Parallel & Cluster Computing

MPI Introduction

Henry Neeman, Director
OU Supercomputing Center for Education & Research
University of Oklahoma

SC08 Education Program's Workshop on Parallel & Cluster computing
August 10-16 2008

Okla. Supercomputing Symposium

Tue Oct 7 2008 @ OU

Over 250 registrations already!

Over 150 in the first day, over 200 in the first week, over 225 in the first month.

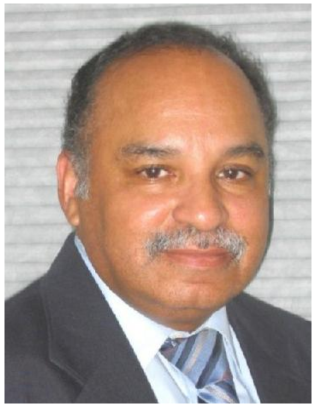
2003 Keynote: Peter Freeman NSF Computer & Information Science & Engineering Assistant Director

2004 Keynote: Sangtae Kim NSF Shared Cyberinfrastructure Division Director

2005 Keynote: Walt Brooks NASA Advanced Supercomputing Division Director

2006 Keynote: Dan Atkins Head of NSF's Office of Cyber-infrastructure

2007 Keynote: Jay Boisseau Director Texas Advanced Computing Center U. Texas Austin


2008 Keynote: José Munoz Deputy Office Director/ Senior Scientific Advisor Office of Cyber-infrastructure National Science Foundation

FREE! Parallel Computing Workshop Mon Oct 6 @ OU sponsored by SC08

FREE! Symposium Tue Oct 7 @ OU

`http://symposium2008.oscer.ou.edu/`

What Is MPI?

The ***Message-Passing Interface*** (MPI) is a standard for expressing distributed parallelism via message passing.

MPI consists of a ***header file***, a ***library of routines*** and a ***runtime environment***.

When you compile a program that has MPI calls in it, your compiler links to a local implementation of MPI, and then you get parallelism; if the MPI library isn't available, then the compile will fail.

MPI can be used in Fortran, C and C++.

MPI Calls

MPI calls in **Fortran** look like this:

```
CALL MPI_Funcname(…, errcode)
```

In C, MPI calls look like:

```
errcode = MPI_Funcname(…);
```

In C++, MPI calls look like:

```
errcode = MPI::Funcname(…);
```

Notice that `errcode` is returned by the MPI routine `MPI_Funcname`, with a value of `MPI_SUCCESS` indicating that `MPI_Funcname` has worked correctly.

MPI is an API

- MPI is actually just an ***<u>Application Programming Interface</u>*** (API).
- An API specifies what a call to each routine should look like, and how each routine should behave.
- An API does not specify how each routine should be implemented, and sometimes is intentionally vague about certain aspects of a routine's behavior.
- Each platform has its own MPI implementation.

Example MPI Routines

- **`MPI_Init`** starts up the MPI runtime environment at the beginning of a run.
- **`MPI_Finalize`** shuts down the MPI runtime environment at the end of a run.
- **`MPI_Comm_size`** gets the number of processes in a run, N_p (typically called just after **`MPI_Init`**).
- **`MPI_Comm_rank`** gets the process ID that the current process uses, which is between 0 and N_p-1 inclusive (typically called just after **`MPI_Init`**).

More Example MPI Routines

- `MPI_Send` sends a message from the current process to some other process (the ***destination***).
- `MPI_Recv` receives a message on the current process from some other process (the ***source***).
- `MPI_Bcast` ***broadcasts*** a message from one process to all of the others.
- `MPI_Reduce` performs a ***reduction*** (e.g., sum, maximum) of a variable on all processes, sending the result to a single process.

MPI Program Structure (F90)

```
PROGRAM my_mpi_program
  IMPLICIT NONE
  INCLUDE "mpif.h"
  [other includes]
  INTEGER :: my_rank, num_procs, mpi_error_code
  [other declarations]
  CALL MPI_Init(mpi_error_code)      !! Start up MPI
  CALL MPI_Comm_Rank(my_rank,   mpi_error_code)
  CALL MPI_Comm_size(num_procs, mpi_error_code)
  [actual work goes here]
  CALL MPI_Finalize(mpi_error_code) !! Shut down MPI
END PROGRAM my_mpi_program
```

Note that MPI uses the term "*rank*" to indicate process identifier.

