C-DAC Four Days Technology Workshop

ON

Hybrid Computing – Coprocessors/Accelerators
Power-Aware Computing – Performance of Applications Kernels

hyPACK-2013
(Mode-1:Multi-Core)

Lecture Topic:
Multi-Core Processors: MPI 1.0 Overview (Part-I)

Venue : CMSD, UoHYD ; Date : October 15-18, 2013

Introduction to Message Passing Interface (MPI)

Quick overview of what this Lecture is all about

- Basics of MPI
- How to compile and execute MPI programs?
- MPI library calls used in Example program
- MPI point-to-point communication

Source : Reference : [11], [12], [25], [26]

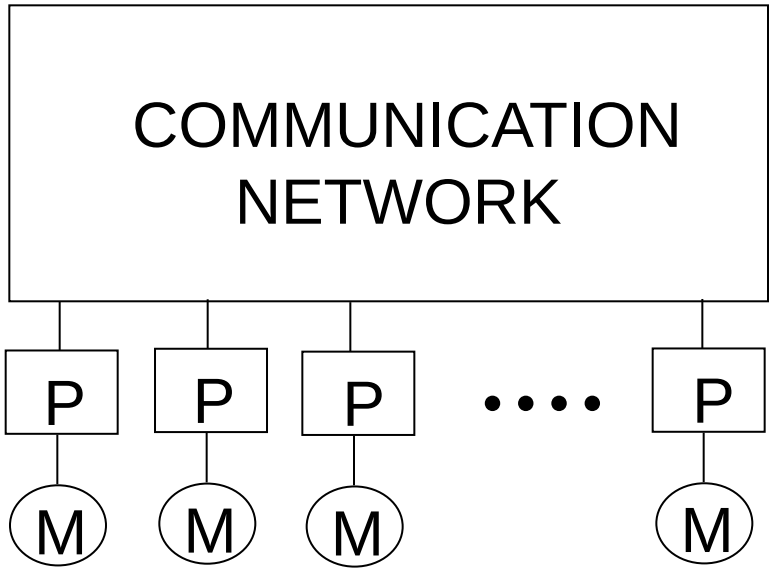
MPI Goals

1. Deals with the principles of parallel programming by passing messages among processing nodes.
2. Understand the effect of different MPI Functions that accomplish the same communication
3. Understand how MPI implementation work
4. Know how to use different MPI functions to solve performance problems
5. Understanding the way in which the different MPI operations are implemented is critical in tuning for performance
6. Other Message Passing Libraries : PVM

Source : Reference : [11], [12], [25], [26]

Message Passing Architecture Model

Message-Passing Programming Paradigm : Processors are connected using a message passing interconnection network.



- On most Parallel Systems, the processes involved in the execution of a parallel program are identified by a sequence of non-negative integers. If there are *p* processes executing a program, they will have ranks 0, 1,2, ……., *p-1.*

Information about MPI

(Contd…)

- MPI Forum – A different approach to developing a **standard** for programming parallel computing systems.
- The foundation of MPI library is a small group of functions that can be used to achieve parallelism by **message-passing**
- A message passing function that explicitly transmits data from one process to another.
- Message Passing programs can be used to create extremely efficient parallel programs.
- Difficult to design and develop programs using message passing

Source : Reference : [11], [12], [25], [26]

Information about MPI

(Contd…)

Where to use MPI ?

- You need a portable parallel program
- You are writing a parallel Library
- You have irregular data relationships that do not fit a data parallel model

Why learn MPI?

- Portable
- Expressive
- Good way to learn about subtle issues in parallel computing
- Universal acceptance

What Is MPI

- A message-passing library specification
  - Message-passing model
  - Not a compiler specification;
  - Not a specific product
- Used for parallel computers, clusters, and heterogeneous networks as a message passing library
- Designed to permit the development of parallel software libraries
- Designed to provide access to advanced parallel hardware for
  - End users
  - Library writers
  - Tool developers

