Παράλληλη Επεξεργασία

Εαρινό Εξάμηνο 2023-24
«MPI Programming Model – III»

Παναγιώτης Χατζηδούκας, Ευστράτιος Γαλλόπουλος

Schedule and Goals

- MPI - part 3
  - uncovered topics on MPI communication

Outline

- Send modes revisited
- Non-blocking neighboring communication
- More on collective operations
  - Broadcast of multiple values
  - `MPI_Alltoall`: good to know, hopefully not needed
  - `MPI_Scan`, `MPI_Exscan`

I. Blocking and non-blocking sends

- blocking: return only when the buffer is ready to be reused
- non-blocking: return immediately

```
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
```

```
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)
```

MPI's Send modes

- Represent different choices of
  - buffering (where is the data kept until it is received)
  - synchronization (when does a send complete)

- `MPI_Send`: will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).
- `MPI_Bsend`: May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.
- `MPI_Ssend`: will not return until matching receive posted
- `MPI_Rsend`: May be used ONLY if matching receive already posted. User responsible for writing a correct program.

Source: http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

MPI's Send modes

- `MPI_Isend`: Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received. An immediate send must return to the user without requiring a matching receive at the destination.
- `MPI_Ibsend`: buffered nonblocking
- `MPI_Issend`: Synchronous nonblocking. Note that a Wait/Test will complete only when the matching receive is posted.
- `MPI_Irsend`: As with MPI_Rsend, but nonblocking.

Source: http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

MPI's Send modes - Recommendations

- The best performance is likely if you can write your program so that you could use just `MPI_Ssend`; in that case, an MPI implementation can completely avoid buffering data.
- Use `MPI_Send` instead; this allows the MPI implementation the maximum flexibility in choosing how to deliver your data.
- If nonblocking routines are necessary, then try to use `MPI_Isend` or `MPI_Irecv`.
- Use `MPI_Bsend` only when it is too inconvenient to use `MPI_Isend`.
- The remaining routines, `MPI_Rsend`, `MPI_Issend`, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.

Source: http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

II. Non-blocking neighbor communication

- Periodic domain

```
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;
}
```

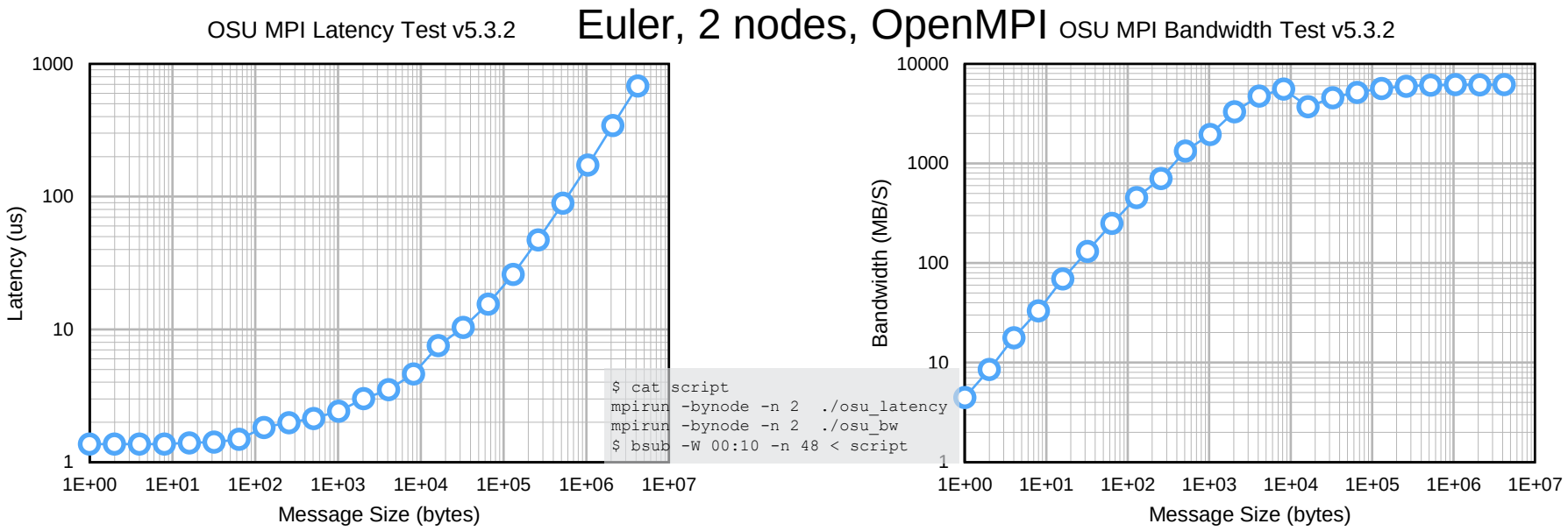
II. Non-blocking neighbor communication

- Non-periodic domain and use of MPI_PROC_NULL

```
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 = MPI_PROC_NULL;
 if (rank == (numtasks - 1))  next = MPI_PROC_NULL;

 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;
}
```

OSU Benchmarks

- http://mvapich.cse.ohio-state.edu/benchmarks/



How do these numbers relate to Computation-Transfer (CT) overlap?

