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-II)

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

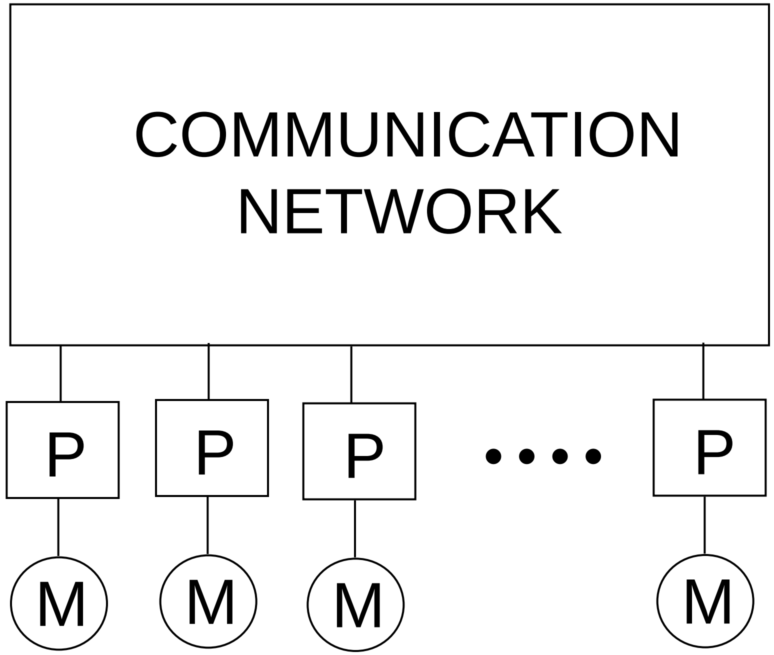
Quick overview of what this Lecture is all about

- Review of MPI Point-to-Point Library Calls
- MPI library calls used in Example program
- MPI Collective Communication Library Calls
- MPI Collective Communication and Computations Library Calls

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

Message Passing Architecture Model

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



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

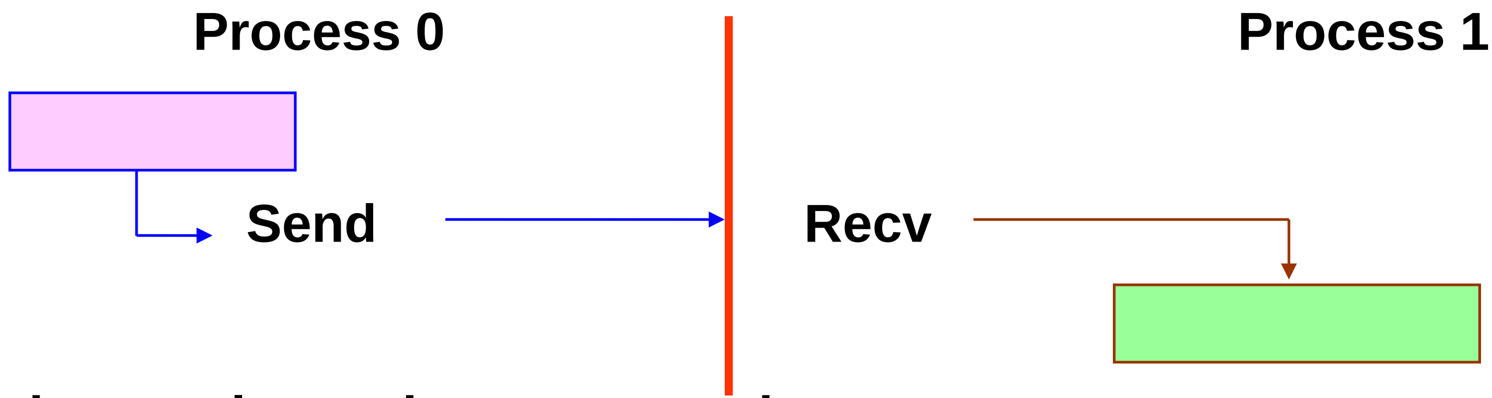
MPI Basics

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.

MPI Send and Receive

Blocking Sending and Receiving messages



Fundamental questions answered

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

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

MPI Point-to-Point Communication

(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

(Contd…)

<u>MPI Message Passing</u> : 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

(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

MPI Basic Datatypes

MPI Basic Datatypes - Fortran

MPI Datatype	Fortran Datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

MPI Basic Datatypes

(Contd…)

MPI Basic Datatypes - C

MPI Datatype	C datatype
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long double
MPI_BYTE	
MPI_PACKED	

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

Is MPI Large or Small?

(Contd…)

The MPI Message Passing Interface Small or Large

MPI can be small.

One can begin programming with 6 MPI function calls

MPI_INIT	*Initializes MPI*
MPI_COMM_SIZE	*Determines number of processors*
MPI_COMM_RANK	*Determines the label of the calling process*
MPI_SEND	*Sends a message*
MPI_RECV	*Receives a message*
MPI_FINALIZE	*Terminates MPI*

MPI can be large

One can utilize any of 125 functions in MPI.