MPI Program Structure (in C)

```
#include <stdio.h>
#include "mpi.h"
 [other includes]

int main (int argc, char* argv[])
{ /* main */
  int my_rank, num_procs, mpi_error;
  [other declarations]
  mpi_error = MPI_Init(&argc, &argv); /* Start up MPI  */
  mpi_error = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  mpi_error = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  [actual work goes here]
  mpi_error = MPI_Finalize();         /* Shut down MPI */
} /* main */
```

MPI is SPMD

MPI uses kind of parallelism known as
Single Program, Multiple Data (SPMD).

This means that you have one MPI program – a single executable – that is executed by all of the processes in an MPI run.

So, to differentiate the roles of various processes in the MPI run, you have to have **`if`** statements:

```
if (my_rank == server_rank) {
      …
}
```

Example: Hello World

1. Start the MPI system.
2. Get the rank and number of processes.
3. If you're **not** the server process:
    1. Create a "hello world" string.
    2. Send it to the server process.
4. If you **are** the server process:
    1. For each of the client processes:
        1. Receive its "hello world" string.
        2. Print its "hello world" string.
5. Shut down the MPI system.

hello_world_mpi.c

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

int main (int argc, char* argv[])
{ /* main */
  const int  maximum_message_length = 100;
  const int  server_rank            =   0;
  char       message[maximum_message_length+1];
  MPI_Status status;      /* Info about receive status  */
  int        my_rank;     /* This process ID            */
  int        num_procs;   /* Number of processes in run */
  int        source;      /* Process ID to receive from */
  int        destination; /* Process ID to send to      */
  int        tag = 0;     /* Message ID                 */
  int        mpi_error;   /* Error code for MPI calls   */
```

[work goes here]

```
} /* main */
```

Hello World Startup/Shut Down

```
[header file includes]
int main (int argc, char* argv[])
{ /* main */
  [declarations]
  mpi_error = MPI_Init(&argc, &argv);
  mpi_error = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  mpi_error = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  if (my_rank != server_rank) {
    [work of each non-server (worker) process]
  } /* if (my_rank != server_rank) */
  else {
    [work of server process]
  } /* if (my_rank != server_rank)…else */
  mpi_error = MPI_Finalize();
} /* main */
```

Hello World Client’s Work

```
[header file includes]
int main (int argc, char* argv[])
{ /* main */
  [declarations]
  [MPI startup (MPI_Init etc)]
  if (my_rank != server_rank) {
    sprintf(message, "Greetings from process #%d!“,
        my_rank);
    destination = server_rank;
    mpi_error =
      MPI_Send(message, strlen(message) + 1, MPI_CHAR,
        destination, tag, MPI_COMM_WORLD);
  } /* if (my_rank != server_rank) */
  else {
    [work of server process]
  } /* if (my_rank != server_rank)…else */
  mpi_error = MPI_Finalize();
} /* main */
```

Hello World Server's Work

```
[header file includes]
int main (int argc, char* argv[])
{ /* main */
  [declarations, MPI startup]
  if (my_rank != server_rank)  {
    [work of each client process]
  } /* if (my_rank != server_rank)  */
  else {
    for (source = 0; source < num_procs; source++)  {
      if (source != server_rank)  {
        mpi_error =
          MPI_Recv(message, maximum_message_length + 1,
            MPI_CHAR, source, tag, MPI_COMM_WORLD,
            &status);
        fprintf(stderr, "%s\n", message);
      } /* if (source != server_rank)  */
    } /* for source */
  } /* if (my_rank != server_rank)…else  */
  mpi_error = MPI_Finalize();
} /* main */
```

How an MPI Run Works

- Every process gets a copy of the executable: ***Single Program, Multiple Data*** (SPMD).
- They all start executing it.
- Each looks at its own rank to determine which part of the problem to work on.
- Each process works **completely independently** of the other processes, except when communicating.

Compiling and Running

```
% mpicc  -o  hello_world_mpi  hello_world_mpi.c
% mpirun  -np  1  hello_world_mpi
% mpirun  -np  2  hello_world_mpi
Greetings from process #1!
% mpirun  -np  3  hello_world_mpi
Greetings from process #1!
Greetings from process #2!
% mpirun  -np  4  hello_world_mpi
Greetings from process #1!
Greetings from process #2!
Greetings from process #3!
```

Note: The compile command and the run command vary from platform to platform.

Why is Rank #0 the server?

