Durham University

PHYS52015 Core Ib: Introduction to High Performance Computing (HPC)

Session VI: Non-blocking P2P communication

Christopher Marcotte

Michaelmas term 2023

Outline

Nonblocking point to point communication
Collective operations

`pollev.com/christophermarcotte820`

Buffers

MPI distinguishes different types of buffers:

- variables
- user-level buffers
- hardware/system buffers

MPI implementations are excellent in tuning communication, i.e. avoid copying, but we have to assume that a message runs through all buffers, then through the network, and then bottom-up through all buffers again. This means that Send and Recv are expensive operations.
Even worse, two concurrent sends might deadlock (but only for massive message counts or extremely large messages).

Buffers

MPI distinguishes different types of buffers:

- variables
- user-level buffers
- hardware/system buffers

MPI implementations are excellent in tuning communication, i.e. avoid copying, but we have to assume that a message runs through all buffers, then through the network, and then bottom-up through all buffers again. This means that Send and Recv are expensive operations.
Even worse, two concurrent sends might deadlock (but only for massive message counts or extremely large messages).
$\Rightarrow$ One way to deal with this is to allow MPI to optimize the messaging by giving both Send and Recv commands simultaneously — this is a `MPI_Sendrecv`.

Sendrecv

```
int MPI_Sendrecv(
  const void *sendbuf, int sendcount,
  MPI_Datatype sendtype,
  int dest, int sendtag,
  void *recvbuf, int recvcount,
  MPI_Datatype recvtype,
  int source, int recvtag,
  MPI_Comm comm, MPI_Status *status
)
```

- Shortcut for send followed by receive
- Allows MPI to optimise aggressively
- Anticipates that many applications have dedicated compute and data exchange phases

$\Rightarrow$ Does not really solve our efficiency concerns, just weaken them

MPI_Sendrecv example

We have a program which sends an nentries-length buffer between two processes:

```
if (rank == 0) {
  MPI_Send(sendbuf, nentries, MPI_INT, 1, 0, ...);
  MPI_Recv(recvbuf, nentries, MPI_INT, 1, 0, ...);
} else if (rank == 1) {
  MPI_Send(sendbuf, nentries, MPI_INT, 0, 0, ...);
  MPI_Recv(recvbuf, nentries, MPI_INT, 0, 0, ...);
}
```

- Recall that MPI_Send behaves like MPI_Bsend when buffer space is available, and then behaves like MPI_Ssend when it is not.

MPI_Sendrecv example

We have a program which sends an `nentries`-length buffer between two processes:

```
if (rank == 0) {
  MPI_Send(sendbuf, nentries, MPI_INT, 1, 0, ...);
  MPI_Recv(recvbuf, nentries, MPI_INT, 1, 0, ...);
} else if (rank == 1) {
  MPI_Send(sendbuf, nentries, MPI_INT, 0, 0, ...);
  MPI_Recv(recvbuf, nentries, MPI_INT, 0, 0, ...);
}
```

- Recall that `MPI_Send` behaves like `MPI_Bsend` when buffer space is available, and then behaves like `MPI_Ssend` when it is not.

```
if (rank == 0) {
  MPI_Sendrecv(sendbuf, nentries, MPI_INT, 1, 0, /* Send */
               recvbuf, nentries, MPI_INT, 1, 0, /* Recv */ ...);
} else if (rank == 1) {
  MPI_Sendrecv(sendbuf, nentries, MPI_INT, 0, 0, /* Send */
               recvbuf, nentries, MPI_INT, 0, 0, /* Recv */ ...);
}
```

Nonblocking P2P communication

- Non-blocking commands start with I (immediate return, e.g.)
- Non-blocking means that operation returns immediately though MPI might not have transferred data (might not even have started)
- Buffer thus is still in use and we may not overwrite it
- We explicitly have to validate whether message transfer has completed before we reuse or delete the buffer

```
// Create helper variable (handle)
int a = 1;
// trigger the send
// do some work
// check whether communication has completed
a = 2;
...
```

$\Rightarrow$ We now can overlap communication and computation.

Why non-blocking. . . ?

- Added flexibility of separating posting messages from receiving them.

$\Rightarrow$ MPI libraries often have optimisations to complete sends quickly if the matching receive already exists.

- Sending many messages to one process, which receives them all. . .

Why non-blocking. . . ?

- Added flexibility of separating posting messages from receiving them.

⇒ MPI libraries often have optimisations to complete sends quickly if the matching receive already exists.

- Sending many messages to one process, which receives them all. . .

```
int buf1, buf2;
// ...
for (int k=0; k<100; k++){
    if (rank==0){
            MPI_Recv(&buf2, 1, MPI_INT, k, k, COMM, &status);
    }else{
            MPI_Send(&buf1, 1, MPI_INT, 0, k, COMM);
    }
}
```

- The receiving process waits on each `MPI_Recv` before moving on, because it is blocking.
- If we used a non-blocking `MPI_Irecv`, then all can complete as each `MPI_Send` arrives and we just need to `MPI_Wait` for the results.

