High Performance Computing Course
MPI Tutorial

Delft University of Technology
Information Technology and Systems

September 16, 2002

1 The MPI standard

1.1 Design and History

The MPI (Message Passing Interface) standard, version 1.0 [1], was set in May 1994 by the MPI Forum. It offers a sophisticated set of C, C++ and Fortran functions for writing parallel programs.

One of the main design goals for MPI is portability across a wide range of parallel machines using the Message Passing paradigm. Although this paradigm suits distributed memory supercomputers and clusters of workstations much better than shared memory machines, it is in practice possible to make highly efficient MPI implementations on all kinds of systems. We can benefit from this by learning to use MPI for our parallel programs, and accomplish high performance on whatever architecture we want to use, or more likely, have access to.

In June 1995 the specification of MPI version 1.1 [2] was released, containing some corrections and clarifications to the older document. Further clarifications and corrections were made in July 1997, when version 1.2 of the standard was released. Only seven pages were needed to describe the differences, which indicates that there are not many. MPI version 1.2 is (therefore) contained in the official document “MPI-2: Extensions to the Message-Passing Interface” [3] which describes MPI-2.0.

1.2 Local considerations

We will be using the DAS-2 (Distributed ASCI Supercomputer) cluster located at the faculty ITS. This cluster consists of 32 compute nodes (each node has 2 1-GHz Intel Pentium-III processors) and a file servernode. All nodes are connected by a very fast network (Myrinet)

All of this is only partially to our interest since all we need to know is how we can give the MPI library important hints about our program. Then, hopefully, the local implementation will use these hints to increase the performance on the specific hardware.

1.3 Further reading

The MPI documents ([1], [2] and [3]) can be found on `http://www.mpi-forum.org/` as HTML and Postscript. They give a complete overview of all aspects and functions of MPI and are convenient reference material.

If you prefer a printed manual, with more examples and comments, the best option will probably be “MPI - The Complete Reference” that comes in two volumes, “Volume 1, The MPI Core” [4] and “Volume 2, The MPI Extensions” [5].

For more about the DAS-2 computer see: http://www.cs.vu.nl/das2
Local information about this course can be found on

`http://pds.twi.tudelft.nl/vakken/in4049TU`. Online manpages can be consulted on this site, or from the command prompt using the `man` command.

2 Introduction to MPI

2.1 Hello World

This code lets all the nodes[1] print "Hello World!" using printf(). Included on each output-line is the number of the node (the *rank*), and the total number of nodes.

```
#include <stdio.h>
#include "mpi.h"

int rank, np;

int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  printf("Node %i of %i says: Hello World!\n", rank, np);
  MPI_Finalize();
}
```

At the top of every program you need to include `#include "mpi.h"` so that your program can use all MPI functions and datatypes.

Every MPI program must call **MPI_Init** before calling any other MPI library function. MPI_Init takes two arguments (argc and argv) as shown in the code above. These two parameters are used to pass the command line options (i.e. the things you type after the name of the program on the command line) to MPI. Likewise, **MPI_Finalize** must be called as the last MPI function. This function takes no parameters and terminates the MPI execution environment.

Two more MPI functions are introduced in this example. **MPI_Comm_size** determines the number of nodes that were started and stores this value in the variable `np`. **MPI_Comm_rank** determines the number associated with "this node", which is of course unique to all processes and stores this value in `rank`. Nodes are numbered from 0 to $N-1$.

The MPI constant **MPI_COMM_WORLD** defines a default **communicator** (a set of nodes). Many MPI functions require a communicator as a parameter, and for the time being we will only use MPI_COMM_WORLD for that.

Compile the above program[2] on DAS-2 with:

```
% mpicc -o helloworld helloworld.c
```

[1]Although it is possible to have one processor-node executing more than one of the processes, we will always speak of 'nodes' for the sake of simplicity.

[2]Make sure you use capitals where there are capitals within the code, otherwise your code will not work!

Then run this program on 2, 4 and 8 nodes, using:

```
% prun -pbs-script /usr/local/sitedep/reserve/pbs_script `pwd`/helloworld
2
```

Note: Above is one command line, the last parameter is the number of nodes to be used (in this case 2).