Information about MPI

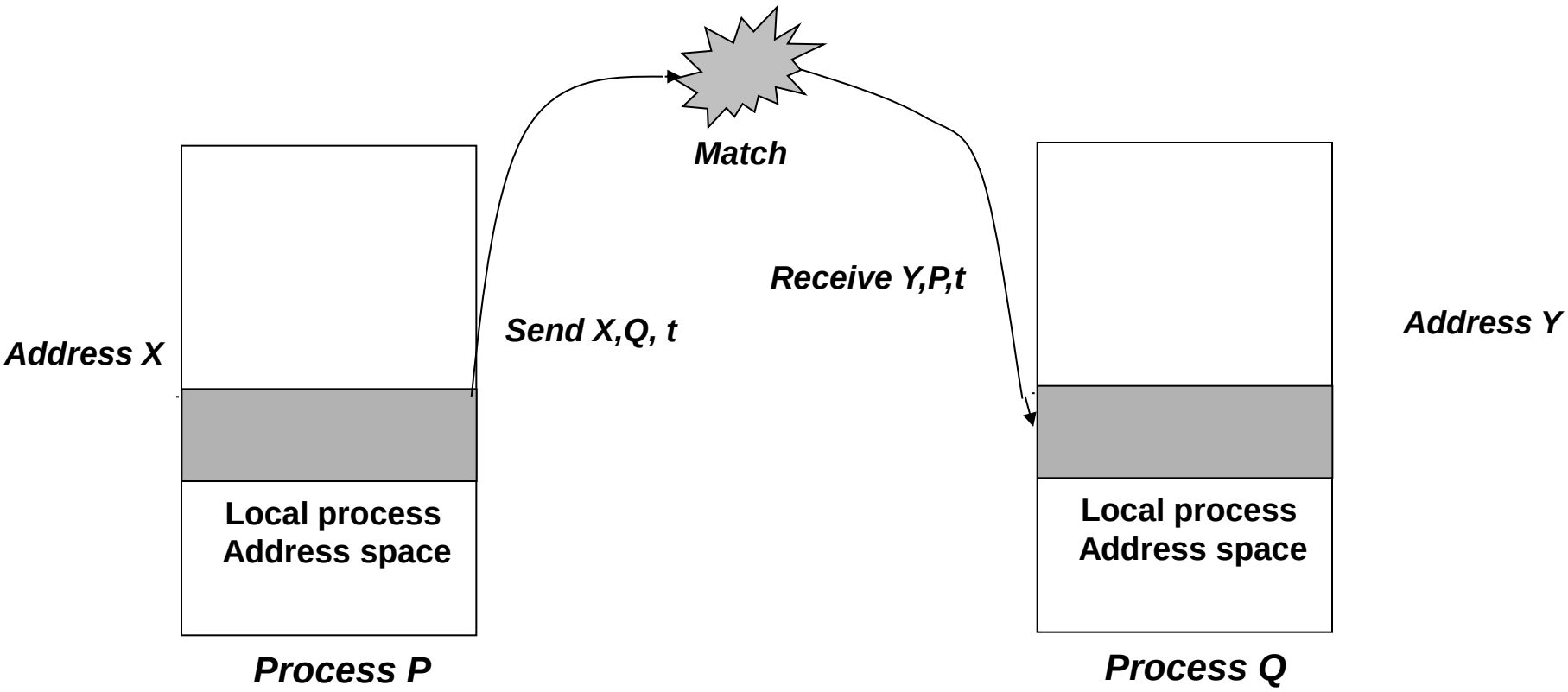
(Contd…)

Why Is A Standard Needed?

- Portability and Ease-of-use
- Provides hardware vendors with well-defined set of routines to implement efficiently
- Pre-requisite for the development of software industry
- MPI is a standard specification for a library of functions developed by *MPI forum*, a broadly based consortium of parallel computer vendors, library writers, and application specialists.
- It is adopted by all parallel computer vendors.

Source : Reference : [11], [12], [25], [26]

The Message Passing Abstraction



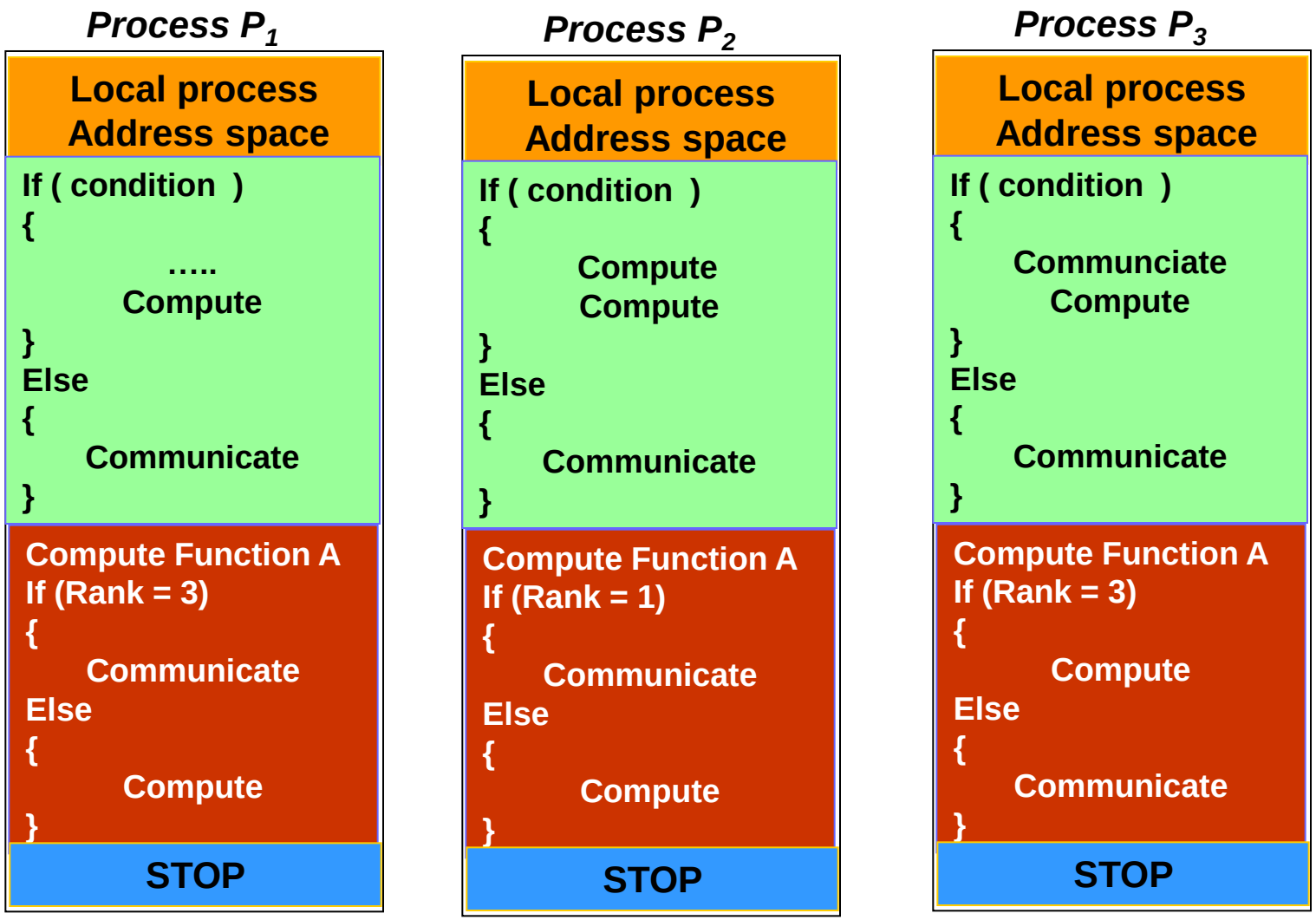
User-Level Send/receive message-passing abstraction : A data transfer from one local address space to another occurs when a *send* to particular processes matches with a *receive* posted by that process

