

A Summary of MPI

Ganesh Gopalakrishnan's Research Group
Author: Michael Bentley

July 2, 2010

Contents

1	Typical Use For MPI	3
1.1	How to Compile an MPI Program	3
1.2	How to Run an MPI Program	3
2	Basic MPI Functions	4
2.1	MPI_Init	5
2.2	MPI_Finalize	5
2.3	MPI_Comm_size	5
2.4	MPI_Comm_rank	6
2.5	MPI_Send	6
2.6	MPI_Recv	7
2.7	MPI_Bcast	8
2.8	MPI_Reduce	9
2.9	MPI_Allreduce	9
2.10	MPI_Gather	10
2.11	MPI_Allgather	11
2.12	MPI_Scatter	12
2.13	MPI_Barrier	13
2.14	MPI_Isend	14
2.15	MPI_Irecv	14
2.16	MPI_Wait	15
2.17	MPI_Probe	16
2.18	MPI_Sendrecv	17
	Appendix A MPI_Datatype Constants in mpi.h	19
	Appendix B MPI_Status Data Structure in mpi.h	21
	Appendix C MPI_Op Constants in mpi.h	23

1 Typical Use For MPI

This text only covers the implementation of MPI in the language of C. Many languages have their own implementation of MPI, most notably Fortran and C++.

Many MPI programs that you will make and that you will see utilize the Single Program Multiple Data (SPMD) model. That is where there is one single program in which different processes work on small portions of that one program.

Typically, an MPI program consists of a single binary that runs in multiple processes. When more than one process is running a single MPI program, it can find out its “rank” from the operating system to determine what it should be doing. You may also find out how many processes are running your code, so that you may know how to divide up the work that needs to be done.

In this document I use the words “node” and “process” interchangeably, but technically, they are not the same thing. Multiple processes may be running on a single processor where the processor is considered to be one node.

1.1 How to Compile an MPI Program

In order to compile a single MPI program in c, you will not use the gcc command. Instead, you will use the mpicc to compile your code. The program mpicc uses gcc, so you must have gcc installed. To compile a file called program.c into program, type the following into the command-line:

```
> mpicc -o program program.c
```

As you can see, the command to compile with mpicc is very similar to the command to compile with gcc. To see more on the options available and a more thorough explanation, look at the man pages for mpicc.

1.2 How to Run an MPI Program

After you have compiled an MPI program using mpicc, you may run it as a normal program. One thing to note is if you run it as a normal program, you will only have one process executing the code. That completely destroys the purpose of writing MPI code!

The way to run the program you have created with more than one process is to use the program called “mpiexec”. Say we wanted to run our program that we have conveniently called “program” and we want to run it with 4 processes. Also say that we need to pass in a parameter to “program” specifying a file name, “tmp.txt”. The command would look like this:

```
> mpiexec -n 4 program tmp.txt
```

It’s just that simple. The ‘-n’ flag is to be preceded by the number of processes that you want your code to run with. There are more features to mpiexec, and you can study them in the man pages.

2 Basic MPI Functions

There are quite a few MPI functions that are often used. Below, there is a brief description of the following functions in this order:

