

Message Passing Interface (MPI)

Reminder:

- MPI programs are compiled using specific compilers, e.g. mpicc
- MPI programs require that MPD (multiprocessing daemon) runs in the background, and can then be launched using “mpirun -np N exec_file” (N: number of nodes)
- MPI programs start with MPI_Init and finish with MPI_Finalize
- MPI_Send takes six parameters, and MPI_Recv seven (remember which ones, and the order!)
- The latest version of the examples (i.e. the ones you must be able to code) will be online, and you will have access to the cluster until the end of the semester (details on Wednesday)

Message Passing Interface (MPI)

Collective communications in MPI

Groups are sets of processors that communicate with each other in a certain way. Such communications permit a more flexible mapping of the language to the problem (allocation of nodes to subparts of the problem etc). MPI implements Groups using data objects called *Communicators*. A special Communicator is defined (called 'MPI_COMM_WORLD') for the group of all processes. Each Group member is identified by a number (its *Rank* 0..n-1).

To create a communicator

There are three steps to create new communication structures: accessing the group corresponding to MPI_COMM_WORLD, using this group to create sub-groups, allocating new communicators for this group. We will see in more details in the last examples.

Message Passing Interface (MPI)

Some sophisticated MPI routines

MPI_Barrier	Synchronise
MPI_Bcast	Broadcast same data to all procs
MPI_Gather	Get data from all procs
MPI_Scatter	Send different data to all procs
MPI_Reduce	Combine data from all onto one proc
MPI_Allreduce	Combine data from all procs onto all procs

The advantage of such global communication routines is that the MPI system can implement them more efficiently than the programmer, involving far less function calls. Also the system will have more opportunity to overlap message transfers with internal processing and to exploit parallelism that might be available in the communications network.

Message Passing Interface (MPI)

Synchronising nodes MPI_Barrier

- MPI_Barrier is used to synchronise a set of nodes.
- It blocks the caller until all group members have called it. The call returns at any process only after all group members have entered the call.
- This function takes only one parameter, the communicator (i.e. group of nodes) to be synchronised.
- As we previously saw with other functions, it will most of the times be used with the default communicator, MPI_COMM_WORLD.

Message Passing Interface (MPI)

Sending data from a node to all the others MPI_Bcast

MPI_Bcast is used to send data from one node to all the other ones in one single command.

This function takes five parameters:

- the location of the data to be sent, and where to store received data (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 sending node
- the communicator to be used for transmission.

Message Passing Interface (MPI)

Receiving data from all nodes MPI_Gather

MPI_Gather is used to gather on a single node data scattered over a group of nodes.

This function takes eight 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 sent (e.g. MPI_INT, MPI_FLOAT, etc.)
- the location where to store data on the receiving node
(i.e. a pointer)
- the number of elements to be received
- the type of data received (e.g. MPI_INT, etc.)
- the rank of the receiving node
- the communicator to be used for transmission.

Message Passing Interface (MPI)

Sending different data to all nodes MPI_Scatter

MPI_Scatter is used to scatter data from a single node to a group of nodes.

This function takes eight 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 sent (e.g. MPI_INT, MPI_FLOAT, etc.)
- the location where to store data on the receiving node
(i.e. a pointer)
- the number of elements to be received
- the type of data received (e.g. MPI_INT, etc.)
- the rank of the sending node
- the communicator to be used for transmission.

Message Passing Interface (MPI)

Reducing data on a single node MPI_Reduce

MPI_Reduce is used to reduce values on all nodes of a group to a single value, on one node.

This function takes seven parameters: - the location of the data to be sent (i.e. a pointer)

- the location where to store data on the receiving node
(i.e. a pointer)
- the number of data elements to be sent from each node
- the type of data sent (e.g. MPI_INT, MPI_FLOAT, etc.)
- the operation to combine the results (e.g. MPI_SUM)
- the rank of the receiving node
- the communicator to be used for transmission.

