Parallel & Cluster Computing

MPI Basics

National Computational Science Institute
August 8-14 2004

Paul Gray, University of Northern Iowa
David Joiner, Shodor Education Foundation
Tom Murphy, Contra Costa College
Henry Neeman, University of Oklahoma
Charlie Peck, Earlham College

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 Application Programming Interface (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: IBM has its own, SGI has its own, Sun has its own, etc.
- Plus, there are portable versions: MPICH, LAM-MPI.

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 processors in a run, N_p (typically called just after `MPI_Init`).

`MPI_Comm_rank` gets the processor 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 processor to some other processor (the destination).

MPI_Recv receives a message on the current processor from some other processor (the source).

MPI_Bcast broadcasts a message from one processor to all of the others.

MPI_Reduce performs a reduction (e.g., sum) of a variable on all processors, sending the result to a single processor.

… and many others.

MPI Program Structure (F90)

```
PROGRAM my_mpi_program
  USE mpi
  IMPLICIT NONE
  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>
[other header includes go here]
#include "mpi.h"

int main (int argc, char* argv[])
{ /* main */
  int my_rank, num_procs, mpi_error;
  [other declarations go here]
  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 */
```

SPMD Computational Model

SPMD: Single Program, Multiple Data

```
int main (int argc, char* argv[])
{
  MPI_Init(&argc, &argv); /* Start up MPI  */
    .
    .
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  if (my_rank == 0)
   master();
  else
   slave();
   .
   .
  mpi_error = MPI_Finalize();  /* Shut down MPI */
}
```

Example: Hello World

1. Start the MPI system.
2. Get the rank and number of processors.
3. If you're not the master process:
   1. Create a "hello world" string.
   2. Send it to the master process.
4. If you are the master process:
   1. For each of the other 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  master_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 != master_rank) {
    [work of each non-master process]
  } /* if (my_rank != master_rank) */
  else {
    [work of master process]
  } /* if (my_rank != master_rank)…else */
  mpi_error = MPI_Finalize();
} /* main */
```

Hello World Non-master's Work

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

Hello World Master's Work

[header file includes]

```
int main (int argc, char* argv[])
{ /* main */
```

[declarations, MPI startup]

```
  if (my_rank != master_rank) {
```

[work of each non-master process]

```
  } /* if (my_rank != master_rank) */
  else {
    for (source = 0; source < num_procs; source++) {
      if (source != master_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 != master_rank) */
    } /* for source */
  } /* if (my_rank != master_rank)…else */
  mpi_error = MPI_Finalize();
} /* main */
```

Compiling and Running

```
% setenv MPIENV gcc
```
[Do this only once per login; use `nag` *for Fortran.]*

```
% 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 Master?

```
const int master_rank = 0;
```

By convention, the master process has rank (process ID) #0. Why?

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.

Note: every MPI run also has a process with rank N_p-1, so you could use N_p-1 as the master 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 != master_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 != master_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
- Rank of sending process (source)
- Rank of process to receive (destination)
- Tag (message ID)
- Communicator (e.g., **MPI_COMM_WORLD**)

MPI Data Types

MPI	C/C++	Fortran
`MPI_CHAR`	`char`	`CHARACTER`
`MPI_INT`	`int`	`INTEGER`
`MPI_FLOAT`	`float`	`REAL`
`MPI_DOUBLE`	`double`	`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 != master_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 != master_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).

We could do this in nondeterministic order, using **`MPI_ANY_SOURCE`**.

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, at least until we get to the last example code (flow in Cartesian coordinates).
- Some libraries (e.g., PETSc) create special library-only communicators, which can simplify keeping track of message tags.

Point-to-point Communication

- Point-to-point means one specific process talks to another specific process.
- The "hello world" program provides a simple implementation of point-to-point communication.
- Many variations! – idea of "local completion" vs. "global completion" of the communication
- Blocking and Nonblocking Routines
- Communication Modes:
  - standard – no assumptions on when the recv is started
  - buffered – send may start before a matching recv, app buf
  - synchronous – complete send & recv together
  - ready – send can only start if matching recv has begun

Collective Communication

- Collective communications involve sets of processes
- Intra-communicators are used to delegate the group members
- Instead of using message tags, communication is coordinated through the use of common variables.
- Examples: broadcast, reduce, scatter/gather, barrier and all-to-all

Collective Routines

`MPI_Bcast()` – broadcast from the root to all other processes

`MPI_Gather()` – gather values from the group of processes

`MPI_Scatter()` – scatters buffer in parts to group of processes

`MPI_Alltoall()` – sends data from all processes to all processes

`MPI_Reduce()` – combine values on all processes to a single value