Introduction to Parallel Programming with MPI

Go to Menu

Outline

- Introduction
- Message Passing Interface (MPI)
- Point to Point Communications
- Collective Communications
- Derived Datatypes
- Resources for Users

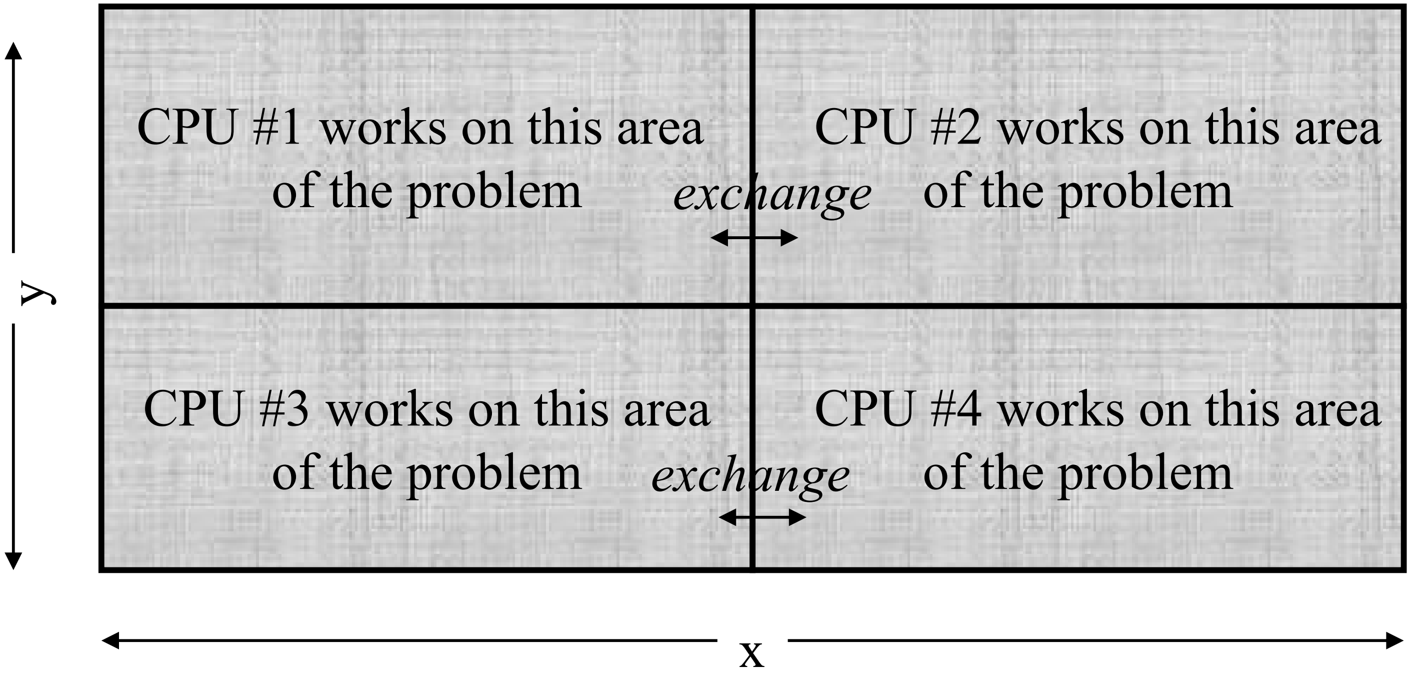
Outline: Introduction

- What is Parallel Computing?
- Why go parallel?
- Types of Parallelism - Two Extremes.
- Parallel Computer Architectures
- Parallel "Architectures"
- Parallel Programming Models

What is Parallel Computing?

- *Parallel computing* - the use of multiple computers, processors or cores working together on a common task.
  - Each processor works on a section of the problem
  - Processors are allowed to exchange information (data in local memory) with other processors

Grid of Problem to be solved



CPU #1 works on this area of the problem

CPU #2 works on this area of the problem

CPU #3 works on this area of the problem

CPU #4 works on this area of the problem

Why go parallel?

