

Ateneo de Manila University

Parallel Programming with MPI: Basics



Ateneo High Performance Computing Group
1st Semester

<http://www.math.admu.edu.ph/ahpc/>
william.s.yu@ieee.org

Section I

Introduction

Parallel Computing Models

- ★ Shared Memory
- ★ Message Passing
- ★ Remote Memory Operation
- ★ Threads
- ★ Combined Models

Issues

- ★ No Network Abstraction
- ★ No standard for porting code to different systems
- ★ No Heterogeneous System Support

Section II

MPI

Advantages of Message Passing

- ★ Universality
- ★ Expressivity
- ★ Ease of Debugging
- ★ Performance
- ★ Portability

MPI

- ★ Message Passing Interface
- ★ standard for creating message passing applications
- ★ aims to be a practical, portable, efficient, and flexible standard

Goals and Aims of MPI

- ★ design an application programming interface
- ★ allow reliable and efficient communications
- ★ allow operations in a heterogeneous environment
- ★ allow convenient bindings to C and Fortran 77
- ★ define an interface that is not too different from existing standards
- ★ define an interface that can be implemented on multiple vendor platforms
- ★ semantics of the interface should be language independent
- ★ interface designed for thread safety

History

- ★ Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia
- ★ 60 people from 40 different organizations
- ★ The workshop aims to develop a standard message passing interface
- ★ Dongarra, Hempel, Hey, and Walker proposed the first draft(v1.0) in November 1992
- ★ On June 1995, MPI v1.1 was released by the MPI Forum
- ★ MPI v2.0 meeting began on April 1995

Flavors of MPI

- ★ MPICH - is a free and portable implementation of the MPI standard developed by MSU/ANL.
- ★ LAM/MPI - is another free implementation of the MPI standard and is being developed at ND.

Basic MPI concepts

- ★ Message - data packet to be exchanged among compute nodes
- ★ Process - computational task to be completed
- ★ Rank - order of the node in the cluster
- ★ Communicator - group of nodes

Section III

Introduction to LAM

Local Area Multicomputer

- ★ a public domain implementation of message passing interface
- ★ developed at the Ohio Supercomputing Center and now hosted at the University of Notre Dame <http://www.mpi.nd.edu/>

Section IV

Using LAM

Getting Started

- ★ Starting LAM
- ★ Run MPI process
- ★ Getting the status of the process
- ★ Display outstanding messages
- ★ Killing a LAM process
- ★ Shutdown LAM

Starting LAM

```
lamboot <-v hostfile>
```

- ★ this program starts the LAM environment for the local user
- ★ it is encouraged that each user start his/her own LAM environment
- ★ LAM uses the default host file if it is not stated explicitly here

Running a MPI process

```
lamrun <args> <processors> program --[args]>
```

- ★ this program enables one to run MPI programs in the LAM environment
- ★ [args] contain the arguments to be passed to the program
- ★ processor refers to the processors to be used. N means all nodes or n0-4 means nodes 0 to 4

Status of a MPI process

`mpitask`

★ returns the list of MPI processes running in your LAM environment

Display outstanding messages

`mpimsg`

- ★ this returns list of outstanding messages included error message from the running MPI processes

Killing a running MPI process

```
lamclean -v
```

- ★ this program terminates the current MPI process in the current LAM environment

Stopping LAM

```
wipe <-v hostfile>
```

- ★ this program will shutdown the LAM environment for the local user
- ★ LAM uses the default host file if it is not stated explicitly here

Final Notes

- ★ lamboot must be run before any other LAM/MPI command
- ★ lamclean is used to "clean up" any residuals from individual run
- ★ once you wipe, you cannot run any other LAM/MPI command

Section V

MPI Programming Basics

Getting Started

- ★ Writing Programs
- ★ Compiling and Linking
- ★ Running MPI Programs

- ★ ST: Parallel Environment
- ★ ST: Message Passing
- ★ ST: Timing

Sample Fortran Program

```
program main
include "mpif.h"
integer ierr

call MPI_INIT(ierr)
print *, 'Hello World'
call MPI_FINALIZE(ierr)

end
```

