Παράλληλη Επεξεργασία

Εαρινό Εξάμηνο 2022-23
«MPI Programming Model – II»

Παναγιώτης Χατζηδούκας, Ευστράτιος Γαλλόπουλος

Schedule and Goals

- MPI - part 2
  - asynchronous communication
  - how MPI works
  - study and discuss more examples

Outline

- Measuring wall-clock time
- Point-to-point communication and deadlock revisited
  - buffered send
  - under the hood: `MPI_Send` and the eager protocol
  - Probing messages and flexible message sending
- Non-blocking point-to-point communication
- More about collective operations
  - `MPI_Barrier`: under the hood
  - `MPI_Bcast`: is there a barrier?

I. Measuring Time

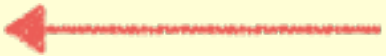
```
#include <stdio.h>  // printf
#include <unistd.h> // sleep
#include <mpi.h>

int main(int argc, char *argv[])
{
   MPI_Init(&argc, &argv); // initialize the environment

   int rank, size;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   double t1 = MPI_Wtime();
   sleep(2);  // do_MPI_parallel_work();
   double t2 = MPI_Wtime();
   if (rank == 0)
      printf("Elapsed time =%f seconds\n", t2-t1);

   MPI_Finalize(); // cleanup
   return 0;
}
```



Use an MPI barrier to synchronize processes and time measurements

II. Point to point communication revisited

- Messages are sent and received through `MPI_Send` and `MPI_Recv` calls

```
int MPI_Send(void* buf, int count, MPI_Datatype type,
             int dest, int tag, MPI_Comm comm);

int MPI_Recv(void* buf, int count, MPI_Datatype type,
             int source, int tag, MPI_Comm comm,
             MPI_Status* status)
```

- An `MPI_Recv` matches a message sent by `MPI_Send` if `tag`, `source`, and `dest` match.
  - `MPI_ANY_TAG`, `MPI_ANY_SOURCE` can be used for `MPI_Recv`
- `MPI_Recv`: buffer size is the allocated memory = maximum message size that can be received
  - Not necessarily the actual size
- `MPI_Recv` is blocking: returns once the message has been received.
  - The status object can be queried for more information about the message

Sending and receiving

- Blocking sends return only when the buffer is ready to be reused. The destination might or might not have received the message yet:

```
int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// synchronous send: returns when the destination has started to receive the message

int MPI_Bsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// buffered send: returns after making a copy of the buffer. The destination might not yet
//                have started to receive the message

int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// standard send: can be synchronous or buffered, depending on message size

int MPI_Rsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
// ready send: an optimized send if the user can guarantee that the destination has already
//             posted the matching receive
```

Synchronous sends and deadlock

- Both ranks wait for the other one to receive the message. We hang forever in a deadlock

```
int main(int argc, char** argv) {
  MPI_Status status;
  int num;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  double d=3.1415927;
  int tag=99;

  if(num==0) {
    MPI_Ssend(&d,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    MPI_Recv (&d,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Ssend(&d,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
    MPI_Recv (&d,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,&status);
  }

  MPI_Finalize();
  return 0;
}
```

- Possible solutions discussed in the previous lecture: reordering of calls, use of `MPI_Sendrecv`

Revisiting attempt 4: hope that you are lucky

- Hope that for such a small message MPI will always buffer it when using a standard send.
- If this works or not depends on the side of the buffer sent.

```
int main(int argc, char** argv) {
  MPI_Status status;
  int num;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  double ds=3.1415927; // to send
  double dr;           // to receive
  int tag=99;

  if(num==0) {
    MPI_Send(&ds,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD);
    MPI_Recv(&dr,1,MPI_DOUBLE,1,tag,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Send(&ds,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD);
    MPI_Recv(&dr,1,MPI_DOUBLE,0,tag,MPI_COMM_WORLD,&status);
  }

  MPI_Finalize();
  return 0;
}
```

What does "lucky" mean?

Attempt 5: buffering

- We can provide a large enough buffer and force a buffered send

```
int main(int argc, char** argv) {
  MPI_Status status;
  int num;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  double ds=3.1415927; // to send
  double dr;           // to receive

  // allocate a buffer and attach it to MPI
  int buffer_size = sizeof(double) + MPI_BSEND_OVERHEAD;
  char* buffer = malloc(buffer_size*sizeof(char));
  MPI_Buffer_attach(buffer,buffer_size);

  if(num==0) {
    MPI_Bsend(&ds,1,MPI_DOUBLE,1,99,MPI_COMM_WORLD);
    MPI_Recv (&dr,1,MPI_DOUBLE,1,99,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Bsend(&ds,1,MPI_DOUBLE,0,99,MPI_COMM_WORLD);
    MPI_Recv (&dr,1,MPI_DOUBLE,0,99,MPI_COMM_WORLD,&status);
  }

  // detach the buffer, making sure all sends are done
  MPI_Buffer_detach(buffer,&buffer_size);
  free(buffer);

  MPI_Finalize();
  return 0;
}
```

Message passing protocols

- **Eager** - An asynchronous protocol that allows a send operation to complete without acknowledgement from a matching receive
- **Rendezvous** - A synchronous protocol which requires an acknowledgement from a matching receive in order for the send operation to complete.

