

Message Passing Interface (MPI)

Preliminaries:

MPI provides support for:

- Point-to-point & collective (i.e. group) communications
- Inquiry routines to query the environment (how many nodes are there, which node number am I, etc.)
- Constants and data-types

We will start with the basics: initialising MPI, and using point-to-point communication

Message Passing Interface (MPI)

Naming convention

By convention, all MPI identifiers are prefixed by 'MPI_'. C routines contain lower case (i.e. 'MPI_Init'), constants are all in upper case (e.g. 'MPI_FLOAT' is an MPI C data-type). C routines are actually integer functions which return a status code (you are strongly advised to check these for errors!).

Running MPI

Number of processors used is specified in the command line, when running the MPI loader that loads the MPI program onto the processors, to avoid hard-coding this into the program e.g.
mpirun -np N exec

Message Passing Interface (MPI)

Writing a program using MPI: what is parallel, what is not

- Only one program is written. By default, every line of the code is executed by each node running the program. For instance, if the code contains “int result=0”, each node will locally create a variable and assign the value.
- When a section of the code needs to be executed by only a subset of nodes, it has to be explicitly specified. For instance, providing that we are using 8 nodes, and that MyID is a variable storing the rank of the node (from 0 to 7, we will see how to get it later), this section of code assign to “result” 0 for the first half of them, and 1 for the second.

```
int result;  
  
if(MyID < 4) result = 0;  
  
else result = 1;
```

Message Passing Interface (MPI)

Common MPI Routines

MPI has a ‘kitchen sink’ approach with 129 different routines, but most basic programs can get away with using six. As usual use `#include "mpi.h"` in C.

| | |
|---------------------|--------------------------------|
| MPI_Init | Initialise MPI computation |
| MPI_Finalize | Terminate MPI computation |
| MPI_Comm_size | Determine number of processes |
| MPI_Comm_rank | Determine my process number |
| MPI_Send, MPI_Isend | Blocking, non-blocking send |
| MPI_Recv, MPI_Irecv | Blocking, non-blocking receive |

Message Passing Interface (MPI)

MPI Initialisation

In all MPI-written programs, MPI must be initialised before use, and finalised at the end. All MPI-related commands and types must be handled within this section of code.

`MPI_Init`

Initialise MPI computation

`MPI_Init` takes two parameters as an input (`argc` and `argv`), and is used to start the MPI environment, e.g. it creates the default communicator (more details later) and assign a rank to each node.

Message Passing Interface (MPI)

MPI Finalisation

MPI_Finalize Terminate MPI computation

This routine cleans up all MPI state. Once this routine is called, no MPI routine (even MPI_INIT) may be called. The user must ensure that all pending communications involving a process completes before the process calls MPI_FINALIZE.

Message Passing Interface (MPI)

Basic Inquiry Routines

At various stages in a parallel-implemented function, it may be useful to know how many nodes the program is using, or what the rank of the current node is.

`MPI_Comm_size` Determine number of processes

This function returns the number of processes/nodes as an integer, and takes only one parameter, a communicator.

If we have time, we will see later in the course what a communicator is, but in most cases you will only use the default one: `MPI_COMM_WORLD`.

Message Passing Interface (MPI)

Basic Inquiry Routines (2)

MPI_Comm_rank Determine my process number

This function is used to determine what the rank of the current process/node on a particular communicator.

If we assume we have two communicators, it is possible, and quite usual, that the ranks of the same node would differ.

Again, in most cases, this function will only be used with the default communicator as an input (MPI_COMM_WORLD), and it will return (as an integer) the rank of the node on that communicator.

