MPI Lab 2: Passing Messages

Now that you’re acquainted with MPI it’s time for a small lab. In this lab you will write a small program that initializes a cluster and sends messages in between nodes. You will learn how to use the message passing interface and some of the different types of messages you can send.

I. Sending and Receiving Data

In order to do this lab you will need to be able to send and receive messages. MPI provides the functions `MPI_Send` and `MPI_Recv` for this purpose. These functions need a buffer containing the data you wish to send, the number of elements in the buffer, the type of data in the buffer, a tag, a destination, a communicator, and a status pointer. The tag of the send or receive call is a way to label different sends and receives so that you can distinguish between them. The destination is the rank of the process you are sending to or receiving from.

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

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

Now create a new C file, `simple_messages.c` and initialize a new grid with two nodes. Now have the master node (node 0) send a message containing its rank to node 1. Have node 1 receive the message and return a message to the master node containing the rank of node 1. Have both nodes print the messages as they receive them.

II. An Easier way to Exchange Messages
As you may imagine the above scenario is not uncommon when writing distributed programs and it can become tedious calling send and then receive and staggering the messages so that there is always one node sending and one receiving can be tedious. To this end, MPI includes a function for exchanging messages.

```c
int MPI_Sendrecv(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);
```

Rewrite `simple_messages.c` as `message_exchange.c` using the `MPI_Sendrecv()` function. Now you no longer need to check which node is executing to make sure that they aren’t sending at the same time.

### III. Nonblocking Messages

While nearly all of the programs you write will require an exchange of messages it is often the case that nodes need to exchange information with more than one neighbor. MPI provides a set of send and receive commands, which allow you to perform nonblocking sends, and receives. Using nonblocking send and receive functions all nodes can send messages to their neighbors and then begin receiving messages.

```c
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);
```
You may note that rather than accepting a status pointer both `MPI_Isend()` and `MPI_Irecv()` accept a request pointer. `MPI_Request` may be used to determine the status of a send or receive. MPI includes the function `MPI_Wait()` which can be used to wait for a send or receive to complete. If you choose to use `MPI_Isend()` or `MPI_Irecv()` you must call `MPI_Wait()` on the resulting request before you may modify the data buffer.

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

Using `MPI_Isend()`, `MPI_Iwait()`, and `MPI_Irecv()` write a new program `more_messages.c`. This time make your program work for an arbitrary number of nodes.

**IV. Arranging the Communicators**

Many distributed programs you write for MPI will deal with 2D arrays and matrices. So far we have dealt with ranks as a system for organizing computers inside of a communicator, the problem with this is that it creates a 1D arrangement of processors. Luckily MPI provides a set of functions for creating and manipulating a Cartesian arrangement of computers. The `MPI_Cart_create()` function may be used to create a new communicator for computers in another communicator automatically arranging them into a grid of specified dimensions.

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

`MPI_Cart_create()` makes a new communicator `comm_cart` which contains processors arranged in `ndim` dimensions (usually 2 for our purposes) each of size `dim[n]`. Now that you have a grid of computers, the next question becomes how does one map between rank and coordinates. MPI also includes the `MPI_Cart_rank()` and `MPI_Cart_coords()` functions to map from coordinates to rank and vice-versa, respectively.
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank);

int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords);

Using the Cartesian functions write a new program message_grid.c that constructs an N by N grid of processors and then have each processor determine its four adjacent neighbors and send them messages as we have done before.

V. Grading

Because this lab consists of small programs that demonstrate a basic skill set. The following is the weighting for this lab:

10% · simple_message.c
  50% · Program meets all of requirements
  50% · Code compiles and demonstrates an honest attempt

10% · message_exchange.c
  50% · Program meets all requirements
  50% · Code compiles and demonstrates an honest attempt

20% · more_messages.c
  25% · Program meets all requirements
  25% · Code demonstrates all of the newly introduced concepts
  50% · Code demonstrates an honest attempt

60% · message_grid.c
  25% · Program meets all requirements
  25% · Code demonstrates correct use of Cartesian grid functions
  25% · Code demonstrates correct use of message passing functions
  25% · Code correctly locates neighboring cells