- Message passing protocols are not defined by the MPI standard, but are left up to implementors, and will vary.
- MPI implementations can use a combination of protocols for the same MPI routine.
  - Eager protocol for a small message, rendezvous protocol for larger messages.

Eager Protocol

- Assumption: the receiving process can store the message if it is sent
- The receiving process must buffer the message upon its arrival, especially if the receive operation has not been posted.
  - Certain amount of available buffer space is required on the receive process.

- Advantages: reduces synchronization delays, simple programming
- Disadvantages: not scalable, memory exhaustion

Source: https://computing.llnl.gov/tutorials/mpi_performance/#EagerVsRendezvous

Rendezvous Protocol

- No assumption about the receiving process
- Handshaking between the two processes
  - Sender sends the message envelope
  - Receiver acknowledges when buffer is ready
  - Sender sends data
  - Destination receives data

- Advantages: scalable, robust, possibility for direct communication
- Disadvantages: inherent synchronization, programming complexity

Take-home message:
Recv's must be posted before Send's!

Source: https://computing.llnl.gov/tutorials/mpi_performance/#EagerVsRendezvous

Attempt 6: identifying the protocol

- For which value of N (#bytes) deadlock occurs in the following code?

```
mpicc -o deadlock6 deadlock6.c
mpirun -n 2 ./deadlock6 1024
```

```
Environment variable for MPICH:
MPIR_CVAR_NEMESIS_SHM_EAGER_MAX_SZ
```

```
int main(int argc, char** argv) {
  MPI_Status status;
  int num;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&num);

  int N = 1024;
  if (argc == 2) N = atoi(argv[1]);

  char *ds=(char *)calloc(1, N*sizeof(char)); // to send
  char *dr=(char *)calloc(1, N*sizeof(char)); // to recv

  int tag=99;

  if(num==0) {
    MPI_Send(ds,N,MPI_CHAR,1,tag,MPI_COMM_WORLD);
    MPI_Recv(dr,N,MPI_CHAR,1,tag,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Send(ds,N,MPI_CHAR,0,tag,MPI_COMM_WORLD);
    MPI_Recv(dr,N,MPI_CHAR,0,tag,MPI_COMM_WORLD,&status);
  }

  MPI_Finalize();
  return 0;
}
```

Under the Hood - 1

- Quick look into the implementation of MPI
  - http://www.mpich.org/downloads/
  - mpich-3.2/src/mpi/pt2pt/send.c

```
/*@
    MPI_Send - Performs a blocking send

Input Parameters:
+ buf - initial address of send buffer (choice)
. count - number of elements in send buffer (nonnegative integer)
. datatype - datatype of each send buffer element (handle)
. dest - rank of destination (integer)
. tag - message tag (integer)
- comm - communicator (handle)

Notes:
This routine may block until the message is received by the destination
process.
```

```
@*/
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
             MPI_Comm comm)
{
```

```
    /* ... body of routine ...  */

    mpi_errno = MPID_Send(buf, count, datatype, dest, tag, comm_ptr,
                          MPID_CONTEXT_INTRA_PT2PT, &request_ptr);
    if (mpi_errno != MPI_SUCCESS) goto fn_fail;
```

Under the Hood - 2

- From `MPI_Send` to `MPID_Send`
  - mpich-3.2/src/mpid/ch3/src/mpid_send.c

```
MPIDI_CH3_GET_EAGER_THRESHOLD(&eager_threshold, comm, vc);

if (data_sz + sizeof(MPIDI_CH3_Pkt_eager_send_t) <= eager_threshold)
{
if (dt_contig)
    {
    mpi_errno = MPIDI_CH3_EagerContigSend( &sreq,
                        MPIDI_CH3_PKT_EAGER_SEND,
                        (char *)buf + dt_true_lb,
                        data_sz, rank, tag, comm,
                        context_offset );
}
else
    {
    MPIDI_Request_create_sreq(sreq, mpi_errno, goto fn_exit);
    MPIDI_Request_set_type(sreq, MPIDI_REQUEST_TYPE_SEND);
    mpi_errno = MPIDI_CH3_EagerNoncontigSend( &sreq,
                                              MPIDI_CH3_PKT_EAGER_SEND,
                                              buf, count, datatype,
                                              data_sz, rank, tag,
                                              comm, context_offset );
}
}
else
{
MPIDI_Request_create_sreq(sreq, mpi_errno, goto fn_exit);
MPIDI_Request_set_type(sreq, MPIDI_REQUEST_TYPE_SEND);
mpi_errno = vc->rndvSend_fn( &sreq, buf, count, datatype, dt_contig,
                                data_sz, dt_true_lb, rank, tag, comm,
                                context_offset );
}
```

Take-home message:

Do not be afraid to look inside the implementation!