1. `MPI_Init` (`int*` argc, `char***` argv)
2. `MPI_Finalize` ()
3. `MPI_Comm_size` (`MPI_Comm` group, `int*` size)
4. `MPI_Comm_rank` (`MPI_Comm` group, `int*` rank)
5. `MPI_Send` (`void*` send_this, `int` how_many, `MPI_Datatype` what_type, `int` send_to, `int` tag, `MPI_Comm` group)
6. `MPI_Recv` (`void*` recv_here, `int` how_many, `MPI_Datatype` what_type, `int` coming_from, `int` tag, `MPI_Comm` group, `MPI_Status*` status)
7. `MPI_Bcast` (`void*` variable, `int` how_many, `MPI_Datatype` what_type, `int` coming_from, `MPI_Comm` group)
8. `MPI_Reduce` (`void*` send_this, `void*` recv_here, `int` how_many, `MPI_Datatype` what_type, `MPI_Op` reduce_type, `int` which_node_receives, `MPI_Comm` group)
9. `MPI_Allreduce` (`void*` send_this, `void*` recv_here, `int` how_many, `MPI_Datatype` what_type, `MPI_Op` reduce_type, `MPI_Comm` group)
10. `MPI_Gather` (`void*` send_this, `int` send_how_many, `MPI_Datatype` send_type, `void*` recv_here, `int` recv_how_many, `MPI_Datatype` recv_type, `int` which_node_receives, `MPI_Comm` group)
11. `MPI_Allgather` (`void*` send_this, `int` send_how_many, `MPI_Datatype` send_type, `void*` recv_here, `int` recv_how_many, `MPI_Datatype` recv_type, `MPI_Comm` group)
12. `MPI_Scatter` (`void*` send_this, `int` send_how_many, `MPI_Datatype` send_type, `void*` recv_here, `int` recv_how_many, `MPI_Datatype` recv_type, `int` which_node_sends, `MPI_Comm` group)
13. `MPI_Barrier` (`MPI_Comm` group)
14. `MPI_Isend` (`void*` send_this, `int` how_many, `MPI_Datatype` what_type, `int` send_to, `int` tag, `MPI_Comm` group, `MPI_Request*` request)
15. `MPI_Irecv` (`void*` recv_here, `int` how_many, `MPI_Datatype` what_type, `int` coming_from, `int` tag, `MPI_Comm` group, `MPI_Request*` request)
16. `MPI_Wait` (`MPI_Request*` request, `MPI_Status*` status)
17. `MPI_Probe` (`int` coming_from, `int` tag, `MPI_Comm` group, `MPI_Status*` status)
18. `MPI_Sendrecv` (`void*` send_this, `int` send_how_many, `MPI_Datatype` send_type, `int` send_here, `int` send_tag, `void*` recv_here, `int` recv_how_many, `MPI_Datatype` recv_type, `int` from_here, `int` recv_tag, `MPI_Comm` group, `MPI_Status*` status)

2.1 MPI_Init

`MPI_Init (int* argc, char*** argv)`

This function starts the MPI service. It will use the command-line arguments to decide what it should do. For example, it will create all of the network connections needed to communicate through ethernet cables to send messages to other processes on other computers.

`argv` and `argc` are passed to let the MPI runtime engine know what arguments were passed to `mpiexec` such as how many processes to make.

As of MPI 2, the `argc` and `argv` parameters are no longer necessary. You may pass in `NULL` to both the the parameters of this function. It will work exactly the same.

argc :

*int** — The address of the amount of command-line arguments.

argv :

*char**** — The address of the list of command-line arguments. An array of arrays of characters.

2.2 MPI_Finalize

`MPI_Finalize()`

This function ends the MPI service. After this function is called, no other MPI functions may be used otherwise an error will be thrown. It executes the cleanup of the MPI routine. You may still continue to run non-MPI code after you execute this method, but you may not enter into the MPI service again in the same instance. That means that if you execute this function and some time after, you execute the `MPI_Init` function, you will receive an error. You can study the `MPI_Init` function in Section 2.1.

The are no parameters for this function.

2.3 MPI_Comm_size

`MPI_Comm_size (MPI_Comm group, int* size)`

This function call asks the operating system how many processes are in a specific group. It will write that value to the variable “size”.

Note: `MPI_COMM_WORLD` is our full group of processes, or our processor group. It is called a “communicator” in MPI language.

`MPI_COMM_WORLD` is the totality of all groups. You can separate the totality into separate groups, yet `MPI_COMM_WORLD` will remain the totality.

group :

MPI_Comm — The group you want to know the size of.

size :

*int** — The address of where to store the size.

2.4 MPI_Comm_rank

MPI_Comm_rank (MPI_Comm group, int* rank)

This function obtains the rank within the group of the process that issued the command. This value is placed into the variable rank.

The rank is which number of process is this current process. This value will start at zero for the first process, and will increment up for every newer process.

This information is used for the process to know what it should be doing. It's really the only thing that is different from the other processes in the group.

group :

MPI_Comm — The group you want to check the rank in.

rank :

*int** — The address of where to store the rank.

2.5 MPI_Send

MPI_Send (void* send_this, int how_many, MPI_Datatype what_type, int send_to, int tag, MPI_Comm group)

This function call sends data from the current process to another process. You can use the data types given by mpi.h or you can define your own. This is one of the most basic MPI functions.

This function is considered to be a “blocking send”. The basic idea of the implementation is that there is a buffer on the local machine that the data to be sent will be copied to. After that data is copied to the local buffer, and when the destination node is known, it will copy the information from the local buffer to a buffer seen by the destination node. This function is called a blocking send because this function will not return until the copying to the local buffer has been completed.

If you do not want to wait for the information to be copied to a buffer, then you can perform an MPI_Isend instead. You can find information on MPI_Isend in Section 2.14.