Sample C Program

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

int main(int argc, char **argv){
    MPI_Init(&argc,&argv);
    printf("Hello World!\n");
    MPI_Finalize();
    return 0;
}
```

What does it all mean?

- ★ "mpif.h" and "mpi.h" are the header files needed to load the mpi libraries
- ★ this program simply lets all the nodes display "Hello World!"

Starting MPI

```
int MPI_Init(int argc, char **argv);  
MPI_INIT(Integer)
```

- ★ this program terminates the current MPI process in the current LAM environment
- ★ must be called before any other MPI call or function apart from `MPI_INITIALIZED`
- ★ this function initializes MPI and returns the error code corresponding to the problem
- ★ accepts `argc` and `argv` which are arguments of `main`

Stopping MPI

```
int MPI_Finalize(void);  
MPI_FINALIZE(Integer)
```

- ★ this function clean up the MPI state
- ★ ensures that all pending communications complete before actually finalizing

Compiling and Linking

★ It would be best to generate a Makefile for ease of compilation

★ to compile in C:

```
hcc -o <output file> <source file>
```

★ to compile in Fortran:

```
h77 -o <outfile file> <source file>
```

Running MPI Programs

★ to run an MPI program:

```
lamrun -v <nodes> <program>
```

★ as an example to compile the sample program:

```
hcc -o ex1 ex1.c
```

```
lamrun -v n0-7 ex1
```

ST: Parallel Environment

- ★ `MPI_COMM_SIZE` - to get the size of the cluster
- ★ `MPI_COMM_RANK` - to get the rank of the current node
- ★ `MPI_COMM_WORLD` - the subset of the entire cluster

Warning: the rank is a number (size-1) since it is base 0.

Sample Fortran Program

```
program main
include "mpif.h"
integer rank,size, ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
print *, 'Hello World! I am',
       rank, ' of ', size, '.'
call MPI_FINALIZE(ierr)

end
```

Sample C Program

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

int main(int argc, char **argv){
    int rank, size;
    MPI_Init();
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    print ("Hello World! I am
           %d of %d.",rank, size);
    MPI_Finalize();
}
```

Getting cluster size

```
int MPI_Comm_size(MPI_Comm communicator,  
                  int *size);
```

```
MPI_COMM_SIZE(Integer communicator,  
               Integer size, Integer error)
```

- ★ this function simply gives back the number of node participating in the application

Who am I?

```
int MPI_Comm_rank(MPI_Comm communicator,  
                  int *rank);
```

```
MPI_COMM_RANK(Integer communicator,  
               Integer rank, Integer error)
```

★ this function simply gives back the rank of the current node

Where am I?

`MPI_COMM_WORLD`

- ★ this constant returns the current global communicator that represents the whole cluster
- ★ is the default communicator and most applications do not need to use any other

Exercise – Getting Started

- Objective: Learn how to write, compile, and run a simple MPI program and become familiar with MPI.
- Compile and run the second “Hello world” program. Try various numbers of processors.
 - Be sure to name your file `lab1.c` or `lab1.f`
 - Compile the programs using `hcc` or `hf77`.
 - Use `lamboot` to start LAM MPI.
 - Use `lamrun` to run your program (use `lamclean` if things go wrong).
 - Use `wipe` when all finished with LAM MPI.
- What does the output look like?