III. Probing for messages

- Instead of directly receiving you can probe whether a message has arrived:

```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status)
// wait for a matching message to arrive

int MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status *status)
// check if a message has arrived.
// flag is nonzero if there is a message waiting

int MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int* count)
// gets the number of elements in the message waiting to be received
```

- The `MPI_Status` object can be queried for information about the message:

```
MPI_Status status;
int count;

// wait for a message
MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, &status);
std::cout << "A message is waiting from " << status->MPI_SOURCE
          << "with tag "                  << status->MPI_TAG;

// get the element count
MPI_Get_count(&status, MPI_INT, &count);
std::cout << "and assuming it contains ints there are " << count << "elements";
```

MPI_Status and MPI_Recv

- Looking into mpi.h:

```
typedef struct MPI_Status {
    int count_lo;
    int count_hi_and_cancelled;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
} MPI_Status;
```

- Example

```
int tag=99;

if(num==0) {
  MPI_Send(ds, N, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
}
else {
 MPI_Recv(dr, N, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

 int count;
 MPI_Get_count(&status, MPI_CHAR, &count);

 printf("status.MPI_SOURCE = %d, status.MPI_TAG=%d, count = %d\n",
        status.MPI_SOURCE, status.MPI_TAG, count);
}
```

- Output

```
$ mpicc -o status0 status0.cpp
$ mpirun -l -n 2 ./status0 1024
[1] status.MPI_SOURCE = 0, status.MPI_TAG = 99, count = 1024
```

MPI_Status and MPI_Recv

- Consider the following code (same as before):

```
int tag=99;

if(num==0) {
  MPI_Send(ds, N, MPI_CHAR, 1,tag, MPI_COMM_WORLD);
}
else {
 MPI_Recv(dr, N, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);

 int count;
 MPI_Get_count(&status, MPI_CHAR, &count);
 printf("status.MPI_SOURCE = %d, status.MPI_TAG=%d, count = %d\n",
           status.MPI_SOURCE, status.MPI_TAG, count);
}
```

- What will happen if:
  1. Rank 0 sends less data?
     (or, equivalently, rank 1 asks for more data)?
  2. Rank 0 sends more data?
     (or, equivalently, rank 1 asks for less data?)

MPI_Status and MPI_Recv

- Case 1: rank 0 sends N/2, rank 1 asks for N

```
int tag=99;

if(num==0) {
    MPI_Send(ds, N/2, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
 }
else {
   MPI_Recv(dr, N, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
   int count;
   MPI_Get_count(&status, MPI_CHAR, &count);
   printf("status.MPI_SOURCE = %d, status.MPI_TAG=%d, count = %d\n",
           status.MPI_SOURCE, status.MPI_TAG, count);
}
```

```
$ mpirun -l -n 2 ./status0_less 1024
[1] status.MPI_SOURCE = 0, status.MPI_TAG = 99, count = 512
```

- Case 2: rank 0 sends N, rank 1 asks for N/2

```
$ mpirun -l -n 2 ./status0_more 1024
[1] status.MPI_SOURCE = 0, status.MPI_TAG = 99, count = 512
[1] Fatal error in MPI_Recv: Message truncated, error stack:
[1] MPI_Recv(200).....................: MPI_Recv(buf=0x7fe9a8002600, count=512,
MPI_CHAR, src=0, tag=99, MPI_COMM_WORLD, status=0x7fff55005968) failed
[1] MPIDI_CH3U_Receive_data_found(131): Message from rank 0 and tag 99 truncated; 1024
bytes received but buffer size is 512
```

How can we deal with incoming messages of unknown size?

Answer: MPI_Status and MPI_Probe

- Solution 1: Probe, get count, and then receive

status1.c

```
int tag=99;

if(num==0) {
    MPI_Send(ds, N/2, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
}
else {
   MPI_Probe(0, tag, MPI_COMM_WORLD, &status);
   int count;
   MPI_Get_count(&status, MPI_CHAR, &count);
   printf("status.MPI_SOURCE = %d, status.MPI_TAG=%d, count = %d\n",
           status.MPI_SOURCE, status.MPI_TAG, count);
   MPI_Recv(dr, count, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
}
```

- Solution 2: Probe, get source+tag+count, and then receive

status2.c

```
int tag=99;

if(num==0) {
    MPI_Send(ds, N/2, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
}
else {
    MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    int count;
    MPI_Get_count(&status,MPI_CHAR,&count);
    printf("status.MPI_SOURCE = %d, status.MPI_TAG = %d, count = %d\n",
                status.MPI_SOURCE, status.MPI_TAG, count);
    MPI_Recv(dr,count, MPI_CHAR, status.MPI_SOURCE, status.MPI_TAG, MPI_COMM_WORLD,&status);
}
```

increased flexibility

Example: Sending data of dynamic size

- In the following example, rank 0 allocates, initializes and finally sends N elements to rank 1
  - N is chosen randomly at runtime
  - The max size is predefined (`MAX_N`)

dynamic1.c

```
int MAX_N = 100; // max number of elements
int N;

if (rank == 0) {
    N = lrand48()%MAX_N;
    double *x = (double *)calloc(1, N*sizeof(double));
    init_data(x, N);
    MPI_Send(x, N, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
}
if (rank == 1) {
   double *y = (double *)calloc(1, MAX_N*sizeof(double));
    MPI_Recv(y, MAX_N, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_DOUBLE, &N);
    print_data(y, N);
}
```

Rank 1 allocates and asks for MAX_N elements.
Can we avoid this?