The type of the send is set by the “what_type” parameter in the function call. It is of the type MPI_Datatype. To see a full list of the MPI_Datatype constants that are defined in mpi.h, see Appendix-A.

send_this :

*void** — This is the address of the data to be sent.

how_many :

int — This is how many objects that there are at the address that are to be sent.

what_type :

MPI_Datatype — This is the type of object that the data to be sent represents. (see Appendix-A)

send_to :

int — This is the “rank” of the process that you want to send this message to.

tag :

int — This is an integer that is used to identify this send. It distinguishes this send from

other sends that you may receive from the same other process. The other process must pass in the same tag in order to receive this message.

group :

MPI_Comm — This is a processor group of which the process you want to send this data resides.

2.6 MPI_Recv

MPI_Recv (*void** *recv_here*, *int* *how_many*, *MPI_Datatype* *what_type*, *int* *coming_from*, *int* *tag*, *MPI_Comm* *group*, *MPI_Status** *status*)

This function call receives data from another process and copies it to the current process to another process.

This function is considered to be a “blocking receive”. The basic idea of the implementation is that there will be another process that is attempting to send data to this process. When this function is processed by this process, it will send a signal to the sending process that it is ready to receive data. The data will then be copied from the local buffer for the sending process to the local buffer for this process. After it is copied to a local buffer for this process, then it will be copied to the address in the first parameter.

This function is considered to be a “blocking receive” because this function will not return until the information has been successfully received by this process. The advantage of this blocking receive over the nonblocking receive is that the received data can be assumed to be completely there and useful.

If you do not want to wait for the information to be received, then you can perform an *MPI_Irecv* instead. More information regarding *MPI_Irecv* can be found in Section 2.15.

The last parameter in this function is the “status” field where the status of the message is recorded. The type of this parameter is a struct called “*MPI_Status*”. More information on the “*MPI_Status*” struct can be found in Appendix-B.

recv_here :

*void** — This is the address of where the data being sent should be written.

how_many :

int — This is how many objects that there are being sent to the variable address above.

what_type :

MPI_Datatype — This is the type of object that the data to be sent represents. (see Appendix-A)

coming_from :

int — This is the “rank” of the process that is going to be sending the data.

tag :

int — This is an integer that is used to identify this send. It distinguishes this send from other sends that you may receive from the same other process. You must pass in the same tag that was used by the sending process in order to receive this data. If more than one send to this process from the same other process both have the same tag, then this will receive the first one.

group :

MPI_Comm — This is the processor group of which the process that is sending the data must reside.

status:

*MPI_Status** — This describes the result of the `MPI_Recv()` command. (see Appendix-B)

2.7 MPI_Bcast

`MPI_Bcast (void* variable, int how_many, MPI_Datatype what_type, int coming_from, MPI_Comm group)`

The following code :

```
if ( rank == 0 )
    for ( i = 1 ; i < nnodes ; i++ )
        MPI_Send ( overallmin, 2, MPI_INT, i, OVRMLIN_MSG, MPI_COMM_WORLD );
else
    MPI_Recv ( overallmin, 2, MPI_INT, 0, OVRMLIN_MSG, MPI_COMM_WORLD, &status );
MPI_Barrier ( MPI_COMM_WORLD );
```

is functionally equivalent to the command :

```
MPI_Bcast ( overallmin, 2, MPI_INT, 0, MPI_COMM_WORLD );
```

This function performs a blocking send to all of the other processes in the group from the node specified by `coming_from`. All other nodes that process this command perform a blocking receive on that information.

Each process that processes the command Broadcast will wait for ALL of the nodes in the group to process the command before any of the processes can advance. In other words, aside from the blocking send that the source process executes and the blocking receives that the other processes execute, there is essentially a barrier at the end of it. To understand barriers a little more, a complete description can be found in Section 2.13.

variable :

*void** — This is the address of the data to be sent or received. The data at this address will be sent if the process executing this function is the one with rank specified by the `coming_from` parameter. All other processes will receive the data sent from the `coming_from` node and place it into this variable address.

how_many :

int — This is how many objects that there are at the address that are to be sent or received.

what_type :

MPI_Datatype — This is the type of object that the data to be sent represents. (see Appendix-A)

coming_from :

int — This is the node number of the process that will be sending the data.

group :

MPI_Comm — This is a processor group of which the process you want to send this data resides.