`Isend` & `Irecv`

- Non-blocking variants of `MPI_Send` and `MPI_Recv`
- Returns immediately, but *buffer is not safe to reuse*

```
int MPI_Isend(const void *buffer, int count, MPI_Datatype dtype,
        int dest, int tag, MPI_Comm comm, MPI_Request *request);
int MPI_Irecv(void *buffer, int count, MPI_Datatype dtype,
        int dest, int tag, MPI_Comm comm, MPI_Request *request);
```

- Note the `request` in the send, and the lack of `status` in recv
- We need to process that `request` before we can reuse the buffers

Isend

```
int MPI_Send(const void *buffer, int count, MPI_Datatype datatype,
        int dest, int tag, MPI_Comm comm
);

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

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

- Pass additional pointer to object of type `MPI_Request`.
- Non-blocking, i.e. operation returns immediately.
- Check for send completition with `MPI_Wait` or `MPI_Test`.
- `MPI_Irecv` analogous.
- The status object is not required for the receive process, as we have to hand it over to wait or test later.

Testing

If we have a request, we can check whether the message it corresponds to has been completed with `MPI_Test`:

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

Testing

If we have a request, we can check whether the message it corresponds to has been completed with `MPI_Test`!

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
```

- `flag` will be true (an `int` of value `1`) if the provided request has been completed, and false otherwise.
- If we don't want to test for completion, we can instead `MPI_Wait`...

Waiting…

```
MPI_Request request1, request2;
MPI_Status status;
int buffer1[10];   int buffer2[10];

MPI_Send(buffer1, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
MPI_Recv(buffer2, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
buffer2[0] = 0;

MPI_Isend(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD, &request1);
MPI_Irecv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &request2);
buffer1[0] = 0;
```

There is an error in this code, what change do we need to make for it to be correct?

Waiting...

```
MPI_Request request1, request2;
MPI_Status status;
int buffer1[10];   int buffer2[10];

MPI_Send(buffer1, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
MPI_Recv(buffer2, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
buffer2[0] = 0;

MPI_Isend(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD, &request1);
MPI_Irecv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &request2);
buffer1[0] = 0;
```

There is an error in this code, what change do we need to make for it to be correct? Before `buffer1[0] = 0;`:

```
MPI_Wait(&request1, &status);
MPI_Wait(&request2, &status);
```

P2P communication in action

```c
MPI_Request request1, request2;
MPI_Status status;
int buffer1[10];   int buffer2[10];
// Variant A
MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
// Variant B
//MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
//MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
// Variant C
//MPI_Irecv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &request1);
//MPI_Isend(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD, &request2);
//MPI_Wait(&request1, &status);
//MPI_Wait(&request2, &status);
```

- Does Variant A deadlock?

P2P communication in action

```
MPI_Request request1, request2;
MPI_Status status;
int buffer1[10];   int buffer2[10];
// Variant A
//MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
//MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
// Variant B
MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
// Variant C
//MPI_Irecv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &request1);
//MPI_Isend(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD, &request2);
//MPI_Wait(&request1, &status);
//MPI_Wait(&request2, &status);
```

- Does Variant A deadlock? *Yes!* `MPI_Recv` is always blocking.
- Does Variant B deadlock?

P2P communication in action

```
MPI_Request request1, request2;
MPI_Status status;
int buffer1[10];   int buffer2[10];
// Variant A
//MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
//MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
// Variant B
//MPI_Send(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD);
//MPI_Recv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &status);
// Variant C
MPI_Irecv(buffer1, 10, MPI_INT, left, 0, MPI_COMM_WORLD, &request1);
MPI_Isend(buffer2, 10, MPI_INT, right, 0, MPI_COMM_WORLD, &request2);
MPI_Wait(&request1, &status);
MPI_Wait(&request2, &status);
```

- Does Variant A deadlock? *Yes!* `MPI_Recv` is always blocking.
- Does Variant B deadlock? Not for only 10 integers (if not too many messages sent before).
- Does Variant C deadlock? Is it correct? Is it fast? May we add additional operations before the first wait?

Concept of building block

- Content
  - Introduce `sendrecv`
  - Introduce concept of non-blocking communication
  - Study variants of P2P communication w.r.t. blocking and call order
- Expected Learning Outcomes
  - The student knows difference of blocking and non-blocking operations
  - The student can explain the idea of non-blocking communication
  - The student can write MPI code where communication and computation overlap

Definition: collective

> Collective operation: A collective (MPI) operation is an operation involving many/all nodes/ranks.