```
const int server_rank = 0;
```

By convention, the server process has rank (process ID) #0. **<u>Why?</u>**

A run must use at least one process but can use multiple processes.

Process ranks are 0 through N_p-1, $N_p \geq 1$.

Therefore, every MPI run has a process with rank #0.

<u>Note</u>: Every MPI run also has a process with rank N_p-1, so you could use N_p-1 as the server instead of 0 … but no one does.

Why "Rank?"

Why does MPI use the term ***rank*** to refer to process ID?

In general, a process has an identifier that is assigned by the operating system (e.g., Unix), and that is unrelated to MPI:

```
% ps
          PID TTY        TIME CMD
     52170812 ttyq57   0:01 tcsh
```

Also, each processor has an identifier, but an MPI run that uses fewer than all processors will use an arbitrary subset.

The rank of an MPI process is neither of these.

Compiling and Running

Recall:

```
% mpicc  -o  hello_world_mpi  hello_world_mpi.c
% mpirun  -np  1  hello_world_mpi
% mpirun  -np  2  hello_world_mpi
Greetings from process #1!
% mpirun  -np  3  hello_world_mpi
Greetings from process #1!
Greetings from process #2!
% mpirun  -np  4  hello_world_mpi
Greetings from process #1!
Greetings from process #2!
Greetings from process #3!
```

Deterministic Operation?

```
% mpirun  -np  4  hello_world_mpi
Greetings from process #1!
Greetings from process #2!
Greetings from process #3!
```

The order in which the greetings are printed is deterministic. **Why?**

```
for (source = 0; source < num_procs; source++) {
  if (source != server_rank) {
    mpi_error =
      MPI_Recv(message, maximum_message_length + 1,
        MPI_CHAR, source, tag, MPI_COMM_WORLD,
        &status);
    fprintf(stderr, "%s\n", message);
  } /* if (source != server_rank) */
} /* for source */
```

This loop **ignores the receive order**.

Message = Envelope+Contents

```
MPI_Send(message, strlen(message) + 1,
     MPI_CHAR, destination, tag,
     MPI_COMM_WORLD);
```

When MPI sends a message, it doesn't just send the contents; it also sends an "envelope" describing the contents:

Size (number of elements of data type)

Data type

Source: rank of sending process

Destination: rank of process to receive

Tag (message ID)

Communicator (e.g., `MPI_COMM_WORLD`)

MPI Data Types

C		Fortran 90	
`char`	`MPI_CHAR`	`CHARACTER`	`MPI_CHARACTER`
`int`	`MPI_INT`	`INTEGER`	`MPI_INTEGER`
`float`	`MPI_FLOAT`	`REAL`	`MPI_REAL`
`double`	`MPI_DOUBLE`	`DOUBLE PRECISION`	`MPI_DOUBLE_PRECISION`

MPI supports several other data types, but most are variations of these, and probably these are all you' ll use.

Message Tags

```
for (source = 0; source < num_procs; source++) {
  if (source != server_rank) {
    mpi_error =
      MPI_Recv(message, maximum_message_length + 1,
        MPI_CHAR, source, tag,
        MPI_COMM_WORLD, &status);
    fprintf(stderr, "%s\n", message);
  } /* if (source != server_rank) */
} /* for source */
```

The greetings are **printed** in **deterministic** order not because messages are sent and received in order, but because each has a ***tag*** (message identifier), and **`MPI_Recv`** asks for a specific message (by tag) from a specific source (by rank).

Parallelism is Nondeterministic

```
for (source = 0; source < num_procs; source++) {
  if (source != server_rank) {
    mpi_error =
      MPI_Recv(message, maximum_message_length + 1,
        MPI_CHAR, MPI_ANY_SOURCE, tag,
        MPI_COMM_WORLD, &status);
    fprintf(stderr, "%s\n", message);
  } /* if (source != server_rank) */
} /* for source */
```

The greetings are **printed** in **non-deterministic** order.

Communicators

An MPI communicator is a collection of processes that can send messages to each other.

`MPI_COMM_WORLD` is the default communicator; it contains all of the processes. It's probably the only one you'll need.

Some libraries create special library-only communicators, which can simplify keeping track of message tags.

Broadcasting

What happens if one process has data that everyone else needs to know?

For example, what if the server process needs to send an input value to the others?