2.8 MPI_Reduce

`MPI_Reduce` (`void*` `send_this`, `void*` `recv_here`, `int` `how_many`, `MPI_Datatype` `what_type`, `MPI_Op` `reduce_type`, `int` `which_node_receives`, `MPI_Comm` `group`)

All of the nodes execute this command. The idea is that all of the various values that the different nodes have for a single variable (Note that each node has an individual copy of every variable and they may contain different values.); these values are reduced into a single value using some sort of criteria. There are different methods of reducing. For example, if you want to reduce them all into the minimum value, then the reduce method you choose is called `MPI_MINLOC`. Each node will contribute their value to a big melting pot of values, and, based on the criteria, the calculated reduced value will be sent to the node that was specified in `which_node_receives`.

The type of the reduce is set by the “`reduce_type`” parameter in the function call. It is of the type `MPI_Op`. To see a full list of the `MPI_Op` constants that are defined in `mpi.h`, see Appendix-C.

`send_this` :

*void** — The address of the variable that each node contributes to the “melting pot” of values.

`recv_here` :

*void** — The address of the location to store the “reduced” value. This value will only be stored for the process specified in “`which_node_receives`”. All other processes are not affected by this parameter.

`how_many` :

int — This is how many pieces of information to be sent this way.

`what_type` :

MPI_Datatype — This is the type of the sent variable to be reduced. (See Appendix-A)

`reduce_type` :

MPI_Op — This is the method of combining all of the different values into one reduced value. i.e. it could be the sum, the maximum, or the minimum, etc. (See Appendix-C)

`which_node_receives`

int — This is the rank of the process that receives the reduced value.

`group` :

MPI_Comm — This is the group that is participating in the reduce. I believe that all of the nodes in the group must participate in the reduce. If you need only a subset of the group, then my guess is that you have to make a new group that contains the subset that you want.

2.9 MPI_Allreduce

`MPI_Allreduce` (`void*` `send_this`, `void*` `recv_here`, `int` `how_many`, `MPI_Datatype` `what_type`, `MPI_Op` `reduce_type`, `MPI_Comm` `group`)

This function is very similar to `MPI_Reduce`. In fact, the following code:

```
MPI_Reduce ( mymin, overallmin, 1, MPI_2INT, MPI_MINLOC, 0, MPI_COMM_WORLD );
MPI_Bcast ( overallmin, 1, MPI_2INT, 0, MPI_COMM_WORLD
```

is functionally equivalent to the following code:

```
MPI_Allreduce ( mymin, overallmin, 1, MPI_2INT, MPI_MINLOC, MPI_COMM_WORLD );
```

There is only one parameter that is different from the `MPI_Reduce`, and that is that the variable “which_node_receives” doesn’t exist in `MPI_Allreduce`. That is because all of the nodes in the group receive the value into the “receive_variable_address”.

send_this :

*void** — The address of the variable that each node contributes to the “melting pot” of values.

recv_here :

*void** — The address that each node will write the “reduced” answer to.

how_many :

int — This is how many pieces of information to be sent this way.

what_type :

MPI_Datatype — This is the type of the sent information to be reduced.

which_reduce :

MPI_Op — This is the method of combining all of the different values into one reduced value. i.e. it could be the sum, the maximum, or the minimum, or others. (See Appendix-C)

group :

This is the group that is participating in the reduce. I believe that all of the nodes in the group must participate in the reduce. If you need only a subset of the group, then my guess is that you have to make a new group that contains the subset that you want.

2.10 MPI_Gather

`MPI_Gather (void* send_this, int send_how_many, MPI_Datatype send_type, void* recv_here, int recv_how_many, MPI_Datatype recv_type, int which_node_receives, MPI_Comm group)`

Many MPI programs are designed to break the information into chunks, and to have each node work on one chunk each. For such a situation, you can divide the data up by using `MPI_Scatter` (see Section 2.12). At the end of some computation in such a program, one of the nodes (usually the one of rank zero) is to gather all of the information calculated from each node, group it together and make some sort of conclusion. The idea of `MPI_Gather` is for one node to receive the information from the calculations of all of the other processes.

Each process (the one receiving included) sends the contents of its send buffer to the process that is receiving. The receiving process receives the messages and stores them in the order of the ranks of the processes that are in the group. The following code :

```
MPI_Send(send_this, how_many, what_type, send_to, tag, group);
if ( rank == receiving_node )
    MPI_Recv(recv_here + i*recv_how_many*size(recv_type), recv_how_many, recv_type,
            i, tag, group, status);
```

is functionally equivalent to:

```
MPI_Gather(send_this, send_how_many, send_type, recv_here, recv_how_many,
           recv_type, receiving_node, group);
```

Note: you cannot simply call size on the receive type, it was just for illustration purposes. Also note that a process can send to and receive from itself.