- Based on the total message size exchanged between two neighboring ranks for the ghost data, we can estimate the required communication time ($T_{transfer}$).
- We can also measure the time for processing the inner domain ($T_{computation}$)
- For perfect CT overlap: $T_{transfer} < T_{computation}$
  - the time of `MPI_Waitall` should be almost zero!

III. More on collective operations

- Parallelizing Simpson integration: only the master rank (0) reads the input data. How do we share it?

```
int main(int argc, char** argv)
{
  MPI_Init(&argc,&argv);

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

  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration

  // read the parameters on the master rank
  if (rank==0)
    std::cin >> a >> b >> nsteps;

  // we need to share the parameters with the other ranks
  ???

  // integrate just one part on each thread
  double delta = (b-a)/size;
  double result = simpson(func,a+rank*delta,a+(rank+1)*delta,nsteps/size);

  //  collect all to the master (rank 0)
  MPI_Reduce(rank == 0 ? MPI_IN_PLACE : &result, &result, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   // the master prints
  if (rank==0)
    std::cout << result << std::endl;

  MPI_Finalize();
  return 0;
}
```

Broadcast of multiple values

- We can use this to broadcast the data

```
// and then broadcast the parameters to the other ranks
 MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
 MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
 MPI_Bcast(&nsteps, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

- This is inefficient since we use three broadcasts.
- "Ugly" solution: change `nsteps` to double and broadcast an array of 3 doubles
- Another solution: pack it all into a struct and send it bitwise

```
// define a struct for the parameters
struct parms {
  double a;            // lower bound of integration
  double b;            // upper bound of integration
  int nsteps; // number of subintervals for integration
};

parms p;

// read the parameters on the master rank
if (rank==0)
  std::cin >> p.a >> p.b >> p.nsteps;

// broadcast the parms as bytes - warning, not portable on heterogeneous machines
MPI_Bcast(&p, sizeof(parms), MPI_BYTE, 0, MPI_COMM_WORLD);
```

- Can be dangerous because it assumes a homogeneous cluster with identical integer and floating point formats.

Packing and unpacking

- Allocate a sufficiently large buffer and then pack the data into it
- Send/receive the packed buffer with type `MPI_PACKED`
- Finally unpack it on the receiving side

```
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype,
             void *outbuf, int outcount, int *position, MPI_Comm comm)
// packs the data given as input into the outbuf buffer starting at a given position.
// outcount is the size of the buffer and position gets updated to point to the first
// free byte after packing in the data.
// An error is returned if the buffer is too small.

int MPI_Unpack(void *inbuf, int insize, int *position,
               void *outbuf, int outcount, MPI_Datatype datatype, MPI_Comm comm)
// unpack data from the buffer starting at given position into the buffer outbuf.
// position is updated to point to the location after the last byte read

int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int *size)
// returns in size an upper bound for the number of bytes needed to pack incount
// values of type datatype. This can be used to determine the required buffer size
```

Packing data into a buffer

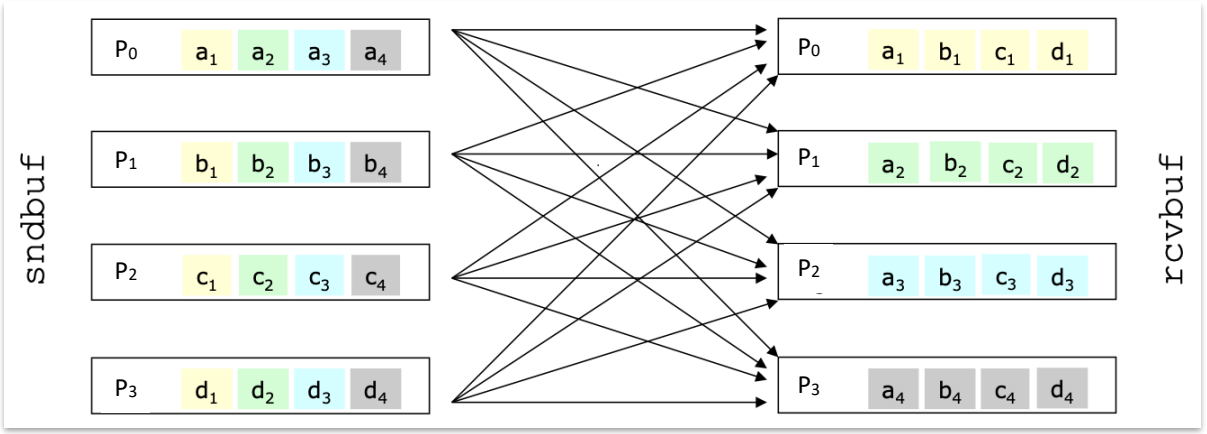
- Pack the input data, broadcast it and unpack