```
MPI_Bcast(length, 1, MPI_INTEGER,
   source, MPI_COMM_WORLD);
```

Note that **`MPI_Bcast`** doesn't use a tag, and that the call is the same for both the sender and all of the receivers.

All processes have to call **`MPI_Bcast`** at the same time; everyone waits until everyone is done.

Broadcast Example: Setup

```
PROGRAM broadcast
  IMPLICIT NONE
  INCLUDE "mpif.h"
  INTEGER,PARAMETER :: server = 0
  INTEGER,PARAMETER :: source = server
  INTEGER,DIMENSION(:),ALLOCATABLE :: array
  INTEGER :: length, memory_status
  INTEGER :: num_procs, my_rank, mpi_error_code

  CALL MPI_Init(mpi_error_code)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank,     &
 &        mpi_error_code)
  CALL MPI_Comm_size(MPI_COMM_WORLD, num_procs,  &
 &        mpi_error_code)
  [input]
  [broadcast]
  CALL MPI_Finalize(mpi_error_code)
END PROGRAM broadcast
```

Broadcast Example: Input

```
PROGRAM broadcast
  IMPLICIT NONE
  INCLUDE "mpif.h"
  INTEGER,PARAMETER  :: server = 0
  INTEGER,PARAMETER  :: source = server
  INTEGER,DIMENSION(:),ALLOCATABLE  :: array
  INTEGER :: length, memory_status
  INTEGER :: num_procs, my_rank, mpi_error_code
```

[MPI startup]

```
  IF (my_rank == server) THEN
    OPEN (UNIT=99,FILE="broadcast_in.txt")
    READ (99,*) length
    CLOSE (UNIT=99)
    ALLOCATE(array(length), STAT=memory_status)
    array(1:length) = 0
  END IF !! (my_rank == server)...ELSE
```

[broadcast]

```
  CALL MPI_Finalize(mpi_error_code)
END PROGRAM broadcast
```

Broadcast Example: Broadcast

```
PROGRAM broadcast
  IMPLICIT NONE
  INCLUDE "mpif.h"
  INTEGER,PARAMETER :: server = 0
  INTEGER,PARAMETER :: source = server
 [other declarations]
 [MPI startup and input]
  IF (num_procs > 1) THEN
    CALL MPI_Bcast(length, 1, MPI_INTEGER, source, &
 &          MPI_COMM_WORLD, mpi_error_code)
    IF (my_rank /= server) THEN
      ALLOCATE(array(length), STAT=memory_status)
    END IF !! (my_rank /= server)
    CALL MPI_Bcast(array, length, MPI_INTEGER, source, &
            MPI_COMM_WORLD, mpi_error_code)
    WRITE (0,*) my_rank, ": broadcast length = ", length
  END IF !! (num_procs > 1)
  CALL MPI_Finalize(mpi_error_code)
END PROGRAM broadcast
```

Broadcast Compile & Run

```
% mpif90 -o broadcast broadcast.f90
% mpirun -np 4 broadcast
 0 : broadcast length =   16777216
 1 : broadcast length =   16777216
 2 : broadcast length =   16777216
 3 : broadcast length =   16777216
```

Reductions

A ***reduction*** converts an array to a scalar: for example, sum, product, minimum value, maximum value, Boolean AND, Boolean OR, etc.

Reductions are so common, and so important, that MPI has two routines to handle them:

`MPI_Reduce`: sends result to a single specified process

`MPI_Allreduce`: sends result to all processes (and therefore takes longer)

Reduction Example

```
PROGRAM reduce
  IMPLICIT NONE
  INCLUDE "mpif.h"
  INTEGER,PARAMETER :: server = 0
  INTEGER :: value, value_sum
  INTEGER :: num_procs, my_rank, mpi_error_code