Message Passing Interface (MPI)

Reducing data, and sending to all nodes MPI_Allreduce

MPI_Allreduce is used to reduce values on all nodes of a group to a single value, and distribute the result back to all nodes. (i.e. it is equivalent to MPI_Reduce + MPI_Bcast)

This function takes seven parameters: - the location of the data to be sent (i.e. a pointer)

- the location where to store data on the receiving node
(i.e. a pointer)
- the number of data elements to be sent from each node
- the type of data sent (e.g. MPI_INT, MPI_FLOAT, etc.)
- the operation to combine the results (e.g. MPI_SUM)
- the communicator to be used for transmission.

Message Passing Interface (MPI)

A new 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)

```
MPI_Bcast(b,4,MPI_INT,0,MPI_COMM_WORLD);
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);
        /* no need to send b here */
    }
else {
    MPI_Recv(line,4,MPI_INT,0,myid, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    /* no need to receive b here */
}
```

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);
}    /* Calculating the trace in one command */
int local_value=line[myid], trace=0;
MPI_Reduce(&local_value,&trace,1,MPI_INT,MPI_SUM,0,
          MPI_COMM_WORLD);
MPI_Finalize(); return 0;
}
```

Message Passing Interface (MPI)

A new 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) {
        printf("Enter the number of points each proc must generate: \n");
        scanf(&points);
    }
    MPI_Bcast(&points,1,MPI_INT, 0, MPI_COMM_WORLD);
}
```

Message Passing Interface (MPI)

```
/* no need for the nodes to “receive”, this was included in the broadcast */  
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;  
    }  
    else  
        MPI_Send(&inside,1,MPI_INT,0,i,MPI_COMM_WORLD);  
=> this is all replaced by:      */  
MPI_Reduce(&inside,&total,1,MPI_INT,MPI_SUM,0, MPI_COMM_WORLD);  
/* (or MPI_Allreduce if all nodes need the value) */  
Pi_comp=4*(double)total/(double)(size*points);  
/* Display goes here (it stays the same as before) */  
MPI_Finalize(); return 0;  
}
```

Message Passing Interface (MPI)

Using communicators

Creating a new group (and communicator) by excluding the first node.

```
MPI_Comm comm_world, comm_worker;  
MPI_Group group_world, group_worker;  
  
comm_world = MPI_COMM_WORLD;  
MPI_Comm_group(comm_world, &group_world);  
MPI_Group_excl(group_world, 1, 0, &group_worker); /* process 0 not member */  
  
MPI_Comm_create(comm_world, group_worker, &comm_worker);
```

Message Passing Interface (MPI)

Warning:

`MPI_Comm_create()` is a collective operation, so all the processes in the old communicator must call it - even those not going to be part of the new communicator.

Message Passing Interface (MPI)

MPI routines to create groups

MPI_Comm_group(MPI_Comm comm, MPI_Group *group)

MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

MPI_Group_intersection(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

MPI_Group_difference(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup)

MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_Group_excl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

MPI_Group_range_incl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)

MPI_Group_range_excl(MPI_Group group, int n, int ranges[][3], MPI_Group *newgroup)

Message Passing Interface (MPI)

Using communicators, II (there may be some change for the version on the cluster)

```
#include "mpi.h"
#include <stdio.h>
#define NPROCS 8
int main(int argc, char* argv[]) {
    int rank, new_rank, sendbuf, recvbuf,
        ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};

    MPI_Group orig_group, new_group;
    MPI_Comm new_comm;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    sendbuf = rank;

    /* Extract the original group handle */

    MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
```

Message Passing Interface (MPI)

```
/* Divide tasks into two distinct groups based upon rank */

if (rank < NPROCS/2)
    MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
else
    MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);

/* Create new communicator and then perform collective communications */

MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
MPI_Group_rank (new_group, &new_rank);

printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

MPI_Finalize();
}
```

Message Passing Interface (MPI)

Final reminder:

- MPI programs need specific compilers (e.g. mpicc), MPD and mpirun
- MPI programs start with MPI_Init and finish with MPI_Finalize
- Four functions for point-to-point communication
- Six more advanced functions, for synchronise, and perform collective communication
- Nine functions (at least three!) to create new groups and communicators
- Many examples to remember everything