- Limits of single CPU computing
  - Available memory
  - Performance
- Parallel computing allows:
  - Solve problems that don’t fit on a single CPU
  - Solve problems that can’t be solved in a reasonable time
- We can run…
  - Larger problems
  - Faster
  - More cases

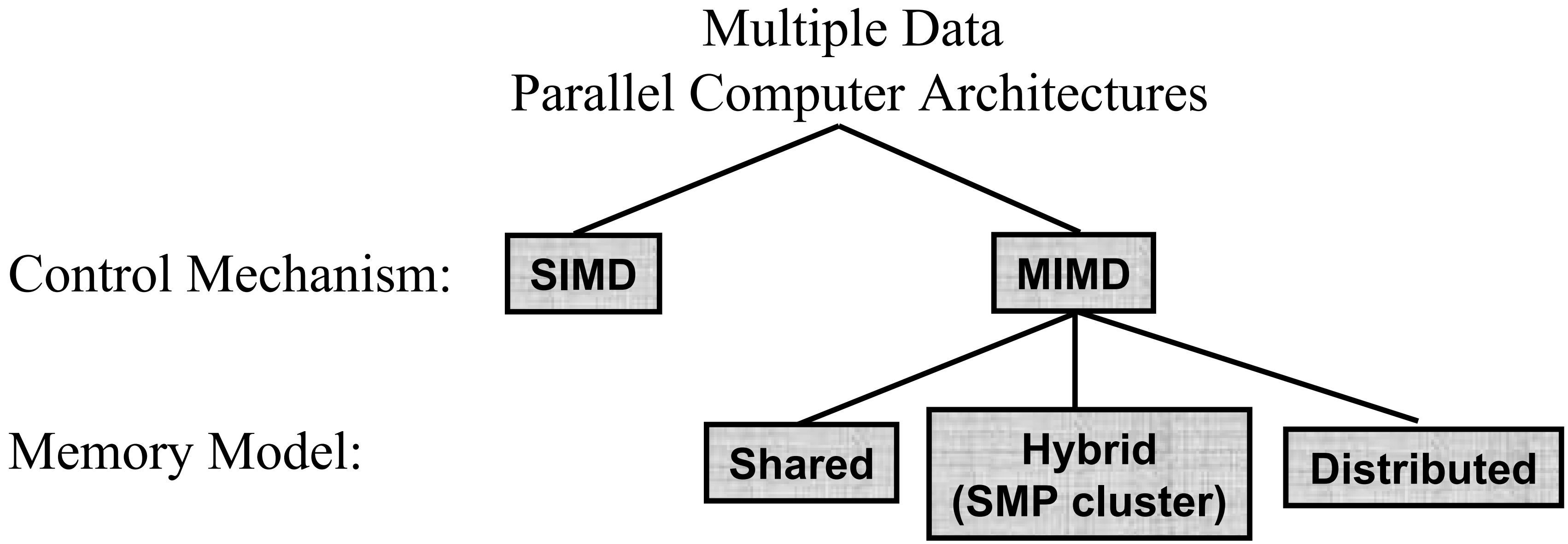
Types of Parallelism - Two Extremes.

- Data parallel
  - Each processor performs the same task on different data
  - Example - grid problems
- Task parallel
  - Each processor performs a different task
  - Example - signal processing
- Most real applications fall somewhere on the continuum between these two extremes and involve more than one type of problem mapping