Based on: http://mpitutorial.com/tutorials/dynamic-receiving-with-mpi-probe-and-mpi-status/

Using MPI_Probe

- Get the number of elements, allocate and receive

dynamic2.c

```
int MAX_N = 100;
int N;

if (rank == 0) {
    N = lrand48()%MAX_N;
    double *x = (double *)calloc(1, N*sizeof(double));
    init_data(x, N);
    MPI_Send(x, N, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
}
if (rank == 1) {
    MPI_Probe(0, 123, MPI_COMM_WORLD, &status);
    MPI_Get_count(&status, MPI_DOUBLE, &N);
    double *y = (double *)calloc(1, N*sizeof(double));
    MPI_Recv(y, N, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, &status);
    print_data(y, N);
}
```

How can we do this without using MPI_Probe?

Sending Count (# Elements)

- Use an extra message to send first the size and then the elements

dynamic3.c

```
int MAX_N = 100;
int N;

if (rank == 0) {
    N = lrand48()%MAX_N;
    double *x = (double *)calloc(1, N*sizeof(double));
    init_data(x, N);
    MPI_Send(&N, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);
    MPI_Send(x, N, MPI_DOUBLE, 1, 124, MPI_COMM_WORLD);
}
if (rank == 1) {
     MPI_Recv(&N, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
    double *y = (double *)calloc(1, N*sizeof(double));
    MPI_Recv(y, N, MPI_DOUBLE, 0, 124, MPI_COMM_WORLD, &status);
    print_data(y, N);
}
```

What if we do not know the data type?

Dynamic Size and Datatype

- Send size and MPI datatype

dynamic4.c

```
if (rank == 0) {
    N = lrand48()%MAX_N;
    double *x = (double *)calloc(1, N*sizeof(double));
    init_data(x, N);
    MPI_Send(&N, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);

    long type = (long) MPI_DOUBLE;
    MPI_Send(&type, sizeof(long), MPI_BYTE, 1, 124, MPI_COMM_WORLD);

    MPI_Send(x, N, MPI_DOUBLE, 1, 125, MPI_COMM_WORLD);
}
if (rank == 1) {
    MPI_Recv(&N, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
    double *y = (double *)calloc(1, N*sizeof(double));
    long type;
    MPI_Recv(&type, sizeof(long), MPI_BYTE, 0, 124, MPI_COMM_WORLD, &status);
    if (type == MPI_DOUBLE) {
        MPI_Recv(y, N, MPI_DOUBLE, 0, 125, MPI_COMM_WORLD, &status);
    } else {
        MPI_Abort(MPI_COMM_WORLD, 911); // from Deino MPI
    }
    print_data(y, N);
}
```

This code here is not safe!

The code above will work in many cases but it is not considered valid / safe.
Rule: "You must never send MPI datatypes as messages"
This should not be enough for us, we need to know why!

Dynamic Size and Datatype

- What is an MPI datatype? Looking again into `mpi.h`
  - MPICH
    - `typedef int MPI_Datatype;`
    - `MPI_DOUBLE` is a predefined constant integer
    - It has the same value on all processes
  - OpenMPI
    - `typedef struct ompi_datatype_t *MPI_Datatype;`
    - `MPI_DOUBLE` is a pointer to a memory location
    - It depends on the memory layout of the process and thus it might be different for each process (rank) *[see code below]*

```
long type  = (long) MPI_DOUBLE;
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

    printf("rank %d: type = %ld, &type = %p\n",
       rank, type, &type);

    MPI_Finalize();
    return 0;
}
```

Processes do not always print the same values
- True: MacOS + MPICH, Linux + OpenMPI
- False: Linux + MVAPICH2 (= MPICH)

```
[chatzidp@eu-login-06 ~]$ mpirun -n 2 ./hello_datatype
I am rank 0 of 2: type = 47390899143520, &type = 0x601060
I am rank 1 of 2: type = 47347376649056, &type = 0x601060
```

Dynamic Size and Datatype

- Solution: use your own mapping for MPI datatypes, e.g.:

```
const long MY_MPI_DOUBLE = 100;

if (rank == 0) {
    N = lrand48()%MAX_N;
    double *x = (double *)calloc(1, N*sizeof(double));
    init_data(x, N);
    MPI_Send(&N, 1, MPI_INT, 1, 123, MPI_COMM_WORLD);

    long type = (long) MY_MPI_DOUBLE;
    MPI_Send(&type, 1, MPI_LONG, 1, 124, MPI_COMM_WORLD);

    MPI_Send(x, N, MPI_DOUBLE, 1, 125, MPI_COMM_WORLD);
}
if (rank == 1) {
    MPI_Recv(&N, 1, MPI_INT, 0, 123, MPI_COMM_WORLD, &status);
    double *y = (double *)calloc(1, N*sizeof(double));
    long type;
    MPI_Recv(&type, 1, MPI_LONG, 0, 124, MPI_COMM_WORLD, &status);
    if (type == MY_MPI_DOUBLE) {
        MPI_Recv(y, N, MPI_DOUBLE, 0, 125, MPI_COMM_WORLD, &status);
    } else {
        MPI_Abort(MPI_COMM_WORLD, 911); // from Deino MPI
    }
    print_data(y, N);
}
```