I am not sure what happens when the amount sent doesn't match the amount received. That is something that can be discovered by some simple tests.

Notice that this is a blocking call, meaning no process will leave this function call until all of them have executed it. That won't really matter if you are using this function to gather the results at the end of the program.

send_this :

*void** — The address of the variable that each node contributes.

send_how_many :

int — How many pieces of information will be sent from each process.

send_type :

MPI_Datatype — The type of the data sent. (see Appendix-A)

recv_here :

*void** — The address of where to store the data. Note, this parameter only effects the node that is receiving the data.

recv_how_many :

int — How many pieces of information total to receive. Note that you can get unexpected behavior if the *recv_here* hasn't allocated enough memory to store this many pieces.

recv_type :

MPI_Datatype — The type of the data received. (see Appendix-A)

which_node_receives :

int — The rank of the one process that is to receive all of the data.

group :

MPI_Comm — The group that this is to run on.

2.11 MPI_Allgather

`MPI_Allgather (void* send_this, int send_how_many, MPI_Datatype send_type, void* recv_here, int recv_how_many, MPI_Datatype recv_type, MPI_Comm group)`

This function is almost exactly the same as `MPI_Gather` (see Section-2.10). The only difference can be noted from its parameters. You will notice that there is no parameter specifying which process is to receive the data. That is because *ALL* of the processes in the group receive the data. The following code:

```
MPI_Gather(send_this, send_how_many, send_type, recv_here, recv_how_many,
           recv_type, which_node_receives, group);
MPI_Bcast(recv_here, recv_how_many, recv_type, which_node_receives, group);
```

is functionally equivalent to:

```
MPI_Allgather(send_this, send_how_many, send_type, recv_here, recv_how_many,
             recv_type, group);
```

Notice that this is a blocking call, meaning no process will leave this function call until all of them have executed it.

send_this :

*void** — The address of the data to be sent (and gathered).

send_how_many :

int — The amount of pieces of data to be sent.

send_type :

MPI_Datatype — The type of data being sent.

recv_here :

*void** — The address of the location to store the gathered data.

recv_how_many :

int — How many pieces of data to store. For best results, this should total the amount of data being sent.

recv_type :

MPI_Datatype — The type of data that is being received. For best results, this should be the same as the *send_type*.

group :

MPI_Comm — The group that is participating in the *MPI_Allgather*.

2.12 MPI_Scatter

MPI_Scatter (*void** *send_this*, *int* *send_how_many*, *MPI_Datatype* *send_type*, *void** *recv_here*, *int* *recv_how_many*, *MPI_Datatype* *recv_type*, *int* *which_node_sends*, *MPI_Comm* *group*)

MPI_Scatter is pretty much the opposite of *MPI_Gather* (see Section-2.10). The basic idea is that one process will have a large set of data that needs to be divided up and sent to all of the processes. The following code:

```
if(rank == sending_node)
  for (i=0 ; i < nnodes ; i++)
    MPI_Send(send_this + i*send_how_many*size(send_type), send_how_many,
            send_type, i, tag, group);
MPI_Recv(recv_here, recv_how_many, recv_type, sending_node, tag, group);
```

is functionally equivalent to

```
MPI_Scatter(send_this, send_how_many, send_type, recv_here, recv_how_many,
           recv_type, sending_node, group);
```

Note: you cannot simply call size on the receive type, it was just for illustration purposes. Also note that a process can send to and receive from itself.

Even the node that sends the data will receive one of the chunks of data. Also note that the amount of data being sent should amount up to the amount of data being received as a whole. I am unsure of the behavior of this function if the amount of data received does not match up to the amount of data sent.

This is a blocking call, meaning no process will leave this function call until all of them have executed this function call.

send_this :

*void** — This is the address of the data that is to be divided up and sent to all of the processes in the group. This parameter is only important to the node that is sending the data.

send_how_many :

int — This indicates how many pieces of data there are to divide up and send to the processes in the group. This parameter is only important to the node that is sending the data.

send_type :

MPI_Datatype — The type of data being sent. (see Appendix-A) This parameter is only important to the node that is sending the data.

recv_here :

*void** — This is the address of where the received data should be stored.

recv_how_many :

int — This is how many pieces of data are to be received by this node.

recv_type :

MPI_Datatype — This is the type of data to be received by this node. (see Appendix-A) Note that *recv_type* does not have to match *send_type*. It is only required that the amount of data received equals the amount of data that was sent.

which_node_sends :

int — The rank of the process that is sending out information to all of the processes in the group.

group :

MPI_Comm — Specifies the group that is participating in the `MPI_Scatter`.

