Parallel Programming with MPI: Day 1

Science & Technology Support
High Performance Computing

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163
Table of Contents

• Brief History of MPI
• MPI Program Structure
• Message Passing
• Point-to-Point Communications
• Problem Set
Brief History of MPI

- What is MPI?
- MPI Forum
- Goals and Scope of MPI
What Is MPI?

- **Message Passing Interface**
- **What is the message?**

**DATA**

- Allows data to be passed between processes in a distributed memory environment
- **Distributed memory (OSC Athlon and Itanium clusters)**
  - Each processor has local memory
  - Cannot directly access the memory of other processors
MPI Forum

- **First message-passing interface standard**
  - Successor to PVM
- **Sixty people from forty different organizations**
- **International representation**
- **MPI 1.1 Standard developed from 92-94**
- **MPI 2.0 Standard developed from 95-97**
- **Standards documents**
  - [http://www.mpi-forum.org/docs/docs.html](http://www.mpi-forum.org/docs/docs.html) (postscript versions)
Goals and Scope of MPI

• **MPI’s prime goals are:**
  – To provide source-code portability
  – To allow efficient implementation

• **It also offers:**
  – A great deal of functionality
  – Support for heterogeneous parallel architectures
Acknowledgments

• Edinburgh Parallel Computing Centre/University of Edinburgh for material on which this course is based
MPI Program Structure

- Handles
- MPI Communicator
- MPI_Comm_world
- Header files
- MPI function format
- Initializing MPI
- Communicator Size
- Process Rank
- Exiting MPI
Handles

• MPI controls its own internal data structures

• MPI releases “handles” to allow programmers to refer to these

• C handles are of defined typedefs

• In Fortran, all handles have type INTEGER
MPI Communicator

- Programmer view: group of processes that are allowed to communicate with each other

- All MPI communication calls have a communicator argument

- Most often you will use MPI_COMM_WORLD
  - Defined when you call MPI_Init
  - It is all of your processors...
MPI_COMM_WORLD Communicator
Header Files

- MPI constants and handles are defined here

C:

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

Fortran:

```fortran
include 'mpif.h'
```
MPI Function Format

C:

    error = MPI_Xxxxx(parameter,...);

    MPI_Xxxxx(parameter,...);

Fortran:

    CALL MPI_XXXXX(parameter,...,IERROR)
Initializing MPI

- Must be the first routine called (only once)

C:

```c
int MPI_Init(int *argc, char ***argv)
```

Fortran:

```fortran
INTEGER IERROR
CALL MPI_INIT(IERROR)
```
Communicator Size

- How many processes are contained within a communicator?

C:

```c
MPI_Comm_size(MPI_Comm comm, int *size)
```

Fortran:

```fortran
INTEGER COMM, SIZE, IERROR
CALL MPI_COMM_SIZE(COMM, SIZE, IERROR)
```
Process Rank

- **Process ID number within the communicator**
  - Starts with zero and goes to (n-1) where n is the number of processes requested
- **Used to identify the source and destination of messages**
- **Also used to allow different processes to execute different code simultaneously**

**C:**

```c
MPI_Comm_rank(MPI_Comm comm, int *rank)
```

**Fortran:**

```fortran
INTEGER COMM, RANK, IERROR
CALL MPI_COMM_RANK(COMM, RANK, IERROR)
```
Exiting MPI

• Must be called last by “all” processes

C:

    MPI_Finalize()

Fortran:

    CALL MPI_FINALIZE(IERROR)
Bones.c

#include <mpi.h>

void main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* … your code here … */

    MPI_Finalize ();
}

PROGRAM skeleton
INCLUDE 'mpif.h'
INTEGER ierror, rank, size
CALL MPI_INIT(ierr)
CALL MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)

C ... your code here ...

CALL MPI_FINALIZE(ierr)
END
What’s in a Message

• Messages
• MPI Basic Datatypes - C
• MPI Basic Datatypes - Fortran
• Rules and Rationale
Messages

• A message contains an array of elements of some particular MPI datatype

• MPI Datatypes:
  – Basic types
  – Derived types

• Derived types can be build up from basic types
  – Covered Later …

• C types are different from Fortran types
### MPI Basic Datatypes - C

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>C Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>Signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
### MPI Basic Datatypes - Fortran

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>Fortran Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td></td>
</tr>
</tbody>
</table>
Rules and Rationale

• Programmer declares variables to have “normal” C/Fortran type, but uses matching MPI datatypes as arguments in MPI routines

• Mechanism to handle type conversion in a heterogeneous collection of machines

• General rule: MPI datatype specified in a receive must match the MPI datatype specified in the send
Point-to-Point Communications

- Definitions
- Communication Modes
- Routine Names (blocking)
- Sending a Message
- Memory Mapping
- Synchronous Send
- Buffered Send
- Standard Send
- Ready Send
- Receiving a Message

