycl::range<2>(N, M));

    sycl::range<2>
      a_range = a_buf.get_range(),
      b_range = b_buf.get_range();

    if (a_range==b_range) {

      // Submit the kernel to the queue
      Q.submit( [&]( sycl::handler &h ) {
  sycl::accessor a = a_buf.get_access<sycl::access::mode::read>(h);
  sycl::accessor b = b_buf.get_access<sycl::access::mode::read>(h);
  sycl::accessor c = c_buf.get_access<sycl::access::mode::write>(h);

  sycl::range<2> c_range = c.get_range();
  if (a_range==c_range) {
    h.parallel_for
      ( a_range,
[=]( sycl::id<2> idx ) {
c[idx] = a[idx] + b[idx];
```

```
      } );
  }
} );
    } else
      throw(std::out_of_range("array size mismatch"));
    // Q.submit( [&]( sycl::handler &h ) {
    //  sycl::id<2> one{1,1};
    //  sycl::accessor c = c_buf.get_access<sycl::access::mode::write>(h);
    //  sycl::range<2> c_range = c.get_range();
    //  c_range -= one;

    //  h.single_task( [&] () {} );
    //   });
  }

  // Check that all outputs match expected value
  bool passed = std::all_of(c.begin(), c.end(), [](int i) {
    return (i == 3);
  });
  std::cout << ((passed) ? "SUCCESS" : "FAILURE") << std::endl;
  return (passed) ? 0 : 1;
}
```

37.11.9 Listing of code code/dpcpp/cxx/hello.cxx

```
#include <CL/sycl.hpp>
#include <vector>

namespace sycl = cl::sycl;

int main(int argc, char **argv) {

  const auto global_range = 4;

  sycl::queue myQueue;
  std::cout << "Hello example running on "
            << myQueue.get_device().get_info<sycl::info::device::name>()
            << std::endl;
  // Create a command_group to issue command to the group
  myQueue.submit
    (
     [&](sycl::handler &cgh) {
       sycl::stream sout(1024, 256, cgh);
       cgh.parallel_for<class hello_world>
 (
  sycl::range<1>(global_range), [=](sycl::id<1> idx) {
    sout << "Hello, World: World rank " << idx << sycl::endl;
  }); // End of the kernel function
     }
     );       // End of the queue commands.
  myQueue.wait();

  return 0;
```

```
}
```

37.11.10 Listing of code code/dpcpp/cxx/jacobi1d.cxx

```
#include <algorithm>
#include <iostream>
#include <iomanip>
#include <numeric>
#include <random>
#include <vector>

#include <CL/sycl.hpp>
namespace sycl = cl::sycl;

int main() {
  sycl::queue queue;

  const size_t N = 16;
  sycl::range<1> unknowns(N);
  sycl::range<1> with_boundary(N + 2);
  std::vector<float>
    old_values(with_boundary.size(),0.),
    new_values(with_boundary.size(),0.);
  old_values.back() = 1.; new_values.back() = 1.;

  {
    sycl::buffer<float,1> new_buf(new_values.data(), with_boundary);
    sycl::buffer<float,1> old_buf(old_values.data(), with_boundary);

    sycl::event last_event = queue.submit( [&] (sycl::handler &h) {
        sycl::accessor
            old_array = old_buf.get_access<sycl::access::mode::write>(h);
        sycl::id<1> offset(1);
        h.parallel_for
            ( unknowns,offset,
              [=] (sycl::id<1> idx) {
                  int i = idx[0];
                  old_array[i] = 0.;
            } );
      } );
    for (int iteration=0; iteration<N; iteration++) {
      last_event = queue.submit( [&] (sycl::handler &h) {
          h.depends_on(last_event);
          sycl::accessor
              new_array = new_buf.get_access<sycl::access::mode::write>(h);
          sycl::accessor
              old_array = old_buf.get_access<sycl::access::mode::read>(h);
          sycl::id<1> offset(1);
          h.parallel_for
              ( unknowns, offset,
                [=] (sycl::id<1> idx) {
                    int i = idx[0];
                    float self = old_array[i];
```

```
                    float left = old_array[i - 1];
                    float righ = old_array[i + 1];
                    new_array[i] = (self + left + righ) / 3.0f;
                } );
        } );
      last_event = queue.submit( [&] (sycl::handler &h) {
          h.depends_on(last_event);
          sycl::accessor
              new_array = new_buf.get_access<sycl::access::mode::read>(h);
          sycl::accessor
              old_array = old_buf.get_access<sycl::access::mode::write>(h);
          sycl::id<1> offset(1);
          h.parallel_for
              ( unknowns, offset,
                [=] (sycl::id<1> idx) {
                    int i = idx[0];
                    old_array[i] = new_array[i];
              } );
        } );
    }
    queue.wait();

    for (int i=0; i<with_boundary.size(); i++) {
      std::cout << std::scientific << std::setprecision(3)
                << new_values[i] << " ";
    }
    std::cout << std::endl;

  }
  return 0;
}
```

PART V

THE REST

Chapter 38

Exploring computer architecture

There is much that can be said about computer architecture. However, in the context of parallel programming we are mostly concerned with the following:

- How many networked nodes are there, and does the network have a structure that we need to pay attention to?
- On a compute node, how many sockets (or other Non-Uniform Memory Access (NUMA) domains) are there?
- For each socket, how many cores and hyperthreads are there? Are caches shared?

38.1 Tools for discovery

An easy way for discovering the structure of your parallel machine is to use tools that are written especially for this purpose.

38.1.1 Intel cpuinfo

The *Intel compiler suite* comes with a tool *cpuinfo* that reports on the structure of the node you are running on. It reports on the number of *package*s, that is: sockets, cores, and threads.

38.1.2 hwloc

The open source package *hwloc* does similar reporting to cpuinfo, but it has been ported to many platforms. Additionally, it can generate ascii and pdf graphic renderings of the architecture.

38.2 Sources used in this chapter

38.2.1 Listing of code header

Chapter 39

Process and thread affinity

In the preceeding chapters we mostly considered all MPI nodes or OpenMP thread as being in one flat pool. However, for high performance you need to worry about *affinity*: the question of which process or thread is placed where, and how efficiently they can interact.

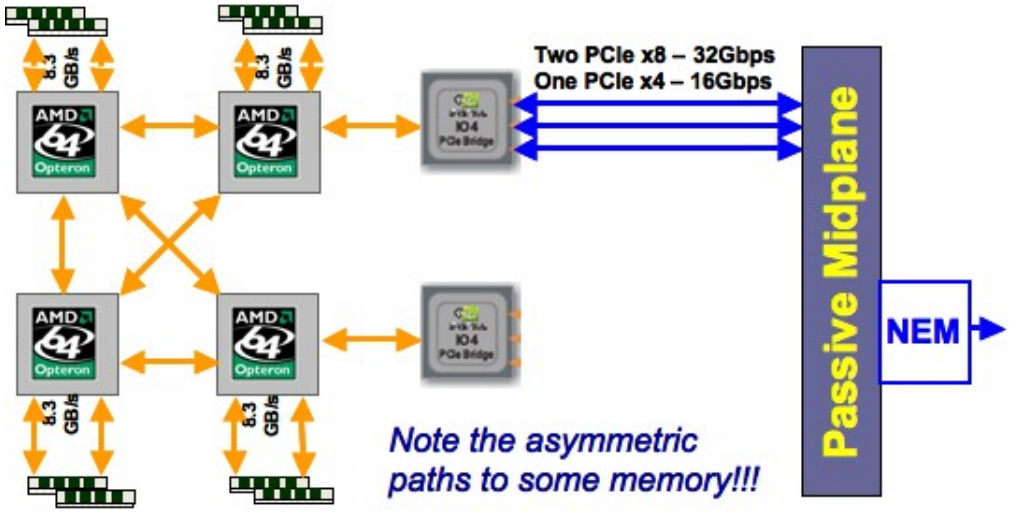


Figure 39.1: The NUMA structure of a Ranger node

Here are some situations where you affinity becomes a concern.

- In pure MPI mode processes that are on the same node can typically communicate faster than processes on different nodes. Since processes are typically placed sequentially, this means that a scheme where process p interacts mostly with $p + 1$ will be efficient, while communication with large jumps will be less so.
- If the cluster network has a structure (*processor grid* as opposed to *fat-tree*), placement of processes has an effect on program efficiency. MPI tries to address this with *graph topology*; section 11.2.
- Even on a single node there can be asymmetries. Figure 39.1 illustrates the structure of the four sockets of the *Ranger* supercomputer (no longer in production). Two cores have no direct connection.

This asymmetry affects both MPI processes and threads on that node.

- Another problem with multi-socket designs is that each socket has memory attached to it. While every socket can address all the memory on the node, its local memory is faster to access. This asymmetry becomes quite visible in the *first-touch* phenomemon; section 23.2.
- If a node has fewer MPI processes than there are cores, you want to be in control of their placement. Also, the operating system can migrate processes, which is detrimental to performance since it negates data locality. For this reason, utilities such as `numactl` (and at TACC `tacc_affinity`) can be used to *pin a thread* or process to a specific core.
- Processors with *hyperthreading* or *hardware threads* introduce another level or worry about where threads go.

39.1 What does the hardware look like?

If you want to optimize affinity, you should first know what the hardware looks like. The `hwloc` utility is valuable here [10] (https://www.open-mpi.org/projects/hwloc/).

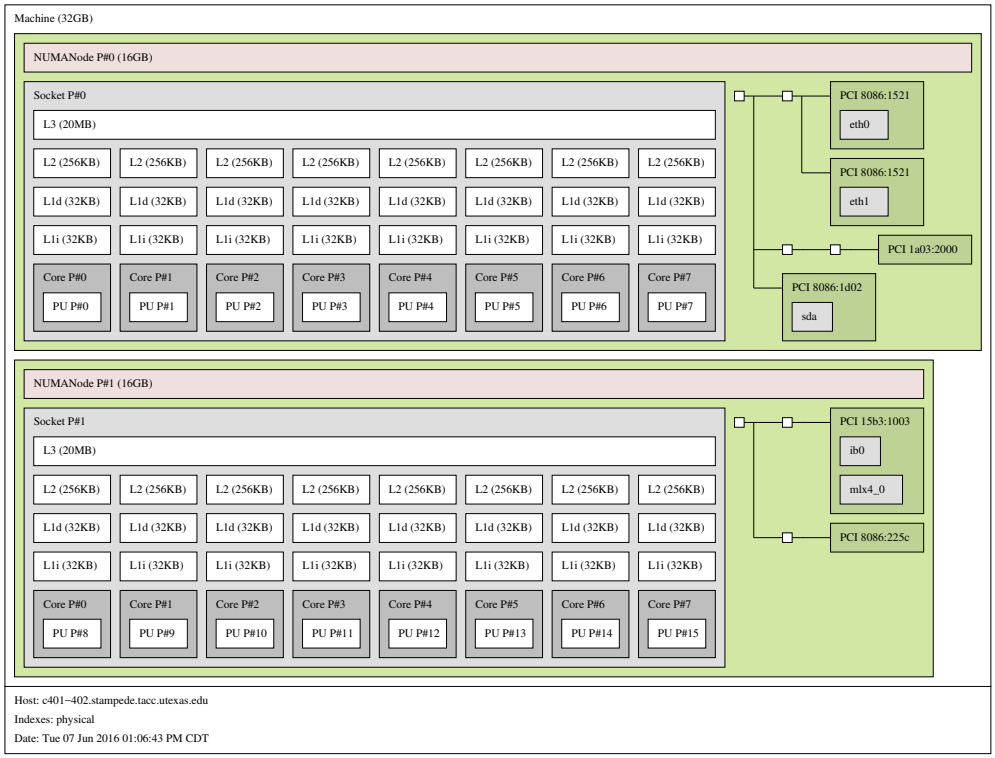


Figure 39.2: Structure of a Stampede compute node

Figure 39.2 depicts a *Stampede compute node*, which is a two-socket *Intel Sandybridge* design; figure 39.3 shows a *Stampede largemem node*, which is a four-socket design. Finally, figure 39.4 shows a *Lonestar5* compute node, a two-socket design with 12-core *Intel Haswell* processors with two hardware threads each.

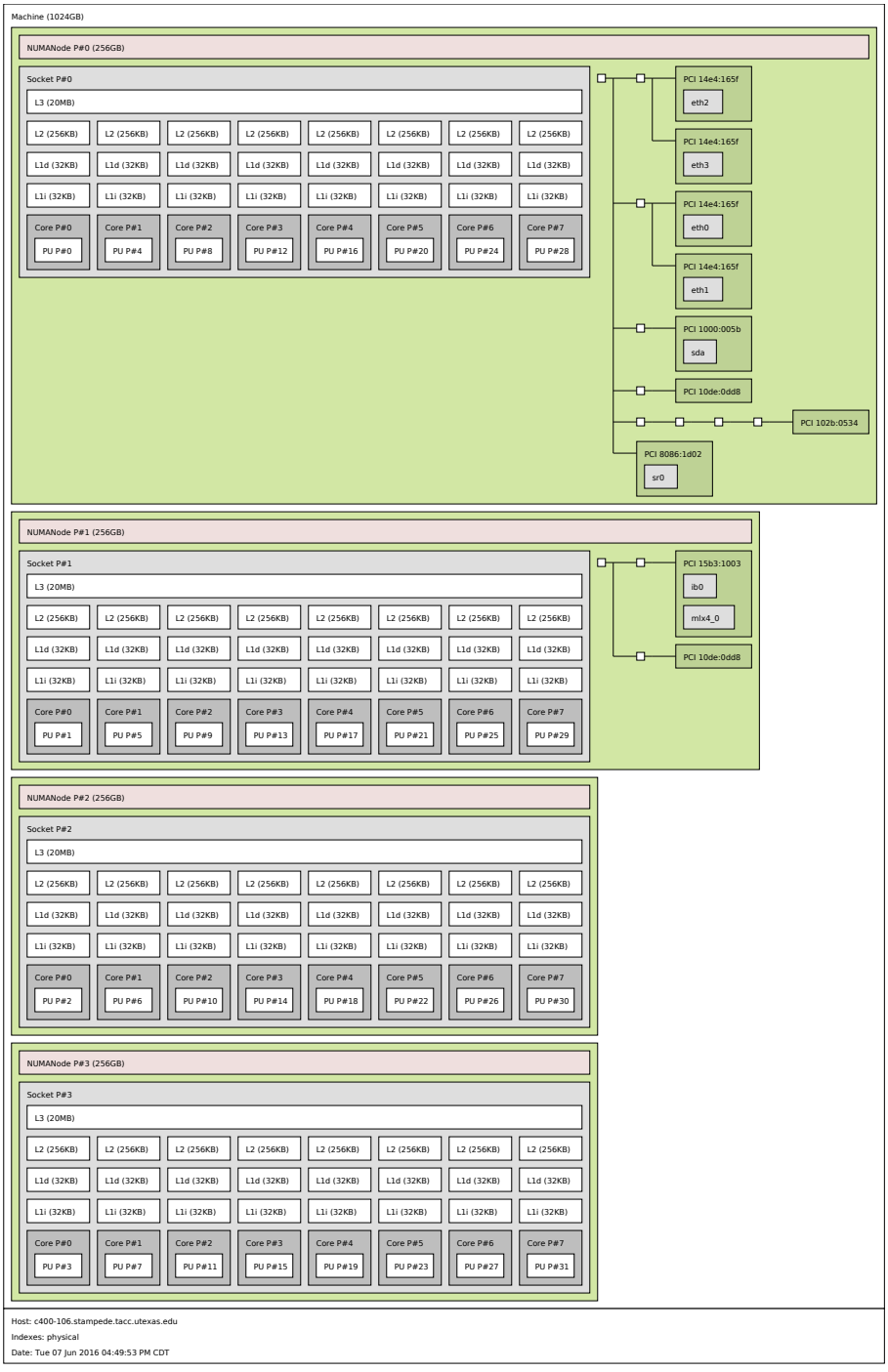


Figure 39.3: Structure of a Stampede largemem four-socket compute node

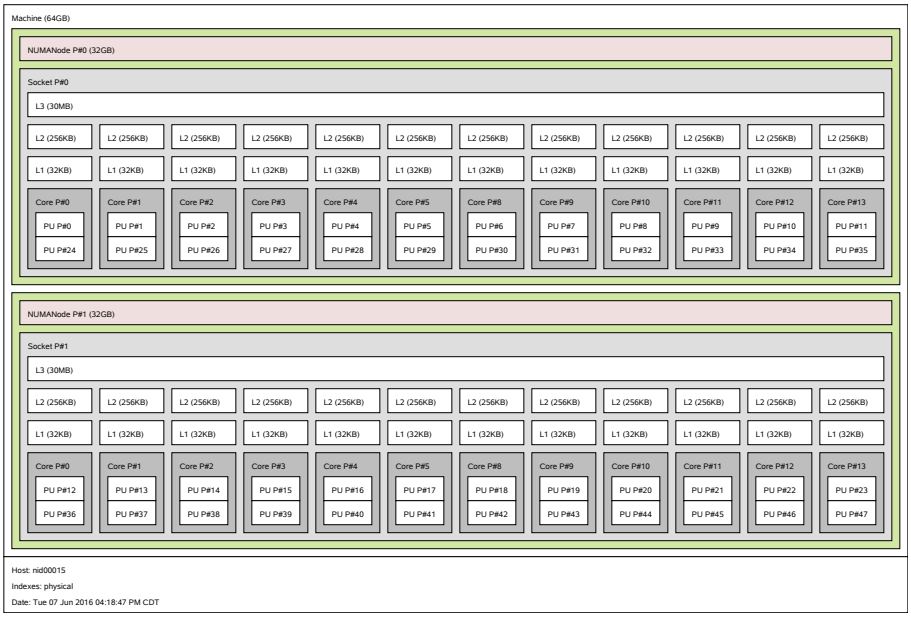


Figure 39.4: Structure of a Lonestar5 compute node

39.2 Affinity control

See chapter 23 for OpenMP affinity control.

39.3 Sources used in this chapter

39.3.1 Listing of code header

Chapter 40

Hybrid computing

So far, you have learned to use MPI for distributed memory and OpenMP for shared memory parallel programming. However, distribute memory architectures actually have a shared memory component, since each cluster node is typically of a multicore design. Accordingly, you could program your cluster using MPI for inter-node and OpenMP for intra-node parallelism.

Say you use 100 cluster nodes, each with 16 cores. You could then start 1600 MPI processes, one for each core, but you could also start 100 processes, and give each access to 16 OpenMP threads.

In your slurm scripts, the first scenario would be specified `-N 100 -n 1600`, and the second as

```
#$ SBATCH -N 100
#$ SBATCH -n 100

export OMP_NUM_THREADS=16
```

There is a third choice, in between these extremes, that makes sense. A cluster node often has more than one socket, so you could put one MPI process on each *socket*, and use a number of threads equal to the number of cores per socket.

The script for this would be:

```
#$ SBATCH -N 100
#$ SBATCH -n 200

export OMP_NUM_THREADS=8
ibrun tacc_affinity yourprogram
```

The `tacc_affinity` script unsets the following variables:

```
export MV2_USE_AFFINITY=0
export MV2_ENABLE_AFFINITY=0
export VIADEV_USE_AFFINITY=0
export VIADEV_ENABLE_AFFINITY=0
```

If you don't use `tacc_affinity` you may want to do this by hand, otherwise *mvapich2* will use its own affinity rules.

Figure 40.1 illustrates these three modes: pure MPI with no threads used; one MPI process per node and full multi-threading; two MPI processes per node, one per socket, and multiple threads on each socket.

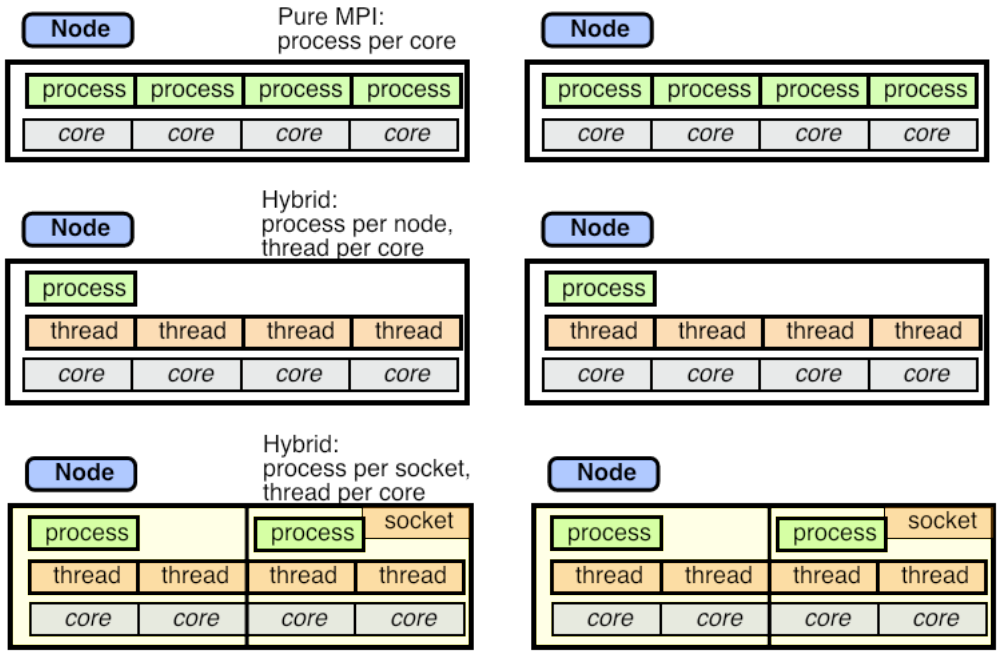


Figure 40.1: Three modes of MPI/OpenMP usage on a multi-core cluster

40.1 Discussion

The performance implications of the pure MPI strategy versus hybrid are subtle.

- First of all, we note that there is no obvious speedup: in a well balanced MPI application all cores are busy all the time, so using threading can give no immediate improvement.
- Both MPI and OpenMP are subject to Amdahl's law that quantifies the influence of sequential code; in hybrid computing there is a new version of this law regarding the amount of code that is MPI-parallel, but not OpenMP-parallel.
- MPI processes run unsynchronized, so small variations in load or in processor behaviour can be tolerated. The frequent barriers in OpenMP constructs make a hybrid code more tightly synchronized, so load balancing becomes more critical.
- On the other hand, in OpenMP codes it is easier to divide the work into more tasks than there are threads, so statistically a certain amount of load balancing happens automatically.
- Each MPI process has its own buffers, so hybrid takes less buffer overhead.

Exercise 40.1. Review the scalability argument for 1D versus 2D matrix decomposition in HPSC-6.2. Would you get scalable performance from doing a 1D decomposition (for instance, of the rows) over MPI processes, and decomposing the other directions (the columns) over OpenMP threads?

Another performance argument we need to consider concerns message traffic. If let all threads make MPI calls (see section 40.2) there is going to be little difference. However, in one popular hybrid computing strategy we would keep MPI calls out of the OpenMP regions and have them in effect done by the master thread. In that case there are only MPI messages between nodes, instead of between cores. This leads

Figure 40.1 `MPI_Init_thread`

Name	Param name	C type	F type	inout
`mpi_init_thread (`				
	`argc`	`int*`		`inout`
	`argv`	`char***`		`inout`
	`required` *desired level of thread support*	`int`	`INTEGER`	`in`
	`provided` *provided level of thread support*	`int*`	`INTEGER`	`out`
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Init_thread(`		
	`required`	`int = Ellipsis`
`) -> int`		

to a decrease in message traffic, though this is hard to quantify. The number of messages goes down approximately by the number of cores per node, so this is an advantage if the average message size is small. On the other hand, the amount of data sent is only reduced if there is overlap in content between the messages.

Limiting MPI traffic to the master thread also means that no buffer space is needed for the on-node communication.

40.2 Hybrid MPI-plus-threads execution

In hybrid execution, the main question is whether all threads are allowed to make MPI calls. To determine this, replace the `MPI_Init` call by `MPI_Init_thread` (figure 40.1) Here the `required` and `provided` parameters can take the following (monotonically increasing) values:

- `MPI_THREAD_SINGLE`: Only a single thread will execute.
- `MPI_THREAD_FUNNELED`: The program may use multiple threads, but only the main thread will make MPI calls.
  The main thread is usually the one selected by the `master` directive, but technically it is the only that executes `MPI_Init_thread`. If you call this routine in a parallel region, the main thread may be different from the master.
- `MPI_THREAD_SERIALIZED`: The program may use multiple threads, all of which may make MPI calls, but there will never be simultaneous MPI calls in more than one thread.
- `MPI_THREAD_MULTIPLE`: Multiple threads may issue MPI calls, without restrictions.

After the initialization call, you can query the support level with `MPI_Query_thread` (figure 40.2).

In case more than one thread performs communication, `MPI_Is_thread_main` (figure 40.3) can determine whether a thread is the main thread.

MPL note 59.

Figure 40.2 `MPI_Query_thread`

Name	Param name	C type	F type	inout
`mpi_query_thread (`				
	`provided`	`int*`	`INTEGER`	`out`
	provided level of thread support			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Query_thread(`		
`) -> int`		

Figure 40.3 `MPI_Is_thread_main`

Name	Param name	C type	F type	inout
`mpi_is_thread_main (`				
	`flag`	`int*`	`LOGICAL`	`out`
	true if calling thread is main thread, false otherwise			
`(opt)`	`ierror`		`INTEGER`	`out`
`)`				

Python:

Name	Param name	param type
`MPI.Is_thread_main(`		
`) -> bool`		

```
enum mpl::threading_modes {
  mpl::threading_modes::single = MPI_THREAD_SINGLE,
  mpl::threading_modes::funneled = MPI_THREAD_FUNNELED,
  mpl::threading_modes::serialized = MPI_THREAD_SERIALIZED,
  mpl::threading_modes::multiple = MPI_THREAD_MULTIPLE
};
threading_modes mpl::environment::threading_mode ();
bool mpl::environment::is_thread_main ();
```

End of MPL note

The *mvapich* implementation of MPI does have the required threading support, but you need to set this environment variable:

```
export MV2_ENABLE_AFFINITY=0
```

Another solution is to run your code like this:

```
  ibrun tacc_affinity <my_multithreaded_mpi_executable
```

Intel MPI uses an environment variable to turn on thread support:

```
I_MPI_LIBRARY_KIND=<value>
where
release : multi-threaded with global lock
release_mt : multi-threaded with per-object lock for thread-split
```

The *mpirun* program usually propagates *environment variables*, so the value of `OMP_NUM_THREADS` when you call `mpirun` will be seen by each MPI process.

- It is possible to use blocking sends in threads, and let the threads block. This does away with the need for polling.
- You can not send to a thread number: use the MPI *message tag* to send to a specific thread.

Exercise 40.2. Consider the 2D heat equation and explore the mix of MPI/OpenMP parallelism:

- Give each node one MPI process that is fully multi-threaded.
- Give each core an MPI process and don't use multi-threading.

Discuss theoretically why the former can give higher performance. Implement both schemes as special cases of the general hybrid case, and run tests to find the optimal mix.

```
// thread.c
MPI_Init_thread(&argc,&argv,MPI_THREAD_MULTIPLE,&threading);
comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm,&procno);
MPI_Comm_size(comm,&nprocs);

if (procno==0) {
  switch (threading) {
  case MPI_THREAD_MULTIPLE : printf("Glorious multithreaded MPI\n"); break;
  case MPI_THREAD_SERIALIZED : printf("No simultaneous MPI from threads\n"); break;
  case MPI_THREAD_FUNNELED : printf("MPI from main thread\n"); break;
  case MPI_THREAD_SINGLE : printf("no threading supported\n"); break;
  }
}
MPI_Finalize();
```

For the full source of this example, see section 40.3.2

40.3 Sources used in this chapter

40.3.1 Listing of code header

40.3.2 Listing of code examples/mpi/c/thread.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "mpi.h"

int main(int argc,char **argv) {

  MPI_Comm comm;
  int procno=-1,nprocs,threading,err;

  MPI_Init_thread(&argc,&argv,MPI_THREAD_MULTIPLE,&threading);
  comm = MPI_COMM_WORLD;
  MPI_Comm_rank(comm,&procno);
  MPI_Comm_size(comm,&nprocs);

  if (procno==0) {
    switch (threading) {
    case MPI_THREAD_MULTIPLE : printf("Glorious multithreaded MPI\n"); break;
    case MPI_THREAD_SERIALIZED : printf("No simultaneous MPI from threads\n"); break;
    case MPI_THREAD_FUNNELED : printf("MPI from main thread\n"); break;
    case MPI_THREAD_SINGLE : printf("no threading supported\n"); break;
    }
  }
  MPI_Finalize();
  return 0;
}
```

Chapter 41

Random number generation

Here is how you initialize the random number generator uniquely on each process:
C:

```
// Initialize the random number generator
srand((int)(mytid*(double)RAND_MAX/ntids));
// compute a random float between [0,1]
randomfraction = (rand() / (double)RAND_MAX);
// compute random integer between [0,N-1]
randomfraction = rand() % N;
```

Fortran:

```
  integer :: randsize
  integer,allocatable,dimension(:) :: randseed
  real :: random_value

  call random_seed(size=randsize)
  allocate(randseed(randsize))
  randseed(:) = 1023*mytid
  call random_seed(put=randseed)
```

41.1 Sources used in this chapter

41.1.1 Listing of code header

Chapter 42

Parallel I/O

Parallel I/O is a tricky subject. You can try to let all processors jointly write one file, or to write a file per process and combine them later. With the standard mechanisms of your programming language there are the following considerations:

- On clusters where the processes have individual file systems, the only way to write a single file is to let it be generated by a single processor.
- Writing one file per process is easy to do, but
  - You need a post-processing script;
  - if the files are not on a shared file system (such as *Lustre*), it takes additional effort to bring them together;
  - if the files *are* on a shared file system, writing many files may be a burden on the metadata server.
- On a shared file system it is possible for all files to open the same file and set the file pointer individually. This can be difficult if the amount of data per process is not uniform.

Illustrating the last point:

```
// pseek.c
FILE *pfile;
pfile = fopen("pseek.dat","w");
fseek(pfile,procid*sizeof(int),SEEK_CUR);
fseek(pfile,procid*sizeof(char),SEEK_CUR);
fprintf(pfile,"%d\n",procid);
fclose(pfile);
```

For the full source of this example, see section 42.1.2

MPI also has its own portable I/O: *MPI I/O*, for which see chapter 10.

Alternatively, one could use a library such as *hdf5*.

