Introduction to MPI

Table of Contents

1. Program Structure
2. Communication Model
   Topology
   Messages
3. Basic Functions
4. Made-up Example Programs
5. Global Operations
6. LaPlace Equation Solver
7. Asynchronous Communication
8. Communication Groups
9 MPI Data Types
Introduction to MPI

MPI Program Structure

1. MPI is a set of precompiled library routines that the user links with their code.

2. An “MPI” parallel program is a sequential program which has been modified to include calls to MPI routines and conditional statements to adapt the execution of the program to its local context.

3. Each processor executes the same program using local processor id to determine its behavior.

4. MPI distributes the programs to the processors, loads them and initiates execution on each processor.

5. Environment specification and execution initiation is external to MPI.
Introduction to MPI

Communication Model

1. A communicator (MPI_Comm) is a collection of processors that can send messages to each other. For basic programs, the only communicator needed is MPI_COMM_WORLD. It is predefined in MPI and consists of all the processors running when program execution begins.

2. Subsets of MPI_COMM_WORLD can be created to partition the processors into smaller communication groups.

3. Message communicators must match between message sender and receiver.

4. Communicators can also be used to determine the number of processors participating in a particular communicator set and the sequence of the processor in the communicator.

5. The processor's location in the communicator sequence is determined by the MPI_Comm_rank function.

6. The total number of processors in the communicator can be determined by the MPI_Comm_size.
Introduction to MPI

Hello World - MPI

PROGRAM hello
#include "mpif.h"
INTEGER error, rank, size, i, tag, status(MPI_STATUS_SIZE), C
CHARACTER*12 message, inmsg
CALL MPI_INIT (error)
C  Find out my rank in the global communicator MPI_COMM_WORLD
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)
C  Find out size of the global communicator MPI_COMM_WORLD
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, error)
C  Do conditional work if my rank is 0
    tag = 17
    IF (rank .EQ. 0) THEN
      message = 'Hello, World'
      DO i=1, size - 1
        CALL MPI_SEND(message, 12, MPI_CHARACTER, i, tag, MPI_COMM_WORLD, error)
      ENDDO
    ELSE
      CALL MPI_RECV(inmsg, 12, MPI_CHARACTER, 0, tag, MPI_COMM_WORLD, status, error)
      WRITE(*,*) node 'rank', i, 'inmsg
    ENDI
C  Exit and finalize MPI
CALL MPI_FINALIZE(error)
STOP
END
Introduction to MPI

MPI_INIT(int *argc, char **argv)
Initialize a computation.
argc, argv are required only in the C language binding,
where they are the main program's arguments.

MPI_FINALIZE()
Shut down a computation.

MPI_COMM_SIZE(comm, size)
Determine the number of processes in a computation.
IN comm communicator (handle)
OUT size number of processes in the group of comm (integer)

MPI_COMM_RANK(comm, pid)
Determine the identifier of the current process.
IN comm communicator (handle)
OUT pid process id in the group of comm (integer)

MPI_SEND(buf, count, datatype, dest, tag, comm)
Send a message.
IN buf address of send buffer (choice)
IN count number of elements to send (integer ≥0)
IN datatype datatype of send buffer elements (handle)
IN dest process id of destination process (integer)
IN tag message tag (integer)
IN comm communicator (handle)

MPI_RECV(buf, count, datatype, source, tag, comm, status)
Receive a message.
OUT buf address of receive buffer (choice)
IN count size of receive buffer, in elements (integer ≥0)
IN datatype datatype of receive buffer elements (handle)
IN source process id of source process, or MPI_ANY_SOURCE (integer)
IN tag message tag, or MPI_ANY_TAG (integer)
IN comm communicator (handle)
OUT status status object (status)
Introduction to MPI

Message Properties

1. MPI messages are one dimensional array of items and are the first argument of the send (MPI_Send) and receive (MPI_Recv) functions.

2. Argument to indicate where the array starts, arguments that indicate the number of elements in the array (count) and the type of each element (datatype) are also passed to the MPI functions.

3. The tag and comm arguments are used to differentiate multiple messages originating from the same processor.

4. The status argument in the receive function stores information about the source, size, and tag of the message. This is useful in cases where the receive is allowed to receive a set of possible sources.
Introduction to MPI

An Parallel Pseudo-Program Using the MPI Library

program main
begin
MPI_INIT() //Initiate computation
MPI_COMM_SIZE(MPI_COMM_WORLD, count)//Find # of processes
MPI_COMM_RANK(MPI_COMM_WORLD, myid) //Find my id
print("I am", myid, "of", count) //Print message
MPI_FINALIZE() //Shut down
end
1. If the program on the previous slide is executed by four processes, we will obtain something like the following output.

