THE UNIVERSITY OF WARWICK

High Performance Computing

Course Notes

Message Passing Programming III

Dr Ligang He

MPI functions

MPI is a complex system comprising of numerous functions with various parameters and variants

Six of them are indispensable, but can write a large number of useful programs already

Other functions add flexibility (datatype), **robustness (non-blocking send/receive)**, efficiency (ready-mode communication), **modularity (communicators, groups)** or **convenience** (**collective operations**, topology).

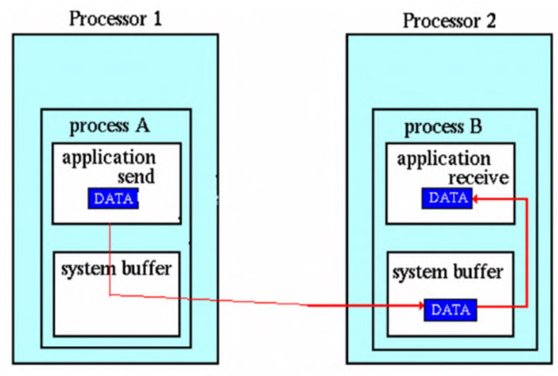
In the lectures, we are going to cover most commonly encountered functions

Communication mode

- The communication mode is only related to the send routines and the receive routine does not specify communication mode

- There are 4 communication modes
  - Standard mode
  - Synchronous mode
  - Buffered mode
  - Ready mode

Standard Mode



Path of a message buffered at the receiving process

- **The messages are handled in the standard way designed by the MPI implementation**

- **The messages may or may not be copied to system buffer depending on the MPI implementations**

- **Can have acceptable performance for all possible communication scenarios, but may not give the best performance in certain situations**

- **Blocking Standard send and non-blocking standard send**
  - Blocking send (MPI_Send): the routine returns after the data has been copied from application buffer to system buffer
  - Non-blocking send (MPI_Isend): the routine returns immediately

Standard Mode

- What happens next after the data has been copied to system buffer?
  - At the sender, the MPI system
    - sends a "ready to send" message to the receiver,
    - waits for a "ready to receive" message from the receiver,
    - Starts transferring the data after receiving the "ready to receive"
  - At the receiver, when the receive routine is called, the MPI system sends a "ready to receive" message to the sender

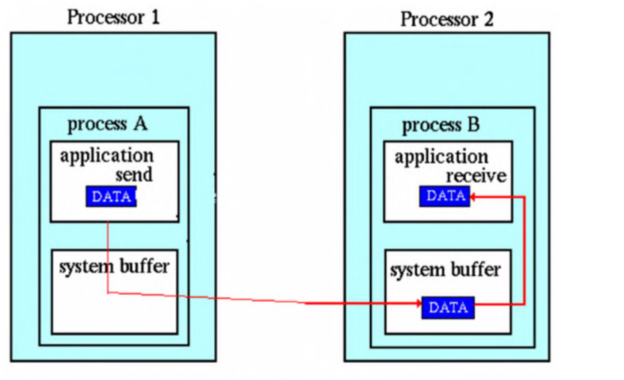
Standard Mode

– **What will happen if the message to be sent is bigger than the system buffer?**

– **The send routine will block until**

– **1) the receive routine starts receiving data and therefore empty the system buffer;**

– **2) the rest of the message has been copied to the system buffer.**



Path of a message buffered at the receiving process

Question 1

Process p0:

1. Call MPI_Send to send message A to p1;

2. Call MPI_Recv to receive message B to p1;

Process p1:

1. Call MPI_Send to send message B to p0;

2. Call MPI_Recv to receive message A to p0;

What will happen if the size of message A exceeds its system buffer?

Question 2

Process p0:

1. Call MPI_Send to send message A to p1;

2. Call MPI_Recv to receive message B to p1;

Process p1:

1. Call MPI_Send to send message B to p0;

2. Call MPI_Recv to receive message A to p0;

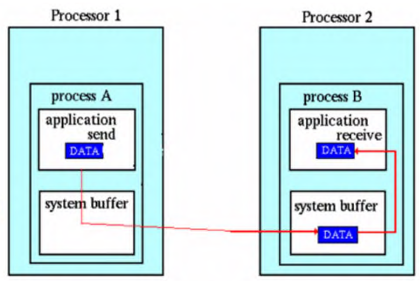
What will happen if the sizes of both message A and B exceed the system buffer?