Is MPI Large or Small?

Is MPI Large or Small?

- MPI is large (125 Functions)
  - MPI's extensive functionality requires many functions
  - Number of functions not necessarily a measure of complexity
- MPI is small (6 Functions)
  - Many parallel programs can be written with just 6 basic functions
- MPI is just **right** candidate for message passing
  - One can access flexibility when it is required
  - One need not master all parts of MPI to use it

The Message Passing Abstraction



MPI Program – Compilation & Execution

1. Compile, Linking and execution of your C or FORTRAN Program with MPI libraries on the system. The details of this, depend on the system you're using.
2. Although details of what happens when the program is executed vary from system to system, the essentials are the same on all systems, provided we run one process on each processor.

   - The user issues a directive to the Operating System that has the effect of placing a copy of the executable program on each processor.
   - Each processor begins execution of its copy of the executable.
   - Different process can execute different statements by branching within the program based on their process ranks

Abstraction : *Send* and *receive* buffers in message passing

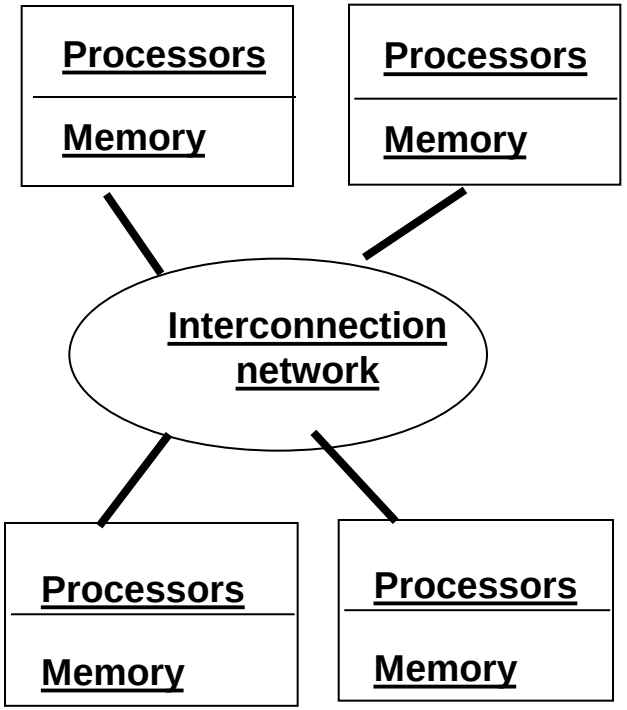
Example : Process P *sends* a message contained in variable M to process Q, which *receives* the message into its variable S.

Process P :	**Process Q :**
M=10; L1 : *send* M to Q L2 : M=20; goto L1;	S= 75; L1 : *receive* S from P L2 : X = S+1;

The variable M is often called the *send message buffer* (or *send buffer*),and S is called the *receive message buffer* (or *receive buffer*)

The Message-Passing Model – Message Passing Modes

- How many processes are involved ?
- How are the processes synchronized ?
- How are communication buffers managed ?



- Blocking Send /Receive
- Non-Blocking Send/Receive
- Synchronous Message Passing

SPMD Program

What is SPMD ?

- Single Program, Multiple Data
  - Same program runs everywhere
  - Restriction on the general message-passing model
  - Some vendors only support SPMD parallel programs

SPMD Program

(Contd…)

Evaluating General Message Passing with SPMD :C program

```
main (int args, char **argv)
{
   if (process is to become a controller process)
   {
      Controller (/* Arguments /*);
   }
   else
   {
      Worker (/* Arguments /*);
   }
}
```

SPMD/MPMD Program

(Contd…)

Evaluating General Message Passing with SPMD : Fortran

```
PROGRAM
      IF (process is to become  a controller process) THEN
         CALL CONTROLLER (/* Arguments /*)
      ELSE
         CALL WORKER (/* Arguments /*)
      ENDIF
END
```

What is MPMD (Non-SPMD)?

- Different programs run on different nodes.
- If one program controls the others then the controlling program is called the *Master* and the others are called the *slaves*.

Compile and Execute MPI programs

(Contd…)

How to compile and execute MPI program?

Compiling

- On some machines, there is a special command to insure that the program links the proper MPI libraries.

  mpif77 program.f **mpicc** program.c

- Compiling a code : Using `Makefile`

  - Include all files for program, appropriate paths to link MPI libraries
  - Used for SPMD and Non-SPMD programs

(Note that this will differ with different MPI libraries).