  CALL MPI_Init(mpi_error_code)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank,   mpi_error_code)
  CALL MPI_Comm_size(MPI_COMM_WORLD, num_procs, mpi_error_code)
  value_sum = 0
  value     = my_rank * num_procs
  CALL MPI_Reduce(value, value_sum, 1, MPI_INT, MPI_SUM, &
 &        server, MPI_COMM_WORLD, mpi_error_code)
  WRITE (0,*) my_rank, ": reduce  value_sum = ", value_sum
  CALL MPI_Allreduce(value, value_sum, 1, MPI_INT, MPI_SUM, &
 &        MPI_COMM_WORLD, mpi_error_code)
  WRITE (0,*) my_rank, ": allreduce value_sum = ", value_sum
  CALL MPI_Finalize(mpi_error_code)
END PROGRAM reduce
```

Compiling and Running

```
% mpif90 -o reduce reduce.f90
% mpirun -np 4 reduce
 3 : reduce  value_sum =  0
 1 : reduce  value_sum =  0
 2 : reduce  value_sum =  0
 0 : reduce  value_sum =  24
 0 : allreduce value_sum =  24
 1 : allreduce value_sum =  24
 2 : allreduce value_sum =  24
 3 : allreduce value_sum =  24
```

Why Two Reduction Routines?

MPI has two reduction routines because of the high cost of each communication.

If only one process needs the result, then it doesn't make sense to pay the cost of sending the result to all processes.

But if all processes need the result, then it may be cheaper to reduce to all processes than to reduce to a single process and then broadcast to all.

Non-blocking Communication

MPI allows a process to start a send, then go on and do work while the message is in transit.

This is called ***non-blocking*** or ***immediate*** communication.

Here, "immediate" refers to the fact that the call to the MPI routine returns immediately rather than waiting for the communication to complete.

Immediate Send

```
mpi_error_code =
    MPI_Isend(array, size, MPI_FLOAT,
        destination, tag, communicator, request);
```

Likewise:

```
mpi_error_code =
    MPI_Irecv(array, size, MPI_FLOAT,
        source, tag, communicator, request);
```

This call starts the send/receive, but the send/receive won't be complete until:

```
MPI_Wait(request, status);
```

What's the advantage of this?

Communication Hiding

In between the call to `MPI_Isend/Irecv` and the call to `MPI_Wait`, both processes can **do work**!

If that work takes at least as much time as the communication, then the cost of the communication is effectively zero, since the communication won't affect how much work gets done.

This is called ***communication hiding***.

Rule of Thumb for Hiding

When you want to hide communication:

- as soon as you calculate the data, send it;
- don't receive it until you need it.

That way, the communication has the maximal amount of time to happen in ***background*** (behind the scenes).

Okla. Supercomputing Symposium

Tue Oct 7 2008 @ OU

Over 250 registrations already!

Over 150 in the first day, over 200 in the first week, over 225 in the first month.

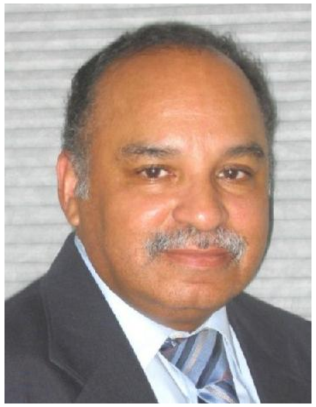
2003 Keynote:
Peter Freeman
NSF
Computer & Information Science & Engineering
Assistant Director

2004 Keynote:
Sangtae Kim
NSF Shared Cyberinfrastructure
Division Director

2005 Keynote:
Walt Brooks
NASA Advanced Supercomputing
Division Director

2006 Keynote:
Dan Atkins
Head of NSF's Office of Cyber-infrastructure

2007 Keynote:
Jay Boisseau
Director
Texas Advanced Computing Center
U. Texas Austin


2008 Keynote:
José Munoz
Deputy Office Director/ Senior Scientific Advisor
Office of Cyber-infrastructure
National Science Foundation

FREE! Parallel Computing Workshop Mon Oct 6 @ OU sponsored by SC08

FREE! Symposium Tue Oct 7 @ OU

`http://symposium2008.oscer.ou.edu/`

To Learn More

http://www.oscer.ou.edu/

http://www.sc-conference.org/

Thanks for your attention!

Questions?

References

[1] P.S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, 1997.

[2] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. MIT Press, 1999.