Message Passing Programming Based on MPI

Point to point communication

Gürsan ÇOBAN

13.06.2012
Outline

• What is MPI?
• Information Routines
• Environment Management Routines
• Basic Point to Point Communication (Send/Receive)
• MPI Datatypes
• Point to Point Communication
MPI = Massage Passing Interface

• MPI is a specification for the developers and users of message passing libraries.
  – A library for creating separate processes for execution on different computers
  – A library of sending and receiving messages
• The goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:
  – practical
  – portable
  – efficient
  – flexible
• Interface specifications have been defined for C/C++ and Fortran programs.
Reasons for Using MPI

• Standardization: MPI is the only message passing library which can be considered a standard. November 1993. Defines routines, not implementation.

• Portability: There is no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

• Performance Opportunities: Vendor implementations should be able to exploit native hardware features to optimize performance.

• Functionality: Over 115 routines are defined in MPI-1 alone.

• Availability: A variety of implementations are available, both vendor and public domain.
Message Passing Interface

- Function categories:
  - Initialization/finalization
  - Point-to-point communication functions
  - Collective communication functions
  - Communicator topologies
  - User-defined data types
  - Utilities (e.g., timing)

Point to point communication 13.06.2012

[uhem]
Single Program Multiple Data (SPMD) Model

Basic MPI way
Minimum MPI Program Skeleton

1. Include MPI header file:
   - `MPI include file`

2. Initialize MPI environment:
   - `MPI_Init`
   - `MPI_Finalize`

3. Do work and make message passing calls:

4. Terminate MPI Environment:
   - `MPI_Finalize`
Format of MPI Calls (C Binding)

- Format:
  \[ rc = \text{MPI} \_\text{Xxxxxx}(\text{parameter}, \ldots) \]

- Example:
  \[ rc = \text{MPI} \_\text{Bsend}(&\text{buf}, \text{count}, \text{type}, \text{dest}, \text{tag}, \text{comm}) \]

- Error code:
  Returned as "rc". MPI\_SUCCESS if successful.
Format of MPI Calls (Fortran Binding)

- Format:

  ```fortran
  CALL MPI_XXXX(parameter, ..., ierr )
  call mpi_xxxx(parameter, ..., ierr )
  ```

- Example:

  ```fortran
  CALL MPI_BSEND(buf,count,type,dest,tag,comm,ierr)
  ```

- Error code:

  Returned as "ierr" parameter. MPI_SUCCESS if successful.
Getting Information

- MPI uses
  - objects,
  - communicators,
  - groups
to define which collection of processes may communicate with each other.
- Most MPI routines require you to specify a communicator as an argument.
- Simply use `MPI_COMM_WORLD` whenever a communicator is required.
- `MPI_COMM_WORLD` includes all of your MPI processes.
Getting Information

- **MPI_Comm_size**: Determines the size of the group associated with a communicator.
- **MPI_Comm_rank**: Determines the rank of the calling process in the communicator.

Point to point communication 13.06.2012
Getting Information

• Rank within a communicator:
  • Every process has its own unique,
  • Integer identifier assigned by the system when the
    process initializes.
  • A rank is sometimes also called a "process ID".
  • Ranks are contiguous and begin at zero.
Environment Management Routines

- Purposes of MPI environment management routines:
  - Initializing the MPI environment,
  - Terminating the MPI environment,
  - Querying the environment,
  - Identity, etc.
Environment Management Routines

MPI_Init

- Initializes the MPI execution environment.
- Must be called in every MPI program,
- Must be called before any other MPI functions
- Must be called only once in an MPI program.
- For C programs, MPI_Init may be used to pass the command line arguments to all processes.

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

MPI_INIT(ierr)
```
Environment Management Routines

MPI_Finalize

- Terminates MPI execution environment.
- After this routine, any of MPI routines does not

```c
int MPI_Finalize (void)

MPI_FINALIZE (ierr)
```
Environment Management Routines

**MPI_Comm_size**

- Determines the number of processes in the group associated with a communicator.
- Generally used within the communicator `MPI_COMM_WORLD` to determine the number of processes being used by your application.

```c
int MPI_Comm_size (MPI_COMM_WORLD,int *size)

MPI_COMM_SIZE (MPI_COMM_WORLD,size,ierr)
```
MPI_Comm_rank

- Determines the rank of the calling process within the communicator.
- A unique integer rank between 0 and (#OfProcs-1) within the communicator **MPI_COMM_WORLD**.
- The rank is referred to as a **task ID**.

`int MPI_Comm_rank (MPI_COMM_WORLD, int *rank)`

`MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierr)`
Environment Management Routines

MPI_Get_processor_name

- Returns the processor name.
- Also returns the length of the name.