MPI Point-to-Point Communication Library Calls

(Contd…)

MPI Message Passing : Send **C - Language**

Subroutine Name	***Message Address***	***Message Count***	***Message Data Type***	***Destination Process ID***	***Message Tag***	***Communicator***
MPI_Send (	&buf,	count,	datatype,	dest,	tag,	MPI_COMM_WORLD)

MPI_Send (&buf, count, datatype, dest, tag, MPI_COMM_WORLD)

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

MPI Collective Communications

Collective Communications

- ❖ The sending and/or receiving of messages to/from groups of processors.
- ❖ A collective communication implies that all processors need participate in a global communication operation.
- ❖ Involves coordinated communication within a group of processes
- ❖ No message tags used
- ❖ All collective routines block until they are locally complete

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

MPI Collective Communications

(Contd…)

- Communications involving a group of processes.
- Called by all processes in a communicator.
- Examples:
  - Barrier synchronization.
  - Broadcast, scatter, gather.
  - Global sum, global maximum, etc.
- Two broad classes :
  - Data movement routines
  - Global computation routines

MPI Collective Communications

(Contd…)

Characteristics of Collective Communication

- Collective action over a communicator
- All processes must communicate
- Synchronization may or may not occur
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size

MPI Collective Communications

(Contd…)

Collective Communications

Communication is coordinated among a group of processes

- ❖ Group can be constructed "**by hand**" with MPI group-manipulation routines or by using MPI topology-definition routines
- ❖ Different communicators are used instead
- ❖ No non-blocking collective operations

MPI Collective Communications

(Contd…)

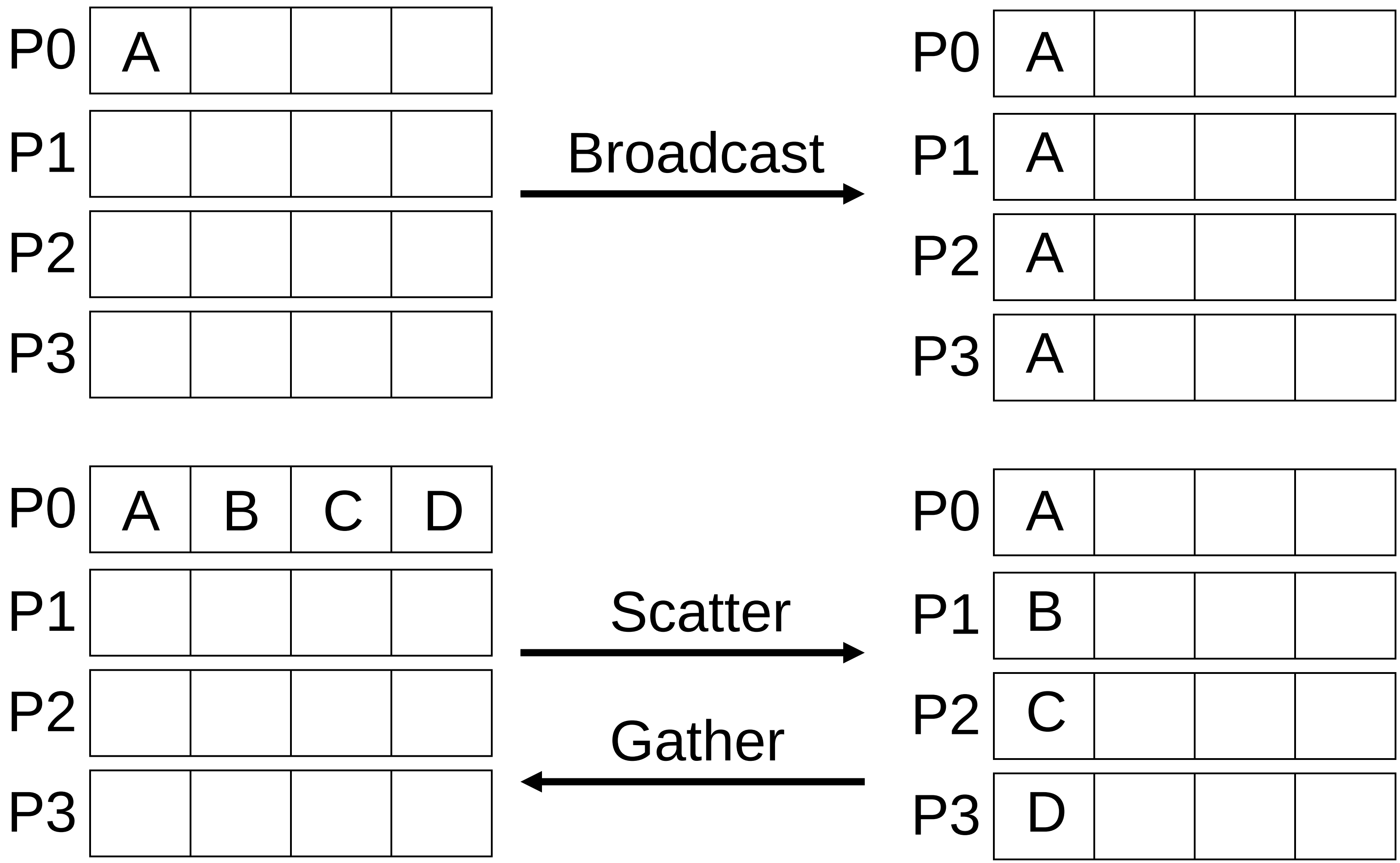
Collective Communication routines

Three classes of collective operations:

