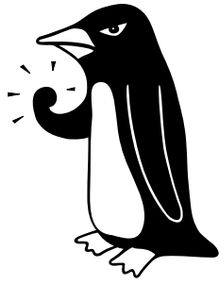
Point to Point Communication



Warwick RSE

Overview

- Most common form of MPI communication
- Relies on one processor (the **source**) sending a message to another processor (the **receiver**)
- The receiver has to post a *matching* receive
- We'll come to what is meant by matching a bit later

Deadlocks

- The most common problem in MPI programming is **deadlocking**
- This is where a send happens without a matching receive
- OR a receive happens for a message that is never sent
- This is because the default MPI send and receive commands are **blocking**
    - They don’t return control until they have completed (sort of, we’ll come back to that)

Sending

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

- `buf` - a buffer containing the data
- `count` - Number of elements to send
- `datatype` - Type of elements to send
- `dest` - **rank** of the **receiver** process
- `tag` - integer code that identifies this message. Has to match in receive
- `comm` - communicator

Receiving

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

- `buf` - buffer to hold the received data
- `count` - Number of elements to receive
- `datatype` - Type of elements to receive
- `source` - **Rank** of **source**
- `tag` - integer code that identifies this message. Has to match in send
- `comm` - Communicator
- `status` - object containing information about the message
  - INTEGER, DIMENSION(MPI_STATUS_SIZE) in Fortran

Matching sends and receives

- Matching sends and receives have three properties

- The **dest** parameter to **MPI_Send** is the rank of the receiver

- The **source** parameter to **MPI_Recv** is the rank of the sender

- The **tag** to both **MPI_Send** and **MPI_Recv** is the same

Matching sends and receives

- There are special values for **source** and **tag** in **MPI_Recv**

- **MPI_ANY_SOURCE** means that a message from any source will be accepted

- **MPI_ANY_TAG** means that a message with any tag will be accepted

- You usually want to have either the tag or the source be fixed or it’s hard to tell messages apart

Simple example

- The simplest example of MPI point to point communications is what is called a ring pass

- Each processor sends it’s rank to it’s neighbour that then prints what it has received

- Doesn’t have much in common with real MPI codes

  - Also has some … interesting pathologies

- Pretty much the standard, so let’s go with it

Simple code

```
int main(int argc, char **argv){
  int rank, nproc, rank_right, rank_left, rank_recv;
  MPI_Status stat;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  rank_left = rank -1;
  /*Ranks run from 0 to nproc-1, so wrap the ends around to make a loop*/
  if(rank_left == -1) rank_left = nproc-1;
  rank_right = rank + 1;
  if(rank_right == nproc) rank_right = 0;

  MPI_Send(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
  MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100, MPI_COMM_WORLD, &stat);

  printf("Rank %i has received value %i from rank %i\n", rank, rank_recv,
      rank_left);

  MPI_Finalize();
}
```

Simple code

```
Rank 5 has received value 4 from rank 4
Rank 1 has received value 0 from rank 0
Rank 2 has received value 1 from rank 1
Rank 3 has received value 2 from rank 2
Rank 4 has received value 3 from rank 3
Rank 14 has received value 13 from rank 13
Rank 0 has received value 15 from rank 15
Rank 6 has received value 5 from rank 5
Rank 7 has received value 6 from rank 6
Rank 8 has received value 7 from rank 7
Rank 9 has received value 8 from rank 8
Rank 12 has received value 11 from rank 11
Rank 13 has received value 12 from rank 12
Rank 15 has received value 14 from rank 14
Rank 10 has received value 9 from rank 9
Rank 11 has received value 10 from rank 10
```

Wait a minute!

- All processors print what you'd expect
- **BUT If you run that code you might get a shock**
- On most computers it will run and work as expected
- But if you go to a cluster it'll deadlock (usually)
- Why?

Simple code

```
int main(int argc, char **argv){
  int rank, nproc, rank_right, rank_left, rank_recv;
  MPI_Status stat;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nproc);

  rank_left = rank -1;
  /*Ranks run from 0 to nproc-1, so wrap the ends around to make a loop*/
  if(rank_left == -1) rank_left = nproc-1;
  rank_right = rank + 1;
  if(rank_right == nproc) rank_right = 0;

  MPI_Send(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
  MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100, MPI_COMM_WORLD, &stat);

  printf("Rank %i has received value %i from rank %i\n", rank, rank_recv,
      rank_left);

  MPI_Finalize();
}
```

Wait a minute!

- Sending and receiving are **blocking**
- So all processors get into the **send** command so the matching **receive** command never happens
- Deadlock is the correct outcome
- So why does it work on my laptop/desktop?

Wait a minute!