Take-home message:
Your code must not be based on assumptions about the underlying software (and hardware)

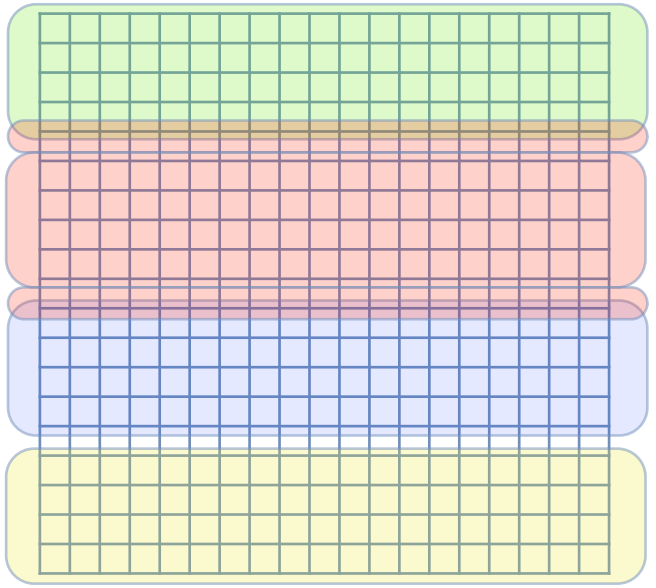
IV. Domain decomposition for PDEs

- Simple example: finite difference solution of a diffusion equation

$$\frac{\partial \phi(\vec{r},t)}{\partial t} = D\Delta\phi(\vec{r},t)$$

- Domain decomposition: split the mesh over the nodes of the parallel computer

- The finite difference stencil needs information from the neighboring domains: stored in *"ghost cells"*
- Message passing is needed to update the ghost cells after each time step



1D diffusion equation in MPI

- We need to exchange the ghost cell values in a deadlock-free way before each iteration

```
for (int t=0; t<iterations; ++t) {
  // first get the ghost cells and send our boundary values to
  // the neighbor for their ghost cells

  // avoid deadlocks by a clear ordering who sends and receives first
  // make sure we have an even number of ranks for this to work
  assert(size %2 == 0);

  if (rank % 2 == 0) {
    MPI_Send(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD);
    MPI_Recv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&status);
    MPI_Send(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD);
    MPI_Recv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&status);
  }
  else {
    MPI_Recv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&status);
    MPI_Send(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD);
    MPI_Recv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&status);
    MPI_Send(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD);
  }

  // do calculation
  for (int i=1; i<local_N-1;++i)
    newdensity[i] = density[i] + coefficient * (density[i+1]+density[i-1]-2.*density[i]);

  // and swap (C++)
  density.swap(newdensity);
}
```

Overlaying communication and computation

- This code will not scale well since we waste time in waiting for the ghost cells to arrive. A better strategy is to overlay computation and communication:
  1. start the communication for the ghost cells
  2. update the interior of the local segment
  3. wait for communication to finish
  4. update the boundary values using the ghost cells

- We now use the wait time for communication to perform most of the calculations. Ideally the ghost cells will have arrived before we finish the computation and communication will then be essentially free.
- This needs non-blocking, asynchronous communication

Nonblocking send and receive

- These functions return immediately, while communication is still ongoing.

```
int MPI_Issend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request)

int MPI_Ibsend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
               MPI_Comm comm, MPI_Request *request)


int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_Request *request)

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Request *request)
```

- They behave the same way as the corresponding blocking versions but perform the communication asynchronously.
- They fill in an `MPI_Request` object that can be used to test for completion.

https://stackoverflow.com/questions/21512975/what-is-the-difference-between-isend-and-issend-in-openmpi

Waiting for completion

- We can wait for one, some, or all communication requests to finish

```
int MPI_Wait(MPI_Request *request, MPI_Status *status)
// waits for the communication to finish and fills in the status

int MPI_Waitall(int count, MPI_Request array_of_requests[],
                MPI_Status array_of_statuses[])
// waits for all given communications to finish and fills in the statuses

int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index,
                MPI_Status *status)
// waits for one of the given communications to finish, sets the index to indicate
// which one and fills in the status

int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[], MPI_Status array_of_statuses[])
// waits for at least one of the given communications to finish, sets the number
// of communication requests that have finished, their indices and status
```

Testing for completion and cancellation

- Instead of waiting we can just test whether they have finished

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
// tests if the communication is finished. Sets flag to 1 and fills in the status if
// finished or sets the flag to 0 if not finished.

int MPI_Testall(int count, MPI_Request array_of_requests[], int *flag,
                MPI_Status array_of_statuses[])
// test whether all given communications are finished. Sts flag to 1 and fills in
// the status aray if all are finished or sets the flag to 0 if not all are finished.

int MPI_Testany(int count, MPI_Request array_of_requests[], int *index,
                int *flag, MPI_Status *status)
// test whether one of the given communications is finished. Sets flag to 1 and fills
// in the index and status if one  finished or sets the flag to 0 if none is
finished.