- Synchronization
- Data movement
- Collective computation

MPI Collective Communications

(Contd…)



Representation of collective data movement in MPI

MPI Collective Communications :Broadcast

(Contd…)

A broadcast sends data from one process to all other processes. The content of the message to all processes (including itself) in the communicator . The contents of the message is identified by the triple (Address, Count, Datatype).

For the root processes, this triple specifies both the *send* and *receive* buffer. For other processes, this triple specifies the *receive* buffer

❖ **C**:

```
int MPI_Bcast ( void *buffer, int count, MPI_Datatype datatype,
                int root, MPI_Comm comm);
```

❖ **Fortran**:

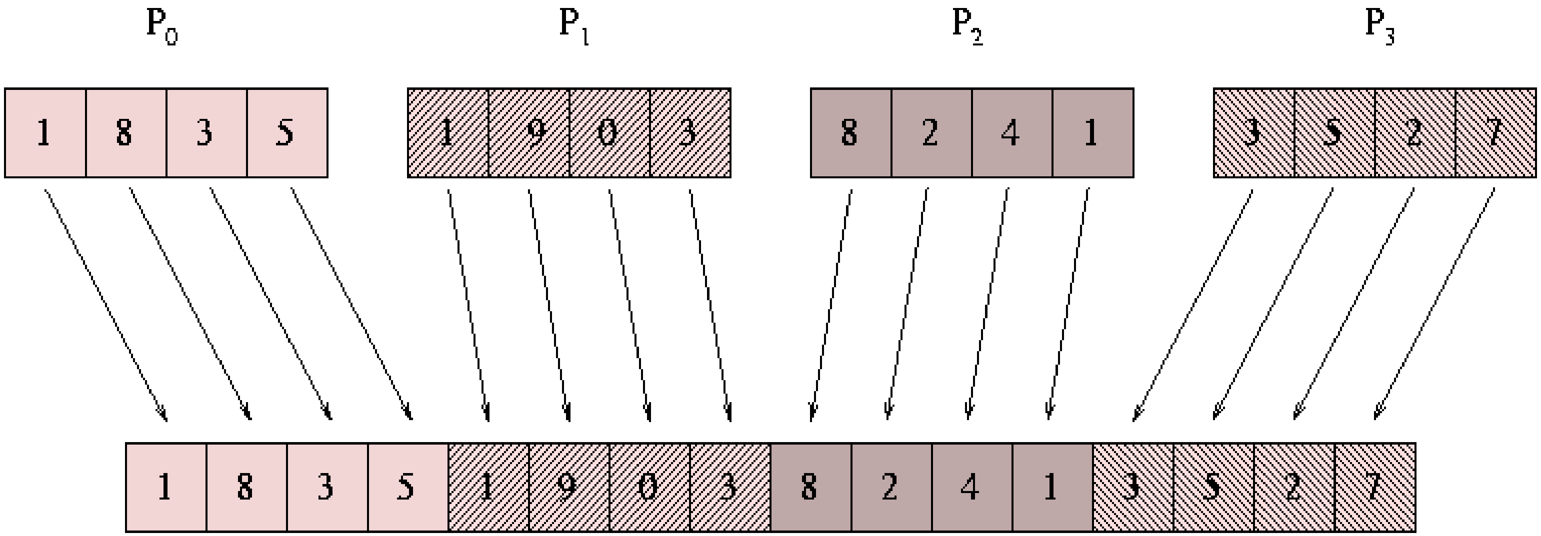
```
MPI_Bcast (buffer, count, datatype, root,  comm, ierror)
<type> buffer(*)
integer count, datatype, root, comm, ierror
```

MPI Collective Communications :Gather

(Contd…)

Get the data that resides on all processes and accumulate onto a single processes.

The root process receives a personalized message from each of the n processes (including itself)



Gather an integer array of size of 4 from each process

MPI Collective Communications :Gather

(Contd…)

Each process in *comm* send the contents of *send-buffer* to the process with rank *root*. The process with rank *root* concatenates the received data in the process rank order in *recv-buffer.* The receive arguments are significant only on process rank *root*. The argument *recv_count* indicates the number of items received from each process –not the total number received.

For the root process, it is identified by the triple (Recv Address, RecvCount, RecvDatatype).