ST: Message Passing

- ★ MPI Data Types - list of data types for passing variable among nodes
- ★ MPI_Send - function for sending data
- ★ MPI_Recv - function for receiving data
- ★ MPI_Status - getting information about the message
- ★ MPI_Bcast - sending data to all nodes
- ★ MPI_Reduce - combining all the node outputs into one value via an MPI_OP

MPI C Data Types

MPI_CHAR - signed char

MPI_SHORT - signed short int

MPI_INT - signed int

MPI_LONG - signed long int

MPI_UNSIGNED_CHAR - unsigned char

MPI_UNSIGNED_SHORT - unsigned short int

MPI_UNSIGNED - unsigned int

MPI_UNSIGNED_LONG - unsigned long int

MPI_FLOAT - float

MPI_DOUBLE - double

MPI_LONG_DOUBLE - long double

MPI_BYTE

MPI_PACKED

MPI FORTRAN Data Types

MPI_INTEGER - INTEGER

MPI_REAL - REAL

MPI_DOUBLE_PRECISION - DOUBLE PRECISION

MPI_COMPLEX - COMPLEX

MPI_LOGICAL - LOGICAL

MPI_CHARACTER - CHARACTER

MPI_BYTE

MPI_PACKED

Blocking Send and Receive

- ★ MPI_SEND and MPI_RECV, are the basic point-to-point communication routines in MPI
- ★ both block the calling process until the communication operation is completed
- ★ blocking creates the possibility of deadlock

Sending Messages

```
int MPI_Send(void *buf, int count,
             MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm);
MPI_SEND(<*> BUF(*), INTEGER COUNT,
        INTEGER DATATYPE, INTEGER DEST,
        INTEGER TAG, INTEGER COMM,
        INTEGER IERROR)
```

MPI_SEND(buf, count, datatype, dest, tag, comm)

IN	buf	initial address of send buffer
IN	count	number of entries to send
IN	datatype data	type of entries
IN	dest	rank of destination
IN	tag	message tag
IN	comm	communicator

- ★ this function is used to send message across the cluster. It performs standard-mode blocking send

Receiving Messages

```
int MPI_Recv(void *buf, int count,
             MPI_Datatype datatype, int dest,
             int tag, MPI_Comm comm,
             MPI_Status *status);
MPI_RECV(<type> BUF(*), INTEGER COUNT,
        INTEGER DATATYPE, INTEGER DEST,
        INTEGER TAG, INTEGER COMM,
        INTEGER STATUS(MPI_STATUS_SIZE),
        INTEGER IERROR)
```

MPI_RECV(buf, count, datatype, source, tag, comm, status)

OUT	buf	initial address of send buffer
IN	count	max number of entries to receive
IN	datatype	data type of entries
IN	dest	rank of source
IN	tag	message tag
IN	comm	communicator
OUT	status	return status information ommunicator

★ this function is used to receive message across the cluster. It performs standard-mode blocking receive

Simple C Example

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

int main(int argc, char **argv)
{
    int i, rank, size, dest;
    int to, src, from, count, tag;
    int st_count, st_source, st_tag;
    double data[100];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Process %d of %d is alive\n", rank, size);

    dest = size - 1;
    src = 0;

    if (rank == src) {
        to = dest;
        count = 100;
        tag = 2001;
        for (i = 0; i < 100; i++)
            data[i] = i;
        MPI_Send(data, count, MPI_DOUBLE, to, tag, MPI_COMM_WORLD);
    }
    else if (rank == dest) {
        tag = MPI_ANY_TAG;
        count = 100;
    }
}
```

```
from = MPI_ANY_SOURCE;
MPI_Recv(data, count, MPI_DOUBLE, from, tag, MPI_COMM_WORLD,
         &status);

MPI_Get_count(&status, MPI_DOUBLE, &st_count);
st_source= status.MPI_SOURCE;
st_tag= status.MPI_TAG;

printf("Status info: source = %d, tag = %d, count = %d\n",
       st_source, st_tag, st_count);
printf(" %d received: ", rank);
for (i = 0; i < st_count; i++)
    printf("%lf ", data[i]);
printf("\n");
}

MPI_Finalize();
return 0;
}
```

Simple C++ Example

```
#include <iostream.h>
#include <mpi++.h>

int main(int argc, char **argv)
{
    int i, rank, size, dest;
    int to, src, from, count, tag;
    int st_count, st_source, st_tag;
    double data[100];
    MPI::Status status;