42.1 Sources used in this chapter

42.1.1 Listing of code header

42.1.2 Listing of code code/mpi/c/pseek.c

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc,char **argv) {
  int ntids,mytid;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&procid);

  FILE *pfile;
  pfile = fopen("pseek.dat","w");
  fseek(pfile,procid*sizeof(int),SEEK_CUR);
  fseek(pfile,procid*sizeof(char),SEEK_CUR);
  fprintf(pfile,"%d\n",procid);
  fclose(pfile);

  MPI_Finalize();

  return 0;
}
```

Chapter 43

Support libraries

There are many libraries related to parallel programming to make life easier, or at least more interesting, for you.

43.1 SimGrid

SimGrid [15] is a simulator for distributed systems. It can for instance be used to explore the effects of architectural parameters. It has been used to simulate large scale operations such as **HPL!** (**HPL!**) [4].

43.2 Other

ParaMesh

Global Arrays

Hdf5 and Silo

43.3 Sources used in this chapter

43.3.1 Listing of code header

PART VI

TUTORIALS

here are some tutorials specific to parallel programming.

1. Debugging, first sequential, then parallel. Chapter 43.4.
2. Tracing and profiling. Chapter 43.5.
3. SimGrid: a simulation tool for cluster execution. Chapter 43.6.
4. Batch systems, in particular Simple Linux Utility for Resource Management (SLURM). Chapter 43.7.

43.4 Debugging

When a program misbehaves, *debugging* is the process of finding out *why*. There are various strategies of finding errors in a program. The crudest one is debugging by print statements. If you have a notion of where in your code the error arises, you can edit your code to insert print statements, recompile, rerun, and see if the output gives you any suggestions. There are several problems with this:

- The edit/compile/run cycle is time consuming, especially since
- often the error will be caused by an earlier section of code, requiring you to edit, compile, and rerun repeatedly. Furthermore,
- the amount of data produced by your program can be too large to display and inspect effectively, and
- if your program is parallel, you probably need to print out data from all proccessors, making the inspection process very tedious.

For these reasons, the best way to debug is by the use of an interactive *debugger*, a program that allows you to monitor and control the behaviour of a running program. In this section you will familiarize yourself with *gdb*, which is the open source debugger of the *GNU* project. Other debuggers are proprietary, and typically come with a compiler suite. Another distinction is that gdb is a commandline debugger; there are graphical debuggers such as *ddd* (a frontend to gdb) or *DDT* and *TotalView* (debuggers for parallel codes). We limit ourselves to gdb, since it incorporates the basic concepts common to all debuggers.

In this tutorial you will debug a number of simple programs with gdb and valgrind. The files can be found in the repository in the directory `tutorials/debug_tutorial_files`.

43.4.1 Step 0: compiling for debug

You often need to recompile your code before you can debug it. A first reason for this is that the binary code typically knows nothing about what variable names corresponded to what memory locations, or what lines in the source to what instructions. In order to make the binary executable know this, you have to include the *symbol table* in it, which is done by adding the `-g` option to the compiler line.

Usually, you also need to lower the *compiler optimization level*: a production code will often be compiled with flags such as `-O2` or `-Xhost` that try to make the code as fast as possible, but for debugging you need to replace this by `-O0` ('oh-zero'). The reason is that higher levels will reorganize your code, making it hard to relate the execution to the source[1].

43.4.2 Invoking `gdb`

There are three ways of using gdb: using it to start a program, attaching it to an already running program, or using it to inspect a *core dump*. We will only consider the first possibility.

Here is an exaple of how to start gdb with program that has no arguments (Fortran users, use `hello.F`):

tutorials/gdb/c/hello.c

1. Typically, actual code motion is done by `-O3`, but at level `-O2` the compiler will inline functions and make other simplifications.

```
%% cc -g -o hello hello.c
# regular invocation:
%% ./hello
hello world
# invocation from gdb:
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
Copyright 2004 Free Software Foundation, Inc. .... copyright info ....
(gdb) run
Starting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. done
hello world

Program exited normally.
(gdb) quit
%%
```

Important note: the program was compiled with the *debug flag* `-g`. This causes the *symbol table* (that is, the translation from machine address to program variables) and other debug information to be included in the binary. This will make your binary larger than strictly necessary, but it will also make it slower, for instance because the compiler will not perform certain optimizations[2].

To illustrate the presence of the symbol table do

```
%% cc -g -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

and compare it with leaving out the `-g` flag:

```
%% cc -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

For a program with commandline input we give the arguments to the `run` command (Fortran users use `say.F`):

tutorials/gdb/c/say.c

```
%% cc -o say -g say.c
%% ./say 2
hello world
hello world
%% gdb say
```

2. Compiler optimizations are not supposed to change the semantics of a program, but sometimes do. This can lead to the nightmare scenario where a program crashes or gives incorrect results, but magically works correctly with compiled with debug and run in a debugger.

```
.... the usual messages ...
(gdb) run 2
Starting program: /home/eijkhout/tutorials/gdb/c/say 2
Reading symbols for shared libraries +. done
hello world
hello world

Program exited normally.
```

43.4.3 Finding errors

Let us now consider some programs with errors.

43.4.3.1 C programs

tutorials/gdb/c/square.c

```
%% cc -g -o square square.c
 %% ./square
5000
Segmentation fault
```

The *segmentation fault* (other messages are possible too) indicates that we are accessing memory that we are not allowed to, making the program abort. A debugger will quickly tell us where this happens:

```
%% gdb square
(gdb) run
50000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x000000000000eb4a
0x00007fff824295ca in __svfscanf_l ()
```

Apparently the error occurred in a function `__svfscanf_l`, which is not one of ours, but a system function. Using the `backtrace` (or `bt`, also `where` or `w`) command we quickly find out how this came to be called:

```
(gdb) backtrace
#0  0x00007fff824295ca in __svfscanf_l ()
#1  0x00007fff8244011b in fscanf ()
#2  0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at square.c:7
```

We take a close look at line 7, and see that we need to change `nmax` to `&nmax`.

There is still an error in our program:

```
(gdb) run
50000
```

```
Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_PROTECTION_FAILURE at address: 0x000000010000f000
0x0000000100000ebe in main (argc=2, argv=0x7fff5fbfc7a8) at square1.c:9
9           squares[i] = 1./(i*i); sum += squares[i];
```

We investigate further:

```
(gdb) print i
$1 = 11237
(gdb) print squares[i]
Cannot access memory at address 0x10000f000
```

and we quickly see that we forgot to allocate `squares`.

By the way, we were lucky here: this sort of memory errors is not always detected. Starting our programm with a smaller input does not lead to an error:

```
(gdb) run
50
Sum: 1.625133e+00

Program exited normally.
```

43.4.3.2 Fortran programs

Compile and run the following program:

tutorials/gdb/f/square.F It should abort with a message such as 'Illegal instruction'. Running the program in gdb quickly tells you where the problem lies:

```
(gdb) run
Starting program: tutorials/gdb//fsquare
Reading symbols for shared libraries ++++. done

Program received signal EXC_BAD_INSTRUCTION, Illegal instruction/operand.
0x0000000100000da3 in square () at square.F:7
7                 sum = sum + squares(i)
```

We take a close look at the code and see that we did not allocate `squares` properly.

43.4.4 Memory debugging with Valgrind

Insert the following allocation of `squares` in your program:

```
squares = (float *) malloc( nmax*sizeof(float) );
```

Compile and run your program. The output will likely be correct, although the program is not. Can you see the problem?

To find such subtle memory errors you need a different tool: a memory debugging tool. A popular (because open source) one is *valgrind*; a common commercial tool is *purify*.

tutorials/gdb/c/square1.c Compile this program with `cc -o square1 square1.c` and run it with `valgrind square1` (you need to type the input value). You will lots of output, starting with:

```
%% valgrind square1
==53695== Memcheck, a memory error detector
==53695== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==53695== Using Valgrind-3.6.1 and LibVEX; rerun with -h for copyright info
==53695== Command: a.out
==53695==
10
==53695== Invalid write of size 4
==53695==    at 0x100000EB0: main (square1.c:10)
==53695==  Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695==    at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695==    by 0x100000E77: main (square1.c:8)
==53695==
==53695== Invalid read of size 4
==53695==    at 0x100000EC1: main (square1.c:11)
==53695==  Address 0x10027e148 is 0 bytes after a block of size 40 alloc'd
==53695==    at 0x1000101EF: malloc (vg_replace_malloc.c:236)
==53695==    by 0x100000E77: main (square1.c:8)
```

Valgrind is informative but cryptic, since it works on the bare memory, not on variables. Thus, these error messages take some exegesis. They state that a line 10 writes a 4-byte object immediately after a block of 40 bytes that was allocated. In other words: the code is writing outside the bounds of an allocated array. Do you see what the problem in the code is?

Note that valgrind also reports at the end of the program run how much memory is still in use, meaning not properly `free`d.

If you fix the array bounds and recompile and rerun the program, valgrind still complains:

```
==53785== Conditional jump or move depends on uninitialised value(s)
==53785==    at 0x10006FC68: __dtoa (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x10003199F: __vfprintf (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x1000738AA: vfprintf_l (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x1000A1006: printf (in /usr/lib/libSystem.B.dylib)
==53785==    by 0x100000EF3: main (in ./square2)
```

Although no line number is given, the mention of `printf` gives an indication where the problem lies. The reference to an 'uninitialized value' is again cryptic: the only value being output is `sum`, and that is not uninitialized: it has been added to several times. Do you see why valgrind calls is uninitialized all the same?

43.4.5 Stepping through a program

Often the error in a program is sufficiently obscure that you need to investigate the program run in detail. Compile the following program

tutorials/gdb/c/roots.c and run it:

```
%% ./roots
sum: nan
```

Start it in gdb as follows:

```
%% gdb roots
GNU gdb 6.3.50-20050815 (Apple version gdb-1469) (Wed May  5 04:36:56 UTC 2010)
Copyright 2004 Free Software Foundation, Inc.
....
(gdb) break main
Breakpoint 1 at 0x100000ea6: file root.c, line 14.
(gdb) run
Starting program: tutorials/gdb/c/roots
Reading symbols for shared libraries +. done

Breakpoint 1, main () at roots.c:14
14        float x=0;
```

Here you have done the following:

- Before calling `run` you set a *breakpoint* at the main program, meaning that the execution will stop when it reaches the main program.
- You then call `run` and the program execution starts;
- The execution stops at the first instruction in main.

If execution is stopped at a breakpoint, you can do various things, such as issuing the `step` command:

```
Breakpoint 1, main () at roots.c:14
14        float x=0;
(gdb) step
15        for (i=100; i>-100; i--)
(gdb)
16          x += root(i);
(gdb)
```

(if you just hit return, the previously issued command is repeated). Do a number of `steps` in a row by hitting return. What do you notice about the function and the loop?

Switch from doing `step` to doing `next`. Now what do you notice about the loop and the function?

Set another breakpoint: `break 17` and do `cont`. What happens?

Rerun the program after you set a breakpoint on the line with the `sqrt` call. When the execution stops there do `where` and `list`.

- If you set many breakpoints, you can find out what they are with `info breakpoints`.
- You can remove breakpoints with `delete n` where `n` is the number of the breakpoint.
- If you restart your program with `run` without leaving gdb, the breakpoints stay in effect.
- If you leave gdb, the breakpoints are cleared but you can save them: `save breakpoints <file>`. Use `source <file>` to read them in on the next gdb run.

43.4.6 Inspecting values

Run the previous program again in gdb: set a breakpoint at the line that does the `sqrt` call before you actually call `run`. When the program gets to line 8 you can do `print n`. Do `cont`. Where does the program stop?

If you want to repair a variable, you can do `set var=value`. Change the variable `n` and confirm that the square root of the new value is computed. Which commands do you do?

If a problem occurs in a loop, it can be tedious keep typing `cont` and inspecting the variable with `print`. Instead you can add a condition to an existing breakpoint: the following:

```
condition 1 if (n<0)
```

or set the condition when you define the breakpoint:

```
break 8 if (n<0)
```

Another possibility is to use `ignore 1 50`, which will not stop at breakpoint 1 the next 50 times.

Remove the existing breakpoint, redefine it with the condition `n<0` and rerun your program. When the program breaks, find for what value of the loop variable it happened. What is the sequence of commands you use?

43.4.7 Parallel debugging

Debugging parallel programs is harder than than sequential programs, because every sequential bug may show up, plus a number of new types, caused by the interaction of the various processes.

Here are a few possible parallel bugs:

- Processes can *deadlock* because they are waiting for a message that never comes. This typically happens with blocking send/receive calls due to an error in program logic.
- If an incoming message is unexpectedly larger than anticipated, a memory error can occur.
- A collective call will hang if somehow one of the processes does not call the routine.

There are few low-budget solutions to parallel debugging. The main one is to create an xterm for each process. We will describe this next. There are also commercial packages such as *DDT* and *TotalView*, that offer a GUI. They are very convenient but also expensive. The *Eclipse* project has a parallel package, *Eclipse PTP*, that includes a graphic debugger.

43.4.7.1 MPI debugging with gdb

You can not run parallel programs in gdb, but you can start multiple gdb processes that behave just like MPI processes! The command

```
mpirun -np <NP> xterm -e gdb ./program
```

create a number of `xterm` windows, each of which execute the commandline `gdb ./program`. And because these xterms have been started with `mpirun`, they actually form a communicator.

43.4.8 Further reading

A good tutorial: http://www.dirac.org/linux/gdb/.

Reference manual: http://www.ofb.net/gnu/gdb/gdb_toc.html.

43.5 Tracing and profiling with TAU

TAU http://www.cs.uoregon.edu/Research/tau/home.php is a utility for profiling and tracing your parallel programs. Profiling is the gathering and displaying of bulk statistics, for instance showing you which routines take the most time, or whether communication takes a large portion of your runtime. When you get concerned about performance, a good profiling tool is indispensible.

Tracing is the construction and displaying of time-dependent information on your program run, for instance showing you if one process lags behind others. For understanding a program's behaviour, and the reasons behind profiling statistics, a tracing tool can be very insightful.

43.5.1 Workflow

43.5.1.1 Instrumentation

Unlike such tools as *VTune* which profile your binary as-is, TAU works by adding *instrumentation* to your code: in effect it is a source-to-source translator that takes your code and turns it into one that generates run-time statistics.

This instrumentation is largely done for you; you mostly need to recompile your code with a script that does the source-to-source translation, and subsequently compiles that instrumented code. You could for instance have the following in your makefile:

```
ifdef TACC_TAU_DIR
  CC = tau_cc.sh
else
  CC = mpicc
endif

% : %.c
<TAB>${CC} -o $@ $^
```

If TAU is to be used (which we detect here by checking for the environment variable `TACC_TAU_DIR`), we define the `CC` variable as one of the TAU compilation scripts; otherwise we set it to a regular MPI compiler.

43.5.1.2 Running

You can now run your instrumented code; trace/profile output will be written to file if environment variables `TAU_PROFILE` and/or `TAU_TRACE` are set:

```
export TAU_PROFILE=1
export TAU_TRACE=1
```

A TAU run can generate many files: typically at least one per process. It is therefore advisabe to create a directory for your tracing and profiling information. You declare them to TAU by setting the environment variables `PROFILEDIR` and `TRACEDIR`.

```
mkdir tau_trace
mkdir tau_profile
```

```
export PROFILEDIR=tau_profile
export TRACEDIR=tau_trace
```

The actual program invocation is then unchanged:

```
mpirun -np 26 myprogram
```

> *TACC note.* At TACC, use `ibrun` without a processor count; the count is derived from the queue submission parameters.

While this example uses two separate directories, there is no harm in using the same for both.

43.5.1.3 Output

The tracing/profiling information is spread over many files, and hard to read as such. Therefore, you need some further programs to consolidate and display the information.

You view profiling information with `paraprof`

```
paraprof tau_profile
```

Viewing the traces takes a few steps:

```
cd tau_trace
rm -f tau.trc tau.edf align.trc align.edf
tau_treemerge.pl
tau_timecorrect tau.trc tau.edf align.trc align.edf
tau2slog2 align.trc align.edf -o yourprogram.slog2
```

If you skip the `tau_timecorrect` step, you can generate the `slog2` file by:

```
tau2slog2 tau.trc tau.edf -o yourprogram.slog2
```

The `slog2` file can be viewed with `jumpshot`:

```
jumpshot yourprogram.slog2
```

43.5.2 Examples

43.5.2.1 Bucket brigade

Let's consider a *bucket brigade* implementation of a broadcast: each process sends its data to the next higher rank.

```
int sendto =
    ( procno<nprocs-1 ? procno+1 : MPI_PROC_NULL )
    ;
int recvfrom =
    ( procno>0 ? procno-1 : MPI_PROC_NULL )
    ;

MPI_Recv( leftdata,1,MPI_DOUBLE,recvfrom,0,comm,MPI_STATUS_IGNORE);
myvalue = leftdata
MPI_Send( myvalue,1,MPI_DOUBLE,sendto,0,comm);
```

We implement the bucket brigade with blocking sends and receives: each process waits to receive from its predecessor, before sending to its successor.

```
// bucketblock.c
for (int i=0; i<N; i++)
  myvalue[i] = (procno+1)*(procno+1) + leftdata[i];
MPI_Send( myvalue,N,MPI_DOUBLE,sendto,0,comm);
```



Figure 43.1: Trace of a bucket brigade broadcast

The TAU trace of this is in figure 43.1, using 4 nodes of 4 ranks each. We see that the processes within each node are fairly well synchronized, but there is less synchronization between the nodes. However, the bucket brigade then imposes its own synchronization on the processes because each has to wait for its predecessor, no matter if it posted the receive operation early.

Next, we introduce pipelining into this operation: each send is broken up into parts, and these parts are sent and received with non-blocking calls.

```
// bucketpipenonblock.c
MPI_Request rrequests[PARTS];
for (int ipart=0; ipart<PARTS; ipart++) {
  MPI_Irecv
    (
     leftdata+partition_starts[ipart],partition_sizes[ipart],
     MPI_DOUBLE,recvfrom,ipart,comm,rrequests+ipart);
}
```

The TAU trace is in figure 43.2.

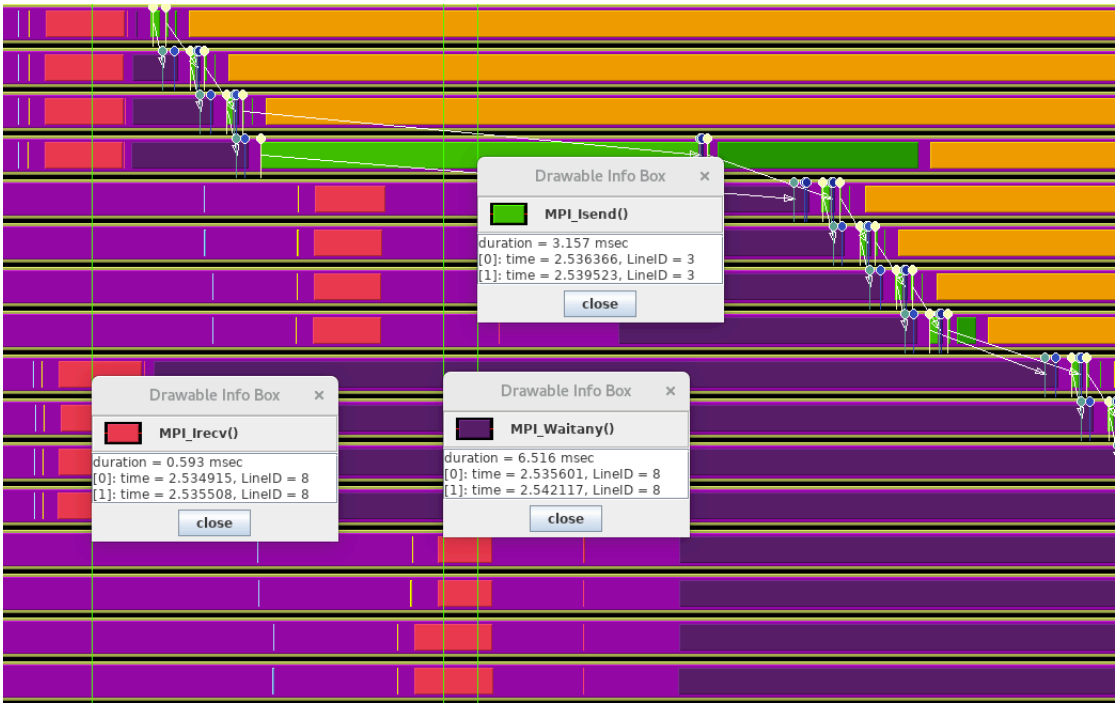


Figure 43.2: Trace of a pipelined bucket brigade broadcast

43.5.2.2 Butterfly exchange

The NAS Parallel Benchmark suite [21] contains a Conjugate Gradients (CG) implementation that spells out its all-reduce operations as a *butterfly exchange*.

```
// cgb.f
    do i = 1, l2npcols
       call mpi_irecv( d,
>                      1,
>                      dp_type,
>                      reduce_exch_proc(i),
>                      i,
>                      mpi_comm_world,
>                      request,
>                      ierr )
       call mpi_send(  sum,
>                      1,
>                      dp_type,
>                      reduce_exch_proc(i),
>                      i,
>                      mpi_comm_world,
>                      ierr )

       call mpi_wait( request, status, ierr )

       sum = sum + d
```

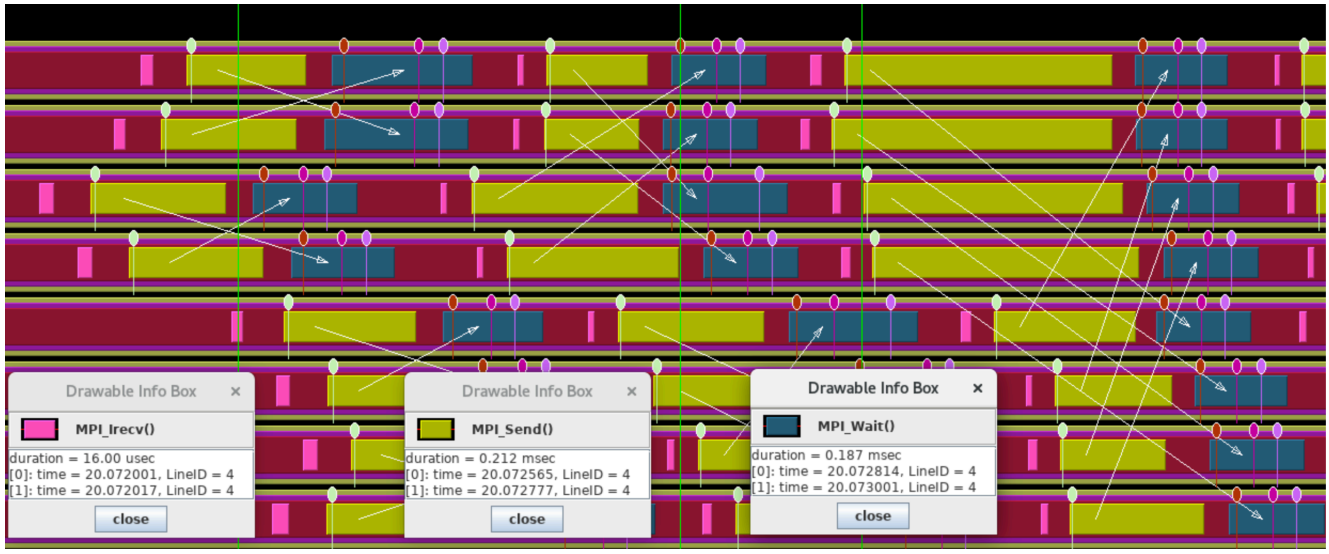


Figure 43.3: Trace of a butterfly exchange

```
    enddo
```

We recognize this structure in the TAU trace: figure 43.3. Upon closer examination, we see how this particular algorithm induces a lot of wait time. Figures 43.5 and 43.6 show a whole cascade of processes waiting for each other.

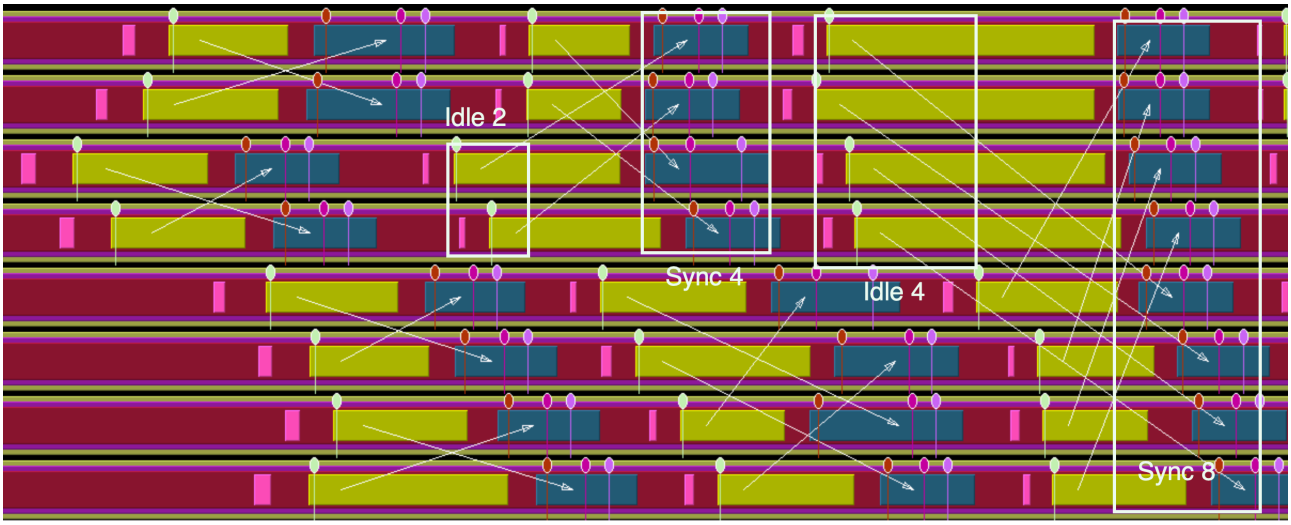


Figure 43.4: Trace of a butterfly exchange

43.6 SimGrid

Many readers of this book will have access to some sort of parallel machine so that they can run simulations, maybe even some realistic scaling studies. However, not many people will have access to more than one cluster type so that they can evaluate the influence of the *interconnect*. Even then, for didactic purposes one would often wish for interconnect types (fully connected, linear processor array) that are unlikely to be available.

In order to explore architectural issues pertaining to the network, we then resort to a simulation tool, *SimGrid*.

Installation

Compilation You write plain MPI files, but compile them with the *SimGrid compiler* `smpicc`.

Running SimGrid has its own version of `mpirun`: `smpirun`. You need to supply this with options:

- `-np 123456` for the number of (virtual) processors;
- `-hostfile simgridhostfile` which lists the names of these processors. You can basically make these up, but are defined in:
- `-platform arch.xml` which defines the connectivity between the processors.

For instance, with a hostfile of 8 hosts, a linearly connected network would be defined as:

```
<?xml version='1.0'?>
<!DOCTYPE platform SYSTEM "http://simgrid.gforge.inria.fr/simgrid/simgrid.dtd">
```

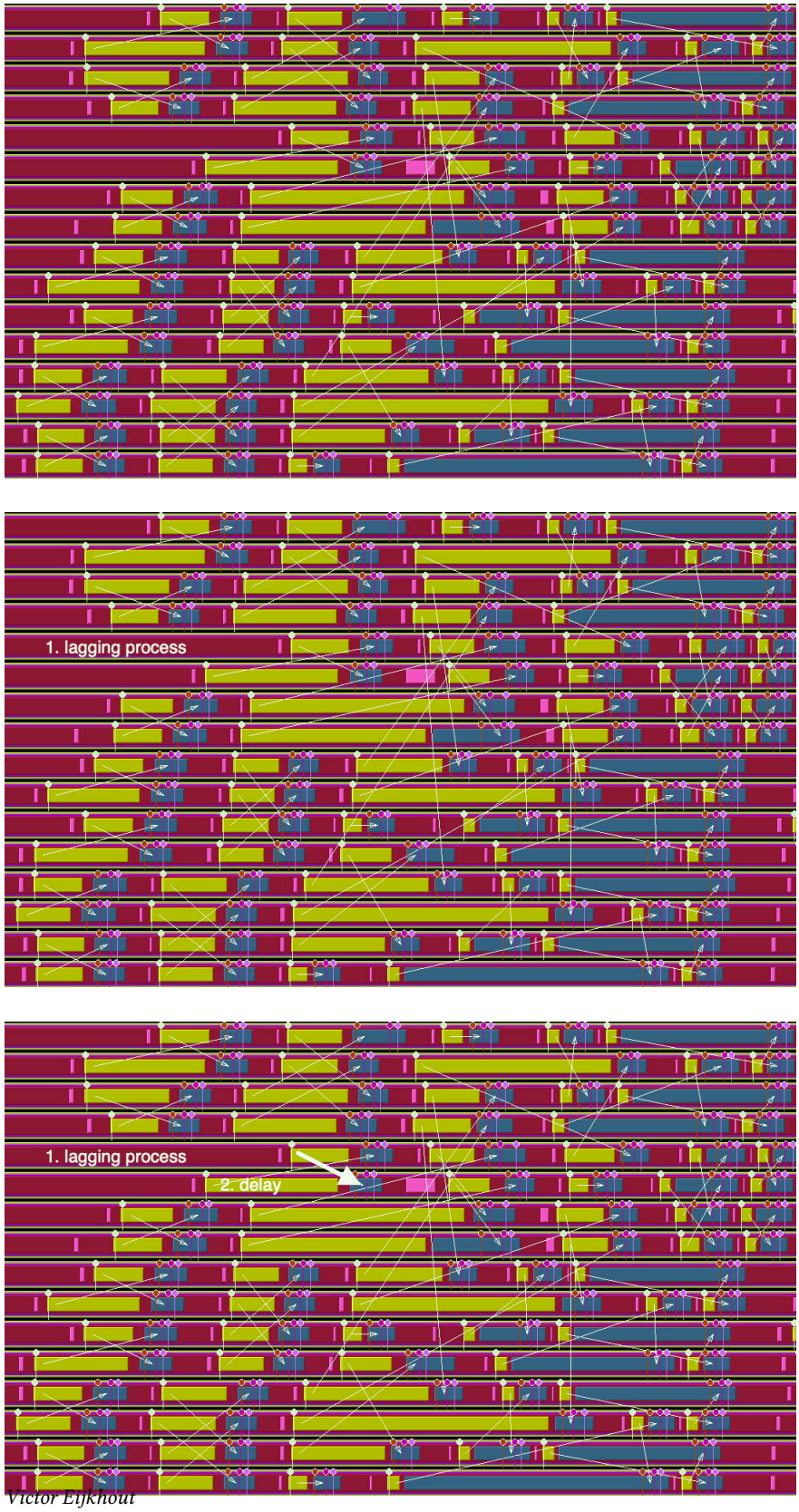


Figure 43.5: Four stages of processes waiting caused by a single lagging process

```
<platform version="4">

<zone id="first zone" routing="Floyd">
  <!-- the resources -->
  <host id="host1" speed="1Mf"/>
  <host id="host2" speed="1Mf"/>
  <host id="host3" speed="1Mf"/>
  <host id="host4" speed="1Mf"/>
  <host id="host5" speed="1Mf"/>
  <host id="host6" speed="1Mf"/>
  <host id="host7" speed="1Mf"/>
  <host id="host8" speed="1Mf"/>
  <link id="link1" bandwidth="125MBps" latency="100us"/>
  <!-- the routing: specify how the hosts are interconnected -->
  <route src="host1" dst="host2"><link_ctn id="link1"/></route>
  <route src="host2" dst="host3"><link_ctn id="link1"/></route>
  <route src="host3" dst="host4"><link_ctn id="link1"/></route>
  <route src="host4" dst="host5"><link_ctn id="link1"/></route>
  <route src="host5" dst="host6"><link_ctn id="link1"/></route>
  <route src="host6" dst="host7"><link_ctn id="link1"/></route>
  <route src="host7" dst="host8"><link_ctn id="link1"/></route>
</zone>

</platform>
```

(such files are easily generated with a shell script).

The `Floyd` designation of the routing means that any route using the transitive closure of the paths given can be used. It is also possible to use `routing="Full"` which requires full specification of all pairs that can communicate.

43.7 Batch systems

Supercomputer *clusters* can have a large number of *nodes*, but not enough to let all their users run simultaneously, and at the scale that they want. Therefore, users are asked to submit *jobs*, which may start executing immediately, or may have to wait until resources are available.

The decision when to run a job, and what resources to give it, is not done by a human operator, but by software called a *batch system*. (The *Stampede* cluster at *TACC* ran close to 10 million jobs over its lifetime, which corresponds to starting a job every 20 seconds.)

This tutorial will cover the basics of such systems, and in particular Simple Linux Utility for Resource Management (SLURM).

43.7.1 Cluster structure

A supercomputer cluster usually has two types of nodes:

- *login nodes*, and
- *compute nodes*.

When you make an *ssh connection* to a cluster, you are connecting to a login node. The number of login nodes is small, typically less than half a dozen.

Exercise 43.1. Connect to your favourite cluster. How many people are on that login node? If you disconnect and reconnect, do you find yourself on the same login node?

Compute nodes are where your jobs are run. Different clusters have different structures here:

- Compute nodes can be shared between users, or they can be assigned exclusively.
  - Sharing makes sense if user jobs have less parallelisn than the core count of a node.
  - ... on the other hand, it means that users sharing a node can interfere with each other's jobs, with one job using up memory or bandwidth that the other job needs.
  - With exclusive nodes, a job has access to all the memory and all the bandwidth of that node.
- Clusters can homogeneous, having the same processor type on each compute node, or they can have more than one processor type. For instance, the TACC *Stampede2* cluster has *Intel Knightslanding* and *Intel Skylake* nodes.
- Often, clusters have a number of 'large memory' nodes, on the order of a Terabyte of memory or more. Because of the cost of such hardware, there is usually only a small number of these nodes.

43.7.2 Queues

Jobs often can not start immediately, because not enough resources are available, or because other jobs may have higher priority (see section 43.7.7). It is thus typical for a job to be put on a *queue*, scheduled, and started, by a batch system such as SLURM.

Batch systems do not put all jobs in one big pool: jobs are submitted to any of a number of queues, that are all scheduled separately.

Queues can differ in the following ways:

- If a cluster has different processor types, those are typically in different queues. Also, there may be separate queues for the nodes that have a Graphics Processing Unit (GPU) attched. Having multiple queues means you have to decide what processor type you want your job to run on, even if your executable is binary compatible with all of them.
- There can be 'development' queues, which have restrictive limits on runtime and node count, but where jobs typically start faster.
- Some clusters have 'premium' queues, which have a higher charge rate, but offer higher priority.
- 'Large memory nodes' are typically also in a queue of their own.
- There can be further queues for jobs with large resource demands, such as large core counts, or longer-than-normal runtimes.

For slurm, the `sinfo` command can tell you much about the queues.