Message Passing Interface (MPI)

Point-to-Point communications in MPI

This involves communication between two processors, one sending, and the other receiving. Certain information is required to specify the message:

- Identification of sender processor
- Identification of destination/receiving processor
- Type of data (MPI_INT, MPI_FLOAT etc)
- Number of data elements to send (i.e. array/vector info)
- Where the data to be sent is in memory (pointer)
- Where the received data should be stored in (pointer)

Message Passing Interface (MPI)

Sending data MPI_Send, MPI_Isend

MPI_Send is used to perform a blocking send, which means that the process waits for the communication to finish before going to the next command.

This function takes six parameters:

- the location of the data to be sent (i.e. a pointer)
- the number of data elements to be sent
- the type of data (e.g. MPI_INT, MPI_FLOAT, etc.)
- the rank of the receiving node
- a tag (for identification of the communication)
- the communicator to be used for transmission

MPI_Isend is non-blocking, so an additional parameter, to allow for verification of communication success is needed. It is a pointer to an element of type MPI_Request. We will see in an example how to use this.

Message Passing Interface (MPI)

Receiving data MPI_Recv, MPI_Irecv

MPI_Recv is used to perform a blocking receive, which means that the process waits for the communication to finish before going to the next command.

This functions takes seven parameters:

- the location where to store received data (i.e. a pointer)
- the number of data elements to be received
- the type of data (e.g. MPI_INT, MPI_FLOAT, etc.)
- the rank of the sending node
- a tag (for identification of the communication)
- the communicator to be used for transmission
- a pointer to a structure of type MPI_Status, containing the rank of the processor sending data, the tag of the communication, and the error status.

For the non-blocking MPI_Irecv, MPI_Request* replaces MPI_Status*.

Message Passing Interface (MPI)

A first example: Hello World

```
#include <mpi.h>

main(int argc, char *argv[]) {

    int myid, size;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("process %d out of %d says Hello\n", myid, size);

    MPI_Finalize();

}
```

Message Passing Interface (MPI)

Our first “real” MPI program: exchanging two values

```
#include <mpi.h>

main(int argc, char *argv[]) {

    int myid, otherid, size, length = 1, tag = 1;

    int myvalue, othervalue;

    MPI_Status status;

    /* initialize MPI and get own id (rank) */

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Message Passing Interface (MPI)

```
if (size!=2) {  
    printf("use exactly two processes\n");  
    exit(1);  
}  
if (myid == 0) {  
    otherid = 1; myvalue = 14;  
}  
else {  
    otherid = 0; myvalue = 25;  
}  
printf("process %d sending %d to process %d\n", myid, myvalue, otherid);
```

Message Passing Interface (MPI)

```
/* Send one integer to the other node (i.e. "otherid") */
MPI_Send(&myvalue,1,MPI_INT,otherid,tag,MPI_COMM_WORLD);

/* Receive one integer from any other node */
MPI_Recv(&othervalue,1,MPI_INT,MPI_ANY_SOURCE,
        MPI_ANY_TAG,MPI_COMM_WORLD, &status);
printf("process %d received a %d\n", myid, othervalue);

/* Terminate MPI */
MPI_Finalize();
}
```

Message Passing Interface (MPI)

Compiling and running

- To compile programs using MPI, you need an “MPI-enabled” compiler. On our cluster, we use mpicc to compile C programs containing MPI commands or mpicxx for C++.
- Before running an executable using MPI, you need to make sure the "multiprocessing daemon" (MPD) is running. It makes the workstations into a "virtual machine" that can run MPI programs. When you run an MPI program, requests are sent to MPD daemons to start up copies of the program. Each copy can then use MPI to communicate with other copies of the same program running in the virtual machine. Just type “mpd &” in the terminal.
- To run the executable, you then type “mpirun -np N ./executable_file”, where N is the number to be used to run the program. This value is then used in your program by MPI_Init to allocate the nodes and create the default communicator.

Message Passing Interface (MPI)

Another example: “Ring” communication

```
int main(int argc, char* argv[] ) {

    int rank, value, size;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    do {
        if (rank == 0) {
            scanf( "%d", &value );
            MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
        }
    }
```

Message Passing Interface (MPI)