You can use

```
% man prun
```

to find out more about starting up MPI programs.

2.2 Sending data to one node

We will now extend our previous example to a program that illustrates the use of

int MPI_Send(buf, count, datatype, dest, tag, comm)	
void *buf	Send buffer address
int count	Number of elements in send buffer
MPI_Datatype datatype	Datatype of each send buffer element
int dest	Rank of message destination
int tag	Message tag
MPI_Comm comm	Communicator

and

int MPI_Recv(buf, count, datatype, source, tag, comm, status)	
void *buf	Receive buffer address
int count	Maximum number of elements in receive buffer
MPI_Datatype datatype	Datatype of each receive buffer element
int source	Rank of message source
int tag	Message tag
MPI_Comm comm	Communicator
MPI_Status *status	Status information object

Suppose we have a float variable f that we want to send to node 0. We pass &f as the send buffer address, and specify the buffer-size (count) to be 1. The corresponding MPI datatype is MPI_FLOAT. Since our destination is node 0, we pass 0 as the dest parameter. The tag parameter can be used to distinguish between different messages and can be choosen freely, so we will just put 11 here. Finally we specify the communicator to be MPI_COMM_WORLD. Our send command will thus be:

```
MPI_Send(&f, 1, MPI_FLOAT, 0, 11, MPI_COMM_WORLD);
```

At the receiving end, node 0, we have to make a corresponding call to MPI_Recv, so MPI can successfully transmit the data for us. We must specify the rank of the source node. Additionally we have to use the same tag or the transmission will not succeed and the program will stall. We also have to pass an address of a status information object, where MPI_Recv will store information about the transmission. Our receive command will thus be:

```
MPI_Recv(&f, 1, MPI_FLOAT, source, 11, MPI_COMM_WORLD, &status);
```

In this exercise we choose node 0 to be the 'main' node, the one that receives and prints all data. Every other node (rank $\neq$ 0) sends to node 0. Extend the Hello World example into a working program. First, you should add these variables right after the ´{ín main:

```
int source;
float f;
MPI_Status status;
```

Then remove the printf line and create an if and else blocks like this:

```
if(rank==0)
{
  ...
}
else
{
  ...
}
```

In the first block, write some code that receives and prints a float value from every other node together with the the nodenumber of the node it came from.[3] In the other block, assign some value to f and send it to node 0. Compile the program and run it with four processors.

2.3 Sending data from one node

Again we choose node 0 to be the 'main' node, but now we have it send some float value to every other node. Each node with rank > 0 should receive the same value and print it to the screen. Compile the program and run it again with four processors.

To find out more about what other kinds of variables (doubles, integers, etc.) can be send, check out MPI_Data_types in the **mpi-man** section of the in4049TU website.

[3]`for(source=1; source<np; source++) { ... }` where source iterates over the ranks of the other nodes. A float can be printed with `printf("%f",...)`

3 Collective communication

3.1 Broadcasting

We will now use

int MPI_Bcast(buffer, count, datatype, root, comm)	
void *buffer	Buffer address
int count	Number of elements in buffer
MPI_Datatype datatype	Datatype of each buffer element
int root	Rank of broadcast root
MPI_Comm comm	Communicator

to distribute a float value from node 0 to every other node. This is functionally equivalent to the exercise in §2.3, but it might be more efficient because we leave the choice of communication scheme to MPI. Some hardware might offer a dedicated broadcast channel, while some other hardware might be optimally used through a certain message tree. Anyway, we should leave these hardware dependent difficulties to the MPI library implementors and just use MPI_Bcast if we broadcast a message.

The same broadcast function should be called by all nodes, whether they are broadcasting or receiving data. The parameter root specifies the rank of the sending node. Now rewrite the code from §2.3 using this template:

```
if(rank==0)
{
  ...
}

MPI_Bcast( ... );

printf( ... );
```

In the if-block node zero sets f to some value. Then, all nodes call the broadcast function to receive a value in their f. After that, every node prints the value it received.