Compile and Execute MPI programs

(Contd…)

How to compile and execute MPI program?

- Execution **: mpirun -np** 4 a.out

  (To run a program across multiple machines; **np** *is the number of processes)*

- Execution
  - Create `ch_p4` procgroup file (File contains users account name, access to the executable of MPI program, number of processes used, for example `run.pg`)
  - Execute the command `make` (`Makefile` generates executable (say `run`)
  - Type `run` on command line

MPI Basics

Basic steps in an MPI program

- Initialize for communications
- Communicate between processors
- Exit in a “clean” fashion from the message-passing system when done communicating.

Source : Reference : [11], [12], [25], [26]

(Contd…)

Format of MPI Calls

C Language Bindings

Return_integer = MPI_Xxxxx(parameter, ...);

- Return_integer is a return code and is type integer. Upon success, it is set to MPI_SUCCESS.
- Note that case is important
- MPI must be capitalized as must be the first character after the underscore. Everything after that must be lower case.
- C programs should include the file `mpi.h` which contains definitions for MPI constants and functions

(Contd…)

Format of MPI Calls

Fortran Language Buildings

Call MPI_XXXXX(parameter,..., ierror)

or

call mpi_xxxxx(parameter,..., ierror)

- Instead of the function returning with an error code, as in C, the Fortran versions of MPI routines usually have one additional parameter in the calling list, `ierror`, which is the return code. Upon success, `ierror` is set to MPI_SUCCESS.
- Note that case is not important
- Fortran programs should include the file `mpif.h` which contains definitions for MPI constants and functions

(Contd…)

Exceptions to the MPI call formats are timing routines

- Timing routines
  - MPI_WTIME and MPI_WTICK are functions for both C and Fortran
- Return double-precision real values.
- These are not subroutine calls

Fortran

```
Double precision MPI_WTIME()
```

C

```
Double precision MPI_Wtime(void);
```

MPI Basics

MPI Messages

- **Message :** data (3 parameters) + envelope (3 parameters)
  - **Data :** startbuf, count, datatype
    - **Startbuf:** address where the data starts
    - **Count:** number of elements (items) of data in the message
  - **Envelope :** dest, tag, comm
    - **Destination or Source**: Sending or Receiving processes
    - **Tag**: Integer to distinguish messages

Communicator

- The communicator is communication “universe.”
- Messages are sent or received within a given “universe.”
- The default communicator is MPI_COMM_WORLD

MPI Basics

(Contd…)

Initializing MPI

- Must be first routine called.
- C

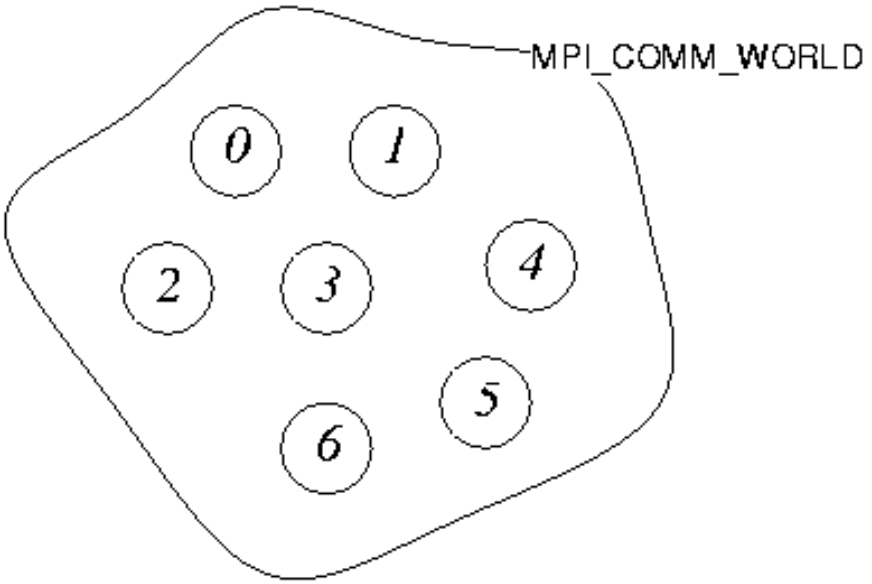
```
int MPI_Init(int *argc,char ***argv);
```

- Fortran

```
MPI_INIT(IERROR)
integer IERROR
```

(Contd…)

MPI_COMM_WORLD communicator



A communicator is MPI's mechanism for establishing individual communication "universe."

(Contd…)

MPI Message Passing Basics

Questions :

- What is my process id number ?
  MPI_COMM_RANK (Rank starts from the integer value 0 to ….)

Fortran

```
call MPI_COMM_RANK (comm, rank, ierror)
integer comm, rank, ierror
```

C

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

(Contd…)

MPI Message Passing Basics

Questions :

- How many processes are contained within a communicator?
- How many processes am I using?

MPI_COMM_SIZE

Fortran

```
call MPI_COMM_SIZE (comm, size, ierror)
```

C

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

(Contd…)

Exiting MPI

- **C**

```
int MPI_Finalize()
```

- **Fortran**

```
MPI_FINALIZE(IERROR)
INTEGER IERROR
```

Note : Must be called last by all processes.