```
int MPI_Get_processor_name (char *name, int *length)
```

```
mpi_get_processor_name (processorName, nameLen, ierr)
```
Simple MPI example

#include <stdio.h>
#include “mpi.h”

int main(int argc, char * argv[]) {
    int taskid, ntasks;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);
    printf("Hello world! I am %d of %d\n", taskid, ntasks);
    MPI_Finalize();
    return(0);
}

Fortran

Program SimpleMPI
implicit none
include "mpif.h"
integer My_Id, Numb_of_Procs, ierr
call MPI_INIT ( ierr )
call MPI_COMM_RANK ( &MPI_COMM_WORLD, My_Id, ierr )
call MPI_COMM_SIZE ( &MPI_COMM_WORLD, Numb_of_Procs, ierr )
print *, "Hello World! This is ", My_id, " of ", Numb_of_Procs
call MPI_FINALIZE ( ierr ) ! bad things happen if you forget ierr

End Program SimpleMPI
MPI_Wtime

- Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

```c
double MPI_Wtime()
```

```c
MPI_Wtime()
```
Environment Management Routines

**MPI_Abort**

- Terminates all MPI processes associated with the communicator.
- In most MPI implementations it terminates ALL processes regardless of the communicator specified.

```c
int MPI_Abort( MPI_COMM_WORLD, int errorcode )

MPI_ABORT( MPI_COMM_WORLD, errorcode, ierr )
```
Sending Data

**MPI_Send**

- Performs a blocking send of the specified data to the specified destination.

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

- Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse.
Sending Data

**MPI_Send**

- Performs a blocking send of the specified data to the specified destination.

```c
MPI_SEND( buf, count,
          MPI_Datatype datatype,
          dest,
          tag,
          comm,
          ierr )
```
Receiving Data

**MPI_Recv**

- Performs a blocking receive of the specified data from the specified source.

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

- Receive a message and block until the requested data is available in the application buffer in the receiving task.
Receiving Data

**MPI_Recv**

- Performs a blocking receive of the specified data from the specified source.