```
# what queues are there?
sinfo -o "%P"
# what queues are there, and what is their status?
sinfo -o "%20P %.5a"
```

Exercise 43.2. Enter these commands. How many queues are there? Are they all operational at the moment?

43.7.2.1 Queue limits

Queues have limits on

- the runtime of a job;
- the node count of a job; or
- how many jobs a user can have in that queue.

43.7.3 Job running

There are two main ways of starting a job on a cluster that is managed by slurm. You can start a program run synchronously with `srun`, but this may hang until resources are available. In this section, therefore, we focus on asynchronously executing your program by submitting a job with `sbatch`.

43.7.3.1 The job submission cycle

In order to run a *batch job*, you need to write a *job script*, or *batch script*. This script describes what program you will run, where its inputs and outputs are located, how many processes it can use, and how long it will run.

In its simplest form, you submit your script without further parameters:

```
sbatch yourscript
```

All options regarding the job run are contained in the script file, as we will now discuss.

As a result of your job submission you get a job id. After submission you can queury your job with `squeue`:

```
squeue -j 123456
```

or queury all your jobs:

```
squeue -u yourname
```

The `squeue` command reports various aspects of your job, such as its status (typically pending or running); and if it is running, the queue (or 'partition') where it runs, its elapsed time, and the actual nodes where it runs.

```
squeue -j 5807991
  JOBID   PARTITION     NAME     USER ST   TIME  NODES NODELIST(REASON)
5807991 development packingt eijkhout  R   0:04      2 c456-[012,034]
```

If you discover errors in your script after submitting it, including when it has started running, you can cancel your job with `scancel`:

```
scancel 1234567
```

43.7.4 The script file

A job script looks like an executable shell script:

- It has an 'interpreter' line such as
  `#!/bin/bash`
  at the top, and
- it contains ordinary unix commands, including
- the (parallel) startup of you program:
  ```
  # sequential program:
  ./yourprogram youroptions
  # parallel program, general:
  mpiexec -n 123 parallelprogram options
  # parallel program, TACC:
  ibrun parallelprogram options
  ```
- ... and then it has many options specifying the parallel run.

43.7.4.1 sbatch options

In addition to the regular unix commands and the interpreter line, your script has a number of SLURM directives, each starting with `#SBATCH`. (This makes them comments to the shell interpreter, so a batch script is actually a legal shell script.)

Directives have the form

```
#SBATCH -option value
```

Common options are:

- `-J`: the jobname. This will be displayed when you call `squeue`.
- `-o`: name of the output file. This will contain all the stdout output of the script.
- `-e`: name of the error file. This will contain all the stderr output of the script, as well as slurm error messages.

It can be a good idea to make the output and error file unique per job. To this purpose, the macro `%j` is available, which at execution time expands to the job number. You will then get an output file with a name such as `myjob.o2384737`.

- `-p`: the *partition* or queue. See above.
- `-t hh:mm:ss`: the maximum running time. If your job exceeds this, it will get *cancelled*. Two considerations:
  1. You can not specify a duration here that is longer than the queue limit.
  2. The shorter your job, the more likely it is to get scheduled sooner rather than later.
- `-A`: the name of the account to which your job should be billed.
- `--mail-user=you@where` Slurm can notify you when a job starts or ends. You may for instance want to connect to a job when it starts (to run `top`), or inspect the results when it's done, but not sit and stare at your terminal all day. The action of which you want to be notified is specified with (among others) `--mail-type=begin/end/fail/all`
- `--dependency=after:123467` indicates that this job is to start after jobs `1234567` finished. Use `afterok` to start only if that job successfully finished. (See https://cvw.cac.cornell.edu/slurm/submission_depend for more options.)
- `--nodelist` allows you to specify specific nodes. This can be good for getting reproducible timings, but it will probably increase your wait time in the queue.
- `--array=0-30` is a specification for 'array jobs': a task that needs to be executed for a range of parameter values.

  TACC note. Arry jobs are not supported at TACC; use a launcher instead; section 43.7.5.3.
- `--mem=10000` specifies the desired amount of memory per node. Default units are megabytes, but can be explicitly indicated with `K/M/G/T`.

  TACC note. This option can not be used to request arbitrary memory: jobs always have access to all available physical memory, and use of shared memory is not allowed.

See https://slurm.schedmd.com/sbatch.html for a full list.

Exercise 43.3. Write a script that executes the `date` command twice, with a `sleep` in between. Submit the script and investigate the output.

43.7.4.2 Environment

Your job script acts like any other shell script when it is executed. In particular, it inherits the calling *environment* with all its environment variables. Additionally, slurm defines a number of environment variables, such as the job ID, the hostlist, and the node and process count.

43.7.5 Parallelism handling

We discuss parallelism options separately.

43.7.5.1 MPI jobs

On most clusters there is a structure with compute nodes, that contain one or more multi-core processors. Thus, you want to specify the node and core count. For this, there are options `-N` and `-n` respectively.

```
#SBATCH -N 4              # Total number of nodes
#SBATCH -n 4              # Total number of mpi tasks
```

It would be possible to specify only the node count or the core count, but that takes away flexibility:

- If a node has 40 cores, but your program stops scaling at 10 MPI ranks, you would use:
  ```
  #SBATCH -N 1
  #SBATCH -n 10
  ```
- If your processes use a large amount of memory, you may want to leave some cores unused. On a 40-core node you would either use
  ```
  #SBATCH -N 2
  #SBATCH -n 40
  ```
  or
  ```
  #SBATCH -N 1
  #SBATCH -n 20
  ```

Rather than specifying a total core count, you can also specify the core count per node with `--ntasks-per-node`.

Exercise 43.4. Go through the above examples and replace the `-n` option by an equivalent `--ntasks-per-node` values.

Python note. MPI programs, except that instead of an executable name you specify the python executable and the script name:

```
ibrun python3 mympi4py.py
```

43.7.5.2 Threaded jobs

The above discussion was mostly of relevance to MPI programs. Some other cases:

- For pure-OpenMP programs you need only one node, so the `-N` value is 1. Maybe surprisingly, the `-n` value is also 1, since only one process needs to be created: OpenMP uses thread-level parallelism, which is specified through the `OMP_NUM_THREADS` environment variable.
- A similar story holds for the *Matlab parallel computing toolbox* (note: note the distributed computing toolbox), and the *Python multiprocessing* module.

Exercise 43.5. What happens if you specify an `-n` value greater than 1 for a pure-OpenMP program?

For *hybrid computing* MPI-OpenMP programs, you use a combination of slurm options and enviroment variables, such that, for instance, the product of the `--tasks-per-node` and `OMP_NUM_THREADS` is less than the core count of the node.

43.7.5.3 Parameter sweeps / ensembles / massively parallel

So far we have focused on jobs where a single parallel executable is scheduled. However, there are use cases where you want to run a sequential (or very modestly parallel) executable for a large number of inputs. This is called variously a *parameter sweep* or an *ensemble*.

Slurm can support this natively with *array jobs*.

> *TACC note.* TACC clusters do not support array jobs. Instead, use the `launcher` or `pylauncher` modules.

43.7.6 Job running

When your job is running, its status is reported as `R` by `squeue`. That command also reports which nodes are allocated to it.

```
squeue -j 5807991
       JOBID   PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
     5807991 development packingt eijkhout  R       0:04      2 c456-[012,034]
```

You can then `ssh` into the compute nodes of your job; normally, compute nodes are off-limits. This is useful if you want to run `top` to see how your processes are doing.

43.7.7 Scheduling strategies

Such a system looks at resource availability and the user's priority to determine when a job can be run.

Of course, if a user is requesting a large number of nodes, it may never happen that that many become available simultaneously, so the batch system will force the availability. It does so by determining a time when that job is set to run, and then let nodes go *idle* so that they are available at that time.

An interesting side effect of this is that, right before the really large job starts, a 'fairly' large job can be run, if it only has a short running time. This is known as *backfill*, and it may cause jobs to be run earlier than their priority would warrant.

43.7.8 File systems

File systems come in different types:

- They can be backed-up or not;
- they can be shared or not; and
- they can be permanent or purged.

On many clusters each node has as local disc, either spinning or a *RAM disc*. This is usually limited in size, and should only be used for temporary files during the job run.

Most of the file system lives on discs that are part of *RAID arrays*. These discs have a large amount of redundancy to make them fault-tolerant, and in aggregate they form a *shared file system*: one unified file system that is accessible from any node and where files can take on any size, or at least much larger than any individual disc in the system.

TACC note. The `HOME` file system is limited in size, but is both permanent and backed up. Here you put scripts and sources.

The `WORK` file system is permanent but not backed up. Here you can store output of your simulations. However, currently the work file system can not immediately sustain the output of a large parallel job.

The `SCRATCH` file system is purged, but it has the most bandwidth for accepting program output. This is where you would write your data. After post-processing, you can then store on the work file system, or write to tape.

Exercise 43.6. If you install software with *cmake*, you typically have
1. a script with all your cmake options;
2. the sources,
3. the installed header and binary files
4. temporary object files and such.

How would you orgnize these entities over your available file systems?

43.7.9 Examples

Very sketchy section.

43.7.9.1 Job dependencies

```
JOB=`sbatch my_batchfile.sh | egrep -o -e "\b[0-9]+$"`

#!/bin/sh

# Launch first job
JOB=`sbatch job.sh | egrep -o -e "\b[0-9]+$"`

# Launch a job that should run if the first is successful
sbatch --dependency=afterok:${JOB} after_success.sh

# Launch a job that should run if the first job is unsuccessful
sbatch --dependency=afternotok:${JOB} after_fail.sh
```

43.7.9.2 Multiple runs in one script

```
ibrun stuff &
sleep 10
for h in hostlist ; do
  ssh $h "top"
done
wait
```

43.7.10 Review questions

For all true/false questions, if you answer False, what is the right answer and why?

Exercise 43.7. T/F? When you submit a job, it starts running immediately once sufficient resources are available.

Exercise 43.8. T/F? If you submit the following script:

```
#!/bin/bash
#SBATCH -N 10
#SBATCH -n 10
echo "hello world"
```
you get 10 lines of 'hello world' in your output.

Exercise 43.9. T/F? If you submit the following script:

```
#!/bin/bash
#SBATCH -N 10
#SBATCH -n 10
hostname
```
you get the hostname of the login node from which your job was submitted.

Exercise 43.10. Which of these are shared with other users when your job is running:

- Memory;
- CPU;
- Disc space?

Exercise 43.11. What is the command for querying the status of your job?

- `sinfo`
- `squeue`
- `sacct`

Exercise 43.12. On 4 nodes with 40 cores each, what's the largest program run, measured in

- MPI ranks;
- OpenMP threads?

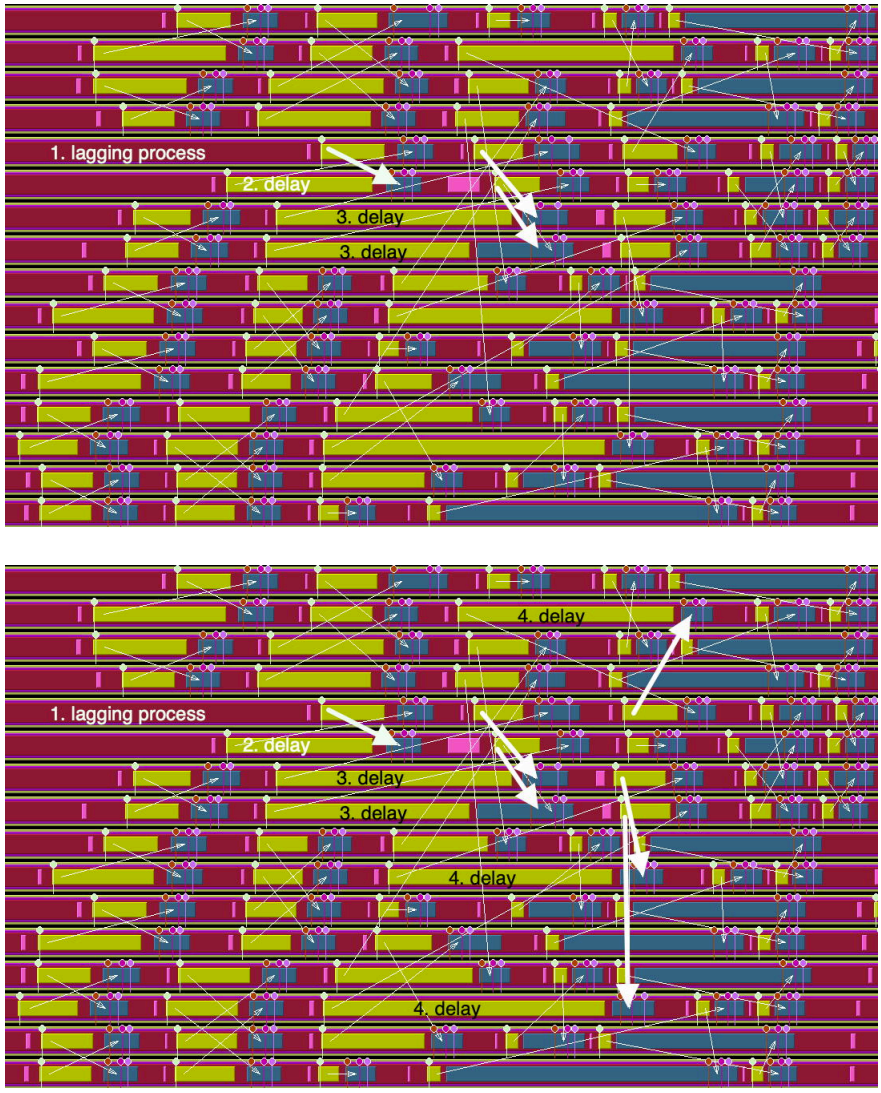


Figure 43.6: Four stages of processes waiting caused by a single lagging process

PART VII

CLASS PROJECTS

Chapter 44

A Style Guide to Project Submissions

Here are some guidelines for how to submit assignments and projects. As a general rule, consider programming as an experimental science, and your writeup as a report on some tests you have done: explain the problem you're addressing, your strategy, your results.

Structure of your writeup Most of the projects in this book use a scientific question to allow you to prove your coding skills. That does not mean that turning in the code is sufficient, nor code plus sample output. Turn in a writeup in pdf form that was generated from a text processing program such (preferably) LaTeX (for a tutorial, see HPSC-31).

Your writeup should have:

- Foremost, a short description of the purpose of your project and your results;
- An explanation of your algorithms or solution strategy;
- Relevant fragments of your code;
- A scientific discussion of what you observed,
- Any code-related observations.
- If applicable: graphs, both of application quantities and performance issues. (For parallel runs possibly TAU plots; see 43.5).

Observe, measure, hypothesize, deduce Your project may be a scientific investigation of some phenomenon. Formulate hypotheses as to what you expect to observe, report on your observations, and draw conclusions.

Quite often your program will display unexpected behaviour. It is important to observe this, and hypothesize what the reason might be for your observed behaviour.

In most applications of computing machinery we care about the efficiency with which we find the solution. Thus, make sure that you do measurements. In general, make observations that allow you to judge whether your program behaves the way you would expect it to.

Including code If you include code samples in your writeup, make sure they look good. For starters, use a mono-spaced font. In LaTeX, you can use the `verbatim` environment or the `verbatiminput` command. In that section option the source is included automatically, rather than cut and pasted. This is to be preferred, since your writeup will stay current after you edit the source file.

Including whole source files makes for a long and boring writeup. The code samples in this book were generated as follows. In the source files, the relevant snippet was marked as

```
... boring stuff
//snippet samplex
  .. interesting! ..
//snippet end
... more boring stuff
```

The files were then processed with the following command line (actually, included in a makefile, which requires doubling the dollar signs):

```
for f in *.{c,cxx,h} ; do
  cat $x | awk 'BEGIN {f=0}
                /snippet end/ {f=0}
                f==1 {print $0 > file}
                /snippet/ && !/end/ {f=1; file=$2 }
               '
done
```

which gives (in this example) a file `samplex`. Other solutions are of course possible.

Code formatting Included code snippets should be readable. At a minimum you could indent the code correctly in an editor before you include it in a `verbatim` environment. (Screenshots of your terminal window are a decidedly suboptimal solution.) But it's better to use the `listing` package which formats your code, include syntax coloring. For instance,

```
\lstset{language=C++} % or Fortran or so
\begin{lstlisting}
for (int i=0; i<N; i++)
  s += 1;
\end{lstlisting}
```

```
for (int i=0; i<N; i++)
  s += 1;
```

Running your code A single run doesn't prove anything. For a good report, you need to run your code for more than one input dataset (if available) and in more than one processor configuration. When you choose problem sizes, be aware that an average processor can do a billion operations per second: you need to make your problem large enough for the timings to rise above the level of random variations and startup phenomena.

When you run a code in parallel, beware that on clusters the behaviour of a parallel code will always be different between one node and multiple nodes. On a single node the MPI implementation is likely optimized to use the shared memory. This means that results obtained from a single node run will be unrepresentative. In fact, in timing and scaling tests you will often see a drop in (relative) performance going from one node to two. Therefore you need to run your code in a variety of scenarios, using more than one node.

Reporting scaling If you do a scaling analysis, a graph reporting runtimes should not have a linear time axis: a logarithmic graph is much easier to read. A speedup graph can also be informative.

Some algorithms are mathematically equivalent in their sequential and parallel versions. Others, such as iterative processes, can take more operations in parallel than sequentially, for instance because the number of iterations goes up. In this case, report both the speedup of a single iteration, and the total improvement of running the full algorithm in parallel.

Repository organization If you submit your work through a repository, make sure you organize your submissions in subdirectories, and that you give a clear name to all files. Object files and binaries should not be in a repository since they are dependent on hardware and things like compilers.

Chapter 45

Warmup Exercises

We start with some simple exercises.

45.0.1 Hello world

For background, see section 2.3.

First of all we need to make sure that you have a working setup for parallel jobs. The example program `helloworld.c` does the following:

```
// helloworld.c
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&ntids);
MPI_Comm_rank(MPI_COMM_WORLD,&mytid);
printf("Hello, this is processor %d out of %d\n",mytid,ntids);
MPI_Finalize();
```

Compile this program and run it in parallel. Make sure that the processors do *not* all say that they are `processor 0 out of 1`!

45.0.2 Collectives

It is a good idea to be able to collect statistics, so before we do anything interesting, we will look at MPI collectives; section 3.1.

Take a look at `time_max.cxx`. This program sleeps for a random number of seconds:

```
// time_max.cxx
wait = (int) ( 6.*rand() / (double)RAND_MAX );
tstart = MPI_Wtime();
sleep(wait);
tstop = MPI_Wtime();
jitter = tstop-tstart-wait;
```

and measures how long the sleep actually was:

```
if (mytid==0)
  sendbuf = MPI_IN_PLACE;
else sendbuf = (void*)&jitter;
MPI_Reduce(sendbuf,(void*)&jitter,1,MPI_DOUBLE,MPI_MAX,0,comm);
```

In the code, this quantity is called 'jitter', which is a term for random deviations in a system.

Exercise 45.1. Change this program to compute the average jitter by changing the reduction operator.

Exercise 45.2. Now compute the standard deviation

$$\sigma = \sqrt{\frac{\sum_i (x_i - m)^2}{n}}$$

where m is the average value you computed in the previous exercise.

- Solve this exercise twice: once by following the reduce by a broadcast operation and once by using an `Allreduce`.
- Run your code both on a single cluster node and on multiple nodes, and inspect the TAU trace. Some MPI implementations are optimized for shared memory, so the trace on a single node may not look as expected.
- Can you see from the trace how the allreduce is implemented?

Exercise 45.3. Finally, use a gather call to collect all the values on processor zero, and print them out. Is there any process that behaves very differently from the others?

45.0.3 Linear arrays of processors

In this section you are going to write a number of variations on a very simple operation: all processors pass a data item to the processor with the next higher number.

- In the file `linear-serial.c` you will find an implementation using blocking send and receive calls.
- You will change this code to use non-blocking sends and receives; they require an `MPI_Wait` call to finalize them.
- Next, you will use `MPI_Sendrecv` to arrive at a synchronous, but deadlock-free implementation.
- Finally, you will use two different one-sided scenarios.

In the reference code `linear-serial.c`, each process defines two buffers:

```
// linear-serial.c
int my_number = mytid, other_number=-1.;
```

where `other_number` is the location where the data from the left neighbour is going to be stored.

To check the correctness of the program, there is a gather operation on processor zero:

```
int *gather_buffer=NULL;
if (mytid==0) {
  gather_buffer = (int*) malloc(ntids*sizeof(int));
  if (!gather_buffer) MPI_Abort(comm,1);
```

```
}
MPI_Gather(&other_number,1,MPI_INT,
           gather_buffer,1,MPI_INT, 0,comm);
if (mytid==0) {
  int i,error=0;
  for (i=0; i<ntids; i++)
    if (gather_buffer[i]!=i-1) {
      printf("Processor %d was incorrect: %d should be %d\n",
             i,gather_buffer[i],i-1);
      error =1;
    }
  if (!error) printf("Success!\n");
  free(gather_buffer);
}
```

45.0.3.1 Coding with blocking calls

Passing data to a neighbouring processor should be a very parallel operation. However, if we code this naively, with `MPI_Send` and `MPI_Recv`, we get an unexpected serial behaviour, as was explained in section 4.1.2.

```
if (mytid<ntids-1)
  MPI_Ssend( /* data: */ &my_number,1,MPI_INT,
             /* to: */ mytid+1, /* tag: */ 0, comm);
if (mytid>0)
  MPI_Recv( /* data: */ &other_number,1,MPI_INT,
            /* from: */ mytid-1, 0, comm, &status);
```

(Note that this uses an `Ssend`; see section 14.7 for the explanation why.)

Exercise 45.4. Compile and run this code, and generate a TAU trace file. Confirm that the execution is serial. Does replacing the `Ssend` by `Send` change this?

Let's clean up the code a little.

Exercise 45.5. First write this code more elegantly by using `MPI_PROC_NULL`.

45.0.3.2 A better blocking solution

The easiest way to prevent the serialization problem of the previous exercises is to use the `MPI_Sendrecv` call. This routine acknowledges that often a processor will have a receive call whenever there is a send. For border cases where a send or receive is unmatched you can use `MPI_PROC_NULL`.

Exercise 45.6. Rewrite the code using `MPI_Sendrecv`. Confirm with a TAU trace that execution is no longer serial.

Note that the `Sendrecv` call itself is still blocking, but at least the ordering of its constituent send and recv are no longer ordered in time.

45.0.3.3 Non-blocking calls

The other way around the blocking behaviour is to use `Isend` and `Irecv` calls, which do not block. Of course, now you need a guarantee that these send and receive actions are concluded; in this case, use `MPI_Waitall`.

Exercise 45.7. Implement a fully parallel version by using `MPI_Isend` and `MPI_Irecv`.

45.0.3.4 One-sided communication

Another way to have non-blocking behaviour is to use one-sided communication. During a `Put` or `Get` operation, execution will only block while the data is being transferred out of or into the origin process, but it is not blocked by the target. Again, you need a guarantee that the transfer is concluded; here use `MPI_Win_fence`.

Exercise 45.8. Write two versions of the code: one using `MPI_Put` and one with `MPI_Get`. Make TAU traces.

Investigate blocking behaviour through TAU visualizations.

Exercise 45.9. If you transfer a large amount of data, and the target processor is occupied, can you see any effect on the origin? Are the fences synchronized?

Chapter 46

Mandelbrot set

If you've never heard the name *Mandelbrot set*, you probably recognize the picture; figure 46.1 Its formal

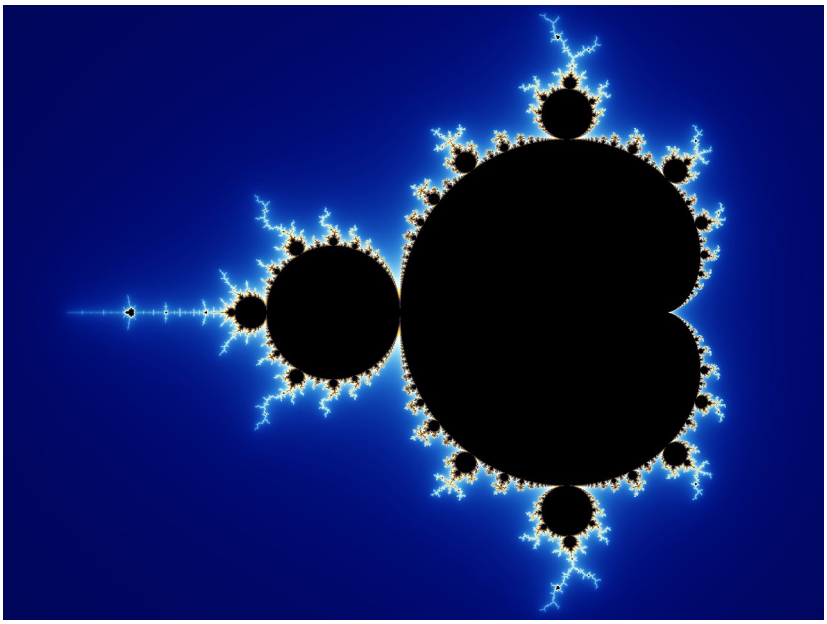


Figure 46.1: The Mandelbrot set

definition is as follows:

> A point c in the complex plane is part of the Mandelbrot set if the series x_n defined by
>
> $$\begin{cases} x_0 = 0 \\ x_{n+1} = x_n^2 + c \end{cases}$$
>
> satisfies
>
> $$\forall_n : |x_n| \leq 2.$$

It is easy to see that only points c in the bounding circle $|c| < 2$ qualify, but apart from that it's hard to say much without a lot more thinking. Or computing; and that's what we're going to do.

In this set of exercises you are going to take an example program `mandel_main.cxx` and extend it to use a variety of MPI programming constructs. This program has been set up as a *manager-worker* model: there is one manager processor (for a change this is the last processor, rather than zero) which gives out work to, and accepts results from, the worker processors. It then takes the results and constructs an image file from them.

46.0.1 Invocation

The `mandel_main` program is called as

```
mpirun -np 123 mandel_main steps 456 iters 789
```

where the `steps` parameter indicates how many steps in x, y direction there are in the image, and `iters` gives the maximum number of iterations in the `belong` test.

If you forget the parameter, you can call the program with

```
mandel_serial -h
```

and it will print out the usage information.

46.0.2 Tools

The driver part of the Mandelbrot program is simple. There is a circle object that can generate coordinates

```
// mandel.h
class circle {
 public :
  circle(int pxls,int bound,int bs);
  void next_coordinate(struct coordinate& xy);
  int is_valid_coordinate(struct coordinate xy);
  void invalid_coordinate(struct coordinate& xy);
```

and a global routine that tests whether a coordinate is in the set, at least up to an iteration bound. It returns zero if the series from the given starting point has not diverged, or the iteration number in which it diverged if it did so.