C:

```
int MPI_Gather( void *send_buffer, int send_count, MPI_Datatype
        send_type, void *recv_buffer, int recv_count,
    MPI_Datatype recv_type, int root, MPI_Comm comm);
```

MPI Collective Communications :Gatherv

Gatterv

Each process in *comm* send the contents of different size of *send-buffer* to the process with rank *root*. The process with rank *root* concatenates the received data in the process rank order in *recv-buffer.* The receive arguments are significant only on process rank *root*. The argument *recv_counts[ ]* indicates the number of items received from each process – not the *total number* received. Extends the functionality of MPI_Gather by allowing different type signatures. Each process has different sizes of *send-buffer. Displacements[ ]* indicates displacement vector

For the root process, it is identified by the triple (Recv Address, RecvCount, RecvDatatype).

C:

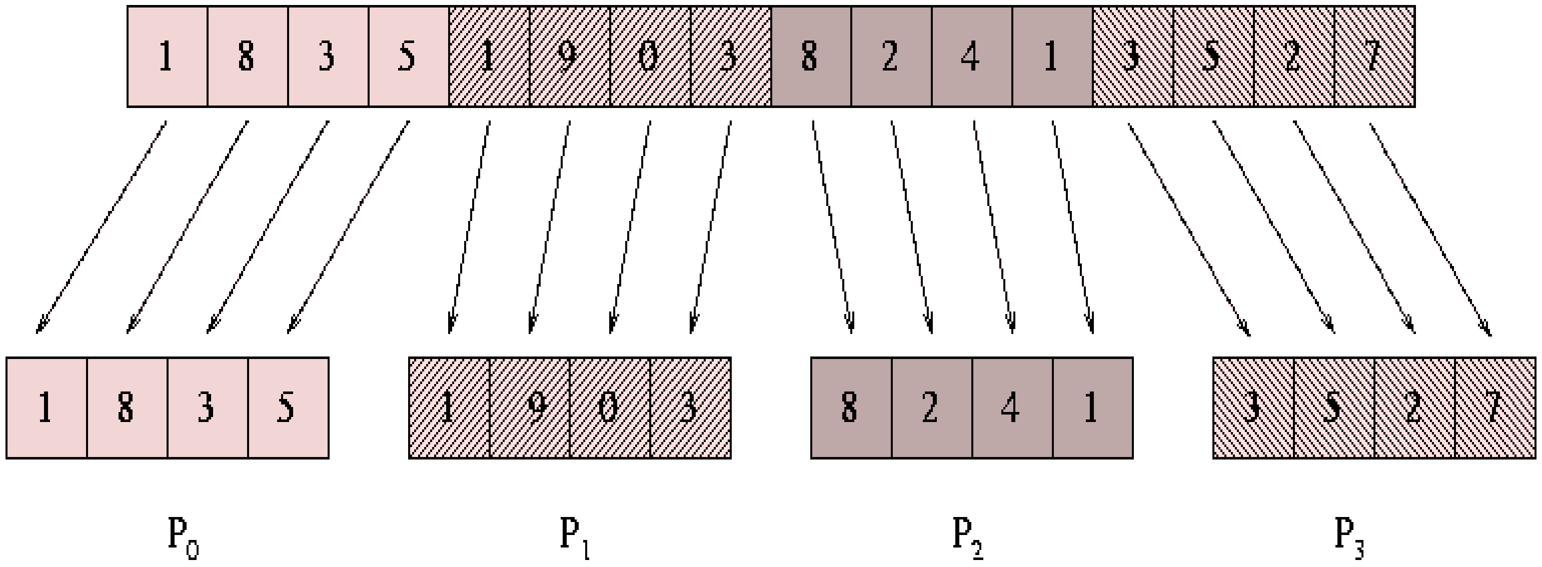
```
int MPI_Gatherv( void *send_buffer, int send_count, MPI_Datatype
                 send_type, void *recv_buffer, int recv_counts[ ],
                 int displacements[  ],  MPI_Datatype recv_type,
                 int root, MPI_Comm comm);
```

MPI Collective Communications :Scatter

(Contd…)

Distribute a set of data from one process to all other processes.

A scatter performs just the opposite operation of a gather.



Scatter an integer array of size 16 on 4 processors

MPI Collective Communications :Scatter

(Contd…)

Scatter

The process with rank *root* distributes the contents of *send-buffer* among the processes. The contents of *send-buffer* are split into *p* segments each consisting of *send_count* elements. The first segment goes to process 0, the second to process 1, etc… The send arguments are significant only on process *root*.