2. The order in which the output appears is not defined; however, we assume here that the output from individual print statements is not interleaved.

   I am 1 of 4
   I am 3 of 4
   I am 0 of 4
   I am 2 of 4
Introduction to MPI

Foundry - Bridge Process

(a) foundry bridge

(b) foundry bridge

9/24/2001 Parallel Programming - Fall 2001
Introduction to MPI

program main
begin
  MPI_INIT()
  MPI_COMM_SIZE(MPI_COMM_WORLD, count)
  if count != 2 then exit
  MPI_COMM_RANK(MPI_COMM_WORLD, myid)
  if myid = 0 then
    foundry(100)
  else
    bridge()
  endif
  MPI_FINALIZE()
end

procedure foundry(numgirders)  Code for process 0
begin
  for i = 1 to numgirders  Send messages
    MPI_SEND(i, i, MPI_INT, 1, 0, MPI_COMM_WORLD)
  endfor
  i = -1
  MPI_SEND(i, i, MPI_INT, 1, 0, MPI_COMM_WORLD)
end

procedure bridge  Code for process 1
begin
  MPI_RECV(msg, i, MPI_INT, 0, 0, MPI_COMM_WORLD, status)
  while msg != -1 do
    use_girders(msg)
    MPI_RECV(msg, i, MPI_INT, 0, 0, MPI_COMM_WORLD, status)
  enddo
end

Program 8.1: MPI implementation of bridge construction problem. This program is designed to be executed by two processes.
Introduction to MPI

Ring Communication

Write a program that takes data from process zero and sends it to all of the other processes by sending it in a ring. That is, process i should receive the data and send it to process i+1, until the last process is reached.

Assume that the data consists of a single integer. Process zero reads the data from the user.
#include <stdio.h>
#include "mpi.h"

int main( argc, argv )
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);
    }
    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;
Introduction to MPI

Global Communication Operations

- One-to-all broadcast: MPI_BCAST
- All-to-one gather: MPI_GATHER
- One-to-all scatter: MPI_SCATTER
Introduction to MPI

Global Communication Operations

Initial Data:

0: 2 4
1: 5 7
2: 0 3
3: 6 2

MPI_REDUCE with MPI_MIN, root = 0:
0 2
- -
- -
- -

MPI_ALL_REDUCE with MPI_MIN:
0 2
0 2
0 2
0 2

MPI_REDUCE with MPI_SUM, root = 1:
- -
13 16
- -
- -
Global Communication Operations

1. MPI_BCAST
2. MPI_SCATTER
3. MPI_SEND/RECV
4. MPI_REDUCEALL
5. MPI_GATHER
Introduction to MPI

MPI_Barrier (comm)
Global synchronization.

IN comm communicator (handle)

MPI_Bcast (inbuf, incount, intype, root, comm)
Broadcast data from root to all processes.

INOUT inbuf address of input buffer, or output buffer at root (choice)
IN incount number of elements in input buffer (integer)
IN intype datatype of input buffer elements (handle)
IN root process id of root process (integer)
IN comm communicator (handle)

MPI_Gather (inbuf, incount, intype, outbuf, outcount, outtype, root, comm)
MPI_Scatter (inbuf, incount, intype, outbuf, outcount, outtype, root, comm)
Collective data movement functions.

IN inbuf address of input buffer (choice)
IN incount number of elements sent to each (integer)
IN intype datatype of input buffer elements (handle)
OUT outbuf address of output buffer (choice)
IN outcount number of elements received from each (integer)
IN outtype datatype of output buffer elements (handle)
IN root process id of root process (integer)
IN comm communicator (handle)

MPI_Reduce (inbuf, outbuf, count, type, op, root, comm)
MPI_Allreduce (inbuf, outbuf, count, type, op, root, comm)
Collective reduction functions.

IN inbuf address of input buffer (choice)
OUT outbuf address of output buffer (choice)
IN count number of elements in input buffer (integer)
IN type datatype of input buffer elements (handle)
IN op operation; see text for list (handle)
IN root process id of root process (integer)
IN comm communicator (handle)
Introduction to MPI

MPI Program for Parallel Implementation of Jacobi iteration for approximating the solution to a linear system of equations.