```
int belongs(struct coordinate xy,int itbound) {
  double x=xy.x, y=xy.y; int it;
  for (it=0; it<itbound; it++) {
    double xx,yy;
    xx = x*x - y*y + xy.x;
    yy = 2*x*y + xy.y;
    x = xx; y = yy;
    if (x*x+y*y>4.) {
      return it;
    }
  }
  return 0;
}
```

In the former case, the point could be in the Mandelbrot set, and we colour it black, in the latter case we give it a colour depending on the iteration number.

```
if (iteration==0)
  memset(colour,0,3*sizeof(float));
else {
  float rfloat = ((float) iteration) / workcircle->infty;
  colour[0] = rfloat;
  colour[1] = MAX((float)0,(float)(1-2*rfloat));
  colour[2] = MAX((float)0,(float)(2*(rfloat-.5)));
}
```

We use a fairly simple code for the worker processes: they execute a loop in which they wait for input, process it, return the result.

```
void queue::wait_for_work(MPI_Comm comm,circle *workcircle) {
  MPI_Status status; int ntids;
  MPI_Comm_size(comm,&ntids);
  int stop = 0;

  while (!stop) {
    struct coordinate xy;
    int res;

    MPI_Recv(&xy,1,coordinate_type,ntids-1,0, comm,&status);
    stop = !workcircle->is_valid_coordinate(xy);
    if (stop) break; //res = 0;
    else {
      res = belongs(xy,workcircle->infty);
    }
    MPI_Send(&res,1,MPI_INT,ntids-1,0, comm);
  }
  return;
}
```

A very simple solution using blocking sends on the manager is given:

```
// mandel_serial.cxx
class serialqueue : public queue {
private :
  int free_processor;
public :
  serialqueue(MPI_Comm queue_comm,circle *workcircle)
    : queue(queue_comm,workcircle) {
    free_processor=0;
  };
  /**
      The `addtask' routine adds a task to the queue. In this
      simple case it immediately sends the task to a worker
      and waits for the result, which is added to the image.

      This routine is only called with valid coordinates;
      the calling environment will stop the process once
      an invalid coordinate is encountered.
```

```
*/
int addtask(struct coordinate xy) {
  MPI_Status status; int contribution, err;

  err = MPI_Send(&xy,1,coordinate_type,
           free_processor,0,comm); CHK(err);
  err = MPI_Recv(&contribution,1,MPI_INT,
           free_processor,0,comm, &status); CHK(err);

  coordinate_to_image(xy,contribution);
  total_tasks++;
  free_processor = (free_processor+1)%(ntids-1);

  return 0;
};
```

Exercise 46.1. Explain why this solution is very inefficient. Make a trace of its execution that bears this out.

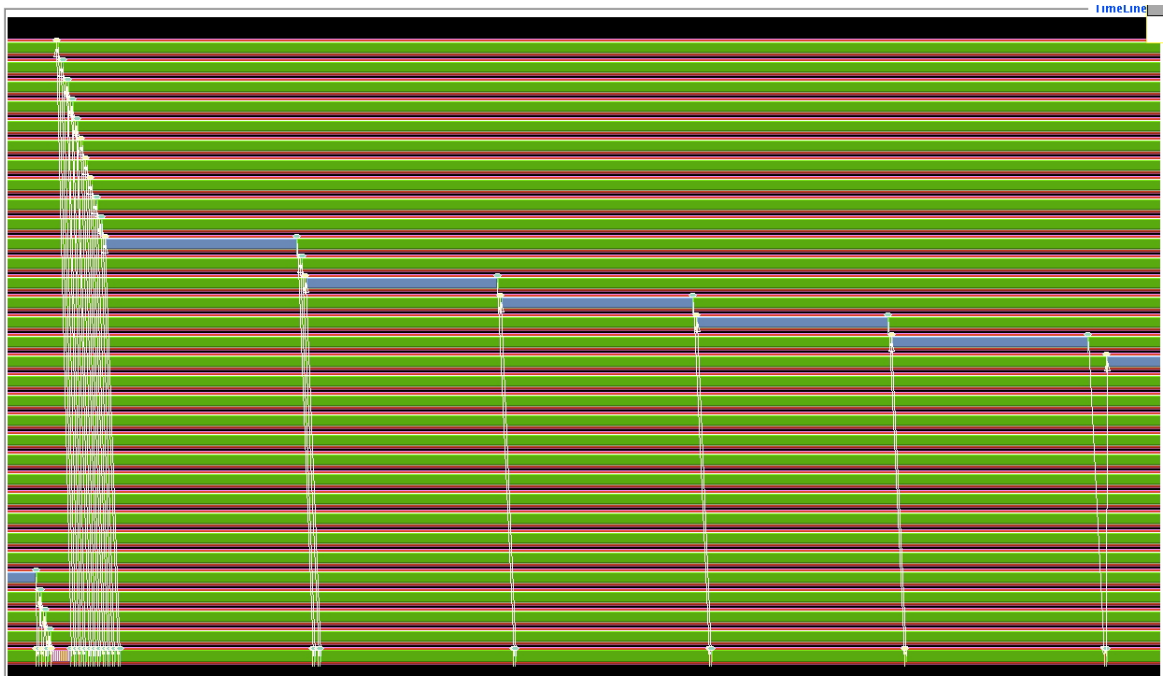


Figure 46.2: Trace of a serial Mandelbrot calculation

46.0.3 Bulk task scheduling

The previous section showed a very inefficient solution, but that was mostly intended to set up the code base. If all tasks take about the same amount of time, you can give each process a task, and then wait on them all to finish. A first way to do this is with non-blocking sends.

Exercise 46.2. Code a solution where you give a task to all worker processes using non-blocking sends and receives, and then wait for these tasks with `MPI_Waitall` to finish before you give a new round of data to all workers. Make a trace of the execution of this and report on the total time.
You can do this by writing a new class that inherits from `queue`, and that provides its own `addtask` method:

```
// mandel_bulk.cxx
class bulkqueue : public queue {
public :
  bulkqueue(MPI_Comm queue_comm,circle *workcircle)
    : queue(queue_comm,workcircle) {
```

You will also have to override the `complete` method: when the circle object indicates that all coordinates have been generated, not all workers will be busy, so you need to supply the proper `MPI_Waitall` call.

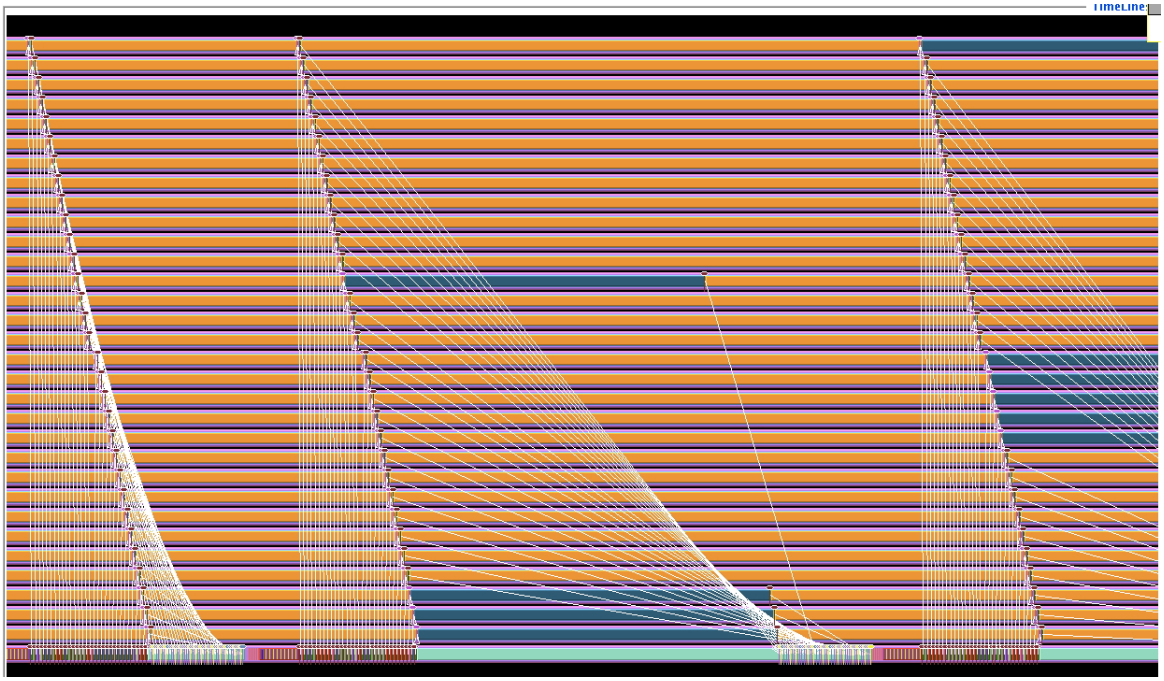


Figure 46.3: Trace of a bulk Mandelbrot calculation

46.0.4 Collective task scheduling

Another implementation of the bulk scheduling of the previous section would be through using collectives.

Exercise 46.3. Code a solution which uses scatter to distribute data to the worker tasks, and gather to collect the results. Is this solution more or less efficient than the previous?

46.0.5 Asynchronous task scheduling

At the start of section 46.0.3 we said that bulk scheduling mostly makes sense if all tasks take similar time to complete. In the Mandelbrot case this is clearly not the case.

Exercise 46.4. Code a fully dynamic solution that uses `MPI_Probe` or `MPI_Waitany`. Make an execution trace and report on the total running time.

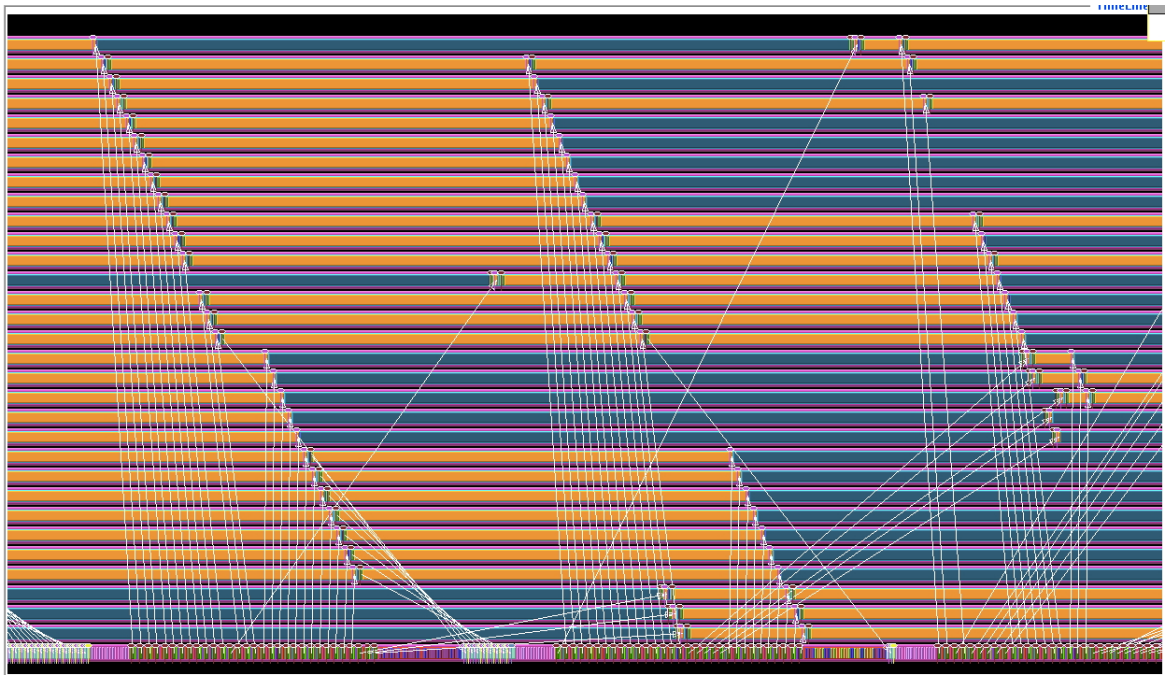


Figure 46.4: Trace of an asynchronous Mandelbrot calculation

46.0.6 One-sided solution

Let us reason about whether it is possible (or advisable) to code a one-sided solution to computing the Mandelbrot set. With active target synchronization you could have an exposure window on the host to which the worker tasks would write. To prevent conflicts you would allocate an array and have each worker write to a separate location in it. The problem here is that the workers may not be sufficiently synchronized because of the differing time for computation.

Consider then passive target synchronization. Now the worker tasks could write to the window on the manager whenever they have something to report; by locking the window they prevent other tasks from interfering. After a worker writes a result, it can get new data from an array of all coordinates on the manager.

It is hard to get results into the image as they become available. For this, the manager would continuously have to scan the results array. Therefore, constructing the image is easiest done when all tasks are concluded.

Chapter 47

Data parallel grids

In this section we will gradually build a semi-realistic example program. To get you started some pieces have already been written: as a starting point look at `code/mpi/c/grid.cxx`.

47.0.1 Description of the problem

With this example you will investigate several strategies for implementing a simple iterative method. Let's say you have a two-dimensional grid of datapoints $G = \{g_{ij} : 0 \leq i < n_i, 0 \leq j < n_j\}$ and you want to compute G' where

$$g'_{ij} = 1/4 \cdot (g_{i+1,j} + g_{i-1,j} + g_{i,j+1} + g_{i,j-1}). \tag{47.1}$$

This is easy enough to implement sequentially, but in parallel this requires some care.

Let's divide the grid G and divide it over a two-dimension grid of $p_i \times p_j$ processors. (Other strategies exist, but this one scales best; see section HPSC-6.5.) Formally, we define two sequences of points

$$0 = i_0 < \cdots < i_{p_i} < i_{p_i+1} = n_i, \quad 0 < j_0 < \cdots < j_{p_j} < i_{p_j+1} = n_j$$

and we say that processor (p, q) computes g_{ij} for

$$i_p \leq i < i_{p+1}, \quad j_q \leq j < j_{q+1}.$$

From formula (47.1) you see that the processor then needs one row of points on each side surrounding its part of the grid. A picture makes this clear; see figure 47.1. These elements surrounding the processor's own part are called the *halo* or *ghost region* of that processor.

The problem is now that the elements in the halo are stored on a different processor, so communication is needed to gather them. In the upcoming exercises you will have to use different strategies for doing so.

47.0.2 Code basics

The program needs to read the values of the grid size and the processor grid size from the commandline, as well as the number of iterations. This routine does some error checking: if the number of processors does not add up to the size of `MPI_COMM_WORLD`, a nonzero error code is returned.

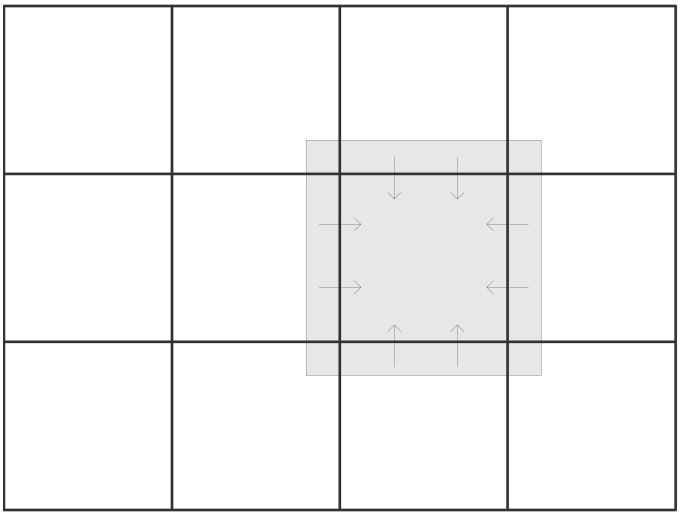


Figure 47.1: A grid divided over processors, with the 'ghost' region indicated

```
ierr = parameters_from_commandline
  (argc,argv,comm,&ni,&nj,&pi,&pj,&nit);
if (ierr) return MPI_Abort(comm,1);
```

From the processor parameters we make a processor grid object:

```
processor_grid *pgrid = new processor_grid(comm,pi,pj);
```

and from the numerical parameters we make a number grid:

```
number_grid *grid = new number_grid(pgrid,ni,nj);
```

Number grids have a number of methods defined. To set the value of all the elements belonging to a processor to that processor's number:

```
grid->set_test_values();
```

To set random values:

```
grid->set_random_values();
```

If you want to visualize the whole grid, the following call gathers all values on processor zero and prints them:

```
grid->gather_and_print();
```

Next we need to look at some data structure details.

The definition of the `number_grid` object starts as follows:

```
class number_grid {
public:
  processor_grid *pgrid;
  double *values,*shadow;
```

where `values` contains the elements owned by the processor, and `shadow` is intended to contain the values plus the ghost region. So how does `shadow` receive those values? Well, the call looks like

```
grid->build_shadow();
```

and you will need to supply the implementation of that. Once you've done so, there is a routine that prints out the shadow array of each processor

```
grid->print_shadow();
```

This routine does the sequenced printing that you implemented in exercise **??**.

In the file `code/mpi/c/grid_impl.cxx` you can see several uses of the macro `INDEX`. This translates from a two-dimensional coordinate system to one-dimensional. Its main use is letting you use (i,j) coordinates for indexing the processor grid and the number grid: for processors you need the translation to the linear rank, and for the grid you need the translation to the linear array that holds the values.

A good example of the use of `INDEX` is in the `number_grid::relax` routine: this takes points from the `shadow` array and averages them into a point of the `values` array. (To understand the reason for this particular averaging, see HPSC-4.2.3 and HPSC-5.5.3.) Note how the `INDEX` macro is used to index in a `ilength` × `jlength` target array `values`, while reading from a (`ilength` + 2) × (`jlength` + 2) source array `shadow`.

```
for (i=0; i<ilength; i++) {
  for (j=0; j<jlength; j++) {
    int c=0;
    double new_value=0.;
    for (c=0; c<5; c++) {
int ioff=i+1+ioffsets[c],joff=j+1+joffsets[c];
new_value += coefficients[c] *
  shadow[ INDEX(ioff,joff,ilength+2,jlength+2) ];
    }
    values[ INDEX(i,j,ilength,jlength) ] = new_value/8.;
  }
}
```

Chapter 48

N-body problems

N-body problems describe the motion of particles under the influence of forces such as gravity. There are many approaches to this problem, some exact, some approximate. Here we will explore a number of them.

For background reading see HPSC-10.

48.0.1 Solution methods

It is not in the scope of this course to give a systematic treatment of all methods for solving the N-body problem, whether exactly or approximately, so we will just consider a representative selection.

1. Full N^2 methods. These compute all interactions, which is the most accurate strategy, but also the most computationally demanding.
2. Cutoff-based methods. These use the basic idea of the N^2 interactions, but reduce the complexity by imposing a cutoff on the interaction distance.
3. Tree-based methods. These apply a coarsening scheme to distant interactions to lower the computational complexity.

48.0.2 Shared memory approaches

48.0.3 Distributed memory approaches

PART VIII

DIDACTICS

Chapter 49

Teaching guide

Based on two lectures per week, here is an outline of how MPI can be taught in a college course. Links to to the relevant exercises.

Topic	Exercises	Week
Block 1: SPMD and collectives		
Intro: cluster structure Functional parallelism	`hello`: 2.1, 2.2 `commrank`: 2.4, 2.5, `prime`: 2.6	week 1
Allreduce, broadcast	3.1, `randommax`: 3.2 `jordan`: 3.8	week 2
Scan, Gather	3.13, `scangather`: 3.11, 3.15	week 3
Block 2: Two-sided point-to-point		
Send and receive	`pingpong`: 4.1, `rightsend`: 4.4	
Sendrecv Nonblocking	`bucketblock`: 4.6, `sendrecv`: 4.8, 4.9 `isendirecv`: 4.13, `isendirecvarray`: 4.14	week 4
	`bucketpipenonblock`: 4.11	week 5
Block 3: Derived datatypes		
Contiguous, Vector, Indexed	`stridesend`: 6.4, `cubegather`: 6.6	
Extent and resizing		
Block 4: Communicators		
Duplication, split Groups		week 6
Block 5: I/O		
Block 6: Neighborhood collectives		

49.1 Sources used in this chapter

49.1.1 Listing of code header

Chapter 50

Teaching from mental modesl

Distributed memory programming, typically through the MPI library, is the *de facto* standard for programming large scale parallelism, with up to millions of individual processes. Its dominant paradigm of Single Program Multiple Data (SPMD) programming is different from threaded and multicore parallelism, to an extent that students have a hard time switching models. In contrast to threaded programming, which allows for a view of the execution with central control and a central repository of data, SPMD programming has a symmetric model where all processes are active all the time, with none privileged, and where data is distributed.

This model is counterintuitive to the novice parallel programmer, so care needs to be taken how to instill the proper 'mental model'. Adoption of an incorrect mental model leads to broken or inefficient code.

We identify problems with the currently common way of teaching MPI, and propose a structuring of MPI courses that is geared to explicit reinforcing the symmetric model. Additionally, we advocate starting from realistic scenarios, rather than writing artificial code just to exercise newly-learned routines.

50.0.1 Introduction

The MPI library [23, 20] is the *de facto* tool for large scale parallelism as it is used in engineering sciences. In this paper we want to discuss the manner it is usually taught, and propose a rethinking.

We argue that the topics are typically taught in a sequence that is essentially dictated by level of complexity in the implementation, rather than by conceptual considerations. Our argument will be for a sequencing of topics, and use of examples, that is motivated by typical applications of the MPI library, and that explicitly targets the required mental model of the parallelism model underlying MPI.

We have written an open-source textbook [8] with exercise sets that follows the proposed sequencing of topics and the motivating applications.

50.0.1.1 Short background on MPI

The MPI library dates back to the early days of cluster computing, the first half of the 1990s. It was an academic/industrial collaboration to unify earlier, often vendor-specific, message passing libraries. MPI is typically used to code large-scale Finite Element Method (FEM) and other physical simulation applications,

which share characteristics of a relatively static distribution of large amounts of data – hence the use of clusters to increase size of the target problem – and the need for very efficient exchange of small amounts of data.

The main motivation for MPI is the fact that it can be scaled to more or less arbitrary scales, currently up to millions of cores [1]. Contrast this with threaded programming, which is limited more or less by the core count on a single node, currently about 70.

Considering this background, the target audience for MPI teaching consists of upper level undergraduate students, graduate students, and even post-doctoral researchers who are engaging for the first time in large scale simulations. The typical participant in an MPI course is likely to understand more than the basics of linear algebra and some amount of numerics of Partial Diffential Equation (PDE).

50.0.1.2 Distributed memory parallelism

Corresponding to its origins in cluster computing, MPI targets distributed memory parallelism[1]. Here, network-connected cluster nodes run codes that share no data, but synchronize through explicit messages over the network. Its main model for parallelism is described as Single Program Multiple Data (SPMD): multiple instances of a single program run on the processing elements, each operating on their own data. The MPI library then implements the communication calls that allow processes to combine and exchange data.

While MPI programs can solve many or all of the same problems that can be solved with a multicore approach, the programming approach is different, and requires an adjustment in the programmer's 'mental model' [6, 27] of the parallel execution. This paper addresses the question of how to teach MPI to best effect this shift in mindset.

Outline of this paper. We use section 50.0.2 to address explicitly the mental models that govern parallel thinking and parallel programming, pointing out why MPI is different, and difficult initially. In section 50.0.3 we consider the way MPI is usually taught, while in section 50.0.4 we offer an alternative that is less likely to lead to an incorrect mental model.

Some details of our proposed manner of teaching are explored in sections 50.0.5, 50.0.6, 50.0.7. We conclude with discussion in sections 50.0.8 and 50.0.9.

50.0.2 Implied mental models

Denning [7] argued how computational thinking consists in finding an abstract machine (a 'computational model') that solves the problem in a simple algorithmic way. In our case of teaching parallel programming, the complication to this story is that the problem to be solved is already a computational system. That doesn't lessen the need to formulate an abstract model, since the full explanation of MPI's workings are unmanageable for a beginning programmer, and often not needed for practical purposes.

In this section we consider in more detail the mental models that students may implicitly be working under, and the problems with them; targeting the right mental model will then be the subject of later

1. Recent additions to the MPI standard target shared memory too.

sections. The two (interrelated) aspects of a correct mental model for distributed memory programming are control and synchronization. We here discuss how these can be misunderstood by students.

50.0.2.1 The traditional view of parallelism

The problem with mastering the MPI library is that beginning programmers take a while to overcome a certain mental model for parallelism. In this model, which we can call 'sequential semantics' (or more whimsically the 'big index finger' model), there is only a single strand of execution[2], which we may think of as a big index finger going down the source code.

This mental model corresponds closely to the way algorithms are described in the mathematical literature of parallelism, and it is actually correct to an extent in the context of threaded libraries such as OpenMP, where there is indeed initially a single thread of execution, which in some places spawns a team of threads to execute certain sections of code in parallel. However, in MPI this model is factually incorrect, since there are always multiple processes active, with none essentially priviliged over others, and no shared or central data store.

50.0.2.2 The misconceptions of centralized control

The sequential semantics mental model that, as described above, underlies much of the theoretical discussion of parallelism, invites the student to adopt certain programming techniques, such as the master-worker approach to parallel programming. While this is often the right approach with thread-based coding, where we indeed have a master thread and spawned threads, it is usually incorrect for MPI. The strands of execution in an MPI run are all long-living processes (as opposed to dynamically spawned threads), and are *symmetric* in their capabilities and execution.

Lack of recognition of this process symmetry also induces students to solve problems by having a form of 'central data store' on one process, rather than adopting a symmetric, distributed, storage model. For instance, we have seen a student solve a data transposition problem by collecting all data on process 0, and subsequently distributing it again in transposed form. While this may be reasonable[3] in shared memory with OpenMP, with MPI it is unrealistic in that no process is likely to have enough storage for the full problem. Also, this introduces a sequential bottleneck in the execution.

In conclusion, we posit that beginning MPI programmers may suffer from a mental model that makes them insufficiently realize the symmetry of MPI processes, and thereby arrive at inefficient and nonscalable solutions.

50.0.2.3 The reality of distributed control

An MPI program run consists of multiple independent threads of control. One problem in recognizing this is that there is only a single source code, so there is an inclination to envision the program execution as a single thread of control: the above-mentioned 'index finger' going down the statements of the source.

2. We carefully avoid the word 'thread' which carries many connotations in the context of parallel programming.
3. To first order; second order effects such as affinity complicate this story.

A second factor contributing to this view is that a parallel code incorporates statements with values (`int x = 1.5;`) that are replicated over all processes. It is easy to view these as centrally executed.

Interestingly, work by Ben-David Kolikant [2] shows that students with no prior knowledge of concurrency, when invited to consider parallel activities, will still think in terms of centralized solutions. This shows that distributed control, such as it appears in MPI, is counterintuitive and needs explicit enforcement in its mental model. In particular, we explicitly target process symmetry and process differentiation.

The centralized model can still be maintained in MPI to an extent, since the scalar operations that would be executed by a single thread become replicated operations in the MPI processes. The distinction between sequential execution and replicated execution escapes many students at first, and in fact, since nothing it gained by explaining this, we do not do so.

50.0.2.4 The misconception of synchronization

Even with multiple threads of control and distributed data, there is still a temptation to see execution as 'bulk synchronous processing' (BSP [26]). Here, the execution proceeds by supersteps, implying that processes are largely synchronized. (The BSP model has several components more, which are usually ignored, notably one-sided communication and processor oversubscription.)

Supersteps as a computational model allow for small differences in control flow, for instance conditional inside a big parallelizable loop, but otherwise imply a form of centralized control (as above) on the level of major algorithm steps. However, codes using the pipeline model of parallelism, such idioms as