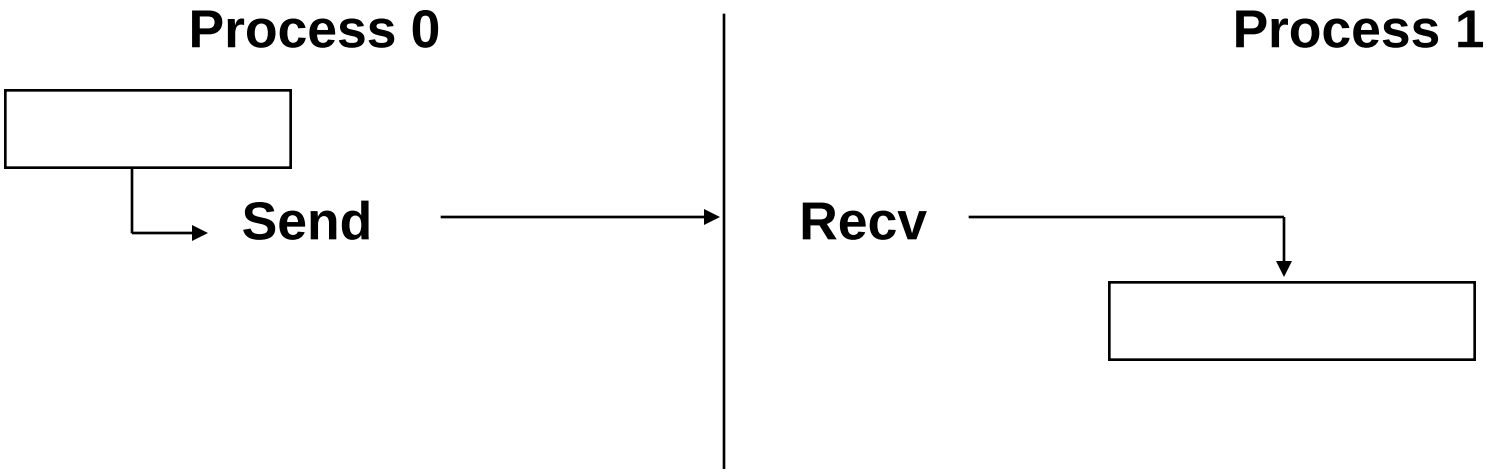
What makes an MPI Program ?

What makes an MPI Program ?

- Include files
  - mpi.h (C)
  - mpif.h (Fortran)
- Initiation of MPI
  - MPI_INIT
- Completion of MPI
  - MPI_FINALIZE

MPI Send and MPI Receive Library Calls

Sending and Receiving messages

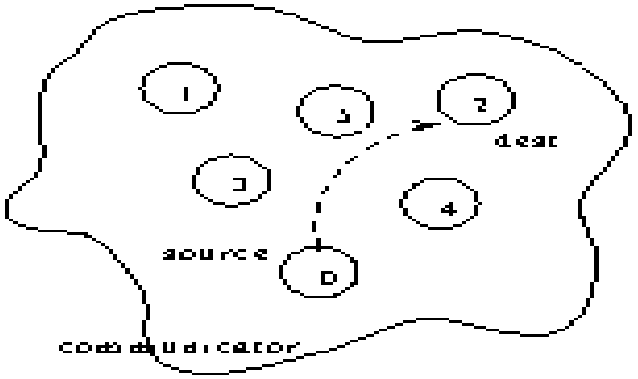


Fundamental questions answered

- To whom is data sent?
- What is sent?
- How does the receiver identify it?

MPI Point-to-Point Communication Library Call

Information on MPI *Send* and MPI *Recv*



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

MPI Point-to-Point Communication Library Calls

(Contd…)

MPI Message Passing : Send

Fortran

MPI_SEND (buf, count, datatype, dest, tag, comm, ierror)

[IN buf] initial address of send buffer (choice)
[IN count] number of elements in send buffer (nonnegative integer)
[IN datatype] datatype of each send buffer element (handle)
[IN dest] rank of destination (integer)
[IN tag] message tag (integer)
[IN comm] communicator (handle)

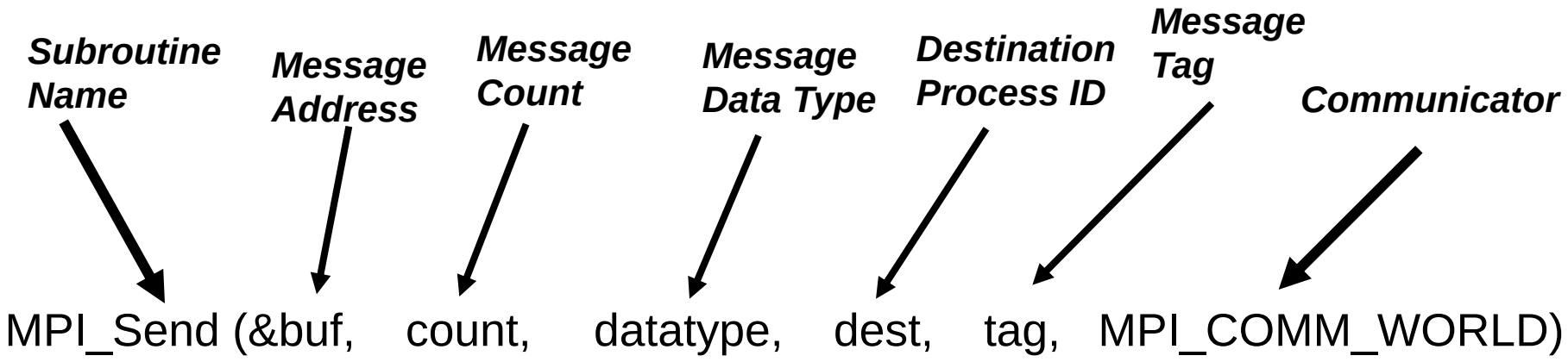
C

MPI_Send (void *Message, int count, MPI_Datatype datatype, int destination, int tag, Mpi_Comm comm);