```
else {
    MPI_Recv( &value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD,
             &status );
    if (rank < size - 1)
        MPI_Send( &value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD );
    }
    printf( "Process %d got %d\n", rank, value );
} while (value >= 0);

MPI_Finalize( );
return 0;
}
```

Message Passing Interface (MPI)

An implementation of the matrix-vector example

```
int main(int argc, char* argv[] ) {
    int A[4][4], b[4], c[4], line[4], temp[4], local_value, myid;
    MPI_Init(&argc, &argv);  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if(myid==0) {
        for(int i=0;i<4;++i) {
            b[i]=4-i;
            for(int j=0; j<4; ++j) {
                A[i][j]=i+j;
            }
        }
        line[0]=A[0][0]; line[1]=A[0][1];
        line[2]=A[0][2]; line[3]=A[0][3];
    }
}
```

Message Passing Interface (MPI)

```
if(myid==0) {  
    for(int i=1;i<4;++i) {  
        temp[0]=A[i][0]; temp[1]=A[i][1]; temp[2]=A[i][2]; temp[3]=A[i][3];  
        MPI_Send(temp,4,MPI_INT,i,i,MPI_COMM_WORLD);  
        MPI_Send(b,4,MPI_INT,i,i,MPI_COMM_WORLD);  
    }  
}  
else {  
    MPI_Recv(line,4,MPI_INT,0,myid, MPI_COMM_WORLD,  
            MPI_STATUS_IGNORE);  
    MPI_Recv(b,4,MPI_INT,0,myid, MPI_COMM_WORLD,  
            MPI_STATUS_IGNORE);  
}
```

Message Passing Interface (MPI)

```
c[myid]=line[0]*b[0]+line[1]*b[1]+line[2]*b[2]+line[3]*b[3];
if(myid!=0) {
    MPI_Send(&(c[myid]),1,MPI_INT,0,myid,MPI_COMM_WORLD);
}
else {
    for(int i=1;i<4;++i) {
        MPI_Recv(&(c[i]),1,MPI_INT,i,i,MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);
    }
}
MPI_Finalize(); return 0;
}
```

Message Passing Interface (MPI)

An implementation of the Pi calculation example

```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    #define INT_MAX_ 1000000000

    int points=10000, myid, size, inside=0, total=0;
    double x,y, Pi_comp, Pi_real=3.141592653589793238462643;

    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if(myid==0) {
        for(int i=1;i<size;++i)
            MPI_Send(&points,1,MPI_INT,i,i,MPI_COMM_WORLD);
    }
}
```

Message Passing Interface (MPI)

```
else
    MPI_Recv(&points,1,MPI_INT,0,i,MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);

rands=new double[2*points];
for(int i=0;i<2*points; ) {
    rands[i]=random();
    if(rands[i] <= INT_MAX_) ++i;
}
for(int i=0; i<points;++i) {
    x=rands[2*i]/INT_MAX_;
    y=rands[2*i+1]/INT_MAX_;
    if((x*x+y*y)<1) ++inside;
}
delete[] rands;
```

Message Passing Interface (MPI)

```
if(myid==0) {
    for(int i=1;i<size;++i) {
        int temp;
        MPI_Recv(&temp,1,MPI_INT,i,i,MPI_COMM_WORLD,
                MPI_STATUS_IGNORE);

        inside+=temp;
    }
    MPI_Send(&inside,1,MPI_INT,0,i,MPI_COMM_WORLD);
    Pi_comp=4*(double)total/(double)(size*points);
    if(myid==0) {
        cout << "Points: " << total << " inside. (Total=" << size*points << ")" << endl;
        cout << "Value obtained: " << Pi_comp << endl << "Pi: " << Pi_real << endl;
        cout << "Error: " << fabs(Pi_comp-Pi_real) << endl << endl;
    }
    MPI_Finalize(); return 0;
}
```