3.2 Reduce

We calculate $\sum_{j=0}^{N-1} f^j$ using N nodes. Node 0 reads in the value of f using

```
scanf("%f", &f);
```

Then node 0 broadcasts f, every node calculates f^j, and node 0 determines the the sum. We will use

int MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)	
void *sendbuf	Send buffer address
void *recvbuf	Receive buffer address (only used for root)
int count	Number of elements in buffer
MPI_Datatype datatype	Datatype of each send buffer element
MPI_Op op	Reduce operation
int root	Rank of receiving node
MPI_Comm comm	Communicator

to calculate the sum of a value (or array of values) over a collection of nodes. All nodes should call this MPI function, because it involves a collective operation. The address of the sendbuf is &f for all nodes, and we set recvbuf to &g (relevant only for the node with rank root).

MPI_Reduce will calculate the sum, if we choose **MPI_SUM** as the reduction operation. There are some other reduction operations predefined in MPI, including MPI_MAX for maximum, MPI_MIN for minimum and MPI_PROD for product. The result is written to recvbuf at the root node. The related function **MPI_Allreduce**, differs from MPI_Reduce only in that it writes the result to recvbuf in *all* nodes. So where with **MPI_Reduce** only the process with rank equal to root will receive the sum, with **MPI_Allreduce** all the processes wil receive the result. This is often usefuli, but we will not need it in this excercise.

Fill in the MPI_Bcast and MPI_Reduce parameters, so that this code will calculate $\sum_{j=0}^{N-1} f^j$:

```
if(rank==0)
{
  printf("Enter a float:\n");
  scanf("%f", &f);
}

MPI_Bcast( ... );
f = pow(f, rank);
MPI_Reduce( ... );

if(rank==0)
{
  printf("Reduced: %f\n", g);
}
```

Because we use the mathematical function pow, we have to add an extra include line to the top of the source code:

```
#include <math.h>
```

and we have to link the mathematical libraries to our program when compiling, with the -lm option:

```
cc -o opg32 opg32.c -lmpi -lm
```

3.3 Scatter and Gather

In many applications there is a need to have arrays of data distributed over several nodes. Every node will always be responsible for computations on its own part of the array. MPI offers special communication functions for use in this common situation.

For the distribution of an array, say of length N, over multiple nodes, we can use

int MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)	
void *sendbuf	Send buffer address (only significant for root node)
void sendcount	Number of elements to send to each node
MPI_Datatype sendtype	Datatype of each send buffer element
void *recvbuf	Receive buffer address
int recvcount	Number of elements to receive for this node
MPI_Datatype recvtype	Datatype of each receive buffer element
int root	Rank of sending node
MPI_Comm comm	Communicator

The root node offers the complete array with sendcount elements to MPI_Scatter in the sendbuf parameter. The recvbuf parameter indicates the starting address where the local sub-part of the array must be stored. This subarray wil contain recvcount elements.

> INTERMEZZO MPI_Scatter assumes that the number of elements to send is the same for each node (sendcount = np * recvcount). If N (=recvcount) is not a multiple of np one is forced to specify the number of elements for each different node in an array. The function to use when N is not a multiple of np is **MPI_Scatterv**.

If, for simplicity, we decide to allocate memory for the complete array at every node, it is also wise to use global indices at every node[4]. This means that the root node would offer a sendbuf like &f[0], and every node offers a recvbuf like &f[begin], where begin is the (integer) index of the first element in f[] that is local to this node.

[4]Alternatively, we could allocate arrays that are just big enough to contain all local elements on all but the root node, and use local indices (0..localN).

For the opposite purpose, the gathering of data from many nodes, we can use

int MPI_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)	
`void *sendbuf`	Send buffer address
`int sendcount`	Number of elements to send to root node
`MPI_Datatype sendtype`	Datatype of each send buffer element
`void *recvbuf`	Receive buffer address (only significant for root node)
`int recvcount`	Number of elements to receive from each node
`MPI_Datatype recvtype`	Datatype of each receive buffer element
`int root`	Rank of receiving node
`MPI_Comm comm`	Communicator