- **C**:

```
int  MPI_Scatter( void *send_buffer, int send_count, MPI_Datatype
                  send_type, void *recv_buffer, int recv_count,
               MPI_Datatype recv_type, int root,MPI_Comm comm);
```

MPI Collective Communications :Scatterv

(Contd…)

Scatterv

The process with rank *root* distributes the contents of different *send-buffer* among the processes. The contents of *send-buffer* are split into different *p* segments each consisting of different array of *send_counts*[ ] elements. The first segment goes to process 0, the second to process 1, etc… The send arguments are significant only on process *root*. Extends the functionality of MPI_Scatter by allowing different type signatures. Each process has different sizes of *recv-buffer. Displacements[ ]* indicates displacement vector

❖ **C**:

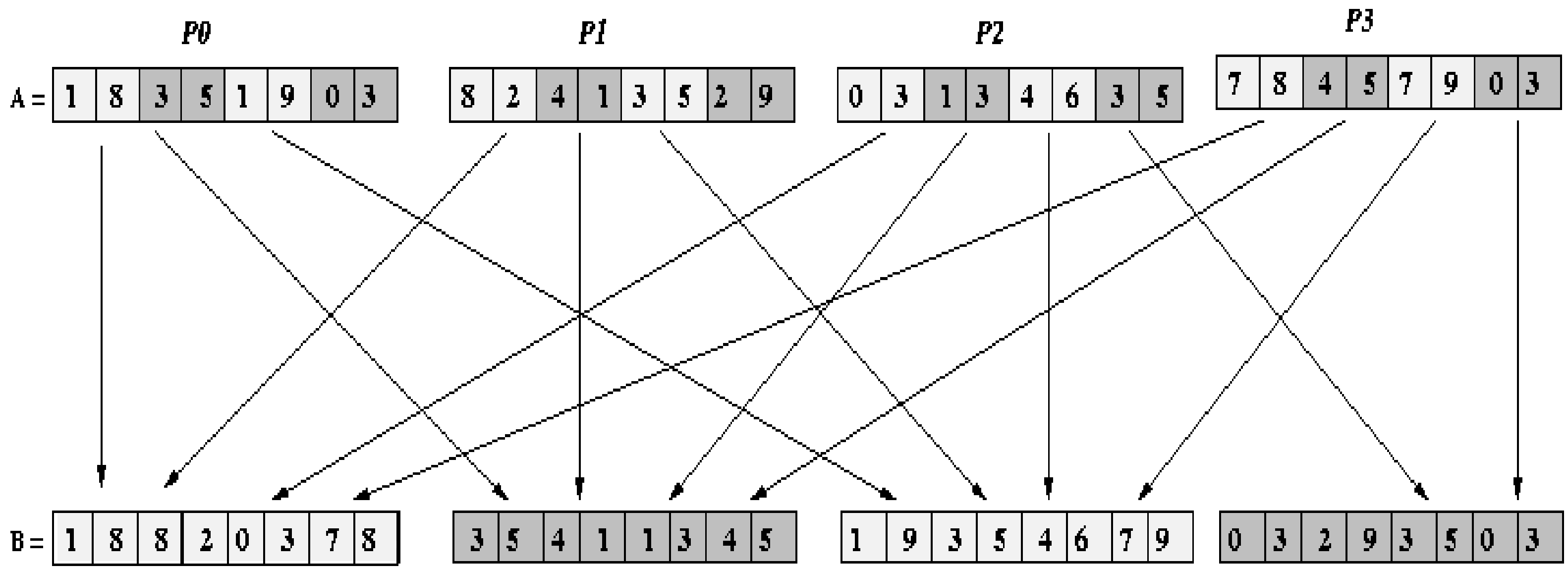
```
int  MPI_Scatterv( void *send_buffer, int send_counts[  ],
                   int displacements[  ], MPI_Datatype send_type,
                   void *recv_buffer, int recv_count,
                    MPI_Datatype recv_type, int root, MPI_Comm comm);
```

MPI Collective Communications

(Contd…)

