

# Ateneo de Manila University

## Parallel Programming with MPI



Ateneo High Performance Computing Group

8 August 2000

<http://www.math.admu.edu.ph/ahpc/>

[william.s.yu@ieee.org](mailto: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

```
mpirun <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
- ★ `iprocessorj` 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:

```
mpirun -v <nodes> <program>
```

★ as an example to compile the sample program:

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

```
mpirun -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

## **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

## 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

# Example: Token Passing

## 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

## 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

# Example: Solve for PI

## **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  $10^{-3}$

**Example: Solve for PI w/ timer**

# Ateneo de Manila University

## Parallel Programming with MPI



Ateneo High Performance Computing Group

8 August 2000

<http://www.math.admu.edu.ph/ahpc/>

[william.s.yu@ieee.org](mailto:william.s.yu@ieee.org)