Go to Menu

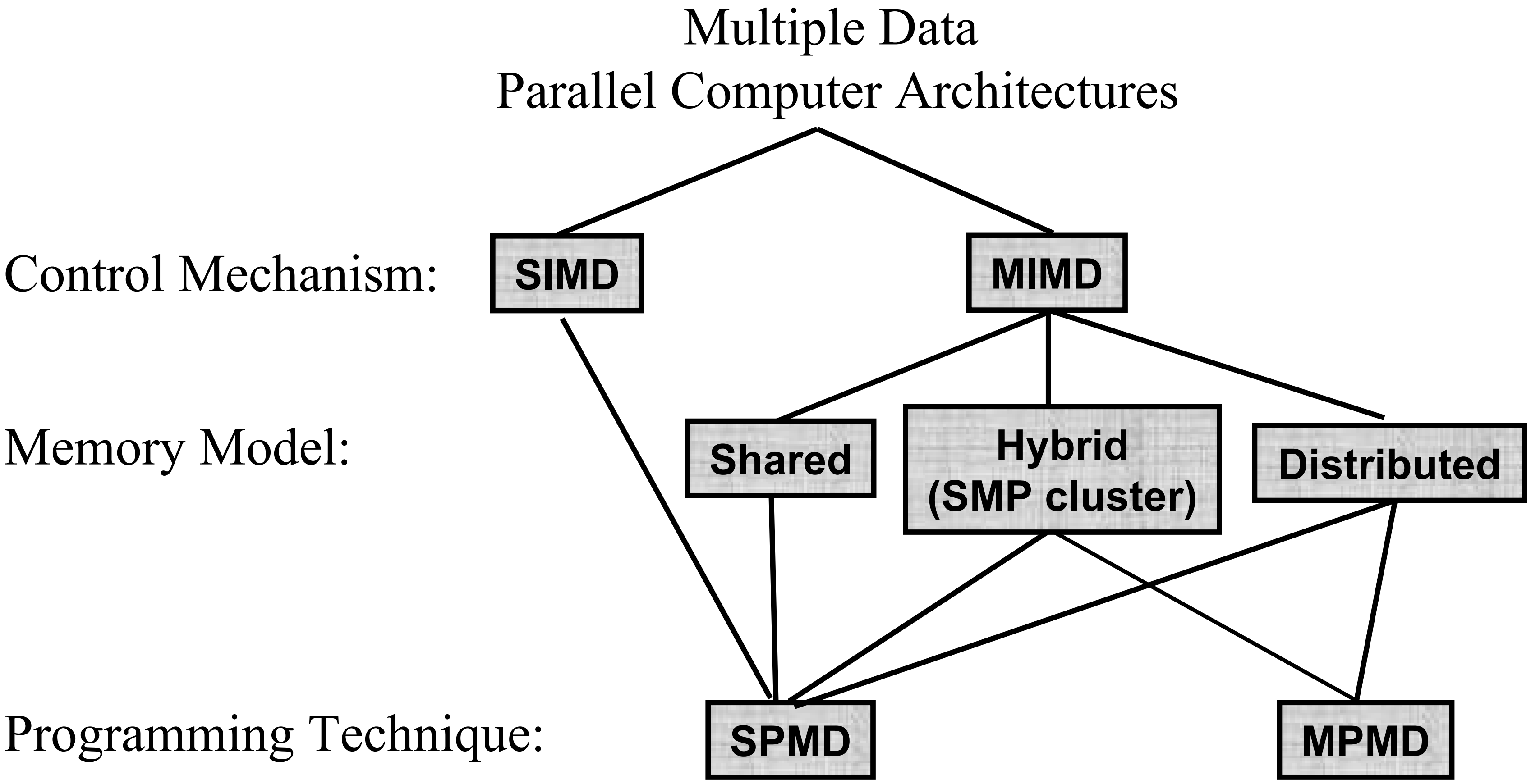
Parallel Computer Architectures

- SIMD - Single Instruction/Multiple Data
- MIMD - Multiple Instructions/Multiple Data



Parallel "Architectures"

- SPMD - Single Process/Multiple Data, or Single Program/Multiple Data
- MPMD - Multiple Process/Multiple Data, or Multiple Program/Multiple Data



Parallel Programming Models

- There are several parallel programming models in common use:
  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Hybrid
  - Other…
- Parallel programming models exist as an abstraction above hardware and memory architectures.
- In hybrid model any two or more parallel programming models are combined.
- Jaguar is an example of a common hybrid model which is the combination of the message passing model (MPI) with the shared memory model (OpenMP) which is also a combination of threads model.
- This document discusses the message passing (MPI) parallel programming model.