All-to-All

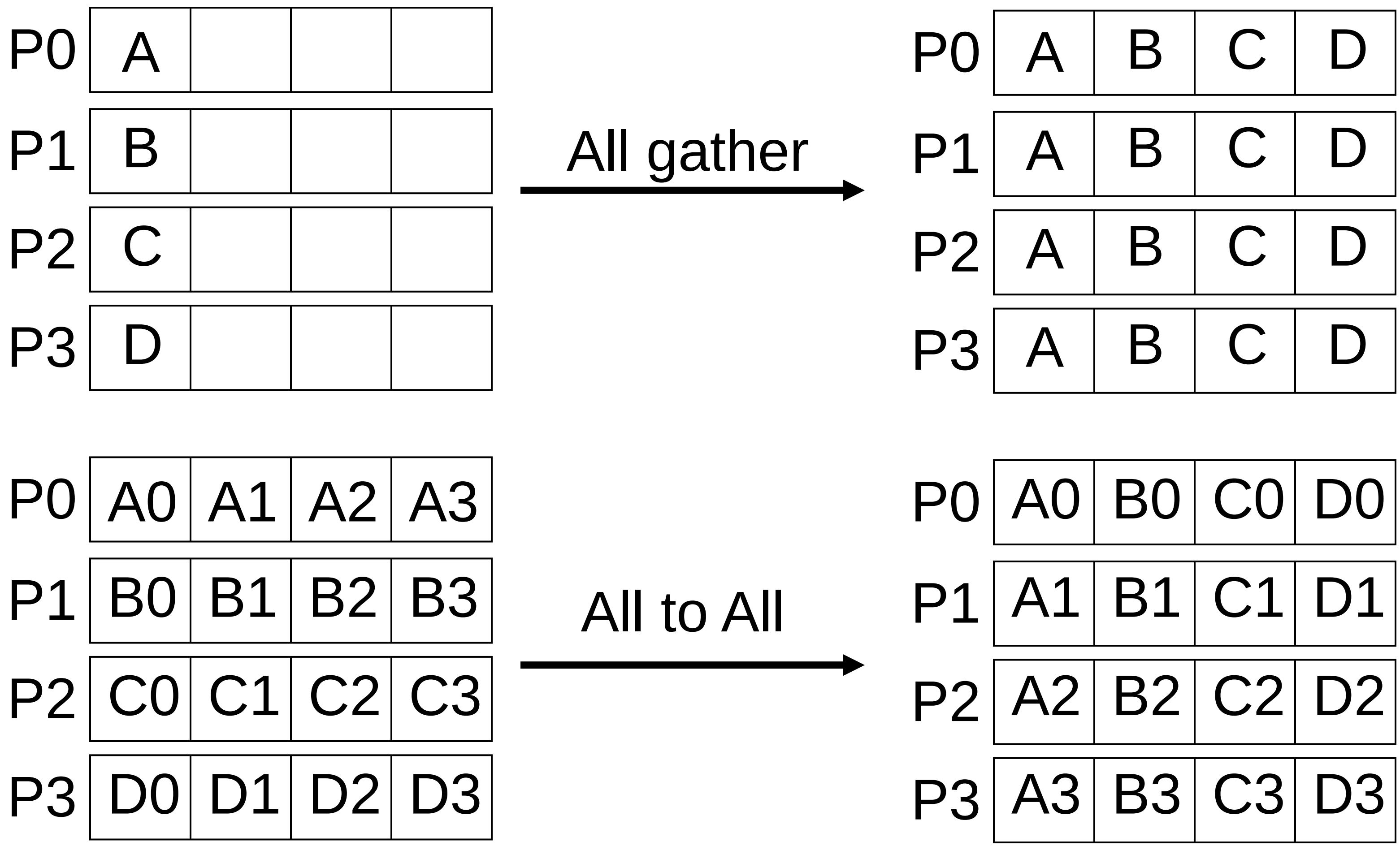
Performs a scatter and gather from all four process to all other four processes. Every process accumulates the final values



All-to-All operation for an integer array of size 8 on 4 processors

MPI Collective Communications

(Contd…)



Representation of collective data movement in MPI

MPI_Alltoall

```
MPI_Alltoall (void* sendAddress, int SendCount,
        MPI_Datatype  SendDatatype, void* RecvAddress,
        int  RecvCount, MPI_Datatype RecvDatatype, MPI_Comm Comm);
```

Every process sends a personalized message to each of the *n* processes, including itself. These *n* messages are originally stored in rank order in its send buffer.Looking at the communication from another way, every process receive a message from each of the *n* processes

These messages "*n*" messages are concatenated in rank order, and stored in the receive buffer. These "*n*" messages are concatenated in rank order, and stored in the receive buffer. Note that a total exchange is equivalent to *n* gathers, each by a different process.

MPI_Alltoallv

```
MPI_Alltoall (void* sendAddress,
        int SendCounts[  ],
         int send _displacements[  ],
         MPI_Datatype  SendDatatype, void* RecvAddress,
         int  RecvCount[  ],
         int recv_displacements[  ],
         MPI_Datatype RecvDatatype, MPI_Comm Comm);
```

MPI Collective Communications

Type	Routine	Functionality
Data Movement	MPI_Bcast	One-to-all, Identical Message
	MPI_Gather	All-to-One, Personalized messages
	MPI_Gatherv	A generalization of MPI_Gather
	MPI_Allgather	A generalization of MPI_Gather
	MPI_Allgatherv	A generalization of MPI_Allgather
	MPI_Scatter	One-to-all Personalized messages
	MPI_Scatterv	A generalization of MPI_Scatter
	MPI_Alltoall	All-to-All, personalized message
	MPI_Scatterv	A generalization of MPI_Alltoall

Characteristics of Collective Communications

- Collective action over a communicator
- All processes must communicate
- Synchronization may or may not occur
- All collective operations are blocking.
- No tags.
- Receive buffers must be exactly the right size

MPI Collective Communications

(Contd…)

Collective Communications

Communication is coordinated among a group of processes

- ❖ Group can be constructed "**by hand**" with MPI group-manipulation routines or by using MPI topology-definition routines
- ❖ Different communicators are used instead
- ❖ No non-blocking collective operations