Synchronous mode

- Blocking synchronous send: MPI_Ssend
  - The routine doesn’t return until 1) the data has been copied to system buffer and 2) the MPI system has received the “ready to receive” message
  - If the receive routine is posted later than the send routine, synchronization overhead is incurred

- Non-blocking synchronous send: MPI_Issend
  - The routine returns immediately
  - The communication is considered complete only after the data has been copied to system buffer and the MPI system has received the “ready to receive” message

Buffered mode



Path of a message buffered at the receiving process

–The sender uses the explicitly defined buffer instead of system buffer (the system buffer is limited)

–Needs to make sure the user-defined buffer is big enough

–Communication is considered complete when

–the application buffer can be reused, which means that the data has been copied from the application buffer to the user-defined buffer

–If the message is bigger than the user-defined buffer, the routine exits (by default).

Blocking buffered send (1)

- **Format:**

  MPI_Bsend(&buf, count, datatype, dest, tag, comm)

- **The sender doesn't return until the application buffer can be reused**

- **Must attach buffer space using:**

  MPI_Buffer_attach(buffer, size)

- **Buffer space is detached using:**

  MPI_Buffer_detach(buffer, size)

Blocking buffered send (2)

Determining the size of the buffer

MPI_Buffer_attach(buffer, size);
MPI_Bsend(..., count=20, datatype=type1, ...);
MPI_Bsend(..., count=40, datatype=type2, ...);

The value of size should be no less than the value computed by

MPI_Pack_size(20, type1, comm, &s1);
MPI_Pack_size(40, type2, comm, &s2);

size = s1 + s2 + 2 * MPI_BSEND_OVERHEAD;

MPI_BSEND_OVERHEAD can be found in mpi.h (for C) and mpif.h (for Fortran)

Ready mode

– In this mode, sender will send the data straightway without waiting for the “ready to receive” message

– This mode can be used only when programmer can make sure that the receive routine will be called before the send routine

– When a ready send is called, but the receive routine has not been called, an error occurs (the default behavior is the routine exits)

Ready mode

– Blocking ready send: MPI_Rsend: the sender returns when the application buffer can be reused

– Non-blocking ready send: MPI_Irsend

MPI functions

MPI is a complex system comprising of numerous functions with various parameters and variants

Six of them are indispensable, but can write a large number of useful programs already

Other functions add flexibility (datatype), robustness (non-blocking send/receive), **efficiency (ready-mode communication)**, modularity (communicators, groups) or convenience (collective operations, topology).

In the lectures, we are going to cover most commonly encountered functions

Blocking and non-blocking forms for the communication modes

All these four communication modes have both blocking and non-blocking forms

The communication modes refers to the send routines

Standard send: MPI_Send (blocking), MPI_Isend (non-blocking)

Synchronous send: MPI_Ssend (blocking), MPI_Issend (non- blocking)

Buffered send: MPI_Bsend (blocking), MPI_Ibsend (non-blocking)

Ready send: MPI_Rsend (blocking), MPI_Irsend (non-blocking)

Two Receive routines

Blocking receive routine: MPI_Recv()

Non-blocking receive routine: MPI_Irecv()

Virtual topology

Is a mechanism for naming the processes in a communicator in a way that fits the communication pattern better

Add convenience to MPI (can make coding easier)

MPI functions

MPI is a complex system comprising of numerous functions with various parameters and variants