```
MPI_Recv( /* from: */ my_process-1)
// do some major work
MPI_Send( /* to  : */ my_process+1)
```

fall completely outside either the sequential semantics or BSP model and require an understanding of one process' control being dependent on another's. Gaining an mental model for this sort of unsynchronized execution is nontrivial to achieve. We target this explicitly in section 50.0.5.1.

50.0.3 Teaching MPI, the usual way

The MPI library is typically taught as follows. After an introduction about parallelism, covering concepts such as speedup and shared versus distributed memory parallelism, students learn about the initialization and finalization routines, and the `MPI_Comm_size` and `MPI_Comm_rank` calls for querying the number of processes and the rank of the current process.

After that, the typical sequence is

1. two-sided communication, with first blocking and later nonblocking variants;
2. collectives; and
3. any number of advanced topics such as derived data types, one-sided communication, subcommunicators, MPI I/O et cetera, in no particular order.

This sequence is defensible from a point of the underlying implementation: the two-sided communication calls are a close map to hardware behaviour, and collectives are both conceptually equivalent to, and can be implemented as, a sequence of point-to-point communication calls. However, this is not a sufficient justification for teaching this sequence of topics.

50.0.3.1 Criticism

We offer three points of criticism against this traditional approach to teaching MPI.

First of all, there is no real reason for teaching collectives after two-sided routines. They are not harder, nor require the latter as prerequisite. In fact, their interface is simpler for a beginner, requiring one line for a collective, as opposed to at least two for a send/receive pair, probably surrounded by conditionals testing the process rank. More importantly, they reinforce the symmetric process view, certainly in the case of the `MPI_All...` routines.

Our second point of criticism is regarding the blocking and nonblocking two-sided communication routines. The blocking routines are typically taught first, with a discussion of how blocking behaviour can lead to load unbalance and therefore inefficiency. The nonblocking routines are then motivated from a point of latency hiding and solving the problems inherent in blocking. In our view such performance considerations should be secondary. Nonblocking routines should instead be taught as the natural solution to a conceptual problem, as explained below.

Thirdly, starting with point-to-point routines stems from a **CSP!** (**CSP!**)[12] view of a program: each process stands on its own, and any global behaviour is an emergent property of the run. This may make sense for the teacher who know how concepts are realized 'under the hood', but it does not lead to additional insight with the students. We believe that a more fruitful approach to MPI programming starts from the global behaviour, and then derives the MPI process in a top-down manner.

50.0.3.2 Teaching MPI and OpenMP

In scientific computing, another commonly used parallel programming system is OpenMP [22]. OpenMP and MPI are often taught together, with OpenMP taught earlier because it is supposedly easier, or because its parallelism would be easier to grasp. Regardless our opinion on the first estimate, we argue that OpenMP should be taught *after* MPI because of its 'central control' parallelism model. If students come to associate parallelism with a model that has a 'master thread' and 'parallel regions' they will find it much harder to make idiomatic use of the symmetric model of MPI.

50.0.4 Teaching MPI, our proposal

As alternative to the above sequence of introducing MPI concepts, we propose a sequence that focuses on practical scenarios, and that actively reinforces the mental model of SPMD execution.

Such reinforcement is often an immediate consequence of our strategy of illustrating MPI constructs in the context of an application: most MPI applications (as we shall briefly discuss next) operate on large 'distributed objects'. This immediately leads to a mental model of the workings of each process being the 'projection' onto that process of the global calculation. The opposing view, where the overall computation is emergent from the individual processes, is the **CSP!** model mentioned above.

50.0.4.1 Motivation from applications

The typical application for MPI comes from Computational Science and Engineering, such as N-body problems, aerodynamics, shallow water equations, Lattice Boltzman methods, weather modeling with

Fast Fourier Transform. Of these, the PDE based applications can readily be explained to need a number of MPI mechanisms.

Nonnumeric applications exist:

- Graph algorithms such as shortest-path or PageRank are usually easily explained. However, the distributed memory algorithms need to be approached fundamentally different from the more naive shared memory variants. Thus they require a good amount of background knowledge. Additionally, they do not feature the regular communications that one-dimensional PDE applications have. Scalability arguments make this story even more complicated. Thus, these algorithms are in fact a logical next topic *after* discussion of parallel PDE algorithm.
- N-body problems, in their naive implementation, are easy to explain to any student who knows inverse-square laws such as gravity. It is a good illustration of some collectives, but nothing beyond that.
- Sorting. Sorting algorithms based on a sorting network (this includes bubblesort, but not quicksort) can be used as illustration. In fact, we use odd-even transposition sort as a 'midterm' exam assignment, which can be solved with `MPI_Sendrecv`. Algorithms such as bitonic sort can be used to illustrate some advanced concepts, but quicksort, which is relatively easy to explain as a serial algorithm, or even in shared memory, is quite hard in MPI.
- Point-to-point operations can also be illustrated by graphics operations such as a 'blur', since these correspond to a 'stencil' applied to a cluster of pixels. Unfortunately, this example suffers from the fact that neither collectives, nor irregular communications have a use in this application. Also, using graphics to illustrate simple MPI point-to-point communication is unrealistic in two ways: first, to start out simple we have to posit a one-dimensional pixel array; secondly, graphics is hardly ever of the scale that necessitates distributed memory, so this example is far from 'real world'. (Ray tracing is naturally done distributed, but that has a completely different computational structure.)

Based on this discussion of possible applications, and in view of the likely background of course attendants, we consider Finite Difference solution of PDEs as a prototypical application that exercises both the simplest and more sophisticated mechanisms. During a typical MPI training, even a one-day short course, we insert a lecture on sparse matrices and their computational structure to motivate the need for various MPI constructs.

50.0.4.2 Process symmetry

Paradoxically, the first way to get students to appreciate the notion of process symmetry in MPI is to run a non-MPI program. Thus, students are asked to write a 'hello world' program, and execute this with `mpiexec`, as if it were an MPI program. Every process executes the print statement identically, bearing out the total symmetry between the processes.

Next, students are asked to insert the initialize and finalize statements, with three different 'hello world' statements before, between, and after them. This will prevent any notion of the code between initialization and finalization being considered as an OpenMP style 'parallel region'.

A simple test to show that while processes are symmetric they are not identical is offered by the exercise of using the `MPI_Get_processor_name` function, which will have different output for some or all of the

processes, depending on how the hostfile was arranged.

50.0.4.3 Functional parallelism

The `MPI_Comm_rank` function is introduced as a way of distinguishing between the MPI processes. Students are asked to write a program where only one process prints the output of `MPI_Comm_size`.

Having different execution without necessarily different data is a case of 'functional parallelism'. At this point there are few examples that we can assign. For instance, in order to code the evaluation of an integral by Riemann sums ($\pi/4 = \int_0^1 \sqrt{1-x^2}dx$ is a popular one) would need a final sum collective, which has not been taught at this point.

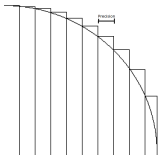


Figure 50.1: Calculation of $\pi/4$ by Riemann sums

A possible example would be primality testing, where each process tries to find a factor of some large integer N by traversing a subrange of $[2, \sqrt{N}]$, and printing a message if a factor is found. Boolean satisfiability problems form another example, where again a search space is partitioned without involving any data space; a process finding a satisfying input can simply print this fact. However, this example requires background that students typically don't have.

50.0.4.4 Introducing collectives

At this point we can introduce collectives, for instance to find the maximum of a random value that is computed locally on each process. This requires teaching the code for random number generation and, importantly, setting a process-dependent random number seed. Generating random 2D or 3D coordinates and finding the center of mass is an examples that requires a send and receive buffer of length greater than 1, and illustrates that reductions are then done pointwise.

These examples evince both process symmetry and a first form of local data. However, a thorough treatment of distributed parallel data will come in the discussion of point-to-point routines.

It is an interesting question whether we should dispense with 'rooted' collectives such as `MPI_Reduce` at first, and start with `MPI_Allreduce`[4] The latter is more symmetric in nature, and has a buffer treatment that is easier to explain; it certainly reinforces the symmetric mindset. There is also essentially no difference in efficiency.

Certainly, in most applications the 'allreduce' is the more common mechanism, for instance where the algorithm requires computations such as

$$\bar{y} \leftarrow \bar{x}/\|\bar{x}\|$$

where x, y are distributed vectors. The quantity $\|\bar{x}\|$ is then needed on all processes, making the `Allreduce` the natural choice. The rooted reduction is typically only used for final results. Therefore we advocate introducing both rooted and nonrooted collectives, but letting the students initially do exercises with the nonrooted variants.

4. The `MPI_Reduce` call performs a reduction on data found on all processes, leaving the result on a 'root' process. With `MPI_Allreduce` the result is left on *all* processes.

This has the added advantage of not bothering the students initially with the asymmetric treatment of the receive buffer between the root and all other processes.

50.0.4.5 Distributed data

As motivation for the following discussion of point-to-point routines, we now introduce the notion of distributed data. In its simplest form, a parallel program operates on a linear array the dimensions of which exceed the memory of any single process.

```
int n;
double data[n];
```

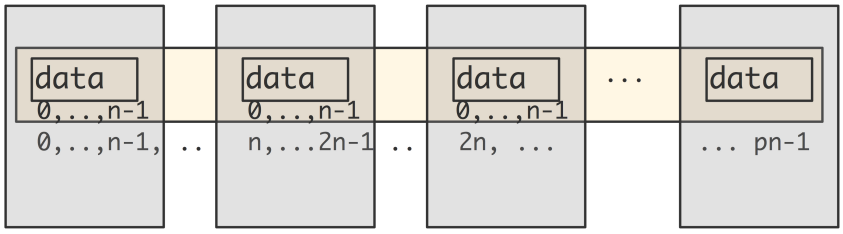


Figure 50.2: A distributed array versus multiple local arrays

The lecturer stresses that the global structure of the distributed array is only 'in the programmer's mind': each MPI process sees an array with indexing starting at zero. The following snippet of code is given for the students to use in subsequent exercises:

```
int myfirst = .....;
for (int ilocal=0; ilocal<nlocal; ilocal++) {
   int iglobal = myfirst+ilocal;
   array[ilocal] = f(iglobal);
}
```

At this point, the students can code a second variant of the primality testing exercise above, but with an array allocated to store the integer range. Since collectives are now known, it becomes possible to have a single summary statement from one process, rather than a partial result statement from each.

The inner product of two distributed vectors is a second illustration of working with distributed data. In this case, the reduction for collecting the global result is slightly more useful than the collective in the previous examples. For this example no translation from local to global numbering is needed.

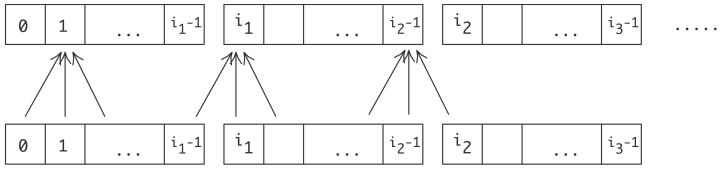
50.0.4.6 Point-to-point motivated from operations on distributed data

We now state the importance of local combining operations such as

$$y_i = \left(x_{i-1} + x_i + x_{i+1}\right)/3 : \; i = 1, \ldots, N-1$$

applied to an array. Students who know about PDEs will recognize that with different coefficients this is the heat equation; for others a graphics 'blur' operation can be used as illustration, if they accept that a one-dimensional pixel array is a stand-in for a true graphic.

Under the 'owner computes' regime, where the process that stores location y_i performs the full calculation of that quantity, we see the need for communication in order to compute the first and last element of the local part of y:



We then state that this data transfer is realized in MPI by two-sided send/receive pairs.

50.0.4.7 Detour: deadlock and serialization

The concept of 'blocking' is now introduced, and we discuss how this can lead to deadlock. A more subtle behaviour is 'unexpected serialization': processes interacting to give serial behaviour on a code that conceptually should be parallel. (The classroom protocol is discussed in detail in section 50.0.5.1.) For completeness, the 'eager limit' can be discussed.

This introduces students to an interesting phenomenon in the concept of parallel correctness: a program may give the right result, but not with the proper parallel efficiency. Asking a class to come up with a solution that does not have a running time proportional the number of processes, will usually lead to at least one student suggesting splitting processes in odd and even subsets. The limits to this approach, code complexity and the reliance on regular process connectivity, are explained to the students as a preliminary to the motivation for nonblocking sends; section 50.0.4.10.

50.0.4.8 Detour: ping-pong

At this point we briefly abandon the process symmetry, and consider the ping-pong operation between two processes A and B[5]. We ask students to consider what the ping-pong code looks like for A and, for B. Since we are working with SPMD code, we arrive at a program where the A code and B code are two branches of a conditional.

We ask the students to implement this, and do timing with `MPI_Wtime`. The implementation of the ping-pong is itself a good exercises in SPMD thinking; finding the right sender/receiver values usually takes the students a nontrivial amount of time. Many of them will initially write a code that deadlocks.

The concepts of latency and bandwidth can be introduced, as the students test the ping-pong code on messages of increasing size. The concept of halfbandwidth can be introduced by letting half of all processes execute a ping-pong with a partner process in the other half.

50.0.4.9 Back to data exchange

The foregoing detours into the behaviour of two-sided send and receive calls were necessary, but they introduced asymmetric behaviour in the processes. We return to the averaging operation given above,

5. In this operations, process A sends to B, and B subsequenty sends to A. Thus the time for a message is half the time of a ping-pong. It is not possible to measure a single message directly, since processes can not be synchronized that finely.

and with it to a code that treats all processes symmetrically. In particular, we argue that, except for the first and last, each process exchanges information with its left and right neighbour.

This could be implemented with blocking sends and receive calls, but students recognize how this could be somewhere between tedious and error-prone. Instead, to prevent deadlock and serialization as described above, we now offer the `MPI_Sendrecv` routine[6]. Students are asked to implement the classroom exercise above with the sendrecv routine. Ideally, they use timing or tracing to gather evidence that no serialization is happening.

As a nontrivial example (in fact, this takes enough programming that one might assign it as an exam question, rather than an exercise during a workshop) students can now implement an odd-even transposition sort algorithm using `MPI_Sendrecv` as the main tool. For simplicity they can use a single array element per process. (If each process has a subarray one has to make sure their solution has the right parallel complexity. It is easy to make errors here and implement a correct algorithm that, however, performs too slowly.)

Note that students have at this point not done any serious exercises with the blocking communication calls, other than the ping-pong. No such exercises will in fact be done.

50.0.4.10 Nonblocking sends

Nonblocking sends are now introduced as the solution to a specific problem: the above schemes required paired-up processes, or careful orchestration of send and receive sequences. In the case of irregular communications this is no longer possible or feasible. Life would be easy if we could declare 'this data needs to be sent' or 'these messages are expected', and then wait for these messages collectively. Given this motivation, it is immediately clear that multiple send or receive buffers are needed, and that requests need to be collected.

Implementing the three-point averaging with nonblocking calls is at this point an excellent exercise.

Note that we have here motivated the nonblocking routines to solve a symmetric problem. Doing this should teach the students the essential point that each nonblocking call needs its own buffer and generates its own request. Viewing nonblocking routines as a performance alternative to blocking routines is likely to lead to students re-using buffers or failing to save the request objects. Doing so is a correctness bug that is very hard to find, and at large scale it induces a memory leak since many requests objects are lost.

50.0.4.11 Taking it from here

At this point various advanced topics can be discussed. For instance, Cartesian topologies can be introduced, extending the linear averaging operation to a higher dimensional one. Subcommunicators can be introduced to apply collectives to rows and columns of a matrix. The recursive matrix transposition algorithm is also an excellent application of subcommunicators.

However, didactically these topics do not require the careful attention that the introduction of the basic concepts needs, so we will not go into further detail here.

---

6. The `MPI_Sendrecv` call combines a send a receive operation, specifying for each process both a sending and receiving communication. The execution guarantees that no deadlock or serialization will occur.

50.0.5 'Parallel computer games'

Part of the problem in developing an accurate mental model of parallel computation is that there is no easy way to visualize the execution. While sequential execution can be imagined with the 'big index finger' model (see section 50.0.2.1), the possibily unsynchronized execution of an MPI program makes this a gross simplification. Running a program in a parallel graphical environment (such as the DDT debugger or the Eclipse PTP IDE) would solve this, but they introduce much learning overhead. Ironically, the low tech solution of

```
mpirun -np 4 xterm -e gdb program
```

is fairly insightful, but having to learn gdb is again a big hurdle.

We have arrived at the somewhat unusual solution of having students act out the program in front of the class. With each student acting out the program, any interaction is clearly visible to an extent that is hard to achieve any other way.

50.0.5.1 Sequentialization

Our prime example is to illustrate the blocking behaviour of `MPI_Send` and `MPI_Recv`[7]. Deadlock is easy enough to understand as a consequence of blocking – in the simplest case of deadlock to processes are both blocked expecting a receive from the other – but there are more subtle effects that will come as a surprise to students. (This was alluded to in section 50.0.4.7.)

Consider the following basic program:

- Pass a data item to the next higher numbered process.

Note that this is conceptually a fully parallel program, so it should execute in time $O(1)$ in terms of the number of processes.

In terms of send and receive calls, the program becomes

- Send data to the next higher process;
- Receive data from the next lower process.

The final detail concerns the boundary conditions: the first process has nothing to receive and the last one has nothing to send. This makes the final version of the program:

- If you are not the last process, send data to the next higher process; then
- If you are not the first process, receive data from the next lower process.

To have students act this out, we tell them to hold a pen in their right hand, and put the left hand in a pocket or behind their back. Thus, they have only one 'communication channel'. The 'send data' instruction becomes 'turn to your right and give your pen', and 'receive data' becomes 'turn to your left and receive a pen'.

Executing this program, the students first all turn to the right, and they see that giving data to a neighbour is not possible because no one is executing the receive instruction. The last process is not sending, so moves on to the receive instruction, after which the penultimate process can receive, et cetera.

7. Blocking is defined as the process executing a send or receive call halting until the corresponding operation is executing.

This exercise makes the students see, better than any explanation or diagram, how a parallel program can compute the right result, but with unexpectedly low performance because of the interaction of the processes. (In fact, we have had explicit feedback that this game was the biggest lightbulb moment of the class.)

50.0.5.2 Ping-pong

While in general we emphasize the symmetry of MPI processes, during the discussion of send and receive calls we act out the ping-pong operation (one process sending data to another, followed by the other sending data back), precisely to demonstrate how asymmetric actions are handled. For this, two students throw a pen back and forth between them, calling out 'send' and 'receive' when they do so.

The teacher then asks each student what program they executed, which is 'send-receive' for the one, and 'receive-send' for the other student. Incorporating this in the SPMD model then leads to a code with conditionals to determine the right action for the right process.

50.0.5.3 Collectives and other games

Other operations can be acted out by the class. For instance, the teacher can ask one student to add the grades of all students, as a proxy for a reduction operation. The class quickly sees that this will take a long time, and strategies such as taking by-row sums in the classroom quickly suggest themselves.

We have at one point tried to have a pair of student act out a 'race condition' in shared memory programming, but modeling this quickly became too complicated to be convincing.

50.0.5.4 Remaining questions

Even with our current approach, however, we still see students writing idioms that are contrary to the symmetric model. For instance, they will write

```
for (p=0; p<nprocs; p++)
  if (p==myrank)
    // do some function of p
```

This code computes the correct result, and with the correct performance behaviour, but it still shows a conceptual misunderstanding. As one of the 'parallel computer games' (section 50.0.5) we have put a student stand in front of the class with a sign 'I am process 5', and go through the above loop out loud ('Am I process zero? No. Am I process one? No.') which quickly drives home the point about the futility of this construct.

50.0.6 Further course summary

We have taught MPI based on the above ideas in two ways. First, we teach an academic class, that covers MPI, OpenMP, and general theory of parallelism in one semester. The typical enrollment is around 30 students, who do lab exercises and a programming project of their own choosing. We also teach a two-day intensive workshop (attendance 10–40 students depending on circumstances) of 6–8 hours per day. Students of the academic class are typically graduate or upper level undergraduate students; the workshops

get attendance from post-docs, academics, and industry too. The typical background is applied math, engineering, physical sciences.

We cover the following topics, with division over two days in the workshop format:

- Day 1: familiarity with SPMD, collectives, blocking and nonblocking two-sided communication.
- Day 2: exposure to: sub-communicators, derived datatypes. Two of the following: MPI-I/O, one-sided communication, process management, the profiling and tools interfaces, neighbourhood collectives.

50.0.6.1 Exercises

On day 1 the students do approximately 10 programming exercises, mostly finishing a skeleton code given by the instructor. For the day 2 material students do two exercises per topic, again starting with a given skeleton. (Skeleton codes are available as part of the repository [8].)

The design of these skeleton codes is an interesting problem in view of our concern with mental models. The skeletons are intended to take the grunt work away from the students, to both indicate a basic code structure and relieve them from making elementary coding errors that have no bearing on learning MPI. On the other hand, the skeletons should leave enough unspecified that multiple solutions are possible, including wrong ones: we want students to be confronted with conceptual errors in their thinking, and a too-far-finished skeleton would prevent them from doing that.

Example: the prime finding exercise mentioned above (which teaches the notion of functional parallelim) has the following skeleton:

```
int myfactor;
// Specify the loop header:
// for ( ... myfactor ... )
for (
    /**** your code here ****/
    ) {
  if (bignum%myfactor==0)
    printf("Process %d found factor %d\n",
           procno,myfactor);
  }
```

This leaves open the possibility of both a blockwise and a cyclic distribution of the search space, as well as incorrect solutions where each process runs through the whole search space.

50.0.6.2 Projects

Students in our academic course do a programming project in place of a final exam. Students can choose between one of a set of standard projects, or doing a project of their own choosing. In the latter case, some students will do a project in context of their graduate research, which means that they have an existing codebase; others will write code from scratch. It is this last category, that will most clearly demonstrate their correct understanding of the mental model underlying SPMD programs. However, we note that this is only a fraction of the students in our course, a fraction made even smaller by the fact that we also give a choice of doing a project in OpenMP rather than MPI. Since OpenMP is, at least to the beginning

programmer, simpler to use, there is an in fact a clear preference for it among the students who pick their own project.

50.0.7 Prospect for an online course

Currently the present author teaches MPI in the form of an academic course or short workshop, as outlined in section 50.0.6. In both cases, lecture time is far less than lab time, making the setup very intensive in teacher time. It also means that this setup is not scalable to a larger number of students. Indeed, while the workshops are usually webcast, we have not sufficiently solved the problem of supporting remote students. (The Pittsburgh Supercomputing Center offers courses that have remotely located teaching assistants, which seems a promising approach.) Such problems of support would be even more severe with an online course, where in-person support is completely absent.

One obvious solution to online teaching is automated grading: a student submits an exercise, which is then run through a checker program that tests the correct output. Especially if the programming assignment takes input, a checker script can uncover programming errors, notably in boundary cases.

However, the whole target of this paper is to uncover conceptual misunderstandings, for instance such as can lead to correct results with sub-optimal performance. In a classroom situation such misunderstandings are easily caught and cleared up, but to achieve this in a context of automated grading we need to go further.

We have started experiments with actually parsing code submitted by the students. This effort started in a beginning programming class taught by the present author, but is now being extended to the MPI courses.

It is possible to uncover misconceptions in students' understanding by detecting the typical manifestations of such misconceptions. For instance, the code in section 50.0.5.4 can be uncovered by detecting a loop where the upper bound involves a variable that was set by `MPI_Comm_size`. Many MPI codes have no need for such a loop over all processes, so detecting one leads to an alert for the student.

Note that no tools exist for such automated evaluation. The source code analysis needed falls far short of full parsing. On the other hand, the sort of constructs is it supposed to detect, are normally not of interest to the writers of compilers and source translators. This means that by writing fairly modest parsers (say, less than 200 lines of python) we can perform a sophisticated analysis of the students' codes. We hope to report on this in more detail in a follow-up paper.

50.0.8 Evaluation and discussion

At the moment, no rigorous evaluation of the efficacy of the above ideas has been done. We intend to perform a comparison between outcomes of the proposed way of teaching and the traditional way by comparing courses at two (or more) different institutions and from different syllabi. The evaluation will then be based on evaluating the independent programming project.

However, anecdotal evidence suggests that students are less likely to develop 'centralized' solutions as described in section 50.0.2.2. This was especially the case in our semester-long course, where the students have to design and implement a parallel programming project of their own choosing. After teaching the 'symmetric' approach, no students wrote code based on a manager-worker model, or using centralized

storage. In earlier semesters, we had seen students do this, even though this model was never taught as such.

50.0.9 Summary

In this paper we have introduced a nonstandard sequence for presenting the basic mechanisms in MPI. Rather than starting with sends and receives and building up from there, we start with mechanisms that emphasize the inherent symmetry between processes in the SPMD programming model. This symmetry requires a substantial shift in mindset of the programmer, and therefore we target it explicitly.

In general, it is the opinion of this author that it pays off to teach from the basis of instilling a mental model, rather than of presenting topics in some order of (perceived) complexity or sophistication.

Comparing our presentation as outlined above to the standard presentation, we recognize the downplaying of the blocking send and receive calls. While students learn these, and in fact learn them before other send and receive mechanisms, they will recognize the dangers and difficulties in using them, and will have the combined sendrecv call as well as nonblocking routines as standard tools in their arsenal.

50.1 Sources used in this chapter

50.1.1 Listing of code header

PART IX

BIBLIOGRAPHY, INDEX, AND LIST OF ACRONYMS

Chapter 51

Bibliography

[1] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on millions of cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[2] Y. Ben-David Kolikant. Gardeners and cinema tickets: High schools' preconceptions of concurrency. *Computer Science Education*, 11:221–245, 2001.

[3] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. Collective communication: theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19:1749–1783, 2007.

[4] Tom Cornebize, Franz C Heinrich, Arnaud Legrand, and Jérôme Vienne. Emulating High Performance Linpack on a Commodity Server at the Scale of a Supercomputer. working paper or preprint, December 2017.

[5] Lisandro Dalcin. MPI for Python, homepage. https://mpi4py.bitbucket.io/.

[6] Saeed Dehnadi, Richard Bornat, and Ray Adams. Meta-analysis of the effect of consistency on success in early learning of programming. In *Psychology of Programming Interest Group PPIG 2009*, pages 1–13. University of Limerick, Ireland, 2009.

[7] Peter J. Denning. Computational thinking in science. *American Scientist*, pages 13–17, 2017.

[8] Victor Eijkhout. *Parallel Programming in MPI and OpenMP*. 2016. available for download: https://bitbucket.org/VictorEijkhout/parallel-computing-book/src.

[9] Victor Eijkhout. Performance of MPI sends of non-contiguous data. *arXiv e-prints*, page arXiv:1809.10778, Sep 2018.

[10] Brice Goglin. Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc). In *International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, July 2014. IEEE.

[11] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994.

[12] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. ISBN-10: 0131532715, ISBN-13: 978-0131532717.

[13] Torsten Hoefler, Prabhanjan Kambadur, Richard L. Graham, Galen Shipman, and Andrew Lumsdaine. A case for standard non-blocking collective operations. In *Proceedings, Euro PVM/MPI*, Paris, France, October 2007.

[14] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *SIGPLAN Not.*, 45(5):159–168, January 2010.

[15] INRIA. SimGrid homepage. http://simgrid.gforge.inria.fr/.
[16] L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. In *Parallel Programming using C++, G. V. Wilson and P. Lu, editors*, pages 175–213. MIT Press, 1996.
[17] M. Li, H. Subramoni, K. Hamidouche, X. Lu, and D. K. Panda. High performance mpi datatype support with user-mode memory registration: Challenges, designs, and benefits. In *2015 IEEE International Conference on Cluster Computing*, pages 226–235, Sept 2015.
[18] Zhenying Liu, Barbara Chapman, Tien-Hsiung Weng, and Oscar Hernandez. Improving the performance of openmp by array privatization. In *Proceedings of the OpenMP Applications and Tools 2003 International Conference on OpenMP Shared Memory Parallel Programming*, WOMPAT'03, pages 244–259, Berlin, Heidelberg, 2003. Springer-Verlag.
[19] Robert McLay. T3pio: TACC's terrific too for parallel I/O. https://github.com/TACC/t3pio.
[20] MPI forum: MPI documents. http://www.mpi-forum.org/docs/docs.html.
[21] NASA Advaned Supercomputing Division. NAS parallel benchmarks. https://www.nas.nasa.gov/publications/npb.html.
[22] The OpenMP API specification for parallel programming. http://openmp.org/wp/openmp-specifications/.
[23] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference, Volume 1, The MPI-1 Core*. MIT Press, second edition edition, 1998.
[24] Jeff Squyres. Mpi-request-free is evil. Cisco Blogs, January 2013. https://blogs.cisco.com/performance/mpi_request_free-is-evil.
[25] R. Thakur, W. Gropp, and B. Toonen. Optimizing the synchronization operations in MPI one-sided communication. *Int'l Journal of High Performance Computing Applications*, 19:119–128, 2005.
[26] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.
[27] P.C. Wason and P.N. Johnson-Laird. *Thinking and Reasoning*. Harmondsworth: Penguin, 1968.

Chapter 52

List of acronyms

API Application Programmer Interface
AMG Algebraic MultiGrid
AVX Advanced Vector Extensions
BLAS Basic Linear Algebra Subprograms
BSP Bulk Synchronous Parallel
CAF Co-array Fortran
CRS Compressed Row Storage
CG Conjugate Gradients
CUDA Compute-Unified Device Architecture
DAG Directed Acyclic Graph
DPCPP Data Parallel C++
DSP Digital Signal Processing
FEM Finite Element Method
FPU Floating Point Unit
FFT Fast Fourier Transform
FSA Finite State Automaton
GPU Graphics Processing Unit
HPC High-Performance Computing
HPF High Performance Fortran
ICV Internal Control Variable
LAPACK Linear Algebra Package
MIC Many Integrated Cores
MPMD Multiple Program Multiple Data
MIMD Multiple Instruction Multiple Data
MPI Message Passing Interface
MPL Message Passing Layer
MTA Multi-Threaded Architecture
NIC Network Interface Card
NUMA Non-Uniform Memory Access
OO Object-Oriented
OS Operating System
PGAS Partitioned Global Address Space
PDE Partial Diffential Equation
PRAM Parallel Random Access Machine
RDMA Remote Direct Memory Access
RMA Remote Memory Access
SAN Storage Area Network
SaaS Software as-a Service
SFC Space-Filling Curve
SIMD Single Instruction Multiple Data
SIMT Single Instruction Multiple Thread
SLURM Simple Linux Utility for Resource Management
SM Streaming Multiprocessor
SMP Symmetric Multi Processing
SOR Successive Over-Relaxation
SP Streaming Processor
SPMD Single Program Multiple Data
SPD symmetric positive definite
SSE SIMD Streaming Extensions
STL Standard Template Library
TACC Texas Advanced Computing Center
TLB Translation Look-aside Buffer
UMA Uniform Memory Access
UPC Unified Parallel C
URI Uniform Resource Identifier
WAN Wide Area Network

Chapter 53

General Index

`-malloc_debug`, 582
`-malloc_test`, 582

active target synchronization, 299, 304
address
    physical, 370
    virtual, 370
adjacency
    graph, 543
affinity, **626**
    process and thread, 626–629
    thread
        on multi-socket nodes, 477
alignment, 218
all-to-all, 40
allocate
    and private/shared data, 445
allreduce, 40
`argc`, 26, 28
`argv`, 26, 28
array
    static, 120
atomic operation, 458, 485
    file, 354
    MPI, 313–317
    OpenMP, 460

backfill, 666
bandwidth, 85
    bisecection, 90
barrier, 395
    implicit, 485
    nonblocking, 83
Basic Linear Algebra Subprograms (BLAS), 521
batch
    job, 15, 662
    scheduler, 15
    script, 662
    system, 661
Beowulf cluster, 14
block row, 531
Boolean satisfiability, 34
boost, 17
breakpoint, 650
broadcast, **38**
`btl_openib_eager_limit`, 126
`btl_openib_rndv_eager_limit`, 126
bucket brigade, 62, 89, **127**, 654
buffer
    MPI, in C, 44
    MPI, in Fortran, 44
    MPI, in MPL, 45
    MPI, in Python, 45
    receive, 122
butterfly exchange, 656

C
    MPI bindigs, *see* MPI, C bindings
C++
    MPI bindings, *see* MPI, C++ bindings
    standard library, 214
        vector, 214
C++ iterators
    in OMP reduction, 455

C99, 192
`c_sizeof`, 198
cacheline, 452
callback, 572
cast, 44
`CC`, 653
Charmpp, 15
chunk, 433
`chunk`, 433
client, 288
cluster, 661
cmake, 667
`Codimension`, 598
collective
  split, 350
collectives, **38**
  neighbourhood, **360**, 398
  nonblocking, 78
    cancelling, 404
column-major storage, 206
communication
  asynchronous, **180**
  blocking, **123–127**
    vs nonblocking, 398
  buffered, 181, **182**, 398
  local, **181**
  nonblocking, **133–146**
  nonlocal, **181**
  one-sided, 299–327
  one-sided, implementation of, 326–327
  partitioned, *see* partitioned,communication
  persistent, 398
  synchronous, **180**
  two-sided, 156
communicator, **30**, 265–276
  info object, 386
  inter, 271, **272**, 272, 285
  intra, 272, 274
  object, 30
  peer, 272
  variable, 30
compare-and-swap, 131
compiler, 217
  optimization level, 645
completion, **320**
  local, 320
  remote, 320
compute
  node, 661
`configure.log`, 516
construct, **416**
contention, 90
contiguous
  data type, 199
control variable
  access, 379
  handle, 379
core, 23, **411**
core dump, 645
`cosubscript`, 599
cpp, 414
cpuinfo, **624**
Cray
  MPI, 16
  T3E, 401
critical section
  flush at, 486
critical section, 451, 459, 494
curly braces, 415

Dalcin
  Lisandro, 18
data dependency, 470
Data Parallel C++ (DPCPP), 603
data race, *see* race condition
datatype, 191–227
  big, 223–225
  derived, 198–345
  different on sender and receiver, 203
  elementary, 192–198
    in C, 192
    in Fortran, 193
    in Python, 194
  signature, 215
datatypes
  derived, 191
  elementary, 191

`date`, 664
ddd, 645
DDT, 645, 651
deadlock, 83, 120, **123**, 133, 398, 399, 651
debug flag, 646
`debug_mt`, 182
debugger, 645
debugging, 645–652
  parallel, 651
dense linear algebra, 267
destructor, 264
directive
  end-of, 415
directives, 414, **414–415**
  cpp, 423
displacement unit, 323
distributed array, 34
distributed shared memory, 299
doubling
  recursive, *see* recursive doubling
dynamic mode, 412

eager limit, **124–126**, 398
Eclipse, 651
  PTP, 651
edge
  cuts, 543
  weight, 543
ensemble, 665
environment
  of batch job, 664
environment variables, 635
epoch, **304**
  access, 305, 312
  communication, 304
  completion, 322
  exposure, 305, 312
  passive target, **317**
error return, 17
ethernet, 16

false sharing, 430, 452
Fast Fourier Transform (FFT), 541
fat-tree, 626
fence, 304
fftw, 516, 541
Fibonacci sequence, 462–464
file
  pointer
    advance by write, 351
    individual, 350
  system
    shared, 666
file system
  shared, 345
first-touch, 481, 627
five-point stencil, 83
fork/join model, **411**, 418, 469
Fortran
  1-based indexing, 140
  90
    bindings, 18
  2008, 30
  array syntax, 440
  assumed-shape arrays in MPI, 393
  fixed-form source, 415
  forall loops, 440
  Fortran90, 26
  line length, 580
  MPI bindings, *see* MPI, Fortran bindings
  MPI equivalences of scalar types, 193
  MPI issues, 393
Fortran2008, 307
  MPI bindings, *see* MPI, Fortran2008 bindings
Fortran77
  PETSc interface, 511
Fortran90
  PETSc interface, 511
  types
    in MPI, 393

gather, **38**
Gauss-Jordan algorithm, 52
Gaussian elimination, 564
gcc
  thread affinity, 482
gdb, 645–651
`gemv`, 537

getrusage, 588
ghost region, 685
GNU, 645
  gdb, *see* gdb
Gram-Schmidt, 43
graph
  partitioning
    packages, 543
  topology, **360**, 398
  unweighted, 362
graph topology, 626
grid
  Cartesian, 357, 549
  periodic, 357
  processor, 626
group, 312
group of
  processors, 313

halo, 685
  update, 309
halo region, 541
handshake, 399
hdf5, 345, 639
heap, 417
heat equation, 478
Hessian, 576
hipsycl, 610
histogram, 461
hostname, 389
hwloc, 624
hwloc, 627
hybrid
  computing, 665
hyperthreading, 627
Hypre, 516, 570, 571
  pilut, 571

I/O
  in OpenMP, 439
I_MPI_ASYNC_PROGRESS, 182
I_MPI_ASYNC_PROGRESS_..., 182
I_MPI_ASYNC_PROGRESS_THREADS, 182
I_MPI_EAGER_THRESHOLD, 126
ibrun, 287, 514, 654
idle, 666
image, **598**
image_index, 599
immediate operation, 134
implicit barrier, 459
  after single directive, 439
incomplete operation, *see* operation, incomplete
indexed
  data type, 199
inner product, 43
input redirection
  shell, 400
instrumentation, 653
Intel, 399
  compiler
    thread affinity, 482
  compiler suite, 624
  Haswell, 627
  Knight's Landing, 411, 489
    thread placement, 480
  Knightslanding, 661
  MPI, 16, 90, 126, 182, 287
  Paragon, 182
  Sandybridge, 411, 627
  Skylake, 661
interconnect, 658
Internal Control Variable (ICV), **493–495**

job, 661
  array, 665
  cancel, 664
job script, 662
jumpshot, 654

KIND, 197
KMP_AFFINITY, 482

Laplace equation, 534
latency, 85
  hiding, **143**, 398, 525, 528
latency hiding, 536
launcher, 665
lcobound, 599

lexical scope, 442
Linear Algebra Package (LAPACK), 521
linked list, 466
linker
    weak symbol, 396
`listing`, 673
load balancing, 430
load imbalance, 430
local
    operation, **181**, 182
local operation, 144
local refinement, 83
lock, 460, **460–462**
    flush at, 486
    nested, 462
login
    node, 661
Lonestar5, 627
LU factorization, 432, 564
Lustre, 639

`make.log`, 516
malloc
    and private/shared data, 445
`malloc`, 417, 481
manager-worker, 144, 149, 153, 680
Mandelbrot set, 34, 88, 448, 679
matching, 399
matching queue, 147
Matlab
    parallel computing toolbox, 665
matrix
    sparse, 79, 536
    transposition, 268
matrix-vector product, 564
    dense, 62
    sparse, 65
Mellanox, 399
memory
    coherent, 322
    shared, MPI, 302
memory leak, 145
message
    collision, 399
    count, 151
    source, **122**
    status, 122, **148–153**
        error, 150
        source, 149
        tag, 150
    tag, 120, 635
Message Passing Layer (MPL), 17, 32
messsage
    target, **120**
MKL, 521
mkl, 517
ML, 571
Monte Carlo codes, 33
motherboard, 410
`move_pages`, 481
MPI
    3
        Fortran2008 interface, 17
    C bindings, 17
    C++ bindings, 17
    constants, 402–403
        compile-time, 402
        link-time, 402
    datatype
        extent, 218
        size, 198
        subarray, 219
        vector, 198
    Fortran bindings, 17–18
    Fortran issues, *see* Fortran, MPI issues
    Fortran2008 bindings, 17–18
    I/O, 392, 639
    initialization, 26
    MPI-1, 357, 364
    MPI-2, 285, 387, 390
    MPI-3, 78, 209, 214, 223, 313, 367, 393
        C++ bindings removed, 17
    MPI-3.1, 402
    MPI-3.2, 404
    MPI-4, 17, 179, 198, 367, 391
    Python bindings, 18
    semantics, 399

tools interface, 397
version, 389
mpi.h, 26
`mpi.h`, 17, 18
`mpi.hpp`, 17
MPI/O, 345–354
`mpi4py`, 18, 32
`mpi_f08`, 17, 18, 307
mpich, 16
mpiexec, 15, 25, 28, 260, 285
options, 16
`mpiexec`, 25
mpif.h, 26
MPIR, **399**
mpirun, 15, 16, 28
and environment variables, 635
MPL, **17**
compiling and linking, 17
`mulpd`, 489
`mulsd`, 489
multicore, 412
Multiple Program Multiple Data (MPMD), 401
mumps, 516
`MV2_IBA_EAGER_THRESHOLD`, 126
mvapich, 634
mvapich2, 126, 631

nested parallelism, 423–425
netcdf, 345
network
card, 399
contention, 399
port
oversubscription, 399
`new`, 417
Newton's method, **576**
node, **23**, 661
cluster, 410
non-blocking communication, 132
norm
one, 78
`np.frombuffer`, 77
`NULL`, 378
null terminator, 383
`num_images`, 599
`numactl`, 482, 627
numerical integration, 429
Numpy, 194
numpy, 18, 45, 196, 302

`od`, 345
offloading
vs onloading, 399
omp
barrier
implicit, 459
for
barrier behaviour, 459
reduction, 451–455
user-defined, **454–455**
`OMP_NUM_THREADS`, 635, 665
OneAPI, **603**
onloading, *see* offloading, vs onloading
opaque handle, 30, 44
OpenMP, 367
accelerator support in, 497
co-processor support in, 497
compiling, 413–414
environment variables, 414, 418, **493–495**
library routines, 418
library routines, **493–495**
OpenMP-3.1, 453
places, 477
running, 414
tasks, 466–473
data, 467–468
dependencies, 470–471
synchronization, 468–470
version 4, 472
OpenMPI, 126
operating system, 497
operation
non-local, 123
operator, 74–78
predefined, 74
user-defined, 75
option
prefix, 587

origin, 299, 312
overlapping computation and communication, *see* latency, hiding
owner computes, 117

package, 624
packing, 225
page
  small, 370
  table, 370
parallel
  data, 481
  embarrassingly, 481
parallel region, 412, 421–425, 438
  barrier at the end of, 459
  dynamic scope, 424, 443
  flush at, 486
parallel regions
  nested, 494
parameter sweep, **665**
`paraprof`, 654
parasails, 570
ParMetis, 398, 543
partition, 664
partitioned communication, 179–180
passive target synchronization, 300, 314, 317
`pbing`, 482
persistent
  collectives, **179**
  communication, 179
  point-to-point, **178–179**
persistent communication, 132, *see* communication, persistent
PETSc, 398
  interoperability with BLAS, 521
  interoperability with MPI, 521
  log files, 516
`PETSC_OPTIONS`, 587
pin a thread, 627
ping-pong, 118, 396, 600
`PMI_RANK`, 401
point-to-point, 117
pointer
  null, 59
polling, 141, 182
posting
  of send/receive, 134
pragma, *see* for list see under 'omp', **414**
preconditioner, 565, **568**
  block jacobi, 587
  field-split, 540
prefix operation, 54
private variables, 418
`proc_bind`, 422
process, 23
processes status of, 401
producer-consumer, 484
`PROFILEDIR`, 653
progress
  asynchronous, 143, 181
purify, 648
PVM, 15, 285
`pylauncher`, 665
Python
  MPI bindigs, *see* MPI, Python bindings
  multiprocessing, 665
  PETSc interface, 512

queue, 661
  in-order, 605
  SYCL, 604

race condition, 313, 451, 459, 485, 495, 599
  in OpenMP, 485–486
radix sort, 64
RAID
  array, 666
RAM
  disc, 666
random number generator, 447, 496
Ranger, 626
`rar`, 324
`raw`, 324
ray tracing, 370
recursive doubling, 90
redirection, *see* shell, input redirection
reduction, **38**
region of code, **416**

register
SSE2, 489
release_mt, 182
residual, 565
Riemann sums, 429
RMA
active, 299
passive, 300
root, 46
root process, 38

sbatch, 662
scalable
in space, 89
in time, 88
scalapack, 574
scan, 40
exclusive, 56
inclusive, 54
scancel, 663
scanf, 583
scatter, **38**
sched_setaffinity, 482
scope
lexical, 412, 416
of variables, 412, 416
SEEK_SET, 354
segfault, 582
segmentation fault, 647
segmented scan, 57
send
buffer, 120
ready mode, 181
synchronous, 181
sentinel, **415**, 422
sequential
semantics, 511
sequential consistency, 486
serialization, 126
server, 288
session, **291**
performance experiment, 380
session model, **291**, 291
SetThreadAffinityMask, 482
shared data, 412
shared memory, *see* memory, shared
shared variables, 418
shell
matrix, 572
shmem, 401
silo, 345
SimGrid, 88, 658–660
compiler, 658
sinfo, 662
Single Program Multiple Data (SPMD), 415, 421
sizeof, 302, 393
sleep, 664
SLURM, 355
smpicc, 658
smpirun, 658
socket, **23**, 369, **410**, 631
sort
odd-even transposition, 131
radix, 64
swap, 131
sparse approximate inverse, 569, 570
sparse matrix vector product, 57
spin-lock, 494
squeue, 662, 663, 666
srun, 662
ssh, 15
connection, 661
ssh, 666
stack, 417, 494
overflow, 443
per thread, 443
Stampede, 661
compute node, 627
largemem node, 627
node, 411
Stampede2, 661
standard deviation, 39
start/affinity, 482
status
of received message, 148
stderr, 400
stdout, 400

stdout/err of, 400
stencil, 549
    nine-point, 358
    star, 358
storage association, 443, 446
`storage_size`, 198
stride, 206
`stringstream`, 422
struct
    data type, 199
structured block, **416**
Sun
    compiler, 482
`SUNW_MP_PROCBIND`, 482
SYCL, **603**
symbol table, 645, 646
`sync`, 599
synchronization
    in OpenMP, 458–464

T3PIO, **352**
TACC, 661
    portal, 512
`tacc_affinity`, 482, 627, 631
`TACC_TAU_DIR`, 653
tag, *see* message, tag
    bound on value, 388
target, 299, 312
    active synchronization, *see* active target synchronization
    passive synchronization, *see* passive target synchronization
task
    scheduler, 466
    scheduling point, 471
`taskset`, 482
TAU, 653–657
`TAU_PROFILE`, 653
`tau_timecorrect`, 654
`TAU_TRACE`, 653
`this_image`, 599
thread
    affinity, 477–480
    initial, **422**
    master, **422**
    migrating a, 480
    primary, **422**
    private data, 446
thread-safe, 495–496
threads, 411
    hardware, 412, 627
    master, 412
    team of, 411, 422
time slicing, 23, 412
time-slicing, 285
timing
    MPI, 394–396
`top`, 664, 666
topology
    virtual, 357
TotalView, 645, 651
`TRACEDIR`, 653
transpose, 82
    and all-to-all, 64
    data, 541
    recursive, 268
    through derived types, 222–223
tree
    traversal
        post-order, 473
        pre-order, 473
tunnel
    ssh, 400

`ucobound`, 599
`ulimit`, 443
Unix
    process, 443

`valarray`, 481
valgrind, 648–649
vector
    data type, 199
    instructions, 488
`verbatim`, 673
virtual shared memory, 299
VTune, 653

wall clock, 394

`war`, 324
`waw`, 324
weak symbol, *see* linker, weak symbol
while loop, 466
while loops, 435
window, 299–305
    consistency, 320
    displacement, 306
    displacement unit, 324
    info object, 386
    memory, *see also* memory model
        model, 322
        separate, 321, 322
        unified, 322
    memory allocation, 301–304
    private, 322
    public, 322
work sharing, 412
work sharing construct, 418, 438
workshare
    flush after, 486
worksharing constructs
    implied barriers at, 459
world model, **291**, 291
wormhole routing, 90
wraparound connections, 357

XSEDE
    portal, 512

Zoltan, 398, 543

Chapter 54

Index of MPI commands and keywords

0_MPI_OFFSET_KIND, **353**

access_style, 387
accumulate_ops, 324
accumulate_ordering, 324
alloc_shared_noncontig, 369

cb_block_size, 387
cb_buffer_size, 387