As an exercise, study the source code template33.c and replace the calls to distribute_f() and collect_g() with appropriate calls to MPI_Scatter() and MPI_Gather():

```
/***
 * template33.c
 ***/

#include <stdio.h>
#include <math.h>
#include "mpi.h"

#define N 128

int rank, np;
int length, begin, end;
MPI_Status status;
float f[N], g[N];


/***
 * init() sets the begin-index, the end-index, and the length of
 * this node's part of f[] and g[].
 ***/

void init()
{
  length = N / np;
  begin = rank * length;
  end = begin + length - 1;
}
```

```
/***
 * set_f() fills f[] with initial values
 ***/

void set_f()
{
  int i;

  for(i=0; i<N; i++)
    f[i] = sin(i * (1.0/N));
}


/***
 * distribute_f() distributes f[] over all nodes, where node 0 sends
 * the appropriate part of f[] to each node. Every node only receives
 * f[begin]...f[end].
 ***/

void distribute_f()
{
  int dest;
  int their_begin;

  if(rank==0) /* Sending node */
  {
    for(dest=1; dest<np; dest++)
    {
      their_begin = dest * length;
      MPI_Send(&f[their_begin], length, MPI_FLOAT, dest, 12, MPI_COMM_WORLD);
    }
  }
  else /* One of the receiving nodes */
  {
    MPI_Recv(&f[begin], length, MPI_FLOAT, 0, 12, MPI_COMM_WORLD, &status);
  }
}


/***
 * calc_g() calculates g[begin]...g[end], based on f[begin]...f[end].
 ***/

void calc_g()
{
  int i;

  for(i=begin; i<=end; i++)
    g[i] = 2.0 * f[i];
```

```
}


/***
 * collect_g() gathers the subresults from each node (g[begin]...g[end]) to
 * node 0.
 ***/

void collect_g()
{
  int src;
  int their_begin;

  if(rank==0) /* Receiving node */
  {
    for(src=1; src<np; src++)
    {
      their_begin = src * length;
      MPI_Recv(
        &g[their_begin], length, MPI_FLOAT, src, 13,
        MPI_COMM_WORLD, &status);
    }
  }
  else /* One of the sending nodes */
  {
    MPI_Send(&g[begin], length, MPI_FLOAT, 0, 13, MPI_COMM_WORLD);
  }
}


/***
 * show_g() prints g[] to the screen
 ***/

void show_g()
{
  int i;

  printf("g[]:\n");
  for(i=0; i<N; i++)
  {
    printf(" %.2f", g[i]);
  }
  printf("\n");
}


int main(int argc, char **argv)
{
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &np);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  init();
  if(rank==0) set_f();
  distribute_f();
  calc_g();
  collect_g();
  if(rank==0) show_g();

  MPI_Finalize();
  return 0;
}
```

4 Advanced topics

4.1 Bechmarking your application

Measuring the performance of a parallel application can be confusing, especially if nodes are depending on each other for critical data and need to wait before they can continue their computational work. On the other hand, benchmarking is the ultimate way to test your application for such latencies in practice, and often leads to ideas about rearranging the algoritm and/or data for higher performance.

With the fairly simple function

double MPI_Wtime(void)

we can measure the wall-clock time in seconds. The term 'wall-clock' indicates that this timer runs just like a clock on the wall. The alternative would be a 'processor-time' measurement with a standard C function such as clock(). This clock only reports the time that the CPU has actually spend on our application. Since this does not include waiting for input nor the time spend on other tasks on this node, a real-time measurement would not learn us much about the efficiency of our code.

Below is some example code showing the use of MPI_Wtime(). The functions start_timer() and stop_timer() can be called to measure the elapsed wall-clock time in an MPI program. Note that these functions can be called repeatedly so a portion of code inside a loop can also be measured.

```
double timer = 0.0;

void start_timer() { MPI_Barrier(MPI_COMM_WORLD);
                     timer = MPI_Wtime() - timer; }
void stop_timer()  { timer = MPI_Wtime() - timer; }

void show_timer()
{
  printf("(Node %i) Timer: %.6f seconds\n", rank, timer);
}
```

With

int MPI_Barrier(comm)	
MPI_Comm comm	Communicator

we block the execution until all nodes are calling the MPI_Barrier function. This is necessary within the start_timer() function if we want to benchmark only a certain part of the coder. Other latencies that have already occured are thus not measured.