Go to Menu

Outline: Message Passing Interface (MPI)

- What is MPI?
- MPI in Summary
- First MPI Program: Hello World!
- Four Major Benefits of MPI
- MPI Message Components
- Typical Message Passing Communication Components
- Message Passing Programming Concept
- Essentials of MPI Programs
- MPI Function Format
- MPI Communicators and Groups
- MPI Environment Management Routines
- MPI Process Identifiers
- MPI_COMM_WORLD communicator
- MPI_COMM_WORLD example
- Example: “Hello From …”

What is MPI?

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded in a source code. The programmer is responsible for determining all the parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum, which had over 40 participating organizations, including vendors, researchers, software library developers, and users was formed with the primary goal of establishing a portable, efficient, and flexible standard specification for message passing implementations.
- As a result of this forum Part 1 of the Message Passing Interface (MPI) was released in 1994. Part 2 (MPI-2) was released in 1996.
- MPI is now the "de facto" industry standard for message passing implementations, replacing virtually all other implementations used for production work.

Go to Menu

MPI in Summary

- MPI is dominant parallel programming approach in the USA.
- By itself, MPI is NOT a library - but rather the specification of what such a library should be.
- MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for message passing implementations.
- As such, MPI is the first standardized vendor independent, message passing specification.
  - the syntax of MPI is standardized!
  - the functional behavior of MPI calls is standardized!
- Popular implementations: MPICH, OpenMPI (not to be confused with openMP), LAM, Cray MPT…
- MPI specifications (as well as MPICH implementation) are available on the web at http://www.mcs.anl.gov/research/projects/mpi/index.htm

First MPI Program: Hello World!

C :

```c
#include <stdio.h>
#include “mpi.h”
int main(int argc, char** argv)  {
   MPI_Init(&argc, &argv);
   printf(“Hello world \n”);
   MPI_Finalize();                }
```

Fortran :

```fortran
program Hello_World
include ‘mpif.h’
integer ierror
call MPI_INIT(ierror)
print *, ‘Hello World’
call MPI_FINALIZE(ierror)
end
```

C++ :

```cpp
#include "mpi.h"
#include <iostream>
int main(int argc, char** argv)  {
   MPI::Init(argc, argv);
   cout <<"Hello world \n";
   MPI::Finalize();               }
```

Four Major Benefits of MPI

Before the standard MPI specification appeared, message passing development was based on the tradeoffs between portability, performance and functionality.

Standardization - MPI is the only message passing specification which can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.

Portability - Usually 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.

Functionality - Over 450 routines are defined in MPI-1 and MPI-2.

Availability - A variety of implementations are available on both vendor and public domains.

MPI Message Components

- Envelope:
  - sending processor (`processor_id`)
  - source location (`group_id, tag`)
  - receiving processor (`processor_id`)
  - destination location (`group_id, tag`)

- Data (letter) :
  - data type (`integer, float, complex, char`...)
  - data length (`buffer size, count, strides`)