cb_nodes, 387
chunked, 387
chunked_item, 387
chunked_size, 387
collective_buffering, 387

file_perm, 387

io_node_list, 387
irequest_pool, 140

KSPSolve, 566

mpi://SELF, 292
mpi://WORLD, 292
MPI_2DOUBLE_PRECISION, **75**
MPI_2INT, **75**
MPI_2INTEGER, **75**
MPI_2REAL, **75**
MPI_Abort, **27**, 78, 392, 401
MPI_Accumulate, 305, **310**, 316, 324
MPI_Add_error_class, **392**
MPI_Add_error_code, **392**
MPI_Add_error_string, **392**
MPI_ADDRESS_KIND, 193, **196**, **307**, 402
MPI_AINT, **192**, **196**
MPI_Aint, 192, 193, **195**, 196, 306
  in Fortran, 196
MPI_Aint_add, **196**
MPI_Aint_diff, **196**
MPI_Allgather, **62**, 92
MPI_Allgather_init, **179**
MPI_Allgatherv, **70**
MPI_Allgatherv_init, **179**
MPI_Alloc_mem, 301, **303**, 318, 390
MPI_Allreduce, **41**, 42, 48, 65, 92
MPI_Allreduce_init, **179**
MPI_Alltoall, **62**, 64, 542
MPI_Alltoall_init, **179**
MPI_Alltoallv, **64**, 64, 70, 74, 212
MPI_Alltoallv_init, **179**
MPI_Alltoallw_init, **179**
MPI_ANY_SOURCE, 68, 88, **122**, 122, 144, 148, 149, 152, 299, 399, 403
MPI_ANY_TAG, **122**, 128, 148, 150, 403
MPI_APPNUM, **388**
MPI_ARGV_NULL, 403
MPI_ARGVS_NULL, 403
MPI_ASYNC_PROTECTS_NONBLOCKING, 402
MPI_Attr_get, **387**
MPI_BAND, **75**
MPI_Barrier, **69**, 353, 395
MPI_Barrier_init, **179**
MPI_Bcast, **50**, 92
MPI_Bcast_init, **179**
MPI_BOR, **75**

MPI_BOTTOM, 196, 325, 402
MPI_Bsend, 181, **182**, 184
MPI_Bsend_init, 178, 181, **185**, 185
MPI_BSEND_OVERHEAD, **184**, 185, 226, 403
MPI_Buffer_attach, **184**, 184
MPI_Buffer_detach, **184**, 184
MPI_BXOR, **75**
MPI_BYTE, **192**, 199, 218
MPI_Cancel, **403**, 404
MPI_CART, **357**
MPI_Cart_coords, **358**
MPI_Cart_create, **358**
MPI_Cart_rank, **358**
MPI_CHAR, **192**
MPI_CHARACTER, **193**
MPI_Close_port, **288**
MPI_Comm, 18–20, 30, **260**, 402
MPI_Comm_accept, **288**, 288
MPI_Comm_compare, **262**, 274
MPI_Comm_connect, 288, **289**, 289, 390
MPI_Comm_create, 265, **269**
MPI_Comm_create_errhandler, **392**, 393
MPI_Comm_create_group, **269**
MPI_Comm_disconnect, **289**
MPI_Comm_dup, 31, **261**, 262, 264, 265, 387, 515
MPI_Comm_dup_with_info, **261**, **387**
MPI_Comm_free, 264, **265**
MPI_Comm_get_attr, 285, **387**
MPI_Comm_get_errhandler, **390**, 392
MPI_Comm_get_info, **386**
MPI_Comm_get_parent, **274**, 287, 292
MPI_Comm_group, **269**, 274
MPI_Comm_idup, **261**
MPI_Comm_idup_with_info, **261**
MPI_Comm_join, **290**, 292
MPI_COMM_NULL, 260, 261, 263, 265, 274, 367
MPI_Comm_rank, **31**, 31, 32, 267, 274, 362
MPI_Comm_remote_group, **274**
MPI_Comm_remote_size, **274**, 287
MPI_COMM_SELF, 260, 261, 522
MPI_Comm_set_errhandler, **390**, 390, 392
MPI_Comm_set_info, **386**
MPI_Comm_set_name, 261
MPI_Comm_size, **31**, 31, 32, 120, 274, 362
MPI_Comm_spawn, 265, **285**, 394
MPI_Comm_spawn_multiple, 288, 388
MPI_Comm_split, 68, 265, **266**, 267, 276
MPI_Comm_split_type, 268, **367**, 371
MPI_Comm_test_inter, **274**
MPI_COMM_TYPE_HW_GUIDED, 367
MPI_COMM_TYPE_HW_UNGUIDED, 367
MPI_COMM_TYPE_SHARED, 367
MPI_COMM_WORLD, 30, 260, 261, 265, 266, 271, 272, 276, 285, 287, 292, 371, 388, 392, 394, 402, 404, 511, 515, 523
MPI_Compare_and_swap, **315**
MPI_COMPLEX, **193**
MPI_CONGRUENT, **263**
MPI_Count, 193, 220, 223, **224**
MPI_COUNT_KIND, **193**, 402
MPI_Cvar_get_num, **378**
MPI_Datatype, 59, **191**, **200**, 200
MPI_DATATYPE_NULL, 193, **200**
MPI_DISPLACEMENT_CURRENT, **352**
MPI_DIST_GRAPH, **357**, 357
MPI_Dist_graph_create, **360**, 360–362, 364
MPI_Dist_graph_create_adjacent, 360
MPI_Dist_graph_neighbors, **362**, **363**, 363
MPI_Dist_graph_neighbors_count, **362**, **363**
MPI_DOUBLE, **192**, 192
MPI_DOUBLE_COMPLEX, **193**
MPI_DOUBLE_INT, **75**, 75
MPI_DOUBLE_PRECISION, 192, **193**
MPI_ERR_ARG, **390**
MPI_ERR_BUFFER, 185, **390**
MPI_ERR_COMM, **390**, 394
MPI_ERR_IN_STATUS, 150, **390**
MPI_ERR_INFO, **390**
MPI_ERR_INTERN, 185, **390**
MPI_ERR_LASTCODE, **390**, 392
MPI_ERR_NO_MEM, 303, **390**
MPI_ERR_OTHER, **390**
MPI_ERR_PORT, 289, **390**
MPI_ERR_PROC_ABORTED, **390**
MPI_ERR_SERVICE, **290**, **390**
MPI_ERRCODES_IGNORE, **285**, 402

MPI_Errhandler, **390**, 391
MPI_Errhandler_c2f, 292
MPI_Errhandler_create, **390**
MPI_Errhandler_f2c, 292
MPI_Errhandler_free, 292, **390**
MPI_ERROR, **150**, 390, 391
MPI_Error_class, 292
MPI_Error_string, 292, **390**, **391**
MPI_ERRORS_ABORT, **391**
MPI_ERRORS_ARE_FATAL, **391**
MPI_ERRORS_RETURN, 391, 392
MPI_Exscan, **54**, **56**, 57
MPI_Exscan_init, **179**
MPI_Fetch_and_op, **313**, 314, 317, 321
MPI_File, **346**
MPI_File_call_errhandler, **390**
MPI_File_close, **346**
MPI_File_delete, **347**
MPI_File_get_size, **353**
MPI_File_get_view, **353**
MPI_File_iread, **348**
MPI_File_iread_all, **348**
MPI_File_iread_at_all, **350**
MPI_File_iread_shared, **350**, **353**
MPI_File_iwrite, **348**
MPI_File_iwrite_all, **348**
MPI_File_iwrite_at_all, **350**
MPI_File_iwrite_shared, **350**, **353**
MPI_File_open, 40, **346**, 387
MPI_File_preallocate, **353**
MPI_File_read, **348**, 355
MPI_File_read_all, **348**
MPI_File_read_all_begin, **350**
MPI_File_read_all_end, **350**
MPI_File_read_at, **348**
MPI_File_read_at_all, **348**
MPI_File_read_ordered, **353**
MPI_File_read_shared, **353**, 355
MPI_File_seek, **347**, **350**, 351, 355
MPI_File_seek_shared, **353**, 355
MPI_File_set_atomicity, **354**
MPI_File_set_errhandler, **390**
MPI_File_set_info, 387
MPI_File_set_size, **353**
MPI_File_set_view, **352**, 352, 355, 387
MPI_File_sync, **347**
MPI_File_write, **347**, 350, 352
MPI_File_write_all, **348**
MPI_File_write_all_begin, **350**
MPI_File_write_all_end, **350**
MPI_File_write_at, **348**, **350**, 350
MPI_File_write_at_all, **348**
MPI_File_write_ordered, **353**
MPI_File_write_shared, **353**
MPI_Finalize, **26**, 27, 28, 291, 515
MPI_Finalized, **28**, 292
MPI_FLOAT, **192**, 192
MPI_FLOAT_INT, **75**
MPI_Gather, **59**, 69, 70, 92, 209, 345
MPI_Gather_init, **179**
MPI_Gatherv, 62, **70**, 70, 212
MPI_Gatherv_init, **179**
MPI_Get, 305, **308**, 316, 318, 321, 323
MPI_Get_accumulate, 311, **313**, 313, 324
MPI_Get_address, 196, **213**, 216, 323
MPI_Get_count, 146, **151**, 223
MPI_Get_count_x, 225
MPI_Get_elements, **151**, 223
MPI_Get_elements_x, 151, **224**
MPI_Get_hw_resource_types, **367**
MPI_Get_library_version, 292
MPI_Get_processor_name, 27, **29**, 29, 389
MPI_Get_version, 292, **389**
MPI_GRAPH, **357**
MPI_Graph_create, **364**
MPI_Graph_get, **364**
MPI_Graph_neighbors, **364**
MPI_Graph_neighbors_count, **364**
MPI_Graphdims_get, **364**
MPI_Group, **269**, 269
MPI_Group_difference, 269
MPI_Group_excl, **269**
MPI_Group_incl, **269**
MPI_HOST (deprecated), 388
MPI_Iallgather, **80**
MPI_Iallgatherv, **80**

MPI_Iallreduce, **80**
MPI_Ialltoall, **81**
MPI_Ialltoallv, **81**
MPI_Ialltoallw, **81**
MPI_Ibarrier, 79, **81**, **83**
MPI_Ibcast, **81**
MPI_Ibsend, 181, 182, 185
MPI_IDENT, 262
MPI_Iexscan, **81**
MPI_Igather, **80**, 82
MPI_Igatherv, **80**
MPI_IN_PLACE, 48, 59, 403
MPI_Ineighbor_allgather, 363
MPI_Ineighbor_allgatherv, 363
MPI_Ineighbor_alltoall, 363
MPI_Ineighbor_alltoallv, 363
MPI_Ineighbor_alltoallw, 363
MPI_Info, 303, 324, 352, 369, **383**, 387
MPI_Info_c2f, 292
MPI_Info_create, 292, **383**
MPI_Info_create_env, 292
MPI_Info_delete, 292, **383**
MPI_Info_dup, 292, **383**
MPI_INFO_ENV, 28, **383**
MPI_Info_f2c, 292
MPI_Info_free, 292, **383**
MPI_Info_get (deprecated), 383
MPI_Info_get, 292, **383**
MPI_Info_get_nkeys, 292, **383**
MPI_Info_get_nthkey, 292, **383**
MPI_Info_get_string, **383**
MPI_Info_get_valuelen (deprecated), 383
MPI_Info_get_valuelen, 292
MPI_INFO_NULL, 352
MPI_Info_set, 292, **383**
MPI_Init, **26**, 27, 28, 291, 378, 402, 404, 514, 594
  in Fortran, 393
MPI_Init_thread, 27, 291, 378, 404, **633**, 633
MPI_Initialized, **28**, 292
MPI_INT, 191, **192**, 404
MPI_INTEGER, **193**, 193
MPI_INTEGER1, **193**
MPI_INTEGER16, **193**, 193
MPI_INTEGER2, **193**
MPI_INTEGER4, **193**
MPI_INTEGER8, **193**
MPI_INTEGER_KIND, 402
MPI_Intercomm_create, 265, **272**
MPI_Intercomm_merge, **274**
MPI_IO, 388
MPI_Iprobe, 84, 146, 182
MPI_Irecv, 80, 90, **134**, 134, 142, 143, 145, 148, 149, 156, 178
MPI_Ireduce, **80**
MPI_Ireduce_scatter, **80**
MPI_Ireduce_scatter_block, **80**
MPI_Irsend, 181
MPI_Is_thread_main, **633**
MPI_Iscan, **81**
MPI_Iscatter, **80**, 82
MPI_Iscatterv, **80**
MPI_Isend, 90, **134**, 134, 156, 178, 184, 311
MPI_Isendrecv, **132**
MPI_Isendrecv_replace, **132**
MPI_Issend, **180**, 181
MPI_KEYVAL_INVALID , 403
MPI_LAND, **75**
MPI_LASTUSEDCODE, **392**
MPI_LOCK_EXCLUSIVE, 318, 403
MPI_LOCK_SHARED, 318, 403
MPI_LOGICAL, **193**
MPI_LONG, **192**
MPI_LONG_DOUBLE, **192**
MPI_LONG_DOUBLE_INT, **75**
MPI_LONG_INT, **75**
MPI_LONG_LONG_INT, **192**
MPI_LOR, **75**
MPI_LXOR, **75**
MPI_MAX, 46, **75**
MPI_MAX_DATAREP_STRING, 402
MPI_MAX_ERROR_STRING, **391**, 402
MPI_MAX_INFO_KEY, **383**, 402
MPI_MAX_INFO_VAL, 402
MPI_MAX_LIBRARY_VERSION_STRING, 402
MPI_MAX_OBJECT_NAME, 402
MPI_MAX_PORT_NAME, **288**, 402

MPI_MAX_PROCESSOR_NAME, **29**, 29, **389**, 402
MPI_MAXLOC, **75**, 75
MPI_Message, **147**
MPI_MIN, **75**
mpi_minimum_memory_alignment, 303, 369, 387
MPI_MINLOC, **75**, 75
MPI_MODE_APPEND, **347**
MPI_MODE_CREATE, **347**
MPI_MODE_DELETE_ON_CLOSE, **347**
MPI_MODE_EXCL, **347**
MPI_MODE_NOCHECK, **319**, 325, 326
MPI_MODE_NOPRECEDE, 326
MPI_MODE_NOPUT, 326
MPI_MODE_NOSTORE, 326
MPI_MODE_NOSUCCEED, 326
MPI_MODE_RDONLY, **347**
MPI_MODE_RDWR, **347**
MPI_MODE_SEQUENTIAL, **347**, 347, 348
MPI_MODE_UNIQUE_OPEN, **347**
MPI_MODE_WRONLY, **347**
MPI_Mprobe, **147**
MPI_Mrecv, **147**
MPI_Neighbor_allgather, **362**, 363
MPI_Neighbor_allgather_init, **179**
MPI_Neighbor_allgatherv, 363
MPI_Neighbor_allgatherv_init, **179**
MPI_Neighbor_allreduce, 363
MPI_Neighbor_alltoall, 363
MPI_Neighbor_alltoall_init, **179**
MPI_Neighbor_alltoallv, 363
MPI_Neighbor_alltoallv_init, **179**
MPI_Neighbor_alltoallw, 363
MPI_Neighbor_alltoallw_init, **179**
MPI_NO_OP, **75**, 311, 314, 324
MPI_Offset, 193, **350**
MPI_OFFSET_KIND, **193**, 353, 402
MPI_Op, 43, 56, **74**, 74, 75, 78, 306, 314, 392
MPI_Op_commutative, **78**
MPI_Op_create, 57, **75**, 78
MPI_Op_free, **78**
MPI_OP_NULL, 78
MPI_Open_port, **288**, 288
MPI_ORDER_C, **209**
MPI_ORDER_FORTRAN, **209**
MPI_Pack, 225
MPI_Pack_size, 184, **226**
MPI_PACKED, **192**, **193**, 218, 225
MPI_Parrived, **180**
MPI_Pready, **180**, 180
MPI_Pready_list, **180**
MPI_Pready_range, **180**
MPI_Precv_init, **180**
MPI_Probe, 122, **146**, 147, 182
MPI_PROC_NULL, 122, **128**, 128, **130**, 132, 142, 273, 306, 359, 388, 403, 677
MPI_PROD, 46, 57, **75**
MPI_Psend_init, **180**, 180
MPI_Publish_name, **290**
MPI_Put, 305, **306**, 307, 309, 316, 318, 321
MPI_Query_thread, **633**
MPI_Raccumulate, 311
MPI_REAL, 192, **193**
MPI_REAL2, **193**
MPI_REAL4, **193**
MPI_REAL8, **193**
MPI_Recv, **121**, 122–124, 126, 127, 133, 143, 148, 149, 151, 181, 403
MPI_Recv_init, 178, 180
MPI_Reduce, **46**, 48, 69, 92, 310
MPI_Reduce_init, **179**
MPI_Reduce_local, **78**
MPI_Reduce_scatter, 65, **67**, 68, 92, 327
MPI_Reduce_scatter_block, **65**, 66
MPI_Reduce_scatter_block_init, **179**
MPI_Reduce_scatter_init, **179**
MPI_REPLACE, **75**, **310**, 311, 314
MPI_Request, 20, 79, **134**, 145, 177, 179, 180, 350
MPI_Request_free, **145**, 145, 177, 179, 404
MPI_Request_get_status, **146**
MPI_REQUEST_NULL, 144, 145
MPI_Rget, 311
MPI_Rget_accumulate, 311
MPI_ROOT, 273, 403
MPI_Rput, **311**
MPI_Rsend, 181, 400
MPI_Rsend_init, 179

MPI_Scan, 52, **54**, 57
MPI_Scan_init, **179**
MPI_Scatter, 58, **59**, 92
MPI_Scatter_init, **179**
MPI_Scatterv, **70**
MPI_Scatterv_init, **179**
MPI_SEEK_CUR, **347**, 351
MPI_SEEK_END, **347**
MPI_SEEK_SET, **347**, 354
MPI_Send, 80, 89, **119**, 122–127, 133, 143, 148, 181, 400
MPI_Send_init, 178, 180
MPI_Sendrecv, **128**, 128, 130, 132, 133, 143, 677
MPI_Sendrecv_init, **132**
MPI_Sendrecv_replace, **131**
MPI_Sendrecv_replace_init, **132**
MPI_Session_call_errhandler, 292, **390**
MPI_Session_create_errhandler, 292
MPI_Session_finalize, **291**
MPI_Session_get_nth_pset, 292
MPI_Session_get_num_psets, 292
MPI_Session_get_pset_info, 292
MPI_Session_init, **291**
MPI_Session_set_errhandler, **390**
MPI_SHORT, **192**
MPI_SHORT_INT, **75**
MPI_SIGNED_CHAR, **192**
MPI_SIMILAR, **263**
mpi_size, 292
MPI_Sizeof, **196**, 196, 198, 393
MPI_SOURCE, 142, 144, **149**, 149, 152, 153
MPI_Ssend, 125, **180**, 181, 400
MPI_Ssend_init, 179, 181
MPI_Start, **177**, 179, 180
MPI_Startall, **177**, 178, 179
MPI_Status, 122, 128, 134, 135, 138, 142, 145, **148**, 148, 152, 153, 347, 348
MPI_STATUS_IGNORE, 122, 135, 142, 148, 402
MPI_STATUS_SIZE, 402
MPI_STATUSES_IGNORE, 138, 142, 402
MPI_SUBARRAYS_SUPPORTED, 402
MPI_SUBVERSION, 389, 402
MPI_SUCCESS, 19, 185, 390, **391**
MPI_SUM, 46, 57, 69, **75**
MPI_T_BIND_NO_OBJECT, **378**
MPI_T_category_changed, **381**
MPI_T_category_get_categories, **381**
MPI_T_category_get_cvars, **381**
MPI_T_category_get_index, **381**
MPI_T_category_get_info, **381**
MPI_T_category_get_num, **381**
MPI_T_category_get_pvars, **381**
MPI_T_cvar_get_index, **378**
MPI_T_cvar_get_info, **378**, **381**
MPI_T_cvar_handle_free, **379**
MPI_T_cvar_read, **379**
MPI_T_cvar_write, **379**
MPI_T_ENUM_NULL, 378
MPI_T_ERR_INVALID_HANDLE, **380**
MPI_T_ERR_INVALID_INDEX, **378**
MPI_T_ERR_INVALID_NAME, **379**
MPI_T_ERR_PVAR_NO_STARTSTOP, **380**
MPI_T_ERR_PVAR_NO_WRITE, **380**
MPI_T_finalize, **378**
MPI_T_init_thread, **378**
MPI_T_PVAR_ALL_HANDLES, **380**, **381**
MPI_T_PVAR_CLASS_AGGREGATE, **379**
MPI_T_PVAR_CLASS_COUNTER, **379**
MPI_T_PVAR_CLASS_GENERIC, **379**
MPI_T_PVAR_CLASS_HIGHWATERMARK, **379**
MPI_T_PVAR_CLASS_LEVEL, **379**
MPI_T_PVAR_CLASS_LOWWATERMARK, **379**
MPI_T_PVAR_CLASS_PERCENTAGE, **379**
MPI_T_PVAR_CLASS_SIZE, **379**
MPI_T_PVAR_CLASS_STATE, **379**
MPI_T_PVAR_CLASS_TIMER, **379**
MPI_T_pvar_get_index, **379**
MPI_T_pvar_get_info, **379**, 380, **381**
MPI_T_pvar_get_num, **379**
MPI_T_pvar_handle_alloc, **380**
MPI_T_pvar_handle_free, **380**
MPI_T_PVAR_HANDLE_NULL, **380**
MPI_T_pvar_read, **380**
MPI_T_pvar_readreset, 381
MPI_T_pvar_session_create, **380**
MPI_T_pvar_session_free, **380**

MPI_T_PVAR_SESSION_NULL, **380**
MPI_T_pvar_start, 379, **380**
MPI_T_pvar_stop, 379, **380**
MPI_T_pvar_write, 380
MPI_TAG, **150**
MPI_TAG_UB, 120, 150, **388**, 388
MPI_Test, 84, **144**, 144, 145, 182, 350, 404
MPI_Testall, **144**
MPI_Testany, **144**
MPI_Testsome, **144**
MPI_THREAD_FUNNELED, **633**
MPI_THREAD_MULTIPLE, **633**
MPI_THREAD_SERIALIZED, **633**
MPI_THREAD_SINGLE, **633**
mpi_thread_support_level, 292
MPI_Topo_test, **357**
MPI_Txxx, 292
MPI_Type_commit, **200**
MPI_Type_contiguous, 200, **201**
MPI_Type_create_f90_complex, **193**
MPI_Type_create_f90_integer, **193**
MPI_Type_create_f90_real, **193**
MPI_Type_create_hindexed, 212
MPI_Type_create_hindexed_block, **213**
MPI_Type_create_resized, **221**
MPI_Type_create_struct, 200, **214**
MPI_Type_create_subarray, 200, **208**, 209
MPI_Type_extent (deprecated), 219
MPI_Type_free, **200**
MPI_Type_get_extent, **218**, 218, 219
MPI_Type_get_extent_x, **225**
MPI_Type_get_true_extent, **219**
MPI_Type_get_true_extent_x, 220, **225**
MPI_Type_hindexed, 200
MPI_Type_indexed, 200, **210**, 213
MPI_Type_lb (deprecated), 219
MPI_Type_match_size, **197**
MPI_Type_size, **198**
MPI_Type_size_x, 193
MPI_Type_struct, 214
MPI_Type_ub (deprecated), 219
MPI_Type_vector, 200, **203**
MPI_TYPECLASS_COMPLEX, 197
MPI_TYPECLASS_INTEGER, 197
MPI_TYPECLASS_REAL, 197
MPI_UB, 217, 222
MPI_UNDEFINED, 357, 403
MPI_UNEQUAL, **263**
MPI_UNIVERSE_SIZE, **285**, 388
MPI_Unpack, 225
MPI_Unpublish_name, **290**, 390
MPI_UNSIGNED, **192**
MPI_UNSIGNED_CHAR, **192**
MPI_UNSIGNED_LONG, **192**
MPI_UNSIGNED_SHORT, **192**
MPI_UNWEIGHTED, 362, 403
MPI_VERSION, 389, 402
MPI_Wait, 79, **134**, 137, 142, 144, 148, 304, 350, 404
MPI_Wait..., 137, 148, 149
MPI_Waitall, **137**, 137, 142, 144, 150, 178, 300
MPI_Waitany, **139**, 141, 142, 144
MPI_Waitsome, 142, 144
MPI_WEIGHTS_EMPTY, **362**, 403
MPI_Win, 195, **300**, 367
MPI_Win_allocate, **302**, 303, 322, 325
MPI_Win_allocate_shared, 302, 322, 325, 368, **369**, 369
MPI_Win_attach, **322**
MPI_Win_call_errhandler, **390**
MPI_Win_complete, 312
MPI_Win_create, 196, **301**, 302, 303, 318, 322, 325
MPI_Win_create_dynamic, 302, **322**, 325
MPI_Win_detach, **323**
MPI_Win_fence, **304**, 318, 320, 325, 326, 678
MPI_WIN_FLAVOR_ALLOCATE, 325
MPI_WIN_FLAVOR_CREATE, 325
MPI_WIN_FLAVOR_DYNAMIC, 325
MPI_WIN_FLAVOR_SHARED, 325
MPI_Win_flush, **321**
MPI_Win_flush..., 311
MPI_Win_flush_all, 321
MPI_Win_flush_local, **320**, 321
MPI_Win_flush_local_all, 321
MPI_Win_get_info, **386**
MPI_Win_lock, 314, **317**, 318, 326
MPI_Win_lock_all, 318, **319**, 321, 326

MPI_Win_post, 312, 325, 326
MPI_WIN_SEPARATE, 325
MPI_Win_set_errhandler, **390**
MPI_Win_set_info, **386**
MPI_Win_shared_query, 368, **370**
MPI_Win_start, 312, 325, 326
MPI_Win_sync, **321**
MPI_WIN_UNIFIED, 325
MPI_Win_unlock, **318**, 320
MPI_Win_unlock_all, **318**, **319**, 320
MPI_Win_wait, 312
MPI_Wtick, **394**, 396
MPI_Wtime, 122, **394**, 588
MPI_WTIME_IS_GLOBAL, 388, 395

nb_proc, 387
no_locks, 324
num_io_nodes, 387

PMPI_..., 396

same_op, 324
same_op_no_op, 324
striping_factor, 387
striping_unit, 387

testany, 141

vector_layout, **203**

wtime, 394

54.1 From the standard document

This is an automatically generated list of every function, type, and constant in the MPI standard document. Where these appear in this book, a page reference is given.

54.1.1 List of all functions

- MPI_Abort 27
- MPI_Accumulate 310
- MPI_Address ??
- MPI_Add_error_class 392
- MPI_Add_error_string 392
- MPI_Aint_add 196
- MPI_Aint_diff 196
- MPI_Allgather 62
- MPI_Allgatherv 70
- MPI_Allgatherv_init 179
- MPI_Allgather_init 179
- MPI_Alloc_mem 303
- MPI_Alloc_mem_cptr ??
- MPI_Allreduce 41
- MPI_Allreduce_init 179
- MPI_Alltoall 62
- MPI_Alltoallv 64
- MPI_Alltoallv_init 179
- MPI_Alltoallw ??
- MPI_Alltoallw_init 179
- MPI_Alltoall_init 179
- MPI_Attr_delete ??
- MPI_Attr_get 387
- MPI_Attr_put ??
- MPI_Accumulate 310
- MPI_Barrier 69
- MPI_Barrier_init 179
- MPI_Bcast 50
- MPI_Bcast_init 179
- MPI_Bsend 182
- MPI_Bsend_init 185
- MPI_Buffer_attach 184
- MPI_Buffer_detach 184
- MPI_Cancel 403
- MPI_Cartdim_get ??
- MPI_Cart_coords 358
- MPI_Cart_create 358
- MPI_Cart_get ??
- MPI_Cart_map ??
- MPI_Cart_rank 358
- MPI_Cart_shift ??
- MPI_Cart_sub ??
- MPI_Close_port 288
- MPI_Comm_accept 288
- MPI_Comm_call_errhandler ??
- MPI_Comm_compare 262
- MPI_Comm_connect 289
- MPI_Comm_create 269
- MPI_Comm_create_errhandler 392
- MPI_Comm_create_from_group ??
- MPI_Comm_create_group 269
- MPI_Comm_create_keyval ??
- MPI_Comm_delete_attr ??
- MPI_Comm_disconnect 289
- MPI_Comm_dup 261
- MPI_Comm_dup_fn ??
- MPI_Comm_dup_with_info 387
- MPI_Comm_free 265
- MPI_Comm_free_keyval ??
- MPI_Comm_get_attr 387
- MPI_Comm_get_errhandler 390
- MPI_Comm_get_info 386
- MPI_Comm_get_name ??
- MPI_Comm_get_parent 274
- MPI_Comm_group 269
- MPI_Comm_idup 261
- MPI_Comm_idup_with_info 261
- MPI_Comm_join 290
- MPI_Comm_null_copy_fn ??
- MPI_Comm_null_delete_fn ??
- MPI_Comm_rank 31
- MPI_Comm_remote_size 274
- MPI_Comm_set_attr ??
- MPI_Comm_set_errhandler 390
- MPI_Comm_set_info 386
- MPI_Comm_set_name ??
- MPI_Comm_size 31
- MPI_Comm_spawn 285
- MPI_Comm_spawn_multiple ??
- MPI_Comm_split 266
- MPI_Comm_split_type 367
- MPI_Comm_test_inter 274
- MPI_Compare_and_swap 315
- MPI_Conversion_fn_null ??
- MPI_Conversion_fn_null_c ??
- MPI_Compare_and_swap 315
- MPI_Dims_create ??
- MPI_Dist_graph_create 360
- MPI_Dist_graph_create_adjacent ??
- MPI_Dist_graph_neighbors 363
- MPI_Dist_graph_neighbors_count 363
- MPI_Dup_fn ??
- MPI_Errhandler_create 390
- MPI_Errhandler_free 390
- MPI_Errhandler_get ??
- MPI_Errhandler_set ??
- MPI_Error_class ??
- MPI_Error_string 391
- MPI_Event_callback_get_info ??
- MPI_Event_handle_get_info ??
- MPI_Exscan 56
- MPI_Exscan_init 179
- MPI_Fetch_and_op 313
- MPI_File_call_errhandler 390
- MPI_File_close 346
- MPI_File_create_errhandler ??
- MPI_File_delete 347
- MPI_File_get_amode ??
- MPI_File_get_atomicity ??
- MPI_File_get_byte_offset ??
- MPI_File_get_errhandler ??
- MPI_File_get_group ??
- MPI_File_get_info ??
- MPI_File_get_position ??
- MPI_File_get_position_shared ??
- MPI_File_get_size 353
- MPI_File_get_type_extent ??
- MPI_File_get_view 353
- MPI_File_iread 348
- MPI_File_iread_all 348
- MPI_File_iread_at ??
- MPI_File_iread_at_all 350
- MPI_File_iread_shared 353
- MPI_File_iwrite 348
- MPI_File_iwrite_all 348
- MPI_File_iwrite_at ??
- MPI_File_iwrite_at_all 350
- MPI_File_iwrite_shared 353
- MPI_File_open 346
- MPI_File_preallocate 353
- MPI_File_read 348
- MPI_File_read_all 348
- MPI_File_read_all_begin 350
- MPI_File_read_all_end 350
- MPI_File_read_at 348

- MPI_File_read_at_all 348
- MPI_File_read_at_all_begin ??
- MPI_File_read_at_all_end ??
- MPI_File_read_ordered 353
- MPI_File_read_ordered_begin ??
- MPI_File_read_ordered_end ??
- MPI_File_read_shared 353
- MPI_File_seek 350
- MPI_File_seek_shared 353
- MPI_File_set_atomicity 354
- MPI_File_set_errhandler 390
- MPI_File_set_info ??
- MPI_File_set_size 353
- MPI_File_set_view 352
- MPI_File_sync 347
- MPI_File_write 347
- MPI_File_write_all 348
- MPI_File_write_all_begin 350
- MPI_File_write_all_end 350
- MPI_File_write_at 350
- MPI_File_write_at_all 348
- MPI_File_write_at_all_begin ??
- MPI_File_write_at_all_end ??
- MPI_File_write_ordered 353
- MPI_File_write_ordered_begin ??
- MPI_File_write_ordered_end ??
- MPI_File_write_shared 353
- MPI_Finalize 26
- MPI_Finalized 28
- MPI_Free_mem ??
- MPI_F_sync_reg ??
- MPI_Gather 59
- MPI_Gatherv 70
- MPI_Gatherv_init 179
- MPI_Gather_init 179
- MPI_Get 308
- MPI_Get_accumulate 313
- MPI_Get_address 213
- MPI_Get_count 151
- MPI_Get_elements 151
- MPI_Get_elements_x 224
- MPI_Get_library_version ??
- MPI_Get_processor_name 29
- MPI_Get_version 389
- MPI_Graphdims_get 364
- MPI_Graph_create 364
- MPI_Graph_get 364
- MPI_Graph_map ??
- MPI_Graph_neighbors 364
- MPI_Graph_neighbors_count 364
- MPI_Grequest_complete ??
- MPI_Grequest_start ??
- MPI_Group_compare ??
- MPI_Group_excl 269
- MPI_Group_free ??
- MPI_Group_from_session_pset ??
- MPI_Group_incl 269
- MPI_Group_range_excl ??
- MPI_Group_range_incl ??
- MPI_Group_rank ??
- MPI_Group_size ??
- MPI_Group_translate_ranks ??
- MPI_Iallgather 80
- MPI_Iallgatherv 80
- MPI_Iallreduce 80
- MPI_Ialltoall 81
- MPI_Ialltoallv 81
- MPI_Ialltoallw 81
- MPI_Ibarrier 83
- MPI_Ibcast 81
- MPI_Ibsend ??
- MPI_Iexscan 81
- MPI_Igather 80
- MPI_Igatherv 80
- MPI_Improbe ??
- MPI_Imrecv ??
- MPI_Ineighbor_allgather ??
- MPI_Ineighbor_allgatherv ??
- MPI_Ineighbor_alltoall ??
- MPI_Ineighbor_alltoallv ??
- MPI_Ineighbor_alltoallw ??
- MPI_Info_create 383
- MPI_Info_create_env ??
- MPI_Info_delete 383
- MPI_Info_dup 383
- MPI_Info_free 383
- MPI_Info_get 383
- MPI_Info_get_nkeys 383
- MPI_Info_get_nthkey 383
- MPI_Info_get_string 383
- MPI_Info_get_valuelen ??
- MPI_Info_set 383
- MPI_Init 26
- MPI_Initialized 28
- MPI_Init_thread 633
- MPI_Intercomm_create 272
- MPI_Intercomm_create_from_groups ??
- MPI_Intercomm_merge 274
- MPI_Iprobe ??
- MPI_Irecv 134
- MPI_Ireduce 80
- MPI_Ireduce_scatter 80
- MPI_Ireduce_scatter_block 80
- MPI_Iscan 81
- MPI_Iscatter 80
- MPI_Iscatterv 80
- MPI_Isend 134
- MPI_Isendrecv 132
- MPI_Isendrecv_replace 132
- MPI_Is_thread_main 633
- MPI_Keyval_create ??
- MPI_Keyval_free ??
- MPI_Lookup_name ??
- MPI_Mprobe 147
- MPI_Mrecv 147
- MPI_Neighbor_allgather 362
- MPI_Neighbor_allgatherv ??
- MPI_Neighbor_allgatherv_init 179
- MPI_Neighbor_alltoall ??
- MPI_Neighbor_alltoallv ??
- MPI_Neighbor_alltoallv_init 179
- MPI_Neighbor_alltoallw ??
- MPI_Neighbor_alltoallw_init 179
- MPI_Null_copy_fn ??
- MPI_Null_delete_fn ??
- MPI_Open_port 288
- MPI_Op_commutative 78
- MPI_Op_create 75
- MPI_Op_create_c ??
- MPI_Pack ??
- MPI_Pack_external ??
- MPI_Pack_size 226
- MPI_Parrived 180
- MPI_Pcontrol ??
- MPI_Pready 180
- MPI_Pready_list 180
- MPI_Pready_range 180
- MPI_Precv_init 180
- MPI_Probe 146
- MPI_Psend_init 180
- MPI_Publish_name 290
- MPI_Put 306
- MPI_Put 306
- MPI_Query_thread 633
- MPI_Raccumulate ??
- MPI_Recv 121
- MPI_Recv_init ??
- MPI_Reduce 46
- MPI_Reduce_init 179
- MPI_Reduce_local 78
- MPI_Reduce_scatter 67
- MPI_Reduce_scatter_block 65
- MPI_Reduce_scatter_block_init 179
- MPI_Reduce_scatter_init 179
- MPI_Register_datarep ??
- MPI_Request_free 145
- MPI_Request_get_status 146
- MPI_Rget ??
- MPI_Rget_accumulate ??

- MPI_Rput 311
- MPI_Rsend ??
- MPI_Register_datarep_c ??
- MPI_Scan 54
- MPI_Scan_init 179
- MPI_Scatter 59
- MPI_Scatterv 70
- MPI_Scatterv_init 179
- MPI_Scatter_init 179
- MPI_Send 119
- MPI_Sendrecv 128
- MPI_Send_init ??
- MPI_Session_call_errhandler 390
- MPI_Session_create_errhandler ??
- MPI_Session_create_keyval ??
- MPI_Session_finalize 291
- MPI_Session_get_info ??
- MPI_Session_get_nth_pset ??
- MPI_Session_get_num_psets ??
- MPI_Session_init 291
- MPI_Session_null_delete_fn ??
- MPI_Sizeof 196
- MPI_Start 177
- MPI_Startall 177
- MPI_Status_set_cancelled ??
- MPI_Status_set_elements ??
- MPI_Status_set_elements_x ??
- MPI_Send 119
- MPI_Test 144
- MPI_Testall 144
- MPI_Testany 144
- MPI_Testsome 144
- MPI_Test_cancelled ??
- MPI_Topo_test 357
- MPI_Type_commit 200
- MPI_Type_contiguous 201
- MPI_Type_create_darray ??
- MPI_Type_create_hindexed ??
- MPI_Type_create_hindexed_block 213
- MPI_Type_create_hvector ??
- MPI_Type_create_indexed_block ??
- MPI_Type_create_keyval ??
- MPI_Type_create_resized 221
- MPI_Type_create_struct 214
- MPI_Type_create_subarray 208
- MPI_Type_delete_attr ??
- MPI_Type_dup ??
- MPI_Type_dup_fn ??
- MPI_Type_extent ??
- MPI_Type_free 200
- MPI_Type_get_attr ??
- MPI_Type_get_contents ??
- MPI_Type_get_envelope ??
- MPI_Type_get_extent 218
- MPI_Type_get_extent_x 225
- MPI_Type_get_name ??
- MPI_Type_get_true_extent 219
- MPI_Type_get_true_extent_x 225
- MPI_Type_hindexed ??
- MPI_Type_hvector ??
- MPI_Type_indexed 210
- MPI_Type_lb ??
- MPI_Type_match_size 197