- **MPI_Send** actually isn't **required** to block until a message is received

  - It's just required to block *until you can reuse the* ***buf*** *variable again*

    - That can be when the data is copied by the MPI library into an internal buffer

    - But you can't count on that, so you shouldn't write a program that relies on that behaviour

- There is a variant **MPI_Ssend** that is guaranteed to block until the message is received. Try it in the example

Wait a minute!

- **MPI_Recv** equivalently only technically blocks until the **buf** variable contains the received value

  - That's the same as blocking until the receive has completed, which can't happen until the send has completed

  - You know that the send has completed by the time the matching receive has completed

- In most use cases you don't care when a send has completed, only that it has done so safely

Fixing the example

- There's lots of ways of fixing the code
  - Simplest is to get rank 0 to send and then receive
  - All other ranks receive and then send
  - Data travels as a "wave" through the processors
    - 0->1->2->…
  - Actually a very inefficient way of doing almost anything (see intermediate MPI course)

Fixed code

```
  MPI_Send(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
  MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100,
MPI_COMM_WORLD, &stat);
```

```
  if (rank == 0) {
    MPI_Ssend(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
    MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100, MPI_COMM_WORLD,
&stat);
  } else {
    MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100, MPI_COMM_WORLD,
&stat);
    MPI_Ssend(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
  }
```

Fixed code

```
Rank 1 has received value 0 from rank 0
Rank 2 has received value 1 from rank 1
Rank 4 has received value 3 from rank 3
Rank 5 has received value 4 from rank 4
Rank 6 has received value 5 from rank 5
Rank 7 has received value 6 from rank 6
Rank 3 has received value 2 from rank 2
Rank 8 has received value 7 from rank 7
Rank 9 has received value 8 from rank 8
Rank 10 has received value 9 from rank 9
Rank 11 has received value 10 from rank 10
Rank 12 has received value 11 from rank 11
Rank 13 has received value 12 from rank 12
Rank 14 has received value 13 from rank 13
Rank 0 has received value 15 from rank 15
Rank 15 has received value 14 from rank 14
```

Fixing the example

- This now works as expected
- Note the use of **MPI_Ssend**
- That ensures that there's no trickery, this will work on any machine

Fixing the example

- Note that the print statements are *nearly* in rank order
  - Much more nearly than for the “simple” code
- This is a feature of the “wave” propagating through the ranks
- This is what makes this approach poorly performing
- Rank **n+1** can’t do anything until rank **n** has finished

Fixing the example

- Note that it doesn't work on one processor!

- That's because rank 0 should both be sending and receiving at the same time

- Wanting to send in one direction while receiving in another is a very common thing to want to do in MPI codes

- There is a command to help!

- **MPI_Sendrecv**

MPI_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)
```

- **MPI_Send** and **MPI_Recv** glued together
- Doesn’t deadlock so long as both the send and the receive parts complete

Sendrecv code

```
  if (rank == 0) {
    MPI_Ssend(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
    MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100, MPI_COMM_WORLD,
&stat);
  } else {
    MPI_Recv(&rank_recv, 1, MPI_INT, rank_left, 100, MPI_COMM_WORLD,
&stat);
    MPI_Ssend(&rank, 1, MPI_INT, rank_right, 100, MPI_COMM_WORLD);
  }
```

```
  MPI_Sendrecv(&rank,      1, MPI_INT, rank_right, 100,
               &rank_recv,1, MPI_INT, rank_left , 100,
               MPI_COMM_WORLD, &stat);
```

Sendrecv code

```
Rank 1 has received value 0 from rank 0
Rank 7 has received value 6 from rank 6
Rank 12 has received value 11 from rank 11
Rank 13 has received value 12 from rank 12
Rank 2 has received value 1 from rank 1
Rank 3 has received value 2 from rank 2
Rank 4 has received value 3 from rank 3
Rank 8 has received value 7 from rank 7
Rank 9 has received value 8 from rank 8
Rank 11 has received value 10 from rank 10
Rank 14 has received value 13 from rank 13
Rank 15 has received value 14 from rank 14
Rank 0 has received value 15 from rank 15
Rank 5 has received value 4 from rank 4
Rank 6 has received value 5 from rank 5
Rank 10 has received value 9 from rank 9
```

Sendrecv code

- Code now works on 1 or more processors
- Output text is mixed up again
- No longer have the “wave” propagating through the processors
- Performance will be much better
- No risk of deadlocking
- This is the preferred solution for most programs that have this “send right, receive left” type behaviour

More than one item

- You will probably have spotted that in all of these examples I've only been sending a single item

- If you send more than one item then MPI simply copies them from memory contiguously

- If you want to send more than 1 item in a single message you have to hand MPI a 1D array

- If you want to send part of a 2D or 3D array then you have to copy it into a 1D array

- In Fortran array subsections work as expected

Recap

- Can now
  - Combine data from all processors
  - Send data from one processor to another
    - Deal with the simplest class of deadlock for "send right-receive left" problems
- That's actually all that you need to get started
- Next, onto how you do actual parallel code