- Wildcarding
- Getting Wildcarded Information
- Received Message Count
- Message Order Preservation
- Sample Programs
- Timers
Point-to-Point Communication

- Communication between two processes
- **Source** process *sends* message to destination process
- **Destination** process *receives* the message
- Communication takes place within a communicator
- **Destination** process is identified by its rank in the communicator
Definitions

• “Completion” of the communication means that memory locations used in the message transfer can be safely accessed
  – Send: variable sent can be reused after completion
  – Receive: variable received can now be used

• MPI communication modes differ in what conditions are needed for completion

• Communication modes can be blocking or non-blocking
  – Blocking: return from routine implies completion
  – Non-blocking: routine returns immediately, user must test for completion
## Communication Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>Completion Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous send</td>
<td>Only completes when the receive has completed</td>
</tr>
<tr>
<td>Buffered send</td>
<td>Always completes (unless and error occurs), irrespective of receiver</td>
</tr>
<tr>
<td>Standard send</td>
<td>Message sent (receive state unknown)</td>
</tr>
<tr>
<td>Ready send</td>
<td>Always completes (unless and error occurs), irrespective of whether the receive has completed</td>
</tr>
<tr>
<td>Receive</td>
<td>Completes when a message has arrived</td>
</tr>
</tbody>
</table>
## Routine Names (blocking)

<table>
<thead>
<tr>
<th>MODE</th>
<th>MPI CALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard send</td>
<td>MPI_SEND</td>
</tr>
<tr>
<td>Synchronous send</td>
<td>MPI_SSEND</td>
</tr>
<tr>
<td>Buffered send</td>
<td>MPI_BSEND</td>
</tr>
<tr>
<td>Ready send</td>
<td>MPI_RSEND</td>
</tr>
<tr>
<td>Receive</td>
<td>MPI_RECV</td>
</tr>
</tbody>
</table>
Sending a Message

C:

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Fortran:

```fortran
<type> BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG
INTEGER COMM, IERROR

CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)
```
Arguments

- **buf**: starting *address* of the data to be sent
- **count**: number of elements to be sent
- **datatype**: MPI datatype of each element
- **dest**: rank of destination process
- **tag**: message marker (set by user)
- **comm**: MPI communicator of processors involved

```c
MPI_SEND(data, 500, MPI_REAL, 6, 33, MPI_COMM_WORLD, IERROR)
```
Memory Mapping

The Fortran 2-D array

<table>
<thead>
<tr>
<th></th>
<th>1,1</th>
<th>1,2</th>
<th>1,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,1</td>
<td>2,2</td>
<td>2,3</td>
<td></td>
</tr>
<tr>
<td>3,1</td>
<td>3,2</td>
<td>3,3</td>
<td></td>
</tr>
</tbody>
</table>

Is stored in memory
Synchronous Send

- **Completion criteria:**
  Completes when message has been received

- **Use if need to know that message has been received**

- **Sending & receiving processes synchronize**
  - regardless of who is faster
  - processor idle time is probable

- **“Fax-type” communication method**
Buffered Send

• Completion criteria:
  Completes when message copied to buffer

• Advantage: Completes immediately

• Disadvantage: User cannot assume there is a pre-allocated buffer

• Control your own buffer space using MPI routines
  MPI_Buffer_attach
  MPI_Buffer_detach
Standard Send

- Completion criteria: **Unknown!**

- May or may not imply that message has arrived at destination

- Don’t make any assumptions (implementation dependent)
Ready Send

- Completion criteria: Completes immediately, but only successful if matching receive already posted

- Advantage: Completes immediately

- Disadvantage: User must synchronize processors so that receiver is ready

- Potential for good performance, but synchronization delays possible
Receiving a Message

C:

```c
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, \
    int source, int tag, MPI_Comm comm, MPI_Status *status)
```

Fortran:

```fortran
                                       BUF(*)
INTEGER COUNT, DATATYPE, DEST, TAG
INTEGER COMM, STATUS(MPI_STATUS_SIZE), IERROR

CALL MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)
```
For a communication to succeed

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
- The communicator must be the same
- Tags must match
- Receiver’s buffer must be large enough
Wildcarding

- Receiver can wildcard
- To receive from any source
  
  \texttt{MPI\_ANY\_SOURCE}

  To receive with any tag
  
  \texttt{MPI\_ANY\_TAG}

- Actual source and tag are returned in the receiver’s \texttt{status} parameter
Using the Status Handle

- Information from a wildcarded receive is returned from `MPI_RECV` in status handle