int MPI_Testsome(int incount, MPI_Request array_of_requests[], int *outcount,
                 int array_of_indices[], MPI_Status array_of_statuses[])
// tests whether some of the given communications is finished, sets the number
// of communication requests that have finished, their indices and statuses.
```

- We can cancel a request if we don't want to wait any longer

```
int MPI_Cancel(MPI_Request *request)
```

Testing for completion and cancellation

- Exchange ghost cells while we compute the interior

```
for (int t=0; t<iterations; ++t) {
  // first start the communications

  if (rank % 2 == 0) {
    MPI_Isend(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[0]);
    MPI_Irecv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[1]);
    MPI_Isend(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[2]);
    MPI_Irecv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[3]);
  }
  else {
    MPI_Irecv(&density[local_N-1],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[0]);
    MPI_Isend(&density[local_N-2],1,MPI_DOUBLE,right,0,MPI_COMM_WORLD,&reqs[1]);
    MPI_Irecv(&density[0],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[2]);
    MPI_Isend(&density[1],1,MPI_DOUBLE,left,0,MPI_COMM_WORLD,&reqs[3]);
  }

  // do calculation  of the interior
  for (int i=2; i<local_N-2;++i)
    newdensity[i] = density[i] + coefficient * (density[i+1]+density[i-1]-2.*density[i]);

  // wait for the ghost cells to arrive
  MPI_Waitall(4,reqs,status);

  // do the boundaries
  newdensity[1] = density[1] + coefficient * (density[2]+density[0]-2.*density[1]);
  newdensity[local_N-2] = density[local_N-2] + coefficient * (
                            density[local_N-1]+density[local_N-3]-2.*density[local_N]);

  // and swap
  density.swap(newdensity);
}
```

Non-blocking communication: Example 1

- Describe the following code

```
int main(int argc, char *argv[])  {
 int rank, size;
 int tag1 = 98, tag2 = 99;
 MPI_Request req0, req1;
 MPI_Status stat0, stat1;

 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 if (rank == 0) {
   int x[2];
   x[0] = 5; x[1] = 10;
   MPI_Isend(&x[0], 1, MPI_INT, 1, tag1, MPI_COMM_WORLD, &req0);
   MPI_Isend(&x[1], 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, &req1);

   MPI_Wait(&req0, &stat0);
   MPI_Wait(&req1, &stat1);
 }
 else if (rank == 1) {
   int y[2];
   MPI_Irecv(&y[0], 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, &req0);
   MPI_Irecv(&y[1], 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, &req1);

   MPI_Wait(&req0, &stat0);
   MPI_Wait(&req1, &stat1);
 }

 MPI_Finalize();
 return 0;
}
```

Non-blocking communication: Example 2

- Identify the issue in this code

```
int main(int argc, char *argv[])  {
 int rank, size;
 int tag1 = 98, tag2 = 99;
 MPI_Request req0, req1;
 MPI_Status stat0, stat1;

 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 if (rank == 0) {
   int x[2];
   x[0] = 5; x[1] = 10;
   MPI_Isend(&x[0], 1, MPI_INT, 1, tag1, MPI_COMM_WORLD, &req0);
   MPI_Isend(&x[1], 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, &req1);
 }
 else if (rank == 1) {
   int y[2];
   MPI_Irecv(&y[0], 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, &req0);
   MPI_Irecv(&y[1], 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, &req1);
 }

 MPI_Wait(&req0, &stat0);
 MPI_Wait(&req1, &stat1);


 MPI_Finalize();

 return 0;
}
```

Incorrect memory management:
x and y do not exist outside their block!

Non-blocking communication: Example 3a

- Identify the issue in this code

```
int main(int argc, char *argv[])  {
 int rank, size;
 int tag1 = 98, tag2 = 99;
 MPI_Request req0, req1;
 MPI_Status stat0, stat1;

 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 int x;    // safe memory management
 int y[2]; // safe memory management

 if (rank == 0) {
   x = 5;
   MPI_Isend(&x, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD, &req0);
   x = 10;
   MPI_Isend(&x, 1, MPI_INT, 1, tag2, MPI_COMM_WORLD, &req1);
 }
 else if (rank == 1) {
   MPI_Irecv(&y[0], 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, &req0);
   MPI_Irecv(&y[1], 1, MPI_INT, 0, tag2, MPI_COMM_WORLD, &req1);
 }

 MPI_Wait(&req0, &stat0);
 MPI_Wait(&req1, &stat1);

 if (rank == 1) printf("y[0] = %d, y[1] = %d\n", y[0], y[1]);

 MPI_Finalize();
 return 0;
}
```

Race condition:
the value of x changes between the two MPI_Isend() calls

The bug might not show up because of the small message size.
But we cannot rely on this assumption!

Non-blocking communication: Example 3b

- Previous example but using large message size

```
int main(int argc, char *argv[])  {
 int rank, size;
 int tag1 = 98, tag2 = 99;
 MPI_Request req0, req1;
 MPI_Status stat0, stat1;