```
// create a buffer and pack the values.
// first get the size for the buffer and allocate a buffer
int size_double, size_int;
MPI_Pack_size(1, MPI_DOUBLE, MPI_COMM_WORLD,&size_double);
MPI_Pack_size(1, MPI_INT,    MPI_COMM_WORLD,&size_int);
int buffer_size = 2*size_double+size_int;
char* buffer = new char[buffer_size];

// pack the values into the buffer on the master
if (rank==0) {
  int pos=0;
  MPI_Pack(&a, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  MPI_Pack(&b, 1, MPI_DOUBLE, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  MPI_Pack(&nsteps, 1, MPI_INT, buffer, buffer_size, &pos, MPI_COMM_WORLD);
  assert ( pos <= buffer_size );
}

// broadcast the buffer
MPI_Bcast(buffer, buffer_size, MPI_PACKED, 0, MPI_COMM_WORLD);

// and unpack on the receiving side
int pos=0;
MPI_Unpack(buffer, buffer_size, &pos, &a, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &b, 1, MPI_DOUBLE, MPI_COMM_WORLD);
MPI_Unpack(buffer, buffer_size, &pos, &nsteps, 1, MPI_INT, MPI_COMM_WORLD);
assert ( pos <= buffer_size );

// and finally delete the buffer
delete[] buffer;
```

All-to-all

- `MPI_Alltoall`: *n*-th rank sends *k*-th portion of its data to rank *k* and receives *n*-th portion from node *k*.
  - Everyone scatters and gathers at the same time
  - like a matrix transpose. Attention: slow!

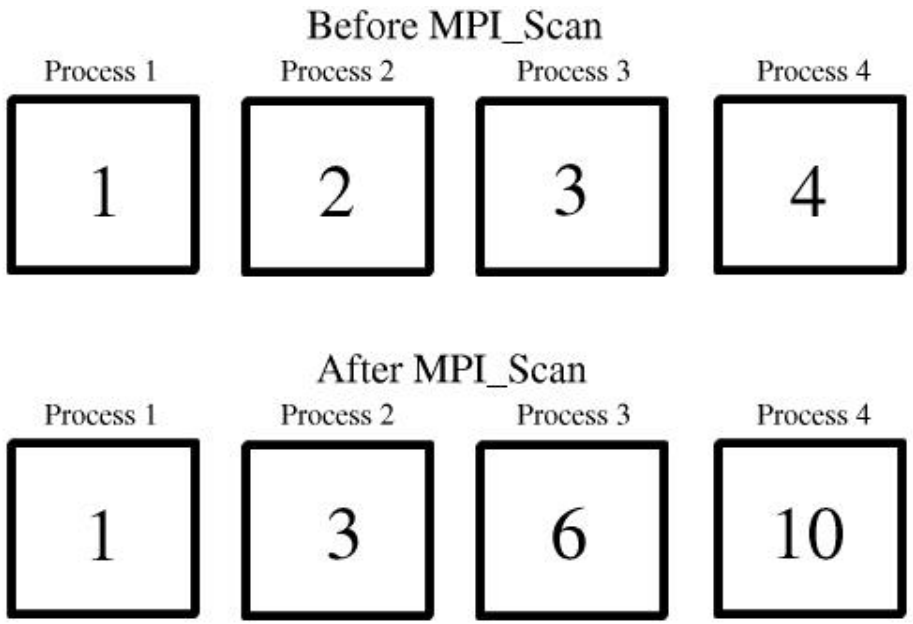
```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                 void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```



MPI_Scan + MPI_Exscan

- A scan or prefix-reduction operation performs partial reductions on distributed data.

```
int MPI_Scan(const void *sbuf, void *rbuf, int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm)
// Computes the scan (partial reductions) of data on a collection of processes

int MPI_Exscan(const void *sbuf, void *rbuf, int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm)
// Computes the exclusive scan (partial reductions) of data on a collection of processes
// MPI_Exscan is like MPI_Scan, except that the contribution from the calling process is
// not included in the result at the calling process (it is contributed to the subsequent
// processes, of course).
```



MPI_Scan + MPI_Exscan

- Example code

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

int main(int argc, char *argv[])
  int rank;

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

  double indata = 10.0*(rank+1);
  double outdata = 0;

  MPI_Scan(&indata, &outdata, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  //MPI_Exscan(&indata, &outdata, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

  printf("process %d: %f -> %f\n", rank, indata, outdata);

  MPI_Finalize();

  return 0;
}
```

```
$ mpirun -n 4 ./scan
process 0: 10.000000 -> 10.000000
process 1: 20.000000 -> 30.000000
process 2: 30.000000 -> 60.000000
process 3: 40.000000 -> 100.000000
```

```
$ mpirun -n 4 ./exscan
process 0: 10.000000 -> 0.000000
process 1: 20.000000 -> 10.000000
process 2: 30.000000 -> 30.000000
process 3: 40.000000 -> 60.000000
```