As an exercise, measure the performance difference of using distribute_f() against MPI_Scatter() and collect_g() against MPI_Gather() in the previous exercise. It might be handy to increase N to 65536 and to remove the line that prints the solution.

4.2 Communication modes

There are several communication modes we can use in MPI. So far, we have been using the **standard mode**, where the send call is blocking. This means that the call to MPI_Send does not return until the data has been delivered to the receiving node, or stored in a temporary buffer. In standard mode, it is up to the MPI system to decide if and when to use message buffering. An MPI pogrammer should not assume that a standard mode send call is buffered, just because that happens to be the case on his system. The following code is erroneous, because it can stall on some systems (this is called a **deadlock**):

```
if(rank==0)
{
  MPI_Send(&data1, 1, MPI_FLOAT, 1, 11, MPI_COMM_WORLD);
  MPI_Recv(&data0, 1, MPI_FLOAT, 1, 12, MPI_COMM_WORLD, &status);
}
else if(rank==1)
{
  MPI_Send(&data0, 1, MPI_FLOAT, 0, 12, MPI_COMM_WORLD);
  MPI_Recv(&data1, 1, MPI_FLOAT, 0, 11, MPI_COMM_WORLD, &status);
}
```

In **synchronous mode**, the send call does not continue until the matching receive call at the other node has been started. This does not forbid MPI to use buffering, but we can be sure that the other node is receiving the data when the send call completes. Synchronous mode commands are indicated by an initial character 'S' as in **MPI_Ssend**.

In **buffered mode**, the send call does continue whether or not a matching receive call has been started. Buffered mode often comes in handy to prevent possible deadlock situations. We must make sure however, that enough buffer memory is available or an error will occur. Take a look at **MPI_Buffer_attach** and **MPI_Buffer_detach** in the manpages to see how additional bufferspace can be specified. Buffered mode commands are indicated by an initial character 'B' as in **MPI_Bsend**.

In **ready mode**, the send call may only be started if a matching receive call has already been posted. This is not a common situation, but sometimes and on some systems it may improve performance because

MPI can skip part of the work needed to communicate the data. Ready mode commands are indicated by an initial character 'R' as in **MPI_Rsend**.

In **immediate mode**, the send call is non-blocking. The program continues immediately and should not alter or even read the contents of the buffer. The latter restriction is made because some hardware might need it for performance reasons.[5] Immediate mode commands are indicated by an initial character 'I' as in **MPI_Isend**.

5 MPI Overview

5.1 Point-to-Point Communication

```
int MPI_Send(void* buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm)
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int
tag, MPI_Comm comm, MPI_Status *status)
int MPI_Get_Count(MPI_Status *status, MPI_Datatype datatype, int *count)
int MPI_BSend(void* buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm)
int MPI_SSend(void* buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm)
int MPI_RSend(void* buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm)
int MPI_Buffer_attach(void* buffer, int size)
int MPI_Buffer_detach(void* buffer, int* size)
int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm, MPI_Request *request)
int MPI_Ibsend(void* buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm, MPI_Request *request)
int MPI_Issend(void* buf, int count, MPI_Datatype datatype, int dest, int
tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irsend(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 dest, int
tag, MPI_Comm comm, MPI_Request *request)
int MPI_Wait(MPI_Request *request, MPI_Status *status)
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
int MPI_Request_free(MPI_Request *request)
int MPI_Waitany(int count, MPI_Request *array_of_requests, int *index,
MPI_Status *status)
int MPI_Testany(int count, MPI_Request *array_of_requests, int *index,
int *flag, MPI_Status *status)
```