MPI Point-to-Point Communication Library Calls

(Contd…)

MPI Message Passing : Send C - Language



- *Anatomy of MPI Components in sending a message*
- *Support Heterogeneous computing*
- *Allow messages from non-contiguous,non-uniform memory sections*

MPI Point-to-Point Communication Library Calls

(Contd…)

MPI Message Passing : Receive

Fortran

MPI_RECV (buf, count, datatype, source, tag, comm, status)

[OUT buf]	initial address of receive buffer (choice)
[IN count]	number of elements in receive buffer (integer)
[IN datatype]	datatype of each receive buffer element (handle)
[IN source]	rank of source (integer)
[IN tag]	message tag (integer)
[IN comm]	communicator (handle)
[OUT status]	status object (Status)

C

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

MPI Point-to-Point Communication Library Call

(Contd…)

Sending and Receiving Messages

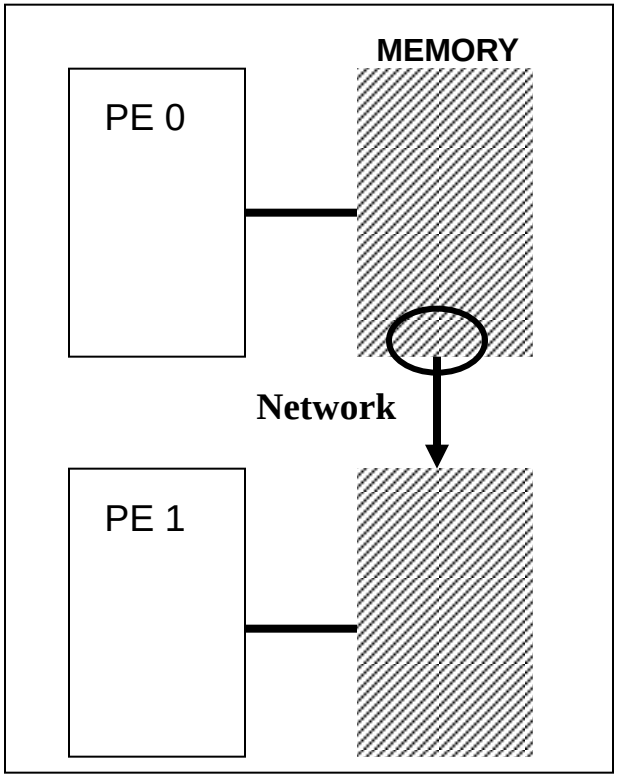
A message consists of several components, outlined below

Sender specifies

- destination PE
- tag
- present address of data on sending PE
- length of message

Receiver specifies

- source PE
- tag
- location for data placement



A message "tag" also known as a message type, is an integer used by the programmer to label different types of message and to restrict reception.

Example Program in MPI

- To write a simple parallel program in which every process with rank greater than 0 sends a message “Hello_World” to a process with rank 0. The processes with rank 0 receives the message and prints out

Example : A Sample MPI program in Fortran

```
program hello
include ‘mpif.h’
integer MyRank, Numprocs, ierror, tag, status (MPI_STATUS_SIZE)
character *12  send_message, recv_message
data  send_message/ ‘Hello_World’/

call MPI_INIT (ierror)
call MPI_COMM_SIZE (MPI_COMM_WORLD, Numprocs, ierror)
call MPI_COMM_RANK (MPI_COMM_WORLD, MyRank, ierror)

 tag=100
```

Example Program in MPI

(Contd…)

Example : A Sample MPI program in Fortran

```
if (MyRank .eq. 0) then
        do i= 1, Numprocs-1
           call MPI_RECV(recv_message, 12, MPI_CHARACTER,
                                i, tag, MPI_COMM_WORLD,status, ierror)
           print *, 'node', MyRank, ':', recv_message

        end  do

else

           call MPI_SEND(send_message, 12, MPI_CHARACTER, 0,
                                tag, MPI_COMM_WORLD, ierror)

endif
call MPI_FINALIZE (ierror)

stop

end
```

Example Program in MPI

(Contd…)

Example : A Sample MPI program in C

```
# include <stdio.h>
# include “mpi.h”
main (int argc, char **argv)
{
   int MyRank, Numprocs, tag, ierror, i;
   MPI_Status status;
   char send_message[12], recv_message[12];
   MPI_Init (&argc, &argv);
   MPI_Comm_size (MPI_COMM_WORLD, &Numprocs);
   MPI_Comm_rank (MPI_COMM_WORLD, &MyRank);
   tag = 100;
   strcpy (send_message, “Hello_World”);
```

Example Program in MPI

(Contd…)

Example : A Sample MPI program in C

```
  if (MyRank==0) {
      for (i=1; i<Numprocs; i++) {
      MPI_Recv (recv_message,12, MPI_CHAR, i, tag,
MPI_COMM_WORLD,&status);
      printf (“node %d : %s \n”, MyRank, recv_message);
      }
  } else
      MPI_Send( send_message, 12, MPI_CHAR,0, tag, MPI_COMM_WORLD);
      MPI_Finalize( );
}
```