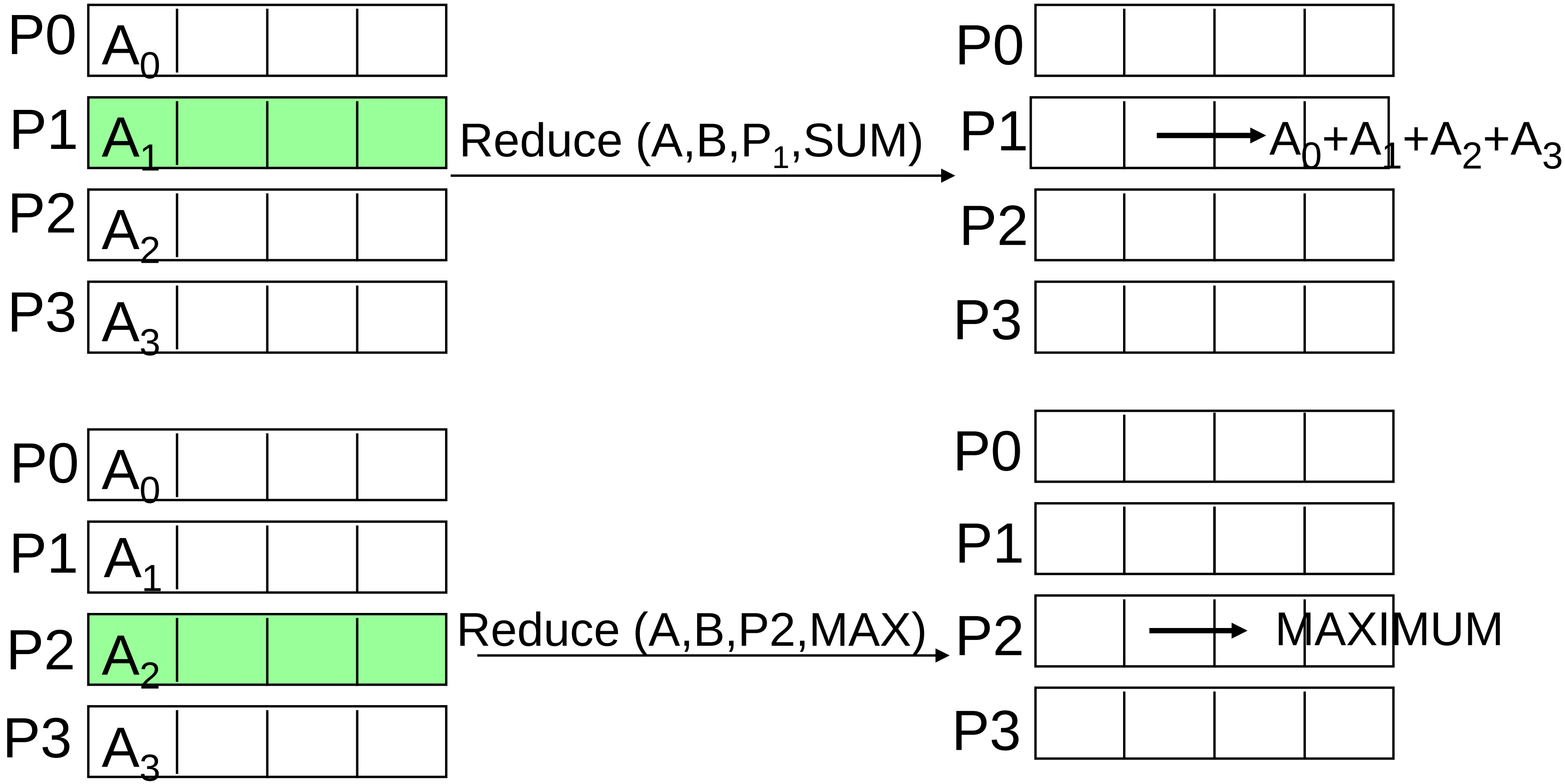
MPI Collective Communications and Computations

Type	Routine	Functionality
Aggregation	MPI_Reduce MPI_Allreduce MPI_Reduce_scatter MPI_Scan	All-to-one reduction, All-to-One, A generalization of MPI_Reduce A generalization of MPI_Reduce All-to-all parallel prefix

Synchronization	MPI_Barrier	Barrier Synchronization

MPI Collective Communications and Computations

(Contd…)



Representation of collective data movement in MPI

MPI Collective Communications and Computations

(Contd…)

Fortran

```
MPI_Reduce (sendbuf, recvbuf, count, datatype, op,
            root, comm, ierror)

<type> sendbuf (*), recvbuf (*)
integer count, datatype, op, root, comm,ierror
```

C

```
int  MPI_Reduce (void* operand, void* result, int count,
                 MPI_Datatype datatype, MPI_Op op,
                 int root, MPI_Comm comm) ;
```

C

```
int  MPI_Allreduce(void* operand, void* result, int count,
                   MPI_Datatype datatype, MPI_Op op,
                   MPI_Comm comm) ;
```

MPI Collective Communications and Computations

(Contd…)

Barrier

A barrier insures that all processes reach a specified location within the code before continuing.