Typical Message Passing Communication Components

- Environment Identifiers

  `processor_id`, `group_id`, `initialization`

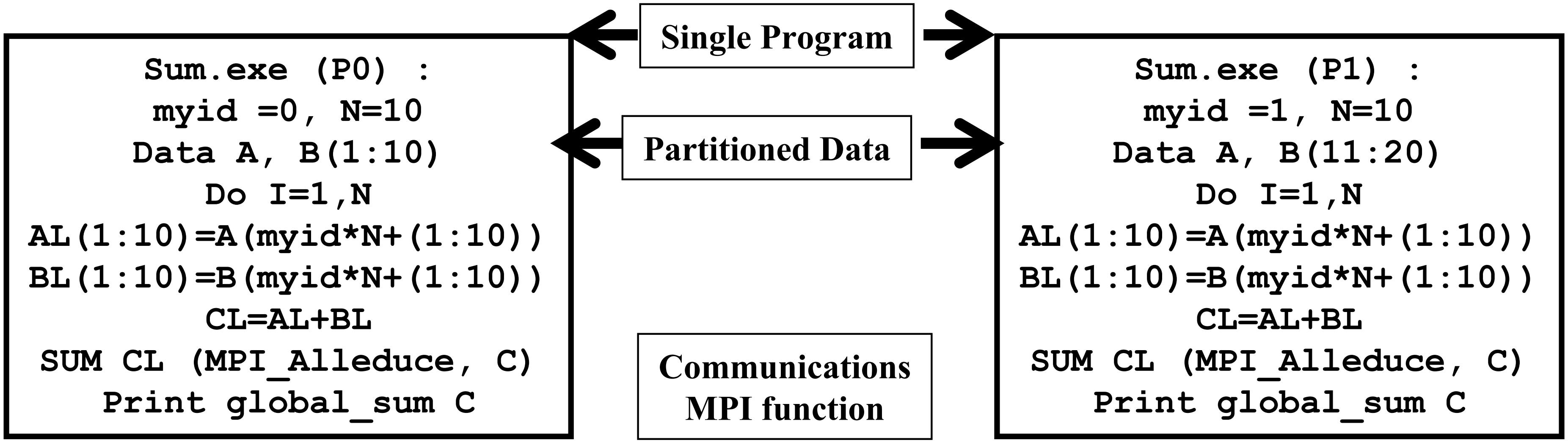
- Point to Point Communications
  - blocking operations
  - non-blocking operations

- Collective Communications
  - barriers
  - broadcast
  - reduction operations

Go to Menu

Message Passing Programming Concept

- Originally was used in distributed memory computing environment
- Can also be used in shared memory environment
- Since memory is local, any data stored in the memory of a remote processor must be explicitly requested by a programmer. In MPI implementations for systems which are collections of SM nodes the support for message passing on a node is included.
- Usually each processor executes the SAME program using partitioned data set (SPMD)
- Written in sequential language (FORTRAN, C, C++) plus MPI functions



```
Sum.exe (P0) :
myid =0, N=10
Data A, B(1:10)
Do I=1,N
AL(1:10)=A(myid*N+(1:10))
BL(1:10)=B(myid*N+(1:10))
CL=AL+BL
SUM CL (MPI_Alleduce, C)
Print global_sum C
```

```
Sum.exe (P1) :
myid =1, N=10
Data A, B(11:20)
Do I=1,N
AL(1:10)=A(myid*N+(1:10))
BL(1:10)=B(myid*N+(1:10))
CL=AL+BL
SUM CL (MPI_Alleduce, C)
Print global_sum C
```

Essentials of MPI Programs

Header files:

C:	`#include “mpi.h”`
Fortran:	`include ‘mpif.h’`
C++:	`#include “mpi.h”`

Initializing MPI:

C:	`int MPI_Init(int argc, char**argv)`
Fortran:	`call MPI_INIT(IERROR)`
C++:	`void MPI::Init(int& argc, char**& argv);`

Exiting MPI:

C:	`int MPI_Finalize(void)`
Fortran:	`call MPI_FINALIZE(IERROR)`
C++:	`void MPI::Finalize();`

MPI Function Format

MPI format:

C: `err = MPI_Xxxx(parameter,…)`
Fortran: `call MPI_XXXX(parameter,…, ierror)`
C++: `void MPI::Xxxx(parameter,…)`

Errors:

C: Returned as `err`. `MPI_SUCCESS` if successful
Fortran: Returned as `ierror` parameter. `MPI_SUCCESS` if successful

C++ functions do not return error codes. If the default error handler has been set to MPI::ERRORS_THROW_EXCEPTIONS, the C++ exception mechanism is used to signal an error by throwing an MPI::Exception object.

It should be noted that the default error handler (i.e., MPI::ERRORS_ARE_FATAL) on a given type has not changed. User error handlers are also permitted. MPI::ERRORS_RETURN simply returns control to the calling function; there is no provision for the user to retrieve the error code.

User callback functions that return integer error codes should not throw exceptions; the returned error will be handled by the MPI implementation by invoking the appropriate error handler.