We solve the Laplace equation in two dimensions with finite differences. Any numerical analysis text will show that iterating

```c
while (not converged) {
    for (i,j)
        xnew[i][j] = (x[i+1][j] + x[i-1][j] + x[i][j+1] + x[i][j-1])/4;
    for (i,j)
        x[i][j] = xnew[i][j];
}
```

will compute an approximation for the solution of Laplace's equation.
Replacement of xnew with the average of the values around it is applied only in the interior; the boundary values are left fixed. In practice, this means that if the mesh is n by n, then the values

- x[0][j]
- x[n-1][j]
- x[i][0]
- x[i][n-1]

are left unchanged. These refer to the complete mesh; you'll have to figure out what to do with for the decomposed data structures (xlocal).

Because the values are replaced by averaging around them, these techniques are called relaxation methods.

We wish to compute this approximation in parallel. Write an MPI program to apply this approximation.
Introduction to MPI

For convergence testing, compute

diffnorm = 0;
for (i,j)
    diffnorm += (xnew[i][j] - x[i][j]) * (xnew[i][j] - x[i][j]);
    diffnorm = sqrt(diffnorm);

Use MPI_Allreduce for this. (Why not use MPI_Reduce?)

Process zero will write out the value of diffnorm and the iteration count at each iteration. When diffnorm is less that 1.0e-2, consider the iteration converged. Also, if you reach 100 iterations, exit the loop.

For simplicity, consider a 12 x 12 mesh on 4 processors.

The boundary values are -1 on the top and bottom, and the rank of the process on the side. The interior points have the same value as the rank of the process.
Introduction to MPI

- Process 3
- Process 2
- Process 1
- Process 0

- Boundary point
- Interior point
Introduction to MPI

This is shown below:

-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
3 3 3 3 3 3 3 3 3 3 3 3
3 3 3 3 3 3 3 3 3 3 3 3
2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
Introduction to MPI

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

/* This example handles a 12 x 12 mesh, on 4 processors only. */
#define maxn 12

int main( argc, argv )
int argc;
char **argv;
{
    int rank, value, size, errcnt, toterr, i, j, itcnt;
    int i_first, i_last;
    MPI_Status status;
    double diffnorm, gdiffnorm;
    double xlocal[(12/4)+2][12];
    double xnew[(12/3)+2][12];

    MPI_Init( &argc, &argv );

    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
Introduction to MPI

MPI_Comm_size( MPI_COMM_WORLD, &size );

if (size != 4) MPI_Abort( MPI_COMM_WORLD, 1 );

/* xlocal[][0] is lower ghostpoints, xlocal[][maxn+2] is upper */

/* Note that top and bottom processes have one less row of interior points */
i_first = 1;
i_last = maxn/size;
if (rank == 0) i_first++;
if (rank == size - 1) i_last--;

/* Fill the data as specified */
for (i=1; i<=maxn/size; i++)
    for (j=0; j<maxn; j++)
        xlocal[i][j] = rank;
    for (j=0; j<maxn; j++) {
        xlocal[i_first-1][j] = -1;
        xlocal[i_last+1][j] = -1;
    }
itcnt = 0;
do {
    /* Send up unless I'm at the top, then receive from below */
    /* Note the use of xlocal[i] for &xlocal[i][0] */
    if (rank < size - 1)
        MPI_Send( xlocal[maxn/size], maxn, MPI_DOUBLE, rank + 1, 0,
                  MPI_COMM_WORLD );
    if (rank > 0)
        MPI_Recv( xlocal[0], maxn, MPI_DOUBLE, rank - 1, 0,
                  MPI_COMM_WORLD, &status );
    /* Send down unless I'm at the bottom */
    if (rank > 0)
        MPI_Send( xlocal[1], maxn, MPI_DOUBLE, rank - 1, 1,
                  MPI_COMM_WORLD );
    if (rank < size - 1)
        MPI_Recv( xlocal[maxn/size+1], maxn, MPI_DOUBLE, rank + 1, 1,
                  MPI_COMM_WORLD, &status );
/* Compute new values (but not on boundary) */

    itcnt ++;
    diffnorm = 0.0;
    for (i=i_first; i<=i_last; i++)
        for (j=1; j<maxn-1; j++) {
            xnew[i][j] = (xlocal[i][j+1] + xlocal[i][j-1] +
            xlocal[i+1][j] + xlocal[i-1][j]) / 4.0;
            diffnorm += (xnew[i][j] - xlocal[i][j]) *
            (xnew[i][j] - xlocal[i][j]);
        }