2.13 MPI_Barrier

`MPI_Barrier` (`MPI_Comm` group)

This is a very simple function. It simply causes all of the processes that execute this call to wait until all of the processes in the given group have executed this call as well. This is used to synchronize all of the processes in one place.

Notice that this is a potential for deadlock for many processes if only one of the processes in the group doesn't execute the `MPI_Barrier`.

group :

MPI_Comm — The group of processes that will synchronize at this function call.

2.14 MPI_Isend

`MPI_Isend` (`void*` `send_this`, `int` `how_many`, `MPI_Datatype` `what_type`, `int` `send_to`, `int` `tag`, `MPI_Comm` `group`, `MPI_Request*` `request`)

This function is very similar to `MPI_Send` (see Section-2.5). The difference is that this function is a non-blocking send. That means that as soon as this function is called, it will return so that the process that called it can start executing more code. Behind the scenes, the MPI service will create another process that copies the data to be sent to the local buffer for the process sending the data. The idea behind this is that you don't have to wait for the data to be copied to the local buffer before you start doing something else.

In essence, `MPI_Send` and `MPI_Isend` are functionally equivalent except for the fact that in the case of `MPI_Isend`, it can be damaging to write to the location that is supposed to be sent afterward because it may not have been completely copied to the local buffer causing a data race. If you need to write to this buffer after this function call, then you will have to make a call to `MPI_Wait`. A full description of `MPI_Wait` can be found in Section-2.16. It is for this reason that the “request” parameter is required.

send_this :

*void** — This is the address of the data to be sent.

how_many :

int — This is how many objects that there are at the address that are to be sent.

what_type :

MPI_Datatype — This is the type of object that the data to be sent represents. (see Appendix-A)

send_to :

int — This is the “rank” of the process that you want to send this message to.

tag :

int — This is an integer that is used to identify this send. It distinguishes this send from other sends that you may receive from the same other process. The other process must pass in the same tag in order to receive this message.

group :

MPI_Comm — This is a processor group of which the process you want to send this data resides.

request :

*MPI_Request** — The address of where to store information necessary to verify if this send has completed or not. This is to be used mainly with `MPI_Wait` (see Section-2.16).

2.15 MPI_Irecv

`MPI_Irecv` (`void*` `recv_here`, `int` `how_many`, `MPI_Datatype` `what_type`, `int` `coming_from`, `int` `tag`, `MPI_Comm` `group`, `MPI_Request*` `request`)

This function is very similar to `MPI_Recv` (see Section-2.6). The difference is that this function is a non-blocking receive. That means that as soon as this function is called, it will return so that the process that called it can start executing more code. Behind the scenes, the MPI service will

create another process that will continue the receive process. That is to say, it will wait until the send has been instantiated, and as soon as it has, it will copy that data to a local buffer and then place that information into the location reserved for the sent data specified by “`recv_here`”. The idea behind this is that you don’t have to wait for the receive to be completed before you start doing something else.

In essence, `MPI_Recv` and `MPI_Irecv` are functionally equivalent except for the fact that in the case of `MPI_Irecv`, it can be unpredictable to read data from the storage location reserved for the received data after the `MPI_Irecv` function call because the data may or may not have been received causing a data race. If you need to read this data after this function call, then you will have to make a call to `MPI_Wait`. A full description of `MPI_Wait` can be found in Section-2.16. It is for this reason that the “request” parameter is required.

recv_here :

*void** — This is the address of where the data being sent should be written.

how_many :

int — This is how many objects that there are being sent to the variable address above.

what_type :

MPI_Datatype — This is the type of object that the data to be sent represents. (see Appendix-A)

coming_from :

int — This is the “rank” of the process that is going to be sending the data.

tag :

int — This is an integer that is used to identify this send. It distinguishes this send from other sends that you may receive from the same other process. You must pass in the same tag that was used by the sending process in order to receive this data. If more than one send to this process from the same other process both have the same tag, then this will receive the first one.

group :

MPI_Comm — This is the processor group of which the process that is sending the data must reside.

request :

*MPI_Request** — The address of where to store information necessary to verify if the receive has completed or not. This is to be used mainly with `MPI_Wait` (see Section-2.16).