MPI Point-to-Point Communication

MPI Routines used in `Hello_World` Program : MPI_Send/MPI_Recv

Synopsis : C

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

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

Synopsis :Fortran

```
MPI_SEND (buf, count, datatype, dest, tag, comm, ierror)

MPI_RECV (buf, count, datatype, source, tag, comm, ierror)

<type> bufffer(*),

integer count, datatype, dest, source, tag, comm, ierror
```

MPI Point-to-Point Communication

(Contd…)

MPI_Send and **MPI_Recv**

- MPI provides for point-to-point communication between pair of processes
- Message selectively is by rank and message tag
- Rank and tag are interpreted relative to the scope of the communication
- The scope is specified by the communicator
- Rank and tag may be wildcarded
- The components of a communicator may not be wildcarded

MPI Blocking Send and Receive

(Contd…)

Point-to-Point Communications

The sending and receiving of messages between pairs of processors.

- **BLOCKING SEND:** returns only after the corresponding RECEIVE operation has been issued and the message has been transferred.

MPI_Send

- **BLOCKING RECEIVE:** returns only after the corresponding SEND has been issued and the message has been received.

MPI_Recv

Message Envelope in MPI : Tag

A *message tag,* also known as a *message type,* is an integer used by the programmer to label different types of message and to restrict message reception.

Example: P and R processes each send a request message to a process Q.

```
Process P:                    Process R:
send(req1,32,Q)               send(req2,32,Q)

Process Q:
while(true) {
recv(received_req, Any_processes, 32);
process received_req;
}
```

with tags

```
Process P:                    Process R:
send(req1,32,Q,tag1)          send(req2,32,Q,tag2)

Process Q:
while(true) {
recv(received_req, Any_processes, 32,Any_Tag,Status);
if(Status.Tag==tag1) process received_req in one way;
if(Status.Tag==tag2)process received_req in other way;
}
```

It is unknown which *send* will be executed first. This is not flexible since all requests are processed the same way.

Message Envelope in MPI : Status

Status is a pointer to a structure which holds various information about the message received.

MPI_Status **Status**

Source process rank and the actual message tag can be found in the two fields

Status. MPI_SOURCE

Status. MPI_TAG

Routine **MPI_Get_count**(&Status, MPI_INT, &C) uses information in Status to determine the actual number of data items of a certain datatype(i.e MPI_INT) and puts the number in C.

Message Envelope in MPI : Communicator

- A communicator is a process group plus a context. A *process* group is a *finite* and *ordered* set of processes.
- The finiteness implies that a group has a finite number *n* of processes, where *n* is called the group size.
- The ordering means that the *n* processes are ranked by integers 0,1,2,,*n-1*
- *A* process is identified by its rank in a communicator (group). The group size and the rank of a process are obtained by calling the two MPI routines.

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

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

Message Envelope in MPI : Intra-Communicators & Context

- Most MPI users only need to use routines for communications within a group (called as intra-communicators in MPI)
- *Context* in MPI are like system-designated supertags that safely separates different communications from adversely interfering with one another
- Each Communicator has a distinct *context*. A message sent in one context can not be received in another *context*.
- MPI is designed so that communications within different communicates are separated and any collective communication is separate from any point-to-point communications, even if they are within the same communicator.
- This communicator concept facilitates the development of libraries
- Managing Communicators :MPI_COMM_WORLD contains set of all process

MPI Point-to-Point Communication: Communication Modes

Synchronous: The send cannot return until the corresponding receive has started. An application buffer is available in the receiver side to hold the arriving message.

Buffered : Buffered send assumes the availability of buffer space which is specified by the MPI_Buffer_attach(buffer,size) which allocates user buffer of *size* bytes.

Standard : The send can be either synchronous or buffered, depending on the implementation.

Ready: The send is certain that the corresponding receive has already started. It does not have to wait as in the synchronous mode.

Source : Reference : [11], [12], [25], [26]

MPI Point-to-Point Communication: Communication Modes

MPI Primitive	Blocking	Nonblocking
Standard Send	MPI_Send	MPI_Isend
Synchronous Send	MPI_Ssend	MPI_Issend
Buffered Send	MPI_Bsend	MPI_Ibsend
Ready Send	MPI_Rsend	MPI_Irsend
Receive	MPI_Recv	MPI_Irecv
Completion Check	MPI_Wait	MPI_Test

Different Send/Receive operations in MPI

MPI Point-to-Point Communication: Message Passing Modes

Process P sends a message contained in variable M to process Q, which receives the message into its variable S.

- Synchronous message passing
- Blocking Send/Receive
- NonBlocking Send/Receive

```
Process P:              Process R:
    M=10;                   S=-100;
L1: send M to Q;        L1: receive S from P;
L2: M=20;               L2: X=S+1;
    goto L1;
```

MPI Point-to-Point Communication: Message Passing Modes

Communication Event	Synchronous	Blocking	Nonblocking
Send start condition	Both send and receive	Send reached	Send reached
Return of send indicates	Message received	Message sent	Message send initiated
Semantics	Clean	In-between	Error-prone
Buffering message	Not needed	Needed	Needed
Status checking	Not needed	Not needed	Needed
Wait overhead	Highest	In-between	Lowest
Overlapping in communications and computations	No	Yes	yes

Key Issues In Message Passing System

Synchronization Speed

- Refers to the time needed for all processors to agree they have finished one step of a problem and are ready to go together to the next step