    /* Only transfer the interior points */
    for (i=i_first; i<=i_last; i++)
        for (j=1; j<maxn-1; j++)
            xlocal[i][j] = xnew[i][j];
Introduction to MPI

MPI_Allreduce( &diffnorm, &gdiffnrm, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD );
gdiffnrm = sqrt( gdiffnrm );
if (rank == 0) printf( "At iteration %d, diff is %e\n", itcnt,
gdiffnrm );
} while (gdiffnrm > 1.0e-2 && itcnt < 100);

MPI_Finalize( );
return 0;
}
## Introduction to MPI

### Asynchronous Communication Operations

**MPI_Probe**

*Poll for a pending message.*

<table>
<thead>
<tr>
<th>In</th>
<th>source</th>
<th>id of source process, or MPI_ANY_SOURCE (integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>tag</td>
<td>message tag, or MPI_ANY_TAG (integer)</td>
</tr>
<tr>
<td>In</td>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
<tr>
<td>Out</td>
<td>flag</td>
<td>(logical/Boolean)</td>
</tr>
<tr>
<td>Out</td>
<td>status</td>
<td>status object (status)</td>
</tr>
</tbody>
</table>

**MPI_Probe**

*Return when message is pending.*

<table>
<thead>
<tr>
<th>In</th>
<th>source</th>
<th>id of source process, or MPI_ANY_SOURCE (integer)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>tag</td>
<td>message tag, or MPI_ANY_TAG (integer)</td>
</tr>
<tr>
<td>In</td>
<td>comm</td>
<td>communicator (handle)</td>
</tr>
<tr>
<td>Out</td>
<td>status</td>
<td>status object (status)</td>
</tr>
</tbody>
</table>

**MPI_Get_Count**

*Determine size of a message.*

<table>
<thead>
<tr>
<th>In</th>
<th>status</th>
<th>status variable from receive (status)</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>datatype</td>
<td>datatype of receive buffer elements (handle)</td>
</tr>
<tr>
<td>Out</td>
<td>count</td>
<td>number of data elements in message (integer)</td>
</tr>
</tbody>
</table>
Introduction to MPI

Creating Communication Groups

**KPI_COMM_DUP(comm, newcomm)**
Create new communicator: same group, new context.

IN  comm       communicator (handle)
    newcomm    communicator (handle)

**KPI_COMM_SPLIT(comm, color, key, newcomm)**
Partition group into disjoint subgroups.

IN  comm       communicator (handle)
    color      subgroup control (integer)
    key        process id control (integer)
    newcomm    communicator (handle)

**KPI_INTERCOMM_CREATE(comm, leader, peer, rleader, tag, inter)**
Create an intercommunicator.

IN  comm       local intracommmunicator (handle)
    leader     local leader (integer)
    peer       peer intracommmunicator (handle)
    rleader    process id of remote leader in peer (integer)
    tag        tag for communicator set up (integer)
    inter      new intracommmunicator (handle)

**KPI_COMM_FREE(comm)**
Destroy a communicator.

IN  comm       communicator (handle)
Communication Groups

A call of the form

MPI_COMM_SPLIT(comm, color, key, newcomm) creates one or more new communicators.

It must be executed by each process in the process group associated with comm.

A new communicator is created for each unique value of color other than the defined constant MPI_UNDEFINED.
Each new communicator comprises those processes that specified its value of color in the MPI_COMM_SPLIT call. These processes are assigned identifiers within the new communicator starting from zero, with order determined by the value of key or, in the event of ties, by the identifier in the old communicator. Thus, a call of the form MPI_COMM_SPLIT(comm, 0, 0, newcomm) in which all processes specify the same color and key, is equivalent to a call MPI_COMM_DUP(comm, newcomm)
The following code creates three new communicators if `comm` contains at least three processes.

```c
MPI_Comm comm, newcomm;
int myid, color;
MPI_Comm_rank(comm, &myid);
color = myid%3;
MPI_Comm_split(comm, color, myid, &newcomm);
```

For example, if `comm` contains eight processes, then processes 0, 3, and 6 form a new communicator of size three, as do processes 1, 4, and 7, while processes 2 and 5 form a new communicator of size two.
Introduction to MPI