[5] Think about DMA channels that are not CPU-cache coupled.

```
int MPI_Waitall(int count, MPI_Request *array_of_requests, MPI_Status *status)
int MPI_Testall(int count, MPI_Request *array_of_requests, int *flag, MPI_Status
*status)
int MPI_Waitsome(int incount, MPI_Request *array_of_requests, int *outcount,
int *array_of_indices, MPI_Status *array_of_statuses)
int MPI_Testsome(int incount, MPI_Request *array_of_requests, int *outcount,
int *array_of_indices, MPI_Status *array_of_statuses)
int MPI_IProbe(int source, int tag, MPI_Comm comm, int *flag, int flag)
int MPI_Probe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status
*status)
int MPI_Cancel(MPI_Request *request)
int MPI_Test_cancelled(MPI_Status *status, int *flag)
int MPI_Send_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Bsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Ssend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Rsend_init(void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Recv_init(void* buf, int count, MPI_Datatype datatype, int source,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Start(MPI_Request *request)
int MPI_Startall(int count, MPI_Request *array_of_requests)
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype, int
dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype,
int source, MPI_Datatype recvtag, MPI_Comm comm, MPI_Status *status)
int MPI_Sendrecv_replace(void* buf, int count, MPI_Datatype datatype, int
dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
int MPI_Type_contiguous(int count, MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype
oldtype, MPI_Datatype *newtype)
int MPI_Type_hvector(int count, int blocklength, MPI_Aint stride, MPI_Datatype
oldtype, MPI_Datatype *newtype)
int MPI_Type_indexed(int count, int *array_of_blocklengths, int *array_of_displacements,
MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_hindexed(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements
MPI_Datatype oldtype, MPI_Datatype *newtype)
int MPI_Type_struct(int count, int *array_of_blocklengths, MPI_Aint *array_of_displacements,
MPI_Datatype *array_of_types, MPI_Datatype *newtype)
int MPI_Address(void* location, MPI_Aint *adress)
int MPI_Type_extent(MPI_Datatype datatype, MPI_Aint *extent)
int MPI_Type_size(MPI_Datatype datatype, int *size)
int MPI_Type_lb(MPI_Datatype datatype, MPI_Aint* displacement)
int MPI_Type_ub(MPI_Datatype datatype, MPI_Aint* displacement)
int MPI_Type_commit(MPI_Datatype datatype)
int MPI_Type_free(MPI_Datatype datatype)
int MPI_Get_elements(MPI_Status *status, MPI_Datatype datatype, int *count)
int MPI_Pack(void *inbuf, int incount, MPI_Datatype datatype, void *outbuf,
int outsize, int *position, MPI_Comm comm)
int MPI_Unpack(void *inbuf, int insize, int *position, void *outbuf, int
outcount, MPI_Datatype datatype, MPI_Comm comm)
int MPI_Pack_size(int incount, MPI_Datatype datatype, MPI_Comm comm, int
*size)
```

5.2 Collective Communication

```
int MPI_Barrier(MPI_Comm comm)
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root,
MPI_Comm comm)
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, int root,
MPI_Comm comm)
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Scatterv(void* sendbuf, int *sendcounts, int *displs, MPI_Datatype
sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root,
MPI_Comm comm)
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int *recvcounts, int *displs, MPI_Datatype recvtype, MPI_Comm
comm)
int MPI_Alltoall(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Alltoallv(void* sendbuf, int *sendcounts, int *sdispls, MPI_Datatype
sendtype, void* recvbuf, int *recvcounts, int *rdispls, MPI_Datatype recvtype,
MPI_Comm comm)
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
MPI_Op op, int root, MPI_Comm comm)
int MPI_Op_create(MPI_User_function *function, int commute, MPI_Op *op)
int MPI_Op_free(MPI_Op *op)
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm)
int MPI_Reduce_scatter(void* sendbuf, void* recvbuf, int *recvcounts, MPI_Datatype
datatype, MPI_Op op, MPI_Comm comm)
int MPI_Scan(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype,
MPI_Op op, MPI_Comm comm)
```

5.3 Groups, Contexts and Communications

```
int MPI_Group_size(MPI_Group group, int *size)
int MPI_Group_rank(MPI_Group group, int *rank)
int MPI_Group_translate_ranks(MPI_Group group1, int n, int *ranks1, MPI_Group
group2, int *ranks2)
int MPI_Group_compare(MPI_Group group1, MPI_Group group2, int *result)
int MPI_Comm_Group(MPI_Comm comm, MPI_Group *group)
int MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)
int MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group
*newgroup)
int MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group
*newgroup)
int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)
int MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group
*newgroup)
int MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group
*newgroup)
int MPI_Group_free(MPI_Group *group)
int MPI_Comm_size(MPI_Comm comm, int *size)
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_compare(MPI_Comm comm1, MPI_Comm comm2, int *result)
int MPI_Comm_dup(MPI_Comm comm, MPI_Comm *newcom)
int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *newcom)
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcom)
int MPI_Comm_free(MPI_Comm *comm)
int MPI_Comm_test_inter(MPI_Comm comm, int *flag)
int MPI_Comm_remote_size(MPI_Comm comm, int *size)
int MPI_Comm_remote_group(MPI_Comm comm, MPI_Group *group)
int MPI_Intercomm_create(MPI_Comm local_comm, int local_leader, MPI_Comm
peer_comm, int remote_leader, int tag, MPI_Comm *newintercomm)
int MPI_Intercomm_merge(MPI_Comm intercomm, int high, MPI_Comm *newintracomm)
int MPI_Keyval_create(MPI_Copy_function *copy_fn, MPI_Delete_function *delete_fn,
int *keyval, void* extra_state)
int MPI_Keyval_free(int *keyval)
int MPI_Attr_put(MPI_Comm comm, int keyval, void* attribute_val)
int MPI_Attr_get(MPI_Comm comm, int keyval, void* attribute_val, int *flag)
int MPI_Attr_delete(MPI_Comm comm, int keyval)
```

5.4 Process Topologies

```
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
int reorder, MPI_Comm *comm_cart)
int MPI_Dims_create(int nnodes, int ndims, int *dims)
int MPI_Graph_create(MPI_Comm comm_old, int nnodes, int *index, int *edges,
int reorder, MPI_Comm *comm_graph)
int MPI_Topo_test(MPI_Comm comm, int *status)
int MPI_Graphdims_get(MPI_Comm comm, int *nnodes, int *nedges)
int MPI_Graph_get(MPI_comm comm, int maxindex, int maxedges, int *index,
int *edges)
int MPI_Cartdim_get(MPI_Comm comm, int *ndims)
int MPI_Cart_get(MPI_Comm comm, int macdims, int *dims, int *periods, int
*coords)
int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)
int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)
int MPI_Graph_neighbors_count(MPI_Comm comm, int rank, int *nneighbors)
int MPI_Graph_neighbors(MPI_Comm comm, int rank, int maxneighbors, int
*neighbors)
int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source,
int *rank_dest)
int MPI_Cart_sub(MPI_Comm comm, int *remain_dims, MPI_Comm *newcomm)
int MPI_Cart_map(MPI_Comm comm, int ndims, int *dims, int *periods, int
*newrank)
int MPI_Graph_map(MPI_Comm comm, int nnodes, int *index, int *edges, int
*newrank)
```

5.5 Environmental Inquiry

```
int MPI_Get_processor_name(char *name, int *resultlen)
int MPI_Errhandler_create(MPI_Handler_function *function, MPI_Errhandler
*errhandler)
int MPI_Errhandler_set(MPI_Comm comm, MPI_Errhandler errhandler)
int MPI_Errhanlder_get(MPI_Comm comm, MPI_Errhandler *errhandler)
int MPI_Errhandler_free(MPI_Errhandler *errhandler)
int MPI_Error_string(int errorcode, char *string, int *resultlen)
int MPI_Error_class(int errorcode, int *errorclass)
double MPI_Wtime(void)
double MPI_Wtick(void)
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize(void)
int MPI_Initialized(int *flag)
int MPI_Abort(MPI_Comm comm, int errorcode)
```

5.6 Profiling

```
int MPI_Pcontrol(const int level, ...)
```

References

[1] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Journal of Supercomputer Applications, 8(3/4):157--416, 1994

[2] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard (version 1.1)*, 1995

[3] Message Passing Interface Forum, *MPI-2: Extensions to the Message-Passing Interface*, 1997

[4] Snir, Otto, Huss-Lederman, Walker, Dongarra, *MPI--The Complete Reference Volume 1, The MPI Core*, (2nd ed.), 1998

[5] Grop, Huss-Lederman, Lumsdaine, Lusk, Nitzberg, Saphir, Snir, *MPI--The Complete Reference Volume 2, The MPI Extensions*, 1998