- MPI_Type_null_copy_fn ??
- MPI_Type_null_delete_fn ??
- MPI_Type_set_attr ??
- MPI_Type_set_name ??
- MPI_Type_size 198
- MPI_Type_size_x ??
- MPI_Type_struct ??
- MPI_Type_ub ??
- MPI_Type_vector 203
- MPI_T_category_changed 381
- MPI_T_category_get_categories 381
- MPI_T_category_get_cvars 381
- MPI_T_category_get_events ??
- MPI_T_category_get_index 381
- MPI_T_category_get_info 381
- MPI_T_category_get_num_events ??
- MPI_T_category_get_pvars 381
- MPI_T_cvar_get_index 378
- MPI_T_cvar_get_info 381
- MPI_T_cvar_get_num ??
- MPI_T_cvar_handle_alloc ??
- MPI_T_cvar_handle_free 379
- MPI_T_enum_get_info ??
- MPI_T_enum_get_item ??
- MPI_T_event_callback_get_info ??
- MPI_T_event_callback_set_info ??
- MPI_T_event_copy ??
- MPI_T_event_get_index ??
- MPI_T_event_get_info ??
- MPI_T_event_get_source ??
- MPI_T_event_get_timestamp ??
- MPI_T_event_handle_alloc ??
- MPI_T_event_handle_free ??
- MPI_T_event_handle_get_info ??
- MPI_T_event_handle_set_info ??
- MPI_T_event_read ??
- MPI_T_event_register_callback ??
- MPI_T_event_set_dropped_handler ??
- MPI_T_finalize 378
- MPI_T_init_thread 378
- MPI_T_pvar_get_index 379
- MPI_T_pvar_get_info 381
- MPI_T_pvar_get_num 379
- MPI_T_pvar_handle_alloc 380
- MPI_T_pvar_handle_free 380
- MPI_T_pvar_read 380
- MPI_T_pvar_readreset ??
- MPI_T_pvar_reset ??
- MPI_T_pvar_start 380
- MPI_T_pvar_stop 380
- MPI_T_pvar_write ??
- MPI_T_source_get_info ??
- MPI_T_source_get_num ??
- MPI_T_source_get_timestamp ??
- MPI_Unpack ??
- MPI_Unpack_external ??
- MPI_Unpublish_name 290
- MPI_Wait 134
- MPI_Waitall 137
- MPI_Waitany 139
- MPI_Waitsome ??
- MPI_Win_allocate 302
- MPI_Win_allocate_cptr ??
- MPI_Win_allocate_shared 369
- MPI_Win_allocate_shared_cptr ??
- MPI_Win_attach 322
- MPI_Win_call_errhandler 390
- MPI_Win_complete ??
- MPI_Win_create 301
- MPI_Win_create_dynamic 322
- MPI_Win_create_errhandler ??
- MPI_Win_create_keyval ??
- MPI_Win_detach 323
- MPI_Win_dup_fn ??
- MPI_Win_fence 304
- MPI_Win_flush 321
- MPI_Win_flush_all ??
- MPI_Win_flush_local 320
- MPI_Win_flush_local_all ??
- MPI_Win_free ??
- MPI_Win_get_attr ??
- MPI_Win_get_group ??
- MPI_Win_get_info 386
- MPI_Win_lock 317
- MPI_Win_lock_all 319
- MPI_Win_null_copy_fn ??
- MPI_Win_null_delete_fn ??
- MPI_Win_post ??
- MPI_Win_set_attr ??
- MPI_Win_set_info 386
- MPI_Win_shared_query 370
- MPI_Win_shared_query_cptr ??
- MPI_Win_start ??
- MPI_Win_sync 321
- MPI_Win_test ??
- MPI_Win_unlock 318

- MPI_Win_unlock_all 319
- MPI_Win_wait ??
- MPI_Wtick 394
- MPI_Wtime 394
- PMPI_ ??
- PMPI_Aint_add ??
- PMPI_Aint_diff ??
- PMPI_Isend ??
- PMPI_T_event_copy ??
- PMPI_T_event_get_source ??
- PMPI_T_event_get_timestamp ??
- PMPI_T_event_read ??
- PMPI_T_pvar_read ??
- PMPI_T_pvar_readreset ??
- PMPI_T_pvar_reset ??
- PMPI_T_pvar_start ??
- PMPI_T_pvar_stop ??
- PMPI_T_pvar_write ??
- PMPI_T_source_get_timestamp ??
- PMPI_Wtick ??
- PMPI_Wtime ??

54.1.2 List of all ctypes

- COMPLEX ??
- Datatype ??
- Group ??
- INTEGER ??
- LOGICAL ??
- MPI_Aint 195
- MPI_Comm_copy_attr_function ??
- MPI_Comm_delete_attr_function ??
- MPI_Comm_errhandler_fn ??
- MPI_Comm_errhandler_function ??
- MPI_Copy_function ??
- MPI_Count 224
- MPI_Datarep_conversion_function ??
- MPI_Datarep_conversion_function_c ??
- MPI_Delete_function ??
- MPI_File_errhandler_fn ??
- MPI_File_errhandler_function ??
- MPI_Offset 350
- MPI_Session_delete_attr_function ??
- MPI_Type_copy_attr_function ??
- MPI_Type_delete_attr_function ??
- MPI_Win_copy_attr_function ??
- MPI_Win_delete_attr_function ??
- MPI_Win_errhandler_fn ??
- MPI_Win_errhandler_function ??
- REAL ??
- Status ??
- _Bool ??
- bool ??
- char ??
- class ??
- double ??
- enum ??
- float ??
- int ??
- long ??
- short ??
- wchar_t ??

54.1.3 List of all ftypes

- ALLOCATABLE ??
- ASYNCHRONOUS ??
- BLOCK ??
- CHARACTER ??
- COMMON ??
- COMM_COPY_ATTR_FUNCTION ??
- COMM_DELETE_ATTR_FUNCTION ??
- COMPLEX ??
- CONTAINS ??
- CONTIGUOUS ??
- COPY_FUNCTION ??
- C_F_POINTER ??
- C_PTR ??
- DATAREP_CONVERSION_FUNCTION ??
- DELETE_FUNCTION ??
- EXTERNAL ??
- FUNCTION ??
- IN ??
- INCLUDE ??
- INOUT ??
- INTEGER ??
- INTENT ??
- INTERFACE ??
- ISO_C_BINDING ??
- ISO_FORTRAN_ENV ??
- KIND ??
- LOGICAL ??
- MODULE ??
- MPI_Send 119
- MPI_Status 148
- MPI_User_function ??
- MPI_Waitall 137
- OPTIONAL ??
- OUT ??
- POINTER ??
- PROCEDURE ??
- REAL ??
- SEQUENCE ??
- SESSION_DELETE_ATTR_FUNCTION ??
- TARGET ??
- TYPE ??
- TYPE_COPY_ATTR_FUNCTION ??
- TYPE_DELETE_ATTR_FUNCTION ??
- USER_FUNCTION ??
- VOLATILE ??
- WIN_COPY_ATTR_FUNCTION ??
- WIN_DELETE_ATTR_FUNCTION ??
- base ??
- foo ??
- int ??
- separated_sections ??

54.1.4 List of all constants

- MPI_ADDRESS_KIND 307
- MPI_ANY_SOURCE 122
- MPI_ANY_TAG 122
- MPI_APPNUM 388
- MPI_ARGVS_NULL ??
- MPI_ARGV_NULL ??
- MPI_ASYNC_PROTECTS_NONBLOCKING ??
- MPI_Aint 195
- MPI_BAND 75

- MPI_BOR 75
- MPI_BOTTOM ??
- MPI_BSEND_OVERHEAD 184
- MPI_BXOR 75
- MPI_CART 357
- MPI_COMBINER_CONTIGUOUS ??
- MPI_COMBINER_DARRAY ??
- MPI_COMBINER_DUP ??
- MPI_COMBINER_HINDEXED ??
- MPI_COMBINER_HINDEXED_BLOCK ??
- MPI_COMBINER_HINDEXED_INTEGER ??
- MPI_COMBINER_HVECTOR ??
- MPI_COMBINER_HVECTOR_INTEGER ??
- MPI_COMBINER_INDEXED ??
- MPI_COMBINER_INDEXED_BLOCK ??
- MPI_COMBINER_NAMED ??
- MPI_COMBINER_RESIZED ??
- MPI_COMBINER_STRUCT ??
- MPI_COMBINER_STRUCT_INTEGER ??
- MPI_COMBINER_SUBARRAY ??
- MPI_COMBINER_VECTOR ??
- MPI_COMM_NULL ??
- MPI_COMM_SELF ??
- MPI_COMM_TYPE_HW_GUIDED ??
- MPI_COMM_TYPE_HW_UNGUIDED ??
- MPI_COMM_TYPE_SHARED ??
- MPI_COMM_WORLD ??
- MPI_CONGRUENT 263
- MPI_COUNT_KIND 193
- MPI_Comm 260
- MPI_Count 224
- MPI_DATATYPE_NULL 200
- MPI_DISPLACEMENT_CURRENT 352
- MPI_DISTRIBUTE_BLOCK ??
- MPI_DISTRIBUTE_CYCLIC ??
- MPI_DISTRIBUTE_DFLT_DARG ??
- MPI_DISTRIBUTE_NONE ??
- MPI_DIST_GRAPH 357
- MPI_Datatype 200
- MPI_ERRCODES_IGNORE 285
- MPI_ERRHANDLER_NULL ??
- MPI_ERROR 150
- MPI_ERRORS_ABORT 391
- MPI_ERRORS_ARE_FATAL 391
- MPI_ERRORS_RETURN ??
- MPI_ERR_ARG 390
- MPI_ERR_BUFFER 390
- MPI_ERR_COMM 390
- MPI_ERR_COUNT ??
- MPI_ERR_DIMS ??
- MPI_ERR_GROUP ??
- MPI_ERR_INTERN 390
- MPI_ERR_IN_STATUS 390
- MPI_ERR_LASTCODE 390
- MPI_ERR_OP ??
- MPI_ERR_OTHER 390
- MPI_ERR_PENDING ??
- MPI_ERR_RANK ??
- MPI_ERR_REQUEST ??
- MPI_ERR_RMA_ATTACH ??
- MPI_ERR_RMA_FLAVOR ??
- MPI_ERR_RMA_RANGE ??
- MPI_ERR_RMA_SHARED ??
- MPI_ERR_ROOT ??
- MPI_ERR_SESSION ??
- MPI_ERR_TAG ??
- MPI_ERR_TOPOLOGY ??
- MPI_ERR_TRUNCATE ??
- MPI_ERR_TYPE ??
- MPI_ERR_UNKNOWN ??
- MPI_ERR_VALUE_TOO_LARGE ??
- MPI_Errhandler 390
- MPI_FILE_NULL ??
- MPI_FLOAT_INT 75
- MPI_F_ERROR ??
- MPI_F_SOURCE ??
- MPI_F_STATUSES_IGNORE ??
- MPI_F_STATUS_IGNORE ??
- MPI_F_STATUS_SIZE ??
- MPI_F_TAG ??
- MPI_File 346
- MPI_Fint ??
- MPI_GRAPH 357
- MPI_GROUP_EMPTY ??
- MPI_GROUP_NULL ??
- MPI_Group 269
- MPI_HOST ??
- MPI_IDENT ??
- MPI_INFO ??
- MPI_INFO_ENV 383
- MPI_INFO_NULL ??
- MPI_INTEGER_KIND ??
- MPI_IN_PLACE ??
- MPI_IO ??
- MPI_Info 383
- MPI_KEYVAL_INVALID ??
- MPI_LAND 75
- MPI_LASTUSEDCODE 392
- MPI_LB ??
- MPI_LOCK_EXCLUSIVE ??
- MPI_LOCK_SHARED ??
- MPI_LOR 75
- MPI_LXOR 75
- MPI_MAX 75
- MPI_MAXLOC 75
- MPI_MAX_DATAREP_STRING ??
- MPI_MAX_ERROR_STRING 391
- MPI_MAX_FROM_GROUP_STRINGTAG ??
- MPI_MAX_FROM_GROUP_TAG ??
- MPI_MAX_INFO_KEY 383
- MPI_MAX_INFO_VAL ??
- MPI_MAX_LIBRARY_VERSION_STRING ??
- MPI_MAX_OBJECT_NAME ??
- MPI_MAX_PORT_NAME 288
- MPI_MAX_PROCESSOR_NAME 389
- MPI_MAX_PSET_NAME_LEN ??
- MPI_MAX_STRINGTAG_LEN ??
- MPI_MESSAGE_NO_PROC ??
- MPI_MESSAGE_NULL ??
- MPI_MIN 75
- MPI_MINLOC 75
- MPI_MODE_APPEND 347
- MPI_MODE_CREATE 347
- MPI_MODE_DELETE_ON_CLOSE 347
- MPI_MODE_EXCL 347
- MPI_MODE_NOCHECK 319
- MPI_MODE_NOPRECEDE ??
- MPI_MODE_NOPUT ??
- MPI_MODE_NOSTORE ??
- MPI_MODE_NOSUCCEED ??
- MPI_MODE_RDONLY 347
- MPI_MODE_RDWR 347
- MPI_MODE_SEQUENTIAL 347
- MPI_MODE_UNIQUE_OPEN 347
- MPI_MODE_WRONLY 347
- MPI_Message 147
- MPI_NO_OP 75
- MPI_OFFSET_KIND 193
- MPI_OP_NULL ??
- MPI_ORDER_C 209
- MPI_ORDER_FORTRAN 209
- MPI_Offset 350
- MPI_Op 74
- MPI_PROC_NULL 130
- MPI_PROD 75
- MPI_REPLACE 310
- MPI_REQUEST_NULL ??
- MPI_ROOT ??
- MPI_Request 134
- MPI_SEEK_CUR 347
- MPI_SEEK_END 347
- MPI_SEEK_SET 347
- MPI_SESSION_NULL ??
- MPI_SHORT_INT 75
- MPI_SIMILAR 263
- MPI_SOURCE 149
- MPI_STATUSES_IGNORE ??
- MPI_STATUS_IGNORE ??
- MPI_STATUS_SIZE ??

- MPI_SUBARRAYS_SUPPORTED ??
- MPI_SUBVERSION ??
- MPI_SUCCESS 391
- MPI_SUM 75
- MPI_Session ??
- MPI_Status 148
- MPI_TAG 150
- MPI_TAG_UB 388
- MPI_THREAD_FUNNELED 633
- MPI_THREAD_MULTIPLE 633
- MPI_THREAD_SERIALIZED 633
- MPI_THREAD_SINGLE 633
- MPI_TYPECLASS_COMPLEX ??
- MPI_TYPECLASS_INTEGER ??
- MPI_TYPECLASS_REAL ??
- MPI_T_BIND_MPI_COMM ??
- MPI_T_BIND_MPI_DATATYPE ??
- MPI_T_BIND_MPI_ERRHANDLER ??
- MPI_T_BIND_MPI_FILE ??
- MPI_T_BIND_MPI_GROUP ??
- MPI_T_BIND_MPI_INFO ??
- MPI_T_BIND_MPI_MESSAGE ??
- MPI_T_BIND_MPI_OP ??
- MPI_T_BIND_MPI_REQUEST ??
- MPI_T_BIND_MPI_SESSION ??
- MPI_T_BIND_MPI_WIN ??
- MPI_T_BIND_NO_OBJECT 378
- MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE ??
- MPI_T_CB_REQUIRE_MPI_RESTRICTED ??
- MPI_T_CB_REQUIRE_NONE ??
- MPI_T_CB_REQUIRE_THREAD_SAFE ??
- MPI_T_CVAR_HANDLE_NULL ??
- MPI_T_ENUM_NULL ??
- MPI_T_ERR_CANNOT_INIT ??
- MPI_T_ERR_CVAR_SET_NEVER ??
- MPI_T_ERR_CVAR_SET_NOT_NOW ??
- MPI_T_ERR_INVALID ??
- MPI_T_ERR_INVALID_HANDLE 380
- MPI_T_ERR_INVALID_INDEX 378
- MPI_T_ERR_INVALID_ITEM ??
- MPI_T_ERR_INVALID_NAME 379
- MPI_T_ERR_INVALID_SESSION ??
- MPI_T_ERR_MEMORY ??
- MPI_T_ERR_NOT_ACCESSIBLE ??
- MPI_T_ERR_NOT_INITIALIZED ??
- MPI_T_ERR_NOT_SUPPORTED ??
- MPI_T_ERR_OUT_OF_HANDLES ??
- MPI_T_ERR_OUT_OF_SESSIONS ??
- MPI_T_ERR_PVAR_NO_ATOMIC ??
- MPI_T_ERR_PVAR_NO_STARTSTOP 380
- MPI_T_ERR_PVAR_NO_WRITE 380
- MPI_T_PVAR_ALL_HANDLES 381
- MPI_T_PVAR_CLASS_AGGREGATE 379
- MPI_T_PVAR_CLASS_COUNTER 379
- MPI_T_PVAR_CLASS_GENERIC 379
- MPI_T_PVAR_CLASS_HIGHWATERMARK 379
- MPI_T_PVAR_CLASS_LEVEL 379
- MPI_T_PVAR_CLASS_LOWWATERMARK 379
- MPI_T_PVAR_CLASS_PERCENTAGE 379
- MPI_T_PVAR_CLASS_SIZE 379
- MPI_T_PVAR_CLASS_STATE 379
- MPI_T_PVAR_CLASS_TIMER 379
- MPI_T_PVAR_HANDLE_NULL 380
- MPI_T_PVAR_SESSION_NULL 380
- MPI_T_SCOPE_ALL ??
- MPI_T_SCOPE_ALL_EQ ??
- MPI_T_SCOPE_CONSTANT ??
- MPI_T_SCOPE_GROUP ??
- MPI_T_SCOPE_GROUP_EQ ??
- MPI_T_SCOPE_LOCAL ??
- MPI_T_SCOPE_READONLY ??
- MPI_T_SOURCE_ORDERED ??
- MPI_T_SOURCE_UNORDERED ??
- MPI_T_VERBOSITY_MPIDEV_ALL ??
- MPI_T_VERBOSITY_MPIDEV_BASIC ??
- MPI_T_VERBOSITY_MPIDEV_DETAIL ??
- MPI_T_VERBOSITY_TUNER_ALL ??
- MPI_T_VERBOSITY_TUNER_BASIC ??
- MPI_T_VERBOSITY_TUNER_DETAIL ??
- MPI_T_VERBOSITY_USER_ALL ??
- MPI_T_VERBOSITY_USER_BASIC ??
- MPI_T_VERBOSITY_USER_DETAIL ??
- MPI_T_cb_safety ??
- MPI_T_cvar_handle ??
- MPI_T_enum ??
- MPI_T_event_instance ??
- MPI_T_event_registration ??
- MPI_T_pvar_handle ??
- MPI_T_pvar_session ??
- MPI_T_source_order ??
- MPI_UB ??
- MPI_UNDEFINED ??
- MPI_UNEQUAL 263
- MPI_UNIVERSE_SIZE 285
- MPI_UNWEIGHTED ??
- MPI_VAL ??
- MPI_VERSION ??
- MPI_WEIGHTS_EMPTY 362
- MPI_WIN_BASE ??
- MPI_WIN_CREATE_FLAVOR ??
- MPI_WIN_DISP_UNIT ??
- MPI_WIN_FLAVOR_ALLOCATE ??
- MPI_WIN_FLAVOR_CREATE ??
- MPI_WIN_FLAVOR_DYNAMIC ??
- MPI_WIN_FLAVOR_SHARED ??
- MPI_WIN_MODEL ??
- MPI_WIN_NULL ??
- MPI_WIN_SEPARATE ??
- MPI_WIN_SIZE ??
- MPI_WIN_UNIFIED ??
- MPI_WTIME_IS_GLOBAL ??
- MPI_Win 300

54.1.5 List of all callbacks

- COMM_COPY_ATTR_FUNCTION ??
- COMM_DELETE_ATTR_FUNCTION ??
- COPY_FUNCTION ??
- DELETE_FUNCTION ??
- MPI_Comm_copy_attr_function ??
- MPI_Comm_delete_attr_function ??
- MPI_Comm_errhandler_function ??
- MPI_Copy_function ??
- MPI_Datarep_conversion_function ??
- MPI_Datarep_conversion_function_c ??
- MPI_Delete_function ??
- MPI_File_errhandler_function ??
- MPI_Handler_function ??
- MPI_Session_errhandler_function ??
- MPI_Type_delete_attr_function ??
- MPI_User_function ??
- MPI_User_function_c ??
- MPI_Win_errhandler_function ??

54.2 MPL commands and topics

absolute, 217, 218
any-source, 122
any_source, 122

blocking-send-and-receive, 120
broadcast, 52
bsend_buffer, **184**
bsend_overhead, 185
bsend_size, **184**
buffer-attach-and-detach, 185
buffered-send, 184

comm_world, 30
communicator, 32, 268
communicator-copying, 30
communicator-duplication, 261
communicator-errhandler, 392
communicator-passing, 31
communicator-splitting, 267
contiguous-composing, 203
contiguous-type, 202
contiguous_layout, 45, **203**

data-types, 192
derived-type-handling, 200
displacement-handling-in-gatherv, 72
dist_graph_communicator, **361**
distributed-graph-creation, 361

environment, 261
extent-resizing, 221

gather, **59**
gather-on-nonroot, 60
gather-scatter, 59

header-file, 26
heterogeneous_layout, 217

indexed-block-type, 212
indexed-type, 212
indexed_block_layout, **212**
indexed_layout, **212**
init,-finalize, 27
irequest, 81, **137**, 137, 177
irequest_pool, **140**, 177

iterator-buffers, 45, 206
iterator-layout, 206
iterator_layout, **206**

lambda-operator, 77
layout, 72, 212
layouts, **72**, **212**
layouts-for-gatherv, 212

make_absolute, 218
message-tag, 150

non-root-variants, 50
nonblocking-collectives, 81
notes-format, 17

other-types, 191

persistent-requests, 177
predefined-communicators, 261
prequest, **177**
prequest_pool, **177**
processor-name, 29
processor_name, 29

rank, **32**
rank-and-size, 32
receive-count, 151
reduce-in-place, 49
reduction-operator, 42
request-pools, 140
requests-from-nonblocking-calls, 135

scalar-buffers, 45
scan-operations, 56
scatter, **59**
send-recv-call, 128
sending-arrays, 120
size, **32**
split, **268**
split-by-shared-memory, 368
split_shared, **368**
status, 137, 150
status-object, 148
status-source-querying, 150
strided_vector_layout, **203**, **205**
struct-type-general, 217

struct-type-scalar, 217
subarray-layout, 208
`subarray_layout`, **208**
`submit`, 609

`tag`, 150
`tag::any`, 150
`tag::up`, 150
`testall`, 140
`testany`, 140
`testsome`, 140
threading-support, 633
timing, 394

user-defined-operators, 77

vector-buffers, 45
vector-type, 205

wait-any, 140
`waitall`, 140
`waitany`, 140
`waitsome`, 140
world-communicator, 30
`wtick`, 394
`wtime_is_global`, 394

54.3 Python

54.3.1 Notes

Big data, 225
Buffers from numpy, 44

Comm split key is optional, 267
Communicator object, 515
Communicator objects, 30
Communicator rank and size, 32
Communicator types, 261

Data types, 191
Define reduction operator, 75
Derived type handling, 199
Displacement byte computations, 302

File open is class method, 346

Graph communicators, 362

Import mpi module, 26
In-place collectives, 49
Init, and with commandline options, 515

No initialize/finalize calls, 27

P, 665
Petsc options, 586
Petsc print and python print, 583
Python notes, 18

Reduction function, 77
Request arrays, 138
Running mpi4py programs, 16

Sending objects, 51

Vector creation, 522
Vector type, 205

Window buffers, 303

54.3.2 Buffer specifications

54.3.3 Listing of python routines

Class Comm: Class Cartcomm: Class Distgraphcomm: Class Graphcomm: Class Intercomm: Class Intracomm: Class Topocomm: Class Group:
Class Request: Class Grequest: Class Prequest: Class Status:
Class Win:
Class Datatype: Class File: Class Info: Class Op:
Class Errhandler: Class Message:

Chapter 55

Index of OpenMP commands

_OPENMP, 414

chunk, 430
critical, 430

dynamic, 496

firstprivate, 506
for, 430

nowait, 496

omp
- atomic, 460, 485
- barrier, 458
- cancel, **472**
- critical, 460, 496
- declare simd, 488
- flush, 461, 486
- lastprivate, 436
- master, **439**, 440, 633
- ordered, 434
- parallel, 415, 421, 480
- parallel for, 428
- private, 443
- section, 438
- sections, 438, 446
- simd, **488**, 488
- single, **439**
- task, 466, 469, 470
- taskgroup, 469, 470
- taskwait, 469--471
- taskyield, 471
- threadprivate, 446, 481, 496
- workshare, 440

omp clause
- aligned, 488
- collapse, 434
- copyin, 447
- copyprivate, **440**, 447
- default, 444
  - firstprivate, 445
  - none, 445
  - private, 444
  - shared, 444
- depend, 470
- firstprivate, 446, 467
- lastprivate, 446
- linear, 488
- nowait, 435, 459, 496
- ordered, 434
- private, 443
- proc_bind, 478, 480
- reduction, 451, 454
- safelen(*n*), 488
- schedule, 496
  - auto, 432
  - chunk, 430
  - guided, 431
  - runtime, 432
- untied, 471

OMP_CANCELLATION, **493**
OMP_DEFAULT_DEVICE, **493**
omp_destroy_nest_lock, **462**
OMP_DISPLAY_ENV, **477**, 493

OMP_DYNAMIC, 448, **493**, 494
omp_get_active_level, 493
omp_get_ancestor_thread_num, 493
omp_get_dynamic, 493, 494
omp_get_level, 493
omp_get_max_active_levels, **423**, 493
omp_get_max_threads, **419**, 493, 494
omp_get_nested, 493, 494
omp_get_num_procs, 416, **419**, 493, 494
omp_get_num_threads, 416, **419**, 421, 423, 493, 494
omp_get_schedule, 433, 493, 494
omp_get_team_size, 493
omp_get_thread_limit, 493
omp_get_thread_num, 416, 421, 423, 493, 494
omp_get_wtick, 493, 495
omp_get_wtime, 493, 495
omp_in, 454
omp_in_parallel, 424, 493, 494
omp_init_nest_lock, **462**
OMP_MAX_ACTIVE_LEVELS, **423**, **493**
OMP_MAX_TASK_PRIORITY, **494**
OMP_NESTED (deprecated), 423
OMP_NESTED, 419, **494**, 494
OMP_NUM_THREADS, 414, 415, 418, 419, **494**, 494
omp_out, 454
OMP_PLACES, **477**, 479, **494**
omp_priv, 454
OMP_PROC_BIND, **477**, 479, **494**, 495
omp_sched_affinity, 433
omp_sched_auto, 433
omp_sched_dynamic, 433
omp_sched_guided, 433
omp_sched_runtime, 433
omp_sched_static, 433
OMP_SCHEDULE, 432, 433, **494**, 494
omp_set_dynamic, 493, 494
omp_set_max_active_levels, **423**, 493
omp_set_nest_lock, **462**
omp_set_nested, 493, 494
omp_set_num_threads, 418, 493, 494
omp_set_schedule, 433, 493, 494
OMP_STACKSIZE, 443, **494**, 494
omp_test_nest_lock, **462**
OMP_THREAD_LIMIT, **494**
omp_unset_nest_lock, **462**
OMP_WAIT_POLICY, **423**, **494**, 494
ordered, 435

parallel, 415, 422, 430, 506
private, 506

reduction, 430, 432

schedule, 430, 432, 433

wait-policy-var, **494**

Chapter 56

Index of PETSc commands

--sub_ksp_monitor, 587
-da_grid_x, 549
-da_refine, 553
-da_refine_x, 553
-download-blas-lapack, 521
-download_mpich, 516
-ksp_atol, 566
-ksp_converged_reason, 567
-ksp_divtol, 566
-ksp_gmres_restart, 568
-ksp_max_it, 566
-ksp_monitor, 573, 587
-ksp_monitor_true_residual, 573
-ksp_rtol, 566
-ksp_type, 567
-ksp_view, 566, 585, 587
-log_summary, 586
-log_view, 588
-malloc_dump, 589
-mat_view, 585
-pc_factor_levels, 571
-snes_fd, 577
-snes_fd_color, 577
-vec_view, 585
-with-precision, 516
-with-scalar-type, 516

ADD_VALUES, 528, 536
AO, **543**
AOViewFromOptions, 585

CHKERRA, **580**
CHKERRQ, **580**
CHKMEMA, **582**
CHKMEMQ, 580, **582**

DM, 549, 550, 554
DM_BOUNDARY_GHOSTED, 549
DM_BOUNDARY_NONE, 549
DM_BOUNDARY_PERIODIC, 549
DM_STENCIL_BOX, 549
DM_STENCIL_STAR, 549
DMBoundaryType, **549**
DMCreateGlobalVector, **554**
DMCreateLocalVector, **554**
DMDA, **549**, 552, 554
DMDACreate1d, **549**
DMDACreate2d, **549**, 549
DMDAGetCorners, **550**, 554
DMDAGetLocalInfo, **550**
DMDALocalInfo, **550**, 552
DMDASetRefinementFactor, **553**
DMDAVecGetArray, **552**
DMGlobalToLocal, **554**
DMGlobalToLocalBegin, **554**
DMGlobalToLocalEnd, **554**
DMLocalToGlobal, **554**
DMLocalToGlobalBegin, **554**
DMLocalToGlobalEnd, **554**
DMStencilType, **549**
DMViewFromOptions, 585

INSERT_VALUES, 528, 536
IS, 543

ISCreate, **541**
ISCreateBlock, **541**
ISCreateGeneral, 541
ISCreateStride, **541**
ISGetIndices, **541**
ISLocalToGlobalMappingViewFromOptions, 585
ISRestoreIndices, **541**
ISViewFromOptions, 585

KSP, **564**
KSPBuildResidual, **573**
KSPBuildSolution, **573**
KSPConvergedDefault, **573**
KSPConvergedReason, 566
KSPConvergedReasonViewFromOptions, 585
KSPConvergenceReasonView, 567
KSPCreate, **565**
KSPGetConvergedReason, **566**
KSPGetIterationNumber, 567
KSPGetRhs, **573**
KSPGetSolution, **573**
KSPGMRESSetRestart, **568**
KSPMatSolve, **568**
KSPMonitorDefault, **573**
KSPMonitorSet, **573**
KSPMonitorTrueResidualNorm, **573**
KSPReasonView (deprecated), 567
KSPSetConvergenceTest, **573**
KSPSetFromOptions, 566, 568, **574**
KSPSetOperators, 566
KSPSetOptionsPrefix, **587**
KSPSetTolerances, **566**
KSPSetType, **567**
KSPView, 566, 584
KSPViewFromOptions, 585

MAT_FLUSH_ASSEMBLY, **536**
MatAssemblyBegin, **536**, 536
MatAssemblyEnd, **536**, 536
MatCoarsenViewFromOptions, 585
MatCreate, **531**
MatCreateFFT, **541**
MatCreateShell, **538**
MatCreateSubMatrices, **538**
MatCreateSubMatrix, **538**, 543
MatCreateVecs, 525, **532**
MatCreateVecsFFTW, 541
MatDenseGetArray, 536
MatDenseRestoreArray, 536
MatGetArray (deprecated), 536
MatGetRow, **536**
MatImaginaryPart, **521**
MatMatMult, **537**
MATMPIAIJ, 531
MatMPIAIJSetPreallocation, **534**
MATMPIBIJ, **540**
MATMPIDENSE, 531
MatMult, **537**, 538
MatMultAdd, **537**
MatMultHermitianTranspose, 537
MatMultTranspose, **537**
MatPartitioning, **543**
MatPartitioningApply, **543**
MatPartitioningCreate, **543**
MatPartitioningDestroy, **543**
MatPartitioningSetType, **543**
MatPartitioningViewFromOptions, 585
MatRealPart, **521**
MatRestoreArray (deprecated), 536
MatRestoreRow, **536**
MATSEQAIJ, 531
MatSeqAIJGetArray, 536
MatSeqAIJRestoreArray, 536
MatSeqAIJSetPreallocation, **534**
MATSEQDENSE, 531
MatSetOptionsPrefix, **587**
MatSetSizes, **532**
MatSetType, **531**
MatSetValue, **535**
MatSetValues, 535
MatShellGetContext, **538**
MatShellSetContext, **538**
MatShellSetOperation, **538**
MatSizes, **532**
MatView, 584, 585
MatViewFromOptions, 585

MPIU_COMPLEX, **521**
MPIU_REAL, **521**
MPIU_SCALAR, **521**

PCCOMPOSITE, **572**
PCFactorSetLevels, **571**
PCGAMG, **571**
PCHYPRESetType, **570**
PCMG, **571**
PCSetOptionsPrefix, **587**
PCSHELL, **572**
PCShellGetContext, **572**
PCShellSetApply, **572**
PCShellSetContext, **572**
PCShellSetSetUp, **572**
PCViewFromOptions, 585
PETSC_ARCH, 512, 513
PETSC_CC_INCLUDES, 513
PETSC_COMM_SELF, 522, 554, 581
PETSC_COMM_WORLD, **515**, 523, 581
PETSC_DECIDE, 520, **523**
PETSC_DEFAULT, 566
PETSC_DIR, 512, 513
PETSC_ERR_ARG_OUTOFRANGE, 521
PETSC_FALSE, **522**
PETSC_FC_INCLUDES, 513
PETSC_i, **521**
PETSC_MEMALIGN, **589**
PETSC_NULL, 550
PETSC_NULL_CHARACTER, 515
PETSC_NULL_INTEGER, 512
PETSC_NULL_IS, 542
PETSC_NULL_OBJECT, 512
PETSC_NULL_VIEWER, 584
PETSC_STDOUT, 583
PETSC_TRUE, **522**
PETSC_USE_DEBUG, **580**
PETSC_VIEWER_STDOUT_WORLD, **584**
PetscBLASInt, **521**, 521
PetscBLASIntCast, **521**
PetscBool, **522**
PetscCalloc1, **588**
PetscComm, **594**
PetscComplex, **521**, 521
PetscDataType, 583
PetscDrawViewFromOptions, 585
PetscDSViewFromOptions, 585
PetscDualSpaceViewFromOptions, 585
PetscErrorCode, **521**, **580**
PetscFEViewFromOptions, 585
PetscFinalize, **515**, 589
PetscFree, **588**
PetscFunctionBegin, **580**
PetscFunctionReturn, **580**
PetscFVViewFromOptions, 585
PetscGetCPUTime, **588**
PetscImaginaryPart, **521**
PetscInitialize, **514**, 515, 586, 587, 594
PetscInitializeFortran, **515**
PetscInt, **521**, 521, 541
petscksp.h, 568
PetscLimiterViewFromOptions, 585
PetscLogDouble, **588**
PetscLogView, **588**
PetscLogViewFromOptions, 585
PetscMalloc, 530, **588**
PetscMalloc1, **588**
PetscMallocDump, **589**
PetscMPIInt, **521**, 521
PetscMPIIntCast, **521**
PetscNew, **588**
PetscObjectSetOptionsPrefix, **587**
PetscObjectViewFromOptions, 585
PetscOptionsBegin, **586**
PetscOptionsEnd, **586**
PetscOptionsGetInt, **586**
PetscOptionsSetValue, **587**, 587
PetscPartitionerViewFromOptions, 585
PetscPrintf, **583**, 583
PetscRandomViewFromOptions, 585
PetscReal, 197, **521**, 521, 588
PetscRealPart, **521**
PetscScalar, 197, **520**, 521
PetscSectionViewFromOptions, 585
PetscSFViewFromOptions, 585
PetscSpaceViewFromOptions, 585
PetscSplitOwnership, **520**

PetscSynchronizedFlush, **583**
PetscSynchronizedPrintf, **583**
PetscTime, **588**
PetscViewer, **584**
PETSCVIEWERASCII, **584**
PETSCVIEWERBINARY, **584**
PetscViewerCreate, **584**
PETSCVIEWERDRAW, **584**
PETSCVIEWERHDF5, **584**
PetscViewerPopFormat, **585**
PetscViewerPushFormat, **585**
PetscViewerRead, **583**
PetscViewerSetOptionsPrefix, **587**
PetscViewerSetType, **584**
PETSCVIEWERSOCKET, **584**
PETSCVIEWERSTRING, **584**
PetscViewerViewFromOptions, 585
PETSCVIEWERVTK, **584**
PFViewFromOptions, 585

SETERRA, **580**
SETERRQ, **580**
SETERRQ1, **580**, 581
SETERRQ2, **580**
SNES, **576**
SNESConvergedReasonViewFromOptions, 585
SNESJacobianFunction, **577**
SNESSetJacobian, **577**
SNESSetOptionsPrefix, **587**
SNESSolve, **576**
SNESViewFromOptions, 585

TaoLineSearchViewFromOptions, 585
TaoViewFromOptions, 585
TSSetIFunction, **578**
TSSetOptionsPrefix, **587**
TSSetRHSFunction, **578**
TSTrajectoryViewFromOptions, 585
TSViewFromOptions, 585

VecAssemblyBegin, **528**, 528
VecAssemblyEnd, 528
VecCreate, **522**, 552
VecCreateMPIWithArray, **524**
VecCreateSeqWithArray, **524**
VecDestroy, **522**
VecDestroyVecs, **523**
VecDot, **525**
VecDotBegin, **526**
VecDotEnd, **526**
VecDuplicate, **523**
VecDuplicateVecs, **523**
VecGetArray, **528**
VecGetArrayF90, **530**
VecGetArrayRead, **528**
VecGetLocalSize, **524**
VecGetOwnershipRange, **524**
VecGetSize, **524**
VecImaginaryPart, **521**
VecLoad, **531**
VECMPI, 523
VecNorm, **525**
VecNormBegin, **526**
VecNormEnd, **526**
VecPlaceArray, **530**, 530
VecRealPart, **521**
VecReplaceArray, **530**
VecResetArray, **530**
VecRestoreArray, **530**
VecRestoreArrayF90, **530**
VecRestoreArrayRead, **530**
VecScale, **525**
VecScatter, **542**
VecScatterCreate, **542**
VecScatterViewFromOptions, 585
VECSEQ, 522
VecSet, **526**
VecSetOptionsPrefix, **587**
VecSetSizes, **523**, 541, 552
VecSetType, **522**
VecSetValue, **526**, 526, 528
VecSetValues, **526**, 526, 528, 552
VecView, **525**, 531
VecViewFromOptions, **585**, 585

Chapter 57

Index of SYCL commands

accessor, 610, 612

buffer, 610

combine, **610**
cout, 613
cpu_selector, 604

endl, 613

free, 611

get_access, 612
get_range, 612

host_selector, 605

id<1>, 607
id<nd>, 607
is_cpu, 604
is_gpu, 604
is_host, 604

malloc, 611
malloc_device, 610, 611
malloc_host, 611

nd_item, 608

offset, 614

parallel_for, 607, 610, 614

range, **607**, 607
read, 612

reduction, **610**
runtime_error, 605

submit, 606

wait, 613
write, 612


ISBN 978-1-387-40028-7
90000
9 781387 400287