All processes in the communicator "*Comm*" synchronize with one another; I.e., they wait until processes execute their respective MPI_Barrier function.

- C:

  ```
  int MPI_Barrier (MPI_Comm comm);
  ```

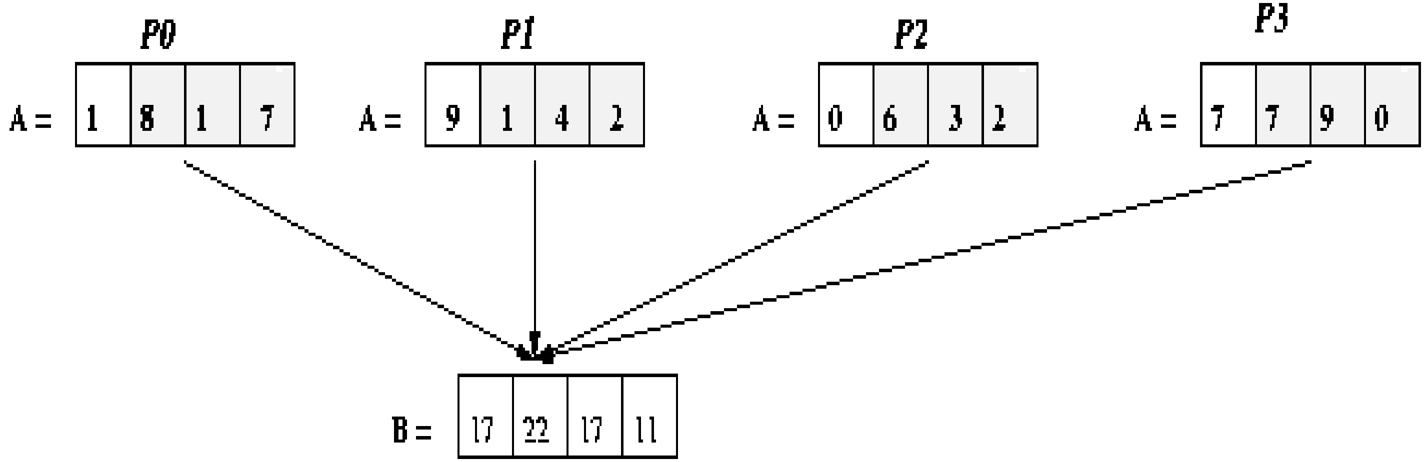
- Fortran:

  ```
  MPI_barrier (comm, ierror)
  integer comm, ierror
  ```

MPI Collective Communications and Computations

Reduction

A reduction compares or computes using a set of data stored on all processes and saves the final result on one specified process.



Global Reduction (sum) of an integer array of size 4 on each process and accumulate the same on process P1

MPI Collective Communications and Computations

(Contd…)

Global Reduction Operations

- Used to compute a result involving data distributed over a group of processes.
- MPI provides two types of *aggregation* : reduction and *Scan*.
- Examples:
  - Global sum or product
  - Global maximum or minimum
  - Global user-defined operation

MPI Collective Communications and Computations

(Contd…)

Collective Computation Operations

MPI_Name	Operation
MPI_LAND	Logical and
MPI_LOR	Logical or
MPI_LXOR	Logical exclusive or (xor)
MPI_BAND	Bitwise AND
MPI_BOR	Bitwise OR
MPI_BXOR	Bitwise exclusive OR

MPI Collective Communications and Computations

(Contd…)

Collective Computation Operation

MPI Name	Operation
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_PROD	Product
MPI_SUM	Sum
MPI_MAXLOC	Maximum and location
MPI_MAXLOC	Maximum and location

MPI Collective Communications and Computations

Allgather	Allgatherv	Allreduce
Alltoall	Alltoallv	Bcast
Gather	Gatherv	Reduce
Reduce Scatter	Scan	Scatter
Scatterv		

- All versions deliver results to all participating processes
- *V* -version allow the chunks to have different non-uniform data sizes (Scatterv, Allgatherv, Gatherv)
- All reduce, Reduce , ReduceScatter, and Scan take both built-in and user-defined combination functions

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

Features of MPI

(Contd…)

- **Positives**
  - MPI is De-facto standard for message-passing in a box
  - Performance was a high-priority in the design
  - Simplified sending message
  - Rich set of collective functions
  - Do not require any daemon to start application
  - No language binding issues

Features of MPI

(Contd…)

- **Positives**
  - Best scaling seen in practice
  - Simple memory model
  - Simple to understand conceptually
  - Can send messages using any kind of data
  - Not limited to “shared -data”

Conclusions

- MPI is De-facto standard for message-passing in a box
- Rich set of Point-to-Point and Collective functions
- No language binding issues
- Scalability can be achieved as we increase the problem size
- Performance tuning can be done

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

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 ?