<table>
<thead>
<tr>
<th>Information</th>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>source</td>
<td><code>status.MPI_SOURCE</code></td>
<td><code>status(MPI_SOURCE)</code></td>
</tr>
<tr>
<td>tag</td>
<td><code>status.MPI_TAG</code></td>
<td><code>status(MPI_TAG)</code></td>
</tr>
<tr>
<td>count</td>
<td><code>MPI_Get_count</code></td>
<td><code>MPI_GET_COUNT</code></td>
</tr>
</tbody>
</table>
Received Message Count

- Message received may not fill receive buffer
- \texttt{count} is number of elements actually received

\textbf{C:}

\begin{verbatim}
int MPI_Get_count (MPI_Status *status,
                 MPI_Datatype datatype, int *count)
\end{verbatim}

\textbf{Fortran:}

\begin{verbatim}
INTEGER STATUS(MPI_STATUS_SIZE), DATATYPE
INTEGER COUNT,IERROR

CALL MPI_GET_COUNT(STATUS,DATATYPE,COUNT,IERROR)
\end{verbatim}
Message Order Preservation

- Messages do not overtake each other
- Example: Process 0 sends two messages
  Process 2 posts two receives that match either message
  Order preserved
Sample Program #1 - C

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/* Run with two processes */
void main(int argc, char *argv[]) {
    int rank, i, count;
    float data[100], value[200];
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank==1) {
        for(i=0;i<100;++i) data[i]=i;
        MPI_Send(data,100,MPI_FLOAT,0,55,MPI_COMM_WORLD);
    } else {
        MPI_Recv(value,200,MPI_FLOAT,MPI_ANY_SOURCE,55,MPI_COMM_WORLD,&status);
        printf("P:%d Got data from processor %d \n",rank, status.MPI_SOURCE);
        MPI_Get_count(&status,MPI_FLOAT,&count);
        printf("P:%d Got %d elements \n",rank,count);
        printf("P:%d value[5]=%f \n",rank,value[5]);
    }
    MPI_Finalize();
}
```

```
P: 0 Got data from processor 1
P: 0 Got 100 elements
P: 0 value[5]=5.000000
```
Sample Program #1 - Fortran

```fortran
PROGRAM p2p
C Run with two processes
INCLUDE 'mpif.h'
INTEGER err, rank, size
real data(100), value(200)
integer status(MPI_STATUS_SIZE)
integer count
CALL MPI_INIT(err)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rank,err)
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,size,err)
if (rank.eq.1) then
  data=3.0
  call MPI_SEND(data,100,MPI_REAL,0,55,MPI_COMM_WORLD,err)
else
  call MPI_RECV(value,200,MPI_REAL,MPI_ANY_SOURCE,55,
                 MPI_COMM_WORLD,status,err)
  print *, "P:",rank," got data from processor ",
  call MPI_GET_COUNT(status,MPI_REAL,count,err)
  print *, "P:",rank," got ",count," elements"
  print *, "P:",rank," value(5)=","value(5)
end if
CALL MPI_FINALIZE(err)
END
```

P: 0 Got data from processor 1
P: 0 Got 100 elements
Timers

- Time is measured in seconds

- Time to perform a task is measured by consulting the timer before and after

C:
   `double MPI_Wtime(void);`

Fortran:
   `DOUBLE PRECISION MPI_WTIME()`
Problem Set

• At the end of each day of the course, the attendees should do all the problems for that day.

• Before next week’s class, you should email your solutions to me – dje@osc.edu - and I will provides constructive comments on your code.

• You should ask you local AG personnel for instructions on how to compile and run an MPI program (asking for a certain number of processes) on your local machines.
Problem Set

a) Write an MPI program in which each processor prints out the message “Parallel Processing”. Pick the number of processors you wish to use as long as it is four or more.

b) Revise your program written for part a) so that each processor prints out its rank as well as the message. Run this program six times. What do you notice?

c) Revise your program written for part b) so that only the processor whose rank equals half the total number of processors performs the output. (When you run your code, use an even number of processors.)

Rewrite Sample Program #1 (in the handouts) so that a message is sent from P1 (P1= processor with rank 1) to P2. The message should be an array of 50 integers: each element equal to 6. P2 should receive the message with both the source and tag wildcarded. P2 should then print out who was the source, what was the tag, and one of the array elements received.

Write a program in which data are exchanged between P1 and P3. P1 should calculate the squares of the first 200 integers. The resulting array should then be transferred to P3. P3 should divide the integers between 20 and 119 by 53.0. The resulting (real) array should then be transferred to P1.

Output whatever you feel is necessary to confirm that the data transfers occurred correctly. Use NO wildcarding in your sends and receives.
Problem Set

Write a program which will be run on four processors. P3 will receive messages from all the other processors with the source rank being wildcarded. Each processor should send a different integer (you pick them). When P3 receives a message from P0, it should print out 2 raised to the integer power contained in the message. When P3 receives a message from P1, it should print out the factorial of the integer received. And, finally, when P3 receives a message from P2 it should print out the sum of all integers up to and including the integer in the message.