    MPI::Init(argc, argv);
    rank = MPI::COMM_WORLD.Get_rank();
    size = MPI::COMM_WORLD.Get_size();

    cout << "Process " << rank << " of " << size << " is alive" << endl;

    dest = size - 1;
    src = 0;

    if (rank == src) {
        to = dest;
        count = 100;
        tag = 2001;
        for (i = 0; i < 100; i++)
            data[i] = i;
        MPI::COMM_WORLD.Send(data, count, MPI::DOUBLE, to, tag);
    }
    else if (rank == dest) {
        tag = MPI::ANY_TAG;
    }
}
```

```
count = 100;
from = MPI::ANY_SOURCE;
MPI::COMM_WORLD.Recv(data, count, MPI::DOUBLE, from, tag, status);
st_count = status.Get_count(MPI::DOUBLE);
st_source= status.Get_source();
st_tag= status.Get_tag();

cout << "Status info: source = " << st_source << ", tag = " << st_tag
<< ", count = " << st_count << endl;
cout << rank << " received: ";
for (i = 0; i < st_count; i++)
    cout << data[i] << " ";
cout << endl;
}

MPI::Finalize();
return 0;
}
```

Simple Fortran Example

```
program main
include 'mpif.h'

integer rank, size, to, from, tag, count, i, ierr
integer src, dest
integer st_source, st_tag, st_count
integer status(MPI_STATUS_SIZE)
double precision data(100)

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierr)
print *, 'Process ', rank, ' of ', size, ' is alive'
dest = size - 1
src = 0
C
if (rank .eq. src) then
to      = dest
count   = 100
tag     = 2001
do 10 i=1, 100
10      data(i) = i
call MPI_SEND(data, count, MPI_DOUBLE_PRECISION, to,
+          tag, MPI_COMM_WORLD, ierr)
else if (rank .eq. dest) then
tag     = MPI_ANY_TAG
count   = 100
from    = MPI_ANY_SOURCE
call MPI_RECV(data, count, MPI_DOUBLE_PRECISION, from,
+          tag, MPI_COMM_WORLD, status, ierr)
```

```
        call MPI_GET_COUNT(status, MPI_DOUBLE_PRECISION,  
+           st_count, ierr)  
        st_source = status(MPI_SOURCE)  
        st_tag    = status(MPI_TAG)  
C  
        print *, 'Status info: source = ', st_source,  
+           ' tag = ', st_tag, ' count = ', st_count  
        print *, rank, ' received', (data(i),i=1,100)  
    endif  
  
    call MPI_FINALIZE(ierr)  
end
```

Six Function MPI

MPI is very simple. These six functions allow you to write many programs:

- MPI_INIT
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_SEND
- MPI_RECV
- MPI_FINALIZE

Exercise – Message Ring

- Objective: Pass a message around a ring n times. Use blocking `MPI_SEND` and `MPI_RECV`.
 - Write a program to do the following:
 - * Process 0 should read in a single integer (> 0) from standard input
 - * Use MPI send and receive to pass the integer around a ring
 - * Use the user-supplied integer to determine how many times to pass the message around the ring
 - * Process 0 should decrement the integer each time it is received.
 - * Processes should exit when they receive a “0”.

Getting Status Info

`MPI_Status`

★ this data type defines a record that contains information about the message

Getting Status Info

- ★ `MPI_Status.MPI_TAG` - returns the tag of the data when `MPI_ANY_TAG` is used in the receive
- ★ `MPI_Status.MPI_SOURCE` - returns the source of the data when `MPI_ANY_SOURCE` is used in the receive
- ★ `MPI_Get_count(MPI_Status status, MPI_Datatype datatype, int count)` - returns the number of datatype objects that was sent

Getting Information About a Message

- The (non-opaque) `status` object contains information about a message

```
/* In C */
MPI_Status status;
MPI_Recv(..., &status);

recvd_tag    = status.MPI_TAG;
recvd_source = status.MPI_SOURCE;
MPI_Get_count(&status, datatype, &recvd_count);

