Lesson 1

The goals of this lesson are:

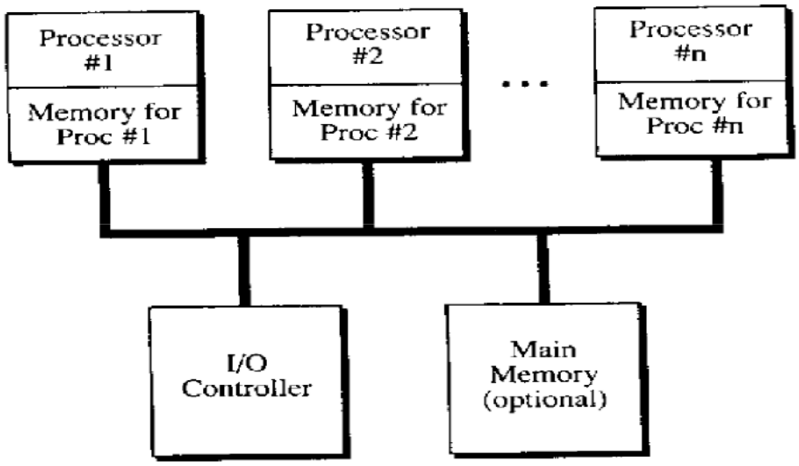
- understanding the MPI programming model
- managing the MPI environment
- handling errors
- point-to-point communication

1. The MPI Environment

MPI (Message Passing Interface) is a message passing library standard, defined by the MPI Forum (http://www.mpi-forum.org/), aimed to offer efficiency, portability and flexibility. All laboratory lessons will refer to the MPICH implementation of MPI (http://www.mpich.org/).

1.1. The MPI Programming Model

MPI runs on distributed memory systems, shared memory systems, or hybrid systems.



The MPI programming model assumes the distributed memory model regardless of the system's physical architecture.

An MPI program has the following structure:

<Header files>

<Declarations and prototypes>

<Main program>

…..
<Serial code>
…..

<Initialization of the MPI Environment>
<Parallel code goes here>
….
<Send messages, receive messages, perform computations>
….
<Finalize the MPI Environment>

<Program End>

The header file **“mpi.h”** is required by all MPI programs. Don't forget to:

```
#include “mpi.h”
```

1.2. Managing the MPI Environment

1.2.1 Initializing and closing the MPI environment

The MPI Environment is initialized by calling the *MPI_Init* function. *MPI_Init* must be called before any other MPI function call, and it must be called only once per program.

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

Parameters *argc*, and *argv* are the command line arguments and they will be passed to all processes.

To check whether the MPI_Init has been called and the MPI environment has been successfully initialized, you can call:

```
int MPI_Initialized(int *flag)
```

The *flag* will store a value different than zero if the environment has been initialized, otherwise it will store zero.

An MPI program ends with a call to:

```
int MPI_Finalize( void )
```

To terminate the execution of all processes inside a communicator, you need to call:

```
int MPI_Abort(MPI_Comm comm, int errorcode)
```

The *errorcode* will be returned to the invoking environment.

1.2.2 MPI Communicators:

MPI achieves communication goals by using communicators and groups. Communicators and groups define collections of processes that may communicate with each other. Communicators and groups will be discussed later.

For now, we will use the MPI_COMM_WORLD whenever a communicator is needed. MPI_COMM_WORLD is a predefined communicator that includes all the processes in the MPI Environment.

Within a communicator, every process has a unique identifier, called Rank, which is an integer between 0 and the number of processes in the communicator. Every process will get a rank in the initialization phase. The rank is needed when sending/receiving messages in order to uniquely identify the source and destination processes.

To obtain the rank of the current process you can call:

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

Parameter *comm* represents the communicator (MPI_COMM_WORLD for now) while the *rank* parameter is an output integer variable which will store the rank.

The size of a communicator can be obtained by calling:

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

Parameter *comm* represents the communicator while the output parameter *size* will store the size of the communicator.

1.2.3 Other routines:

To get the processor's name you need to call:

```
int MPI_Get_processor_name( char *name, int *resultlen )
```

Parameter *name* will point to the name of the processor, while parameter *resultlength* will store the length of the processor's name. The *name* buffer must be at least MPI_MAX_PROCESSOR_NAME characters long.

To get the MPI version, you need to call:

```
int MPI_Get_version( int *version, int *subversion )
```

The two parameters will hold the version and subversion of MPI.