Task Model versus Process Model

Diagram showing the comparison between a task model and a process model.
Introduction to MPI

Communication Pattern for Program on Next Slide
Introduction to MPI

integer comm, intercomm, ierr, status(MPI_STATUS_SIZE)
C For simplicity, we require an even number of processes
   call MPI_COMM_SIZE(MPI_COMM_WORLD, count, ierr)
   if(mod(count, 2) .ne. 0) stop
C Split processes into two groups: odd and even numbered
   call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
   call MPI_COMM_SPLIT(MPI_COMM_WORLD, mod(myid, 2), myid,
                       name, ierr)
   if(mod(myid, 2) .eq. 0) then
     Group 0: create intercommunicator and send message
     Arguments: 0=local leader; 1=remote leader; 00=tag
                 comm, 0, MPI_COMM_WORLD, 1, 00,
                 intercomm, ierr)
                 call MPI_SEND(msg, i, type, newid, 0, intercomm, ierr)
               else
     Group 1: create intercommunicator and receive message
     Note that remote leader has id 0 in MPI_COMM_WORLD
     call MPI_INTERCOM_CREATE(comm, 0, MPI_COMM_WORLD, 0, 99,
                                intercomm, ierr)
     call MPI_RECV(msg, i, type, newid, 0, intercomm,
                    status, ierr)
   endif
C Free communicators created during this operation
   call MPI_COMM_FREE(intercomm, ierr)
   call MPI_COMM_FREE(comm, ierr)

Program 5.7 : An MPI program illustrating creation and use of an intercommunicator.
Introduction to MPI

MPI Data Type Creation Operations

MPI_TYPE_CONTIGUOUS(count, oldtype, newtype)
Construct datatype from contiguous elements.

IN  count   number of elements (integer \geq 0)
IN  oldtype input datatype (handle)
OUT newtype output datatype (handle)

MPI_TYPE_VECTOR(count, blocklen, stride, oldtype, newtype)
Construct datatype from blocks separated by stride.

IN  count   number of elements (integer \geq 0)
IN  blocklen elements in a block (integer \geq 0)
IN  stride  elements between start of each block (integer)
IN  oldtype input datatype (handle)
OUT newtype output datatype (handle)

MPI_TYPE_INDEXED(count, blocklen, indices, oldtype, newtype)
Construct datatype with variadic indices and sizes.

IN  count   number of blocks (integer \geq 0)
IN  blocklen elements in each block (array of integer \geq 0)
IN  indices displacements for each block (array of integer)
IN  oldtype input datatype (handle)
OUT newtype output datatype (handle)

MPI_TYPE_COMM(type)
Construct datatype so that it can be used in communication.

INOUT type   datatype to be committed (handle)

MPI_TYPE_FREE(type)
Free a derived datatype.

INOUT type   datatype to be freed (handle)
### Introduction to MPI

![Diagram showing rowtype and coltype with numbers 1 to 24]

<table>
<thead>
<tr>
<th>North: rowtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 5 9 13 17 21</td>
</tr>
<tr>
<td>2 6 10 14 18 22</td>
</tr>
<tr>
<td>3 7 11 15 19 23</td>
</tr>
<tr>
<td>4 8 12 16 20 24</td>
</tr>
</tbody>
</table>

- **West**: coltype
- **East**: coltype

9/24/2001 Parallel Programming - Fall 2001
Introduction to MPI

integer coltype, rostype, comm, ierr

C The derived type coltype is 4 contiguous reals.
call MPI_TYPE_CONTIGUOUS(4, MPI_REAL, coltype, ierr)
call MPI_TYPE_COMMIT(coltype, ierr)

C The derived type rostype is 8 reals, located 4 apart.
call MPI_TYPE_VECTOR(8, 1, 4, MPI_REAL, rostype, ierr)
call MPI_TYPE_COMMIT(rostype, ierr)

... call MPI_SEND(array(1,1), 1, coltype, west, 0, comm, ierr)
call MPI_SEND(array(1,8), 1, coltype, east, 0, comm, ierr)
call MPI_SEND(array(1,1), 1, rostype, north, 0, comm, ierr)
call MPI_SEND(array(4,1), 1, rostype, south, 0, comm, ierr)

...
call MPI_TYPE_FREE(rostype, ierr)
call MPI_TYPE_FREE(coltype, ierr)

Program 8.6 : Using derived types to communicate a finite difference stencil. The variables west, east, north, and south refer to the process’s neighbors.