Synchronization

- Waiting until all processes finish a loop (No one can leave until everyone finishes breakfast)
- Waiting until the first of any of the contributing processes finds a particular answer (We can all leave as soon as one of us finds the keys to the car)
- Assigning a unique task to each processor from a list of tasks (Each one of us will search for the keys in a different room)

Conclusions

- MPI is De-facto standard for message-passing in a box
- Performance was a high-priority in the design
- Rich set of Point-to-Point and Collective functions
- No language binding issues
- Scalability can be achieved as we increase the problem size
- Portability problems do not exist
- Performance tuning can be done

Source : Reference : [11], [12], [25], [26]

MPI Resources

The MPI Standard : http://www.mcs.anl.gov/mpi

References

1. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Programming, Boston, MA : Addison-Wesley

2. Butenhof, David R **(1997),** Programming with POSIX Threads , Boston, MA : Addison Wesley Professional

3. Culler, David E., Jaswinder Pal Singh **(1999),** Parallel Computer Architecture - A Hardware/Software Approach , San Francsico, CA : Morgan Kaufmann

4. Grama Ananth, Anshul Gupts, George Karypis and Vipin Kumar **(2003),** Introduction to Parallel computing, Boston, MA : Addison-Wesley

5. Intel Corporation, **(2003),** Intel Hyper-Threading Technology, Technical User's Guide, Santa Clara CA : Intel Corporation Available at : http://www.intel.com

6. Shameem Akhter, Jason Roberts **(April 2006),** Multi-Core Programming - Increasing Performance through Software Multi-threading , Intel PRESS, Intel Corporation,

7. Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell **(1996),** Pthread Programming O'Reilly and Associates, Newton, MA 02164,

8. James Reinders, Intel Threading Building Blocks – (**2007**) , O'REILLY series

9. Laurence T Yang & Minyi Guo (Editors), (**2006**) *High Performance Computing - Paradigm and Infrastructure* Wiley Series on Parallel and Distributed computing, Albert Y. Zomaya, Series Editor

10. Intel Threading Methodology ; Principles and Practices Version 2.0 copy right **(March 2003),** Intel Corporation

References

11. William Gropp, Ewing Lusk, Rajeev Thakur **(1999),** Using MPI-2, Advanced Features of the Message-Passing Interface, The MIT Press..

12. Pacheco S. Peter, **(1992),** Parallel Programming with MPI, , University of Sanfrancisco, Morgan Kaufman Publishers, Inc., Sanfrancisco, California

13. Kai Hwang, Zhiwei Xu, (**1998**), Scalable Parallel Computing (Technology Architecture Programming), McGraw Hill New York.

14. Michael J. Quinn (**2004**), Parallel Programming in C with MPI and OpenMP McGraw-Hill International Editions, Computer Science Series, McGraw-Hill, Inc. Newyork

15. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Progrmaming, Boston, MA : Addison-Wesley

16. SunSoft. Solaris multithreaded programming guide. SunSoft Press, Mountainview, CA, **(1996),** Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill,

17. Chandra, Rohit, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon, **(2001)**,Parallel Programming in OpenMP San Fracncisco Moraan Kaufmann

18. S.Kieriman, D.Shah, and B.Smaalders **(1995),** Programming with Threads, SunSoft Press, Mountainview, CA. 1995

19. Mattson Tim, **(2002),** Nuts and Bolts of multi-threaded Programming Santa Clara, CA : Intel Corporation, Available at : http://www.intel.com

20. I. Foster **(1995,** Designing and Building Parallel Programs ; Concepts and tools for Parallel Software Engineering, Addison-Wesley (1995)

21. J.Dongarra, I.S. Duff, D. Sorensen, and H.V.Vorst **(1999),** Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools) SIAM, 1999

References

22. OpenMP C and C++ Application Program Interface, Version 1.0". **(1998),** OpenMP Architecture Review Board. October 1998

23. D. A. Lewine. *Posix Programmer's Guide:* **(1991),** Writing Portable Unix Programs with the Posix. 1 Standard. O'Reilly & Associates, 1991

24. Emery D. Berger, Kathryn S McKinley, Robert D Blumofe, Paul R.Wilson, *Hoard : A Scalable Memory Allocator for Multi-threaded Applications* ; The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX). Cambridge, MA, November (**2000)**. Web site URL : http://www.hoard.org/

25. Marc Snir, Steve Otto, Steyen Huss-Lederman, David Walker and Jack Dongarra, (**1998**) *MPI-The Complete Reference: Volume 1, The MPI Core, second edition* [MCMPI-07].

26. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir (**1998**) *MPI-The Complete Reference: Volume 2, The MPI-2 Extensions*

27. A. Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill, **(1996)**

28. OpenMP C and C++ Application Program Interface, Version 2.5 (**May 2005**)", From the OpenMP web site, URL **: http://www.openmp.org/**

29. Stokes, Jon 2002 Introduction to Multithreading, Super-threading and Hyper threading *Ars Technica*, October **(2002)**

30. Andrews Gregory R. 2000, Foundations of Multi-threaded, Parallel and Distributed Programming, Boston MA : Addison – Wesley (**2000)**

31. Deborah T. Marr , Frank Binns, David L. Hill, Glenn Hinton, David A Koufaty, J . Alan Miller, Michael Upton, "Hyperthreading, Technology Architecture and Microarchitecture", Intel (**2000-01)**

Thank You

Any questions ?