MPI Communicators and Groups

- MPI uses objects called *communicators* and *groups* to define which collection of processes may communicate with each other.
- All MPI communication calls require a *communicator argument* and MPI processes can only communicate if they share a communicator.
- Every communicator contains a *group of tasks* with a *system supplied identifier* (for message context) as an extra match field.
- The *base group* is the group that contains all processes, which are associated with the *`MPI_COMM_WORLD`* – predefined communicator.
- Communicators are used to create independent "message universe"
- *`MPI_Init`* initializes all tasks with *`MPI_COMM_WORLD`*
- Communicators are particularly important for user supplied libraries

MPI Environment Management Routines

- MPI environment management routines are used for an assortment of purposes, such as initializing and terminating the MPI environment, querying the environment and identity, etc. Most of the commonly used ones are:

```
MPI_Init
MPI_Comm_size
MPI_Comm_rank
MPI_Abort
MPI_Get_processor_name
MPI_Initialized
MPI_Wtime
MPI_Wtick
MPI_Finalize
```

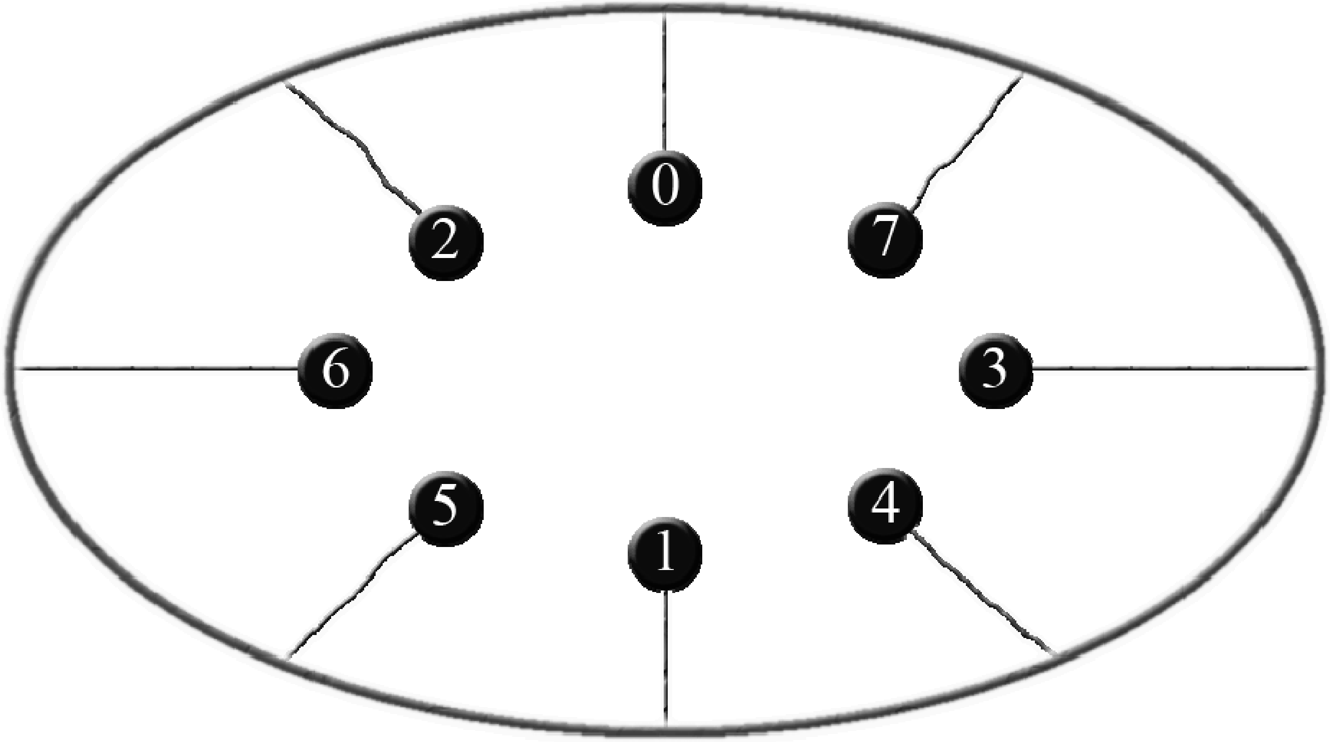
Go to Menu

MPI Process Identifiers