 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
 MPI_Comm_size(MPI_COMM_WORLD, &size);

 int x[1][65000];
 int y[2][65000];

 if (rank == 0) {
   x[0][0] = 5;
   MPI_Isend(&x[0][0], 65000, MPI_INT, 1, tag1, MPI_COMM_WORLD, &req0);
   x[0][0] = 10;
   MPI_Isend(&x[0][0], 65000, MPI_INT, 1, tag2, MPI_COMM_WORLD, &req1);
 }
 else if (rank == 1) {
   MPI_Irecv(&y[0][0], 65000, MPI_INT, 0, tag1, MPI_COMM_WORLD, &req0);
   MPI_Irecv(&y[1][0], 65000, MPI_INT, 0, tag2, MPI_COMM_WORLD, &req1);
 }

 MPI_Wait(&req0, &stat0);
 MPI_Wait(&req1, &stat1);

 if (rank == 1) printf("y[0][0] = %d, y[1][0] = %d\n", y[0][0], y[1][0]);

 MPI_Finalize();
 return 0;
}
```

Some padding here to increase the message size and reproduce the race condition bug

MPI_Wait and MPI_Test

irecv_wait.c

```
if (rank == 0) {
    int x = 5;
    sleep(5);
    MPI_Send(&x, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
}
else if (rank == 1) {
    MPI_Request req;
    MPI_Status stat;

    int y;
    MPI_Irecv(&y, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, &req);
    /* do something useful here */
    MPI_Wait(&req, &stat);

    printf("y = %d\n", y);
}
```

- `MPI_Irecv` can match a `MPI_Send`
  - It does not have to be `MPI_Isend`
- `MPI_Wait` is blocking, like `MPI_Recv`
  - What the calling thread does while waiting depends on the MPI implementation
    - In turn, this can depend on the underlying communication channel (shared memory, sockets, HPC interconnect)
  - Similarly to OpenMP, it can sleep or spin (poll)

MPI_Wait and MPI_Test

- Replacing `MPI_Wait` with `MPI_Test`

irecv_test.c

```
if (rank == 0) {
    int x = 5;
    sleep(5);
    MPI_Send(&x, 1, MPI_INT, 1, tag1, MPI_COMM_WORLD);
}
else if (rank == 1) {
    MPI_Request req;
    MPI_Status stat;

    int y;
    MPI_Irecv(&y, 1, MPI_INT, 0, tag1, MPI_COMM_WORLD, &req);

    int flag = 0;
    while (1) {
        MPI_Test(&req, &flag, &stat);
        if (flag == 1) {
            break;
        } else {
            sleep(1); /* do nothing, do something useful or sleep for a while */
            printf("still here...\n");
        }
    }
    printf("y = %d\n", y);
}
```

But WHY?

- Because we can decide what a waiting thread does!
  - e.g. to avoid spinning on the CPU if the MPI library does that for `MPI_Recv` and `MPI_Wait`, …
  - instead of releasing the CPU to the another thread or process

Neighbor communication: Example 1

- Each process sends its rank to the previous and next process (left and right neighbor)

```
int main(int argc, char *argv[])  {
 int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
 MPI_Request reqs[4];
 MPI_Status stats[4];

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 prev = rank-1;
 next = rank+1;
 if (rank == 0)  prev = numtasks - 1;
 if (rank == (numtasks - 1))  next = 0;

 MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[0]);
 MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[1]);

 MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[2]);
 MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[3]);

 { /* do some work */ }

 MPI_Waitall(4, reqs, stats);

 MPI_Finalize();
 return 0;
}
```

Correct code but not the best approach

Neighbor communication: Example 2

- Reordering of send and receive operations

```
int main(int argc, char *argv[])  {
 int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
 MPI_Request reqs[4];
 MPI_Status stats[4];

 MPI_Init(&argc,&argv);
 MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

 prev = rank-1;
 next = rank+1;
 if (rank == 0)  prev = numtasks - 1;
 if (rank == (numtasks - 1))  next = 0;

 MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[2]);
 MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[3]);

 MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[0]);
 MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[1]);

 { /* do some work */ }

 MPI_Waitall(4, reqs, stats);

 MPI_Finalize();
 return 0;
}
```

Best approach

(Repeated) take-home message:
Recv's must be posted before Send's!

V. Back to MPI Collective Operations

- Measuring the runtime overhead of MPI_Barrier()

barrier_bench.c

```
#define NITERS 10000
int main(int argc, char *argv[])
{
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  const int n_iters = NITERS;
  double t, elapsed;

  if (rank == 0) printf("Entering MPI_Barrier() benchmark\n");

  MPI_Barrier(MPI_COMM_WORLD);

  t = MPI_Wtime();
  for (int i=0; i<n_iters; i++) MPI_Barrier(MPI_COMM_WORLD);
  elapsed = MPI_Wtime() - t;

  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0) printf("Average Barrier Latency = %lf usec\n",
                   (elapsed/(double)n_iters)*1000.0*1000.0);

  MPI_Finalize();
  return 0;
}
```

```
$ cat script
mpirun -bynode -n 2  ./barrier > output1.txt
$ cat output1.txt | grep Lat
Average Barrier Latency = 2.040601 usec
```