```c
MPI_RECV( buf, count,
        MPI_Datatype datatype,
        source,
        tag,
        comm,
        MPI_Status status,ierr )
```
Destination

An argument to send routines that indicates the process where a message should be delivered. Specified as the rank of the receiving process.

Source

An argument to receive routines that indicates the originating process of the message. Specified as the rank of the sending process. This may be set to the wild card MPI_ANY_SOURCE to receive a message from any task.
Tag - Communicator

Tag

Arbitrary non-negative integer assigned by the programmer to uniquely identify a message. Send and receive operations should match message tags. For a receive operation, the wild card MPI_ANY_TAG can be used to receive any message regardless of its tag. The MPI standard guarantees that integers 0-32767 can be used as tags, but most implementations allow a much larger range than this.

Communicator

Indicates the communication context, or set of processes for which the source or destination fields are valid. Unless the programmer is explicitly creating new communicators, the predefined communicator MPI_COMM_WORLD is usually used.
Status

For a receive operation, indicates the source of the message and the tag of the message. In C, this argument is a pointer to a predefined structure MPI_Status (ex. stat.MPI_SOURCE stat.MPI_TAG). In Fortran, it is an integer array of size MPI_STATUS_SIZE (ex. stat(MPI_SOURCE) stat(MPI_TAG)).
MPI_Send and MPI_Recv

Process 1

compute
Send (P2, info);
compute
compute
idle
idle
Receive (P2, reply);

Process 2

idle
idle
Receive (P1, info);
compute
compute
compute
Send (P1, reply);
MPI_Send and MPI_Recv

Point to point communication 13.06.2012
MPI_Send and MPI_Recv

Waits for a message from process 1 with a tag of 5
## MPI Datatypes (C)

<table>
<thead>
<tr>
<th>MPI Type</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed 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_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
</tbody>
</table>
### MPI Datatypes (Fortran)

<table>
<thead>
<tr>
<th>MPI Datatype</th>
<th>Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHARACTER</td>
<td>character</td>
</tr>
<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_DOUBLE_COMPLEX</td>
<td>double complex</td>
</tr>
</tbody>
</table>
# MPI Datatypes (C)

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned 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_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>8 binary digits</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>data packed/unpacked with MPI_Pack() / MPI_Unpack</td>
</tr>
</tbody>
</table>
### MPI Datatypes (Fortran)

<table>
<thead>
<tr>
<th>Datatype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_LOGICAL</td>
<td>logical</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>8 binary digits</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>data packed/unpacked with MPI_Pack() / MPI_Unpack</td>
</tr>
</tbody>
</table>
Point to Point Communication

- The **communication mode** is **selected** with the **send routine**.
- Four blocking send routines,
- Four non-blocking send routines.
- The **receive routine does not specify communication mode**.
- The **receive routine** is simply **blocking or non-blocking**.
# Point to Point Communication

<table>
<thead>
<tr>
<th>Communication Mode</th>
<th>Blocking Routines</th>
<th>Non-Blocking Routines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>MPI_Ssend</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td>Ready</td>
<td>MPI_Rsend</td>
<td>MPI_Irsend</td>
</tr>
<tr>
<td>Buffered</td>
<td>MPI_Bsend</td>
<td>MPI_Ibsend</td>
</tr>
<tr>
<td>Standard</td>
<td>MPI_Send</td>
<td>MPI_Isend</td>
</tr>
<tr>
<td></td>
<td>MPI_Recv</td>
<td>MPI_Irecv</td>
</tr>
<tr>
<td></td>
<td>MPI_Sendrecv</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MPI_Sendrecv_replace</td>
<td></td>
</tr>
</tbody>
</table>

The receive routine is simply blocking (MPI_Recv) or non-blocking (MPI_Irecv).
Before Activity

Dead Lock!

#include “mpi.h”
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &myrsize);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if ( my_PE_num == 0 )
    MPI_Recv(&message, 12, MPI_CHAR, 1, ntag, 
             MPI_COMM_WORLD, &status);
else if ( my_PE_num == 1 ) {
    MPI_Recv(&message, 12, MPI_CHAR, 0, ntag, 
             MPI_COMM_WORLD, &status);
    printf("Node %d : %s\n", my_PE_num, message); }
MPI_Finalize();
Before Activity

Before Activity
Remember!

#include “mpi.h”

Initialize:

MPI_Init(&argc, &arcv);
MPI_Comm_size(MPI_COMM_WORLD, &mysize);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

Finalize:

MPI_Finalize();

Compile with

mpicc -O foo.c -o foo.out

Run with

mpirun -np #procs ./foo.out (or use ‘bsub’)

Point to point communication

13.06.2012
Activity: sendrecv
(Send and Receive data with Message)

- Filling the blanks in sendrecv.c program with following conditions:

I. **Master**: Sends a block of data to processes 1 to N iteratively. It prints the sending message.

II. **Process from 1 to (N)**: Receives the data from rank 0 with ‘any tag’, it prints the receiving message.
Activity: sendrecv
(Send and Receive data with Message)

Point to point communication 13.06.2012
Activity: Hello World
(Hello World with Message)

• Filling the blanks in helloworld1.c program with following conditions:

I. **Master**: Takes a value from stdin, increments it and will sends it to the next process 1.

II. **Process from 1 to (last-1)**: After taking value previous process, it prints the receiving message, and increments the value then sends it to next one.

III. **Last Process**: Takes value from previous one, and prints receiving message.
Activity: Hello World
(Hello World with Message)
Activity

Activity: Hello World
(Hello World with Finish Message)

- Filling the blanks in helloworld2.c program with following conditions:

  I. MPI processes pass an integer variable from left to right, reporting and incrementing its value at each step.

  II. The master process should report when all processes are finished.
Activity: Hello World
(Hello World with Finish Message)

1. **Process 0**
   - Get an integer

2. **Process 1**
   - Increase it and send
   - Print message

3. **Process 2**
   - Increase it and send
   - Print message

4. **Process 0**
   - Increase it and send
   - Reports that all work is finished

Point to point communication

13.06.2012
Activity: ping pong  
(Wall Clock Time for Send and Receive Process)

- **Filling the blanks in pingpong.c program with following conditions:**

  I. Master MPI process (rank 0) sends a block of double sized array of length $2^n$ to rank 1, incrementing its size from $n=1$ to $n=20$ at each iteration. The operation is repeated for 5 times to obtain an average WCT from trials.

  II. For any trial and array size, Process rank 1 receives the array from rank 0 and immediately sends the same amount of data back to rank 0.

  III. Measure elapsed wall clock time (WCT) with MPI_Wtime.

  IV. The master process should report all statistics calculated from the ping-pong operation.
Activity

Activity: ping pong
(Wall Clock Time for Send and Receive Process)

Loop for array size (n=1 to 20)

Loop for trials (trial = 1 to 5)
Final Activity

Calculate Pi

• Open the file mpi_pi.c (mpi_pi.f90). We will edit two lines for sending the value of 'n' and collecting the sum of local variable 'mypi'.
Dimensionality

(Wall Clock Time for Send and Receive Process)

- Analysis of dimensionality (1D, 2D, 3D)
- Ghost point operations
- Domain decomposition

Mesh partition from 3D domain

Data transfer between partitions
Değerlendirme

Genel Değerlendirme

Genel değerlendirme (her bir çalıştayın son günü):

http://workshop.uybhm.itu.edu.tr/feedback.php

Günlük değerlendirme formları:

http://workshop.uybhm.itu.edu.tr/evaluation/general.php