- In MPI, a collective operation involves all ranks of one communicator (introduced later)
- For `MPI_COMM_WORLD`, a collective operation involves all ranks
- Collectives are blocking (though newer ($\geq$3.1) MPI standard introduces non-blocking collectives)
- Blocking collectives always synchronise all ranks, i.e. all ranks have to enter the same collective instruction before any rank proceeds

A (manual) collective

```
double a;

if (myrank==0) {
  for (int i=1; i<mysize; i++) {
    double tmp;
    MPI_Recv(&tmp,1,MPI_DOUBLE, ...);
    a+=tmp;
  }
}
else {
  MPI_Send(&a,1,MPI_DOUBLE,0, ...);
}
```

What type of collective operation is realised here?

A (manual) collective

```
double a;

if (myrank==0) {
  for (int i=1; i<mysize; i++) {
    double tmp;
    MPI_Recv(&tmp,1,MPI_DOUBLE, ...);
    a+=tmp;
  }
}
else {
  MPI_Send(&a,1,MPI_DOUBLE,0, ...);
}
```

What type of collective operation is realised here?

```
double globalSum;
MPI_Reduce(&a, &globalSum, 1,
  MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Flavours of collective operations in MPI

Type of collective	One-to-all	All-to-one	All-to-all
Synchronisation			
Communication			
Computation			

Insert the following MPI operations into the table (MPI prefix and signature neglected):

- Barrier
- Broadcast
- Reduce
- Allgather
- Scatter
- Gather
- Allreduce

$\Rightarrow$ Synchronisation as discussed is simplest kind of collective operation

Flavours of collective operations in MPI

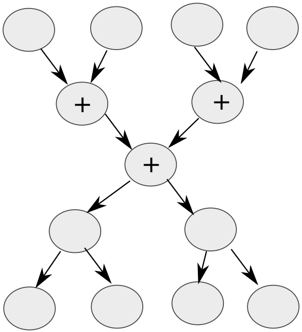
Type of collective	One-to-all	All-to-one	All-to-all
Synchronisation	Barrier		
Communication	Broadcast, Scatter	Gather	Allgather
Computation		Reduce	Allreduce

Insert the following MPI operations into the table (MPI prefix and signature neglected):

- Barrier
- Broadcast
- Reduce
- Allgather
- Scatter
- Gather
- Allreduce

$\Rightarrow$ Synchronisation as discussed is simplest kind of collective operation

Good reasons to use MPI's collective



- Simplicity of code
- Performance through specialised implementations
- Support through dedicated hardware (cf. BlueGene's three network topologies: clique, fat tree, ring)

MPI_Barrier

- Simplest form of collective operation — synchronization of all ranks in `comm`.
- Rarely used:

⇒ `MPI_Barrier` doesn't synchronize non-blocking calls

⇒ Really meant for telling MPI about calls *outside* MPI, like IO

```
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
for ( int ii = 0; ii < size; ++ii ) {
    if ( rank == ii ) {
        // my turn to write to the file
        writeStuffToTheFile();
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

MPI_Bcast & MPI_Scatter

- MPI_Bcast sends the contents of a buffer from root to all other processes.
- MPI_Scatter sends *parts* of a buffer from root to different processes.
- MPI_Bcast is the inverse of MPI_Reduce
- MPI_Scatter is the inverse of MPI_Gather

```
MPI_Comm comm;
int array[100];
int root=0;
//...
MPI_Bcast(array, 100, MPI_INT, root, comm);

int gsize, *sendbuf;
int rbuf[100];
//...
MPI_Comm_size(comm, &gsize);
sendbuf = (int *)malloc(gsize*100*sizeof(int));
//...
MPI_Scatter(sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

MPI_Reduce & MPI_Gather

- MPI_Reduce reduces a value across ranks to a single value on root using a prescribed reduction operator.
- MPI_Gather concatenates the array pieces from all processes onto the root process.

```
MPI_Comm comm;
int sum, root=0;
//...
MPI_Reduce(sum, c, 1, MPI_INT, MPI_SUM, root, comm);

int gsize,sendarray[100];
int myrank, *rbuf;
//...
MPI_Comm_rank(comm, &myrank);
if (myrank == root) {
        MPI_Comm_size(comm, &gsize);
        rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

MPI_Allgather & MPI_Allreduce

- MPI_Allgather is an MPI_Gather which concatenates the array pieces on all processes.
- MPI_Allreduce is an MPI_Reduce which reduces on all processes.

```
MPI_Comm comm;
int sum, root=0;
//...
MPI_Allreduce(sum, c, 1, MPI_INT, MPI_SUM, comm);

int gsize,sendarray[100];
int *rbuf;
//...
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*100*sizeof(int));
MPI_Allgather(sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, comm);
```

- MPI_Allgather is used a lot (for better or worse) debugging distributed arrays — serial checks work on one process!
- MPI_Allreduce Is particularly useful for *convergence* checks — we will see this when we return to the Jacobi iteration problem in MPI.

Concept of building block

- Content
  - Classify different collectives from MPI
  - Understand use cases for main forms of MPI collectives
- Expected Learning Outcomes
  - The student knows which type of collectives do exist (*)
  - The student can explain what collectives do (*)
  - The student can identify collective code fragments (*)
  - The student can use collectives or implement them manually