2.16 MPI_Wait

`MPI_Wait` (*MPI_Request** request, *MPI_Status** status)

This function simply keeps the process that executed it waiting until the non-blocking call that the request item belongs to is finished. After the non-blocking call is finished, this method will place the information about the message into the status variable. The status variable is an address pointing to a variable of the type `MPI_Status`. More information can be found about `MPI_Status` in Appendix-B.

This function call makes operations that were considered unsafe before into safe again. An example would be the following:

```

MPI_Irecv(&recv_here, how_many, what_type, coming_from, tag, group, &request);
// Do some work here not involving recv_here.
MPI_Wait(&request, &status);
// Do some work here involving recv_here.

```

The above example is a safe implementation of `MPI_Irecv` and `MPI_Wait`. The example sets the MPI service to wait for the receive while we are able to do some work. Then the wait function sets the process to wait until the receive is finished. After the receive is finished (guaranteed by the `MPI_Wait` function), then we can perform some computations using the information stored in “recv_here”.

request :

*MPI_Request** — This is the address of the object that has the information regarding the completion of a non-blocking call that you want to wait for.

status :

*MPI_Status** — The information about the non-blocking call will be stored in this variable address. (see Appendix-B)

2.17 MPI_Probe

`MPI_Probe` (int coming_from, int tag, MPI_Comm group, MPI_Status* status)

The `MPI_Probe` operation checks for incoming messages without actually receiving them. This can be useful to determine how to manage your receive. One example could be to determine how long the data to be received is going to be before allocating the amount of storage. The code to do that would be:

```

int count;
MPI_Status status;
MPI_Probe(coming_from, tag, group, &status);
MPI_Get_count(&status, send_type, &count);
int* recv_here = malloc(count*sizeof(int));
MPI_Recv(recv_here, count, MPI_INT, coming_from, tag, group, &status);

```

Information about the send that can be received by this process will be stored in the status variable. Information can be extracted from the status variable. For more information, see Appendix-B.

If there is a send that can be received by this process, yet there is another process that receives it instead, then this method will wait for another send to be issued.

This method is a blocking call, which means that it won't return from this function call until it has found a call from the correct process with the correct tag that is can be received from this process and is available. There is a non-blocking version this called `MPI_Iprobe`. `MPI_Iprobe` is not discussed in this document.

coming_from :

int — This is the rank of the process that is supposed to send data to this process.

tag :

int — This is an integer that is used to identify the send. It distinguishes this send from

other sends that you may receive from the same sending process. You must pass in the same tag that was used by the sending process in order to receive this data. If more than one send to this process from the same other process both have the same tag, then this will receive the first one.

group :

MPI_Comm — This is the group that this process and the sending process both belong to.

status :

*MPI_Status** — This is the address of the status variable. This is where the information about the send is going to be stored. (see Appendix-B)

2.18 MPI_Sendrecv

MPI_Sendrecv (*void** send_this, *int* send_how_many, *MPI_Datatype* send_type, *int* send_here, *int* send_tag, *void** recv_here, *int* recv_how_many, *MPI_Datatype* recv_type, *int* from_here, *int* recv_tag, *MPI_Comm* group, *MPI_Status** status)

This function is a combination of a send and a receive. The following code:

```
if ( rank == sending_node )
    MPI_Send(&send_this, how_many, what_type, recving_node, tag, group);
if ( rank == recving_node )
    MPI_Recv(&recv_here, how_many, what_type, sending_node, tag, group, &status);
```

is functionally equivalent to:

```
MPI_Sendrecv(&send_this, how_many, what_type, recving_node, tag, &recv_here,
             how_many, what_type, sending_node, tag, group, &status);
```

For more information on *MPI_Send*, see Section-2.5. For more information on *MPI_Recv*, see Section-2.6.

send_this :

*void** — This is the address of the data to be sent.

send_how_many :

int — This is how many objects that there are at the address that are to be sent.

send_type :

MPI_Datatype — This is the type of object that the data to be sent represents. (see Appendix-A)

send_to :

int — This is the “rank” of the process that you want to send this message to.

send_tag :

int — This is an integer that is used to identify this send. It distinguishes this send from other sends that you may receive from the same other process. The other process must pass in the same tag in order to receive this message.

recv_here :