/* In C++ */
MPI::Status status;
MPI::COMM_WORLD.Recv(..., status);

recvd_tag = status.Get_tag();
recvd_source = status.Get_source();
recvd_count = status.Get_count(datatype);
```

Broadcasting Messages

```
int MPI_Bcast(void *buf, int count,  
             MPI_Datatype datatype, int root,  
             MPI_Comm comm);  
MPI_BCAST(<type> BUF(*), INTEGER COUNT,  
         INTEGER DATATYPE, INTEGER ROOT,  
         INTEGER COMM, INTEGER IERROR)
```

MPI_BCAST(buf, count, datatype, root, comm)

INOUT	buf	initial address of send buffer
IN	count	max number of entries to receive
IN	datatype	data type of entries
IN	root	rank of broadcast root
IN	comm	communicator

- ★ this function broadcasts from the process with the rank root to all processes in comm group
- ★ note: root and comm on all the nodes must be the same. The variable in buf has been copied to all the nodes

Global Reduction

```
int MPI_Reduce(void *sendbuf, void *recvbuf,  
              int count, MPI_Datatype datatype, MPI_op,  
              int root, MPI_Comm comm);  
MPI_REDUCE(<type> SENDBUF(*), <type> RECVBUF(*),  
          INTEGER COUNT, INTEGER DATATYPE, INTEGER OP,  
          INTEGER ROOT, INTEGER COMM, INTEGER IERROR)
```

MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)

IN	sendbuf	address of the send buffer
OUT	recvbuf	address of the receive buffer
IN	count	max number of entries to receive
IN	datatype	data type of entries
IN	op	reduce operation
IN	root	rank of broadcast root
IN	comm	communicator

- ★ this function combines all elements in the recvbuf of all nodes in comm using the operation op into the sendbuf and returns to combined value to rank root

Solve for PI

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

int main (int argc, char* argv[]) {
    int n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    while (1) {
        if (myid == 0) {
            printf("Number of Intervals(0 exits): ");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
        else {
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i=myid+1; i<=n; i += numprocs){
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi= h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
            if (myid == 0)
```

```
        printf("pi is %.16f, Error is %.16f\n", pi, fabs(pi-PI25DT));
    }
}
MPI_Finalize();
}
```

ST: Timing

- ★ MPI_Wtime - returns a floating point number of seconds
- ★ MPI_Wtick - returns the current time in seconds

Getting Time

```
double MPI_WTime( );
```

```
DOUBLE PRECISION MPI_WTIME( )
```

- ★ gets the value of time as a floating point value in seconds
- ★ greater precision than ticks since the value is in floating point units

Getting Time in Seconds

```
double MPI_WTick();  
DOUBLE PRECISION MPI_WTICK()
```

- ★ gets the value of time in seconds
- ★ returns values in successive clock ticks
- ★ A millisecond in MPI_Wtick is 10^{-3}

Solve for PI w/ timer

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

int main (int argc, char* argv[]) {
    int n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a, t0, t1;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    while (1) {
        if (myid == 0) {
            printf("Number of Intervals(0 exits): ");
            scanf("%d", &n);
        }

        t0 = MPI_Wtime();

        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
        else {
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i=myid+1; i<=n; i += numprocs){
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
        }
    }
}
```

```
    mypi= h * sum;

    t1=MPI_Wtime();

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0)
        printf("pi is %.16f, Error is %.16f\n", pi, fabs(pi-PI25DT));
        printf("time is %.16f.\n",t1-t0);
    }
}
MPI_Finalize();
}
```

Homework

- Study the example provided above
- Search for a problem
- Study the problem and determine if it can be parallelized



Copyright © 2000-2001 by William Emmanuel S. Yu. This material may be distributed only subject to the terms and conditions set forth in the Open Content License, v1.0 or later (the latest version is presently available at <http://opencontent.org/opl.shtml>).