```
$ cat script
mpirun -n 2  ./barrier > output1n.txt
$ cat output1n.txt | grep Lat
Average Barrier Latency = 0.468111 usec
```

- What is the complexity of MPI_Barrier() with respect to number of ranks (P)? How is it implemented?

Looking into the MPI library

- mpich-3.2/src/mpi/coll/barrier.c

```
int MPIR_Barrier_intra( MPID_Comm *comm_ptr, MPIR_Errflag_t *errflag )
{
    int size, rank, src, dst, mask, mpi_errno=MPI_SUCCESS;
    int mpi_errno_ret = MPI_SUCCESS;

    /* Only one collective operation per communicator can be active at any time */
    MPIDU_ERR_CHECK_MULTIPLE_THREADS_ENTER( comm_ptr );

    size = comm_ptr->local_size;
    /* Trivial barriers return immediately */
    if (size == 1) goto fn_exit;

    if (MPIR_CVAR_ENABLE_SMP_COLLECTIVES && MPIR_CVAR_ENABLE_SMP_BARRIER &&
        MPIR_Comm_is_node_aware(comm_ptr)) {
        mpi_errno = barrier_smp_intra(comm_ptr, errflag);
        if (mpi_errno) {
            /* for communication errors, just record the error but continue */
            *errflag = MPIR_ERR_GET_CLASS(mpi_errno);
            MPIR_ERR_SET(mpi_errno, *errflag, "**fail");
            MPIR_ERR_ADD(mpi_errno_ret, mpi_errno);
        }
        goto fn_exit;
    }

    rank = comm_ptr->rank;
    mask = 0x1;
    while (mask < size) {
        dst = (rank + mask) % size;
        src = (rank - mask + size) % size;
        mpi_errno = MPIC_Sendrecv(NULL, 0, MPI_BYTE, dst,
                                  MPIR_BARRIER_TAG, NULL, 0, MPI_BYTE,
                                  src, MPIR_BARRIER_TAG, comm_ptr,
                                  MPI_STATUS_IGNORE, errflag);
        if (mpi_errno) {
            /* for communication errors, just record the error but continue */
            *errflag = MPIR_ERR_GET_CLASS(mpi_errno);
            MPIR_ERR_SET(mpi_errno, *errflag, "**fail");
            MPIR_ERR_ADD(mpi_errno_ret, mpi_errno);
        }
        mask <<= 1;
    }
 fn_exit:
    MPIDU_ERR_CHECK_MULTIPLE_THREADS_EXIT( comm_ptr );
    if (mpi_errno_ret)
        mpi_errno = mpi_errno_ret;
    else if (*errflag != MPIR_ERR_NONE)
        MPIR_ERR_SET(mpi_errno, *errflag, "**coll_fail");
    return mpi_errno;
 fn_fail:
    goto fn_exit;
}
```

mask < size

mask = mask * 2;

We need to analyze this while loop to reach the take-home message

L = loop iterations
mask = 2^L =>
max(mask)=size =>
2^max(L) = size =>
max(L) = log(size)

Take-home message:
complexity of MPI_Barrier = O(log(size))

MPI_Bcast: Example

- The master rank (0) broadcasts a single value several times

bcast1.c

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

int main(int argc, char *argv[])
{
   int size,  rank;
   double data;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   srand48(rank);

   for (int k = 0; k < 10; k++) {
      if (!rank) data = drand48();

      MPI_Bcast(&data, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      /*if (rank)*/
      sleep(1);
      printf("Step %d: I am Process %d Data = %f\n", k, rank, data);
   }

  MPI_Finalize();
  return 0;
}
```

What can happen if I activate this?

Rank 0 prints first all the messages while the other ranks sleep.
Why the processes are not synchronized?

MPI_Bcast: simple approach

- Implementing MPI_Bcast with linear complexity O(size)

bcast2.c

```
int main(int argc, char *argv[])
{
   int rank, size;
   double data;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   srand48(rank);

   for (int k = 0; k < 10; k++) {
      if (!rank) data = drand48();

      if (!rank) {
         for (int i = 1; i < size; i++)
            MPI_Send(&data, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
      } else {
          MPI_Recv(&data, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      }
      if (rank) sleep(1);
      printf("Step %d: I am Process %d Data = %f\n", k, rank, data);
   }

  MPI_Finalize();
  return 0;
}
```

For small messages, MPI_Send is not synchronous

Take-home message:
Collective operations != Execution barrier

MPI_Bcast: Discussion

- Describe and identify the performance issue in the following code.
- Goal: show the importance of non-blocking communication

bcast3.c

```
int main(int argc, char *argv[])
{
   int rank, size;
   double data;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   srand48(rank);

   for (int k = 0; k < 10; k++) {
      if (!rank) data = drand48();

      if (!rank) {
         for (int i = 1; i < size; i++)
            MPI_Ssend(&data, 1, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
      }
      else {
           sleep(lrand48()%5); // work that lasts up to 5 seconds (it can be after MPI_Recv)
           MPI_Recv(&data, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
     }
      printf("Step %d: I am Process %d Data = %f\n", k, rank, data); fflush(0);
   }

  MPI_Finalize();
  return 0;
}
```