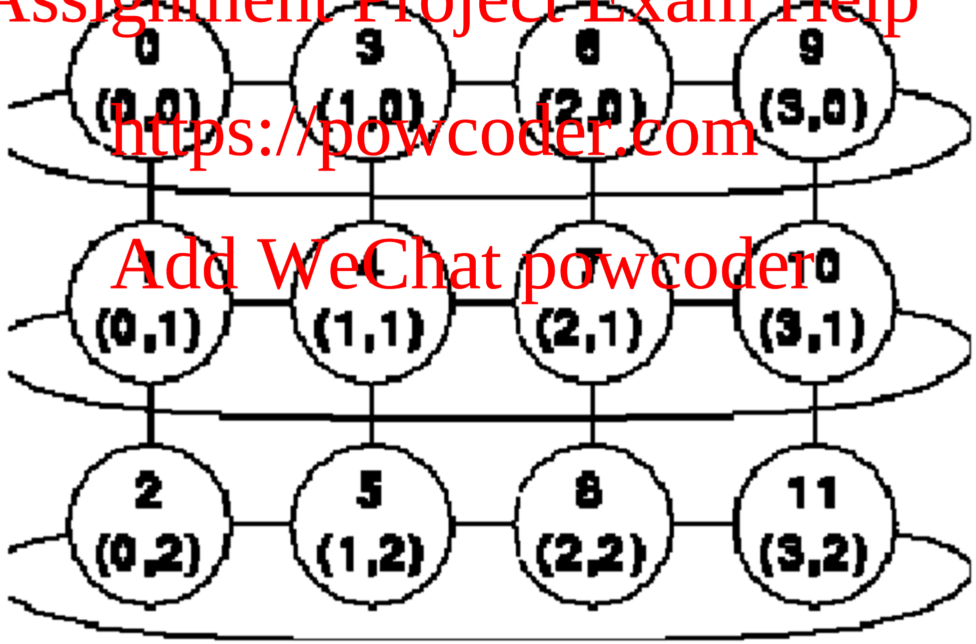
Six of them are indispensable, but can write a large number of useful programs already

Other functions add flexibility (datatype), robustness (non-blocking send/receive), efficiency (ready-mode communication), modularity (communicators, groups) or convenience (collective operations, topology).

In the lectures, we are going to cover most commonly encountered functions

Cartesian Topology

naming the processes in a communicator using Cartesian coordinates



Cartesian topology

Create a Cartesian topology

int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart)

[IN comm_old] input communicator

[IN ndims] number of dimensions of cartesian grid
[IN dims] integer array of size ndims specifying the number of processes in each dimension

[IN periods] logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
[IN reorder] ranking may be reordered (true) or not (false)
[OUT comm_cart] communicator with new Cartesian topology (handle)

The topology is only accessible through the new communicator returned in comm_cart

Converting between ranks and coordinates

MPI_Cart_rank (comm, coords, rank)

converts process grid coordinates to process rank.

It might be used to determine the rank of a particular process whose grid coordinates are known, in order to send a message to it or receive a message from it

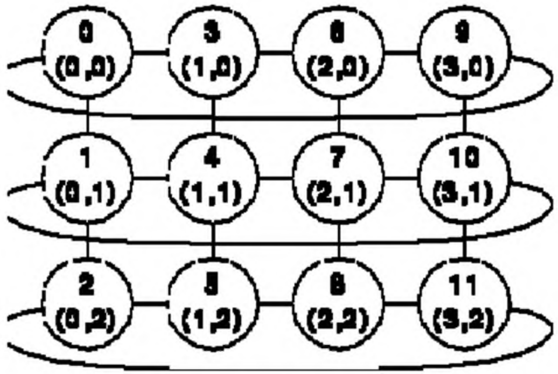
MPI_Cart_coords (comm, rank, ndims, coords)

converts process rank to coordinates.

It might be used to determine the grid coordinates of a particular process from which a message has just been received.

An Example of Cartesian Topology

```
int main(int argc, char *argv[])
{
   int rank, size;
   MPI_Comm comm;
   int dim[2], period[2], reorder;
   int coord[2], id;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if (size != 12) {
     printf("Please run with 12 processes.\n");
     MPI_Abort(MPI_COMM_WORLD, 1);
    }
    dim[0]=4; dim[1]=3;
    period[0]=0; period[1]=1;
    reorder=0;
```

An Example of Cartesian Topology

```
 MPI_Cart_create(MPI_COMM_WORLD, 2, dim,
period, reorder, &comm);
    if (rank == 5) {
        MPI_Cart_coords(comm, rank, 2, coord);
        printf("Rank %d coordinates are %d %d\n",
rank, coord[0], coord[1]);
     }

    if(rank==0) {
         coord[0]=3; coord[1]=1;
         MPI_Cart_rank(comm, coord, &id);
         printf("The processor at position (%d, %d) has
rank %d\n", coord[0], coord[1], id);
     }
     MPI_Finalize();
     return 0
}
```