*void** — This is the address of where the data being sent should be written.

recv_how_many :

int — This is how many objects that there are being sent to the variable address above.

recv_type :

MPI_Datatype — This is the type of object that the data to be sent represents. (see Appendix-A)

recv_from :

int — This is the “rank” of the process that is going to be sending the data.

recv_tag :

int — This is an integer that is used to identify this send. It distinguishes this send from other sends that you may receive from the same other process. You must pass in the same tag that was used by the sending process in order to receive this data. If more than one send to this process from the same other process both have the same tag, then this will receive the first one.

group :

MPI_Comm — This is the processor group of which the process that is sending the data must reside.

status:

*MPI_Status** — This describes the result of the receive portion of the function call. (see Appendix-B)

Appendix A MPI_Datatype Constants in mpi.h

MPI_CHAR :

char

MPI_BYTE :

See standard; like unsigned char

MPI_SHORT :

short

MPI_INT :

int

MPI_LONG :

long

MPI_FLOAT :

float

MPI_DOUBLE :

double

MPI_UNSIGNED_CHAR :

unsigned char

MPI_UNSIGNED_SHORT :

unsigned short

MPI_UNSIGNED :

unsigned int

MPI_UNSIGNED_LONG :

unsigned long

MPI_LONG_DOUBLE :

long double (some systems may not implement)

MPI_LONG_LONG_INT :

long long (some systems may not implement)

The following are datatypes for the MPI functions MPI_MAXLOC and MPI_MINLOC.

MPI_FLOAT_INT :

struct float, int

MPI_LONG_INT :

struct long, int

MPI_DOUBLE_INT :

struct double, int

MPI_SHORT_INT :

struct short, int

MPI_2INT :

struct int, int

MPI_LONG_DOUBLE_INT :

struct long double, int ; this is an optional type, and may be set to NULL.

Appendix B MPI_Status Data Structure in mpi.h

MPI_Status is a struct defined in mpi.h that has the following fields:

count :

int — Specifies how many of the type of item were received. You can get the count by calling the function `MPI_Get_count(&status, datatype, &count)`. That function will read the amount of bytes from status, divide by the size of datatype and store it in the variable given by the address of count.

cancelled :

int — Specifies whether or not a message was cancelled. You can cancel messages using `MPI_Cancel(&req)`, but it's expensive. If you do cancel a message, then you can test to see if it worked by using `MPI_Test_cancelled(&status, &flag)`.

MPI_SOURCE :

int — The rank of the process that sent the message.

MPI_TAG :

int — The tag that the message had.

MPI_ERROR :

int — Any error return.

The error can be deciphered using these defined constants:

MPI_SUCCESS :

Successful return code

MPI_ERR_BUFFER :

Invalid buffer pointer

MPI_ERR_COUNT :

Invalid count argument

MPI_ERR_TYPE :

Invalid datatype argument

MPI_ERR_TAG :

Invalid tag argument

MPI_ERR_COMM :

Invalid communicator

MPI_ERR_RANK :

Invalid rank

MPI_ERR_ROOT :

Invalid root

MPI_ERR_GROUP :

Null group passed to function

MPI_ERR_OP :

Invalid operation

MPI_ERR_TOPOLOGY :

Invalid topology

MPI_ERR_DIMS :

Illegal dimension argument

MPI_ERR_ARG :

Invalid argument

MPI_ERR_UNKNOWN :

Unknown error

MPI_ERR_TRUNCATE :

message truncated on receive

MPI_ERR_OTHER :Other error; use `Error_string`**MPI_ERR_INTERN :**

internal error code

MPI_ERR_IN_STATUS :

Look in status for error value

MPI_ERR_PENDING :

Pending request

MPI_ERR_REQUEST :illegal `mpi_request` handle**MPI_ERR_LASTCODE :**

Last error code – always at end

Appendix C *MPI_Op* Constants in *mpi.h*

MPI_MAX :

return the maximum

MPI_MIN :

return the minimum

MPI_SUM :

return the sum

MPI_PROD :

return the product

MPI_LAND :

return the logical and

MPI_BAND :

return the bitwise and

MPI_LOR :

return the logical or

MPI_BOR :

return the bitwise of

MPI_LXOR :

return the logical exclusive or

MPI_BXOR :

return the bitwise exclusive or

MPI_MINLOC :

return the minimum and the location (actually, the value of the second element of the structure where the minimum of the first is found)

MPI_MAXLOC :

return the maximum and the location