To get elapsed time for the current process, you need to call:

double MPI_Wtime(void)

The function returns the elapsed time in double precision. See also MPI_Wtick.

1.3. Error Handling

All MPI routines, except MPI_Wtime and MPI_Wtick return an error code or include an error code output parameter. The default behavior (MPI_ERRORS_ARE_FATAL) in case of an error is to abort. However this behavior can be replaced by using the function

int MPI_Comm_set_errhandler(MPI_Comm comm, MPI_Errhandler errhandler)

Parameter *errhandler* represents the new error handler to be set.

There is a predefined error handler, called **MPI_ERRORS_RETURN** which in case of an error, will no longer abort, but return the error code.

1.4. Examples

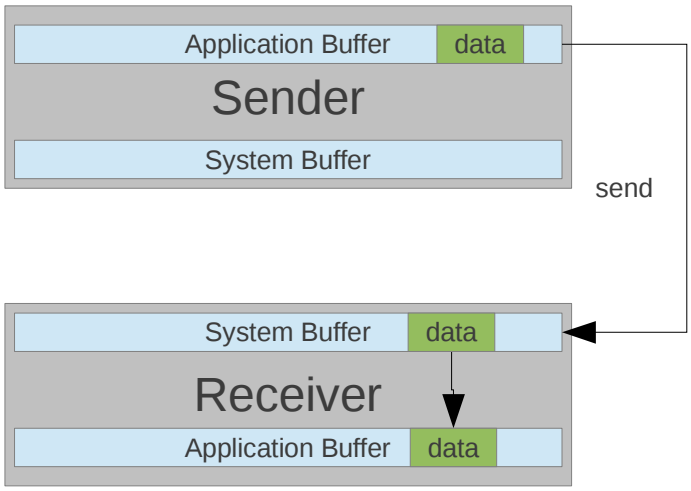
Example 1: Initialization, finalize, the use of ranks, communicator size and processor name

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

int main(int argc, char* argv){
  int  numtasks, rank, len, rc;
  char hostname[MPI_MAX_PROCESSOR_NAME];

  rc = MPI_Init(&argc,&argv);
  if (rc != MPI_SUCCESS) {
    printf ("Error starting MPI program. Terminating.\n");
    MPI_Abort(MPI_COMM_WORLD, rc);
  }

  MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Get_processor_name(hostname, &len);
  printf ("Number of tasks= %d My rank= %d Running on %s\n", numtasks,rank,hostname);

  /*******  do some work *******/

  MPI_Finalize();
  }
```

2.Point to Point Communication

2.1. Point to Point Operations

Point to point communication refers to message exchange between two and only two different MPI processes. One of the processes sends a message while the other receives it. MPI uses buffers for communication. There are two layers of buffers: application buffer and system buffer.



Ideally send/receive requests would be perfectly synchronized. In practice this is not the case. MPI deals with such issues by using buffers. The above figure illustrates how a data object is being sent from one host to the other. The system buffer stores the data until the receiver process is ready for the receive operation. Passing the data through the system buffer achieves asynchronous message passing and allows a receiver which can only accept one message at a time, to receive multiple messages arrived simultaneously.
The system buffer is entirely handled by the MPI library. The programmer will not interact directly with the system buffer.
The application buffer is entirely handled by the programmer and it usually consists of the program's variables and other memory allocations.
MPI routines accomplish blocking and non-blocking communication:

Blocking routines:

- A blocking send returns immediately after the application buffer (data to be sent) is safe for reuse (any change in the buffer won't affect the data that needs to be sent). This doesn't mean the data was actually received. For example, a blocking routine might return immediately after transferring the data into the receiver's system buffer without being received properly by the target process. Blocking send can be synchronous (there will be handshaking that confirms a successful receive), or asynchronous (the send returns immediately after saving the data into the system buffer).
- Blocking receive returns only when the data base been entirely received and is ready for use.

Non-blocking routines:

- Non-blocking send and receive do not wait for any confirmation that the call has been performed. They only request the send/receive and return immediately. No prediction upon the moment when the send/receive will be performed can be made.
- Changing the content of the application buffer in the context of a non-blocking send might compromite the data to be sent.
- MPI exposes routines for checking whether a send/receive has been successfully performed.
- Non-blocking communication are mainly used to overlap computation with communication increasing the performance.

MPI guarantees that the messages sent between two hosts are to be received in exactly the same order that they've been sent.

2.2. Point-to-Point MPI Routines:

2.2.1 Send and Receive

Blocking send/receive:

```
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)
```

Non-blocking send/receive:

```
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)
```

Synchronous send:

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

Parameters:

buf – Data to be sent / received (application buffer).

count – Size (number of elements) of the buffer to be sent / received.

dataType – One of the following values:

C type	MPI type
char	MPI_CHAR
wchar_t	MPI_WCHAR
signed short int	MPI_SHORT
signed int	MPI_INT
signed long int	MPI_LONG
Float	MPI_FLOAT
Double	MPI_DOUBLE
long double	MPI_LONG_DOUBLE
byte	MPI_BYTE

For more types see the MPI documentation.

dest – the rank of the receiver processes

source – the rank of the source process. MPI_ANY_SOURCE allows receiving a message regardless of its source

tag – the message tag. This can be any non-negative integer. Send/Receive perform matching on the tag. MPI_ANY_TAG can be used to receive a message regardless of its tag.

comm – the MPI communicator to be used

status – structure that indicates the source and tag of the message.

request – non-blocking operations are being assigned a request handle returned through this parameter. This handle can be used in wait operations.

2.2.2 Wait and Probe

The following routines wait for requests to be completed:

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

int MPI_Waitany(int count, MPI_Request array_of_requests[], int *index,

```
                MPI_Status *status)
```

int MPI_Waitall(int count, MPI_Request array_of_requests[],

```
                MPI_Status array_of_statuses[])
```

```
int MPI_Waitsome(int incount, MPI_Request array_of_requests[],
                 int *outcount, int array_of_indices[],
                 MPI_Status array_of_statuses[])
```

request – the request to wait for

status – status object

incount – length of array_of_requests

outcount – number of completed requests

array_of_requests – request handles to wait for

array_of_indices – array of indices of operations that have completed (array of integers between 0 and incount-1

array_of_statuses – array of status objects for operations that have completed

2.3. Examples

Example 1: *Hello world – blocking send.*

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  MASTER              0

int main (int argc, char *argv[])
{
        int  numprocs, procid, len;
        char hostname[MPI_MAX_PROCESSOR_NAME];
        int  partner, message;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
```

```
        MPI_Comm_rank(MPI_COMM_WORLD,&procid);
        MPI_Get_processor_name(hostname, &len);
        printf ("Hello from proc %d on %s!\n", procid, hostname);
        if (procid == MASTER)
                printf("MASTER: Number of MPI procs is: %d\n",numprocs);

        /* determine partner and then send/receive with partner */
        if (procid < numprocs/2) {
                partner = numprocs/2 + procid;
                MPI_Send(&procid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
                MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
        }
        else if (procid >= numprocs/2) {
                partner = procid - numprocs/2;
                MPI_Recv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &status);
                MPI_Send(&procid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD);
        }
        /* print partner info and exit*/
        printf("Proc %d is partner with %d\n",procid,message);

        MPI_Finalize();
}
```

Example 2: *Hello world – non-blocking send.*

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define  MASTER                 0

int main (int argc, char *argv[])
{
        int  numprocs, procid, len;
        char hostname[MPI_MAX_PROCESSOR_NAME];
        int  partner, message;
        MPI_Status status;
        MPI_Request reqs[2];

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&procid);
        MPI_Get_processor_name(hostname, &len);
        printf ("Hello from proc %d on %s!\n", procid, hostname);
        if (procid == MASTER)
                printf("MASTER: Number of MPI procs is: %d\n",numprocs);
```

```
/* determine partner and then send/receive with partner */
if (procid < numprocs/2)
  partner = numprocs/2 + procid;
else if (procid >= numprocs/2)
  partner = procid - numprocs/2;

MPI_Irecv(&message, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &reqs[0]);
MPI_Isend(&procid, 1, MPI_INT, partner, 1, MPI_COMM_WORLD, &reqs[1]);

/* now block until requests are complete */
MPI_Waitall(2, reqs, &status);

/* print partner info and exit*/
printf("Proc %d is partner with %d\n",procid,message);

MPI_Finalize();

}
```

3. Exercises:

1. Write a program that prints all the prime numbers less than N using M processes.

2. Write a program that searches an element inside an array and prints its position in case the array contains that element, otherwise it prints 'Not found.'

3. Let there be *n* processes. Each process generates m (m>=100) random numbers (<=1000), prints them to the console, computes their sum and prints it to the console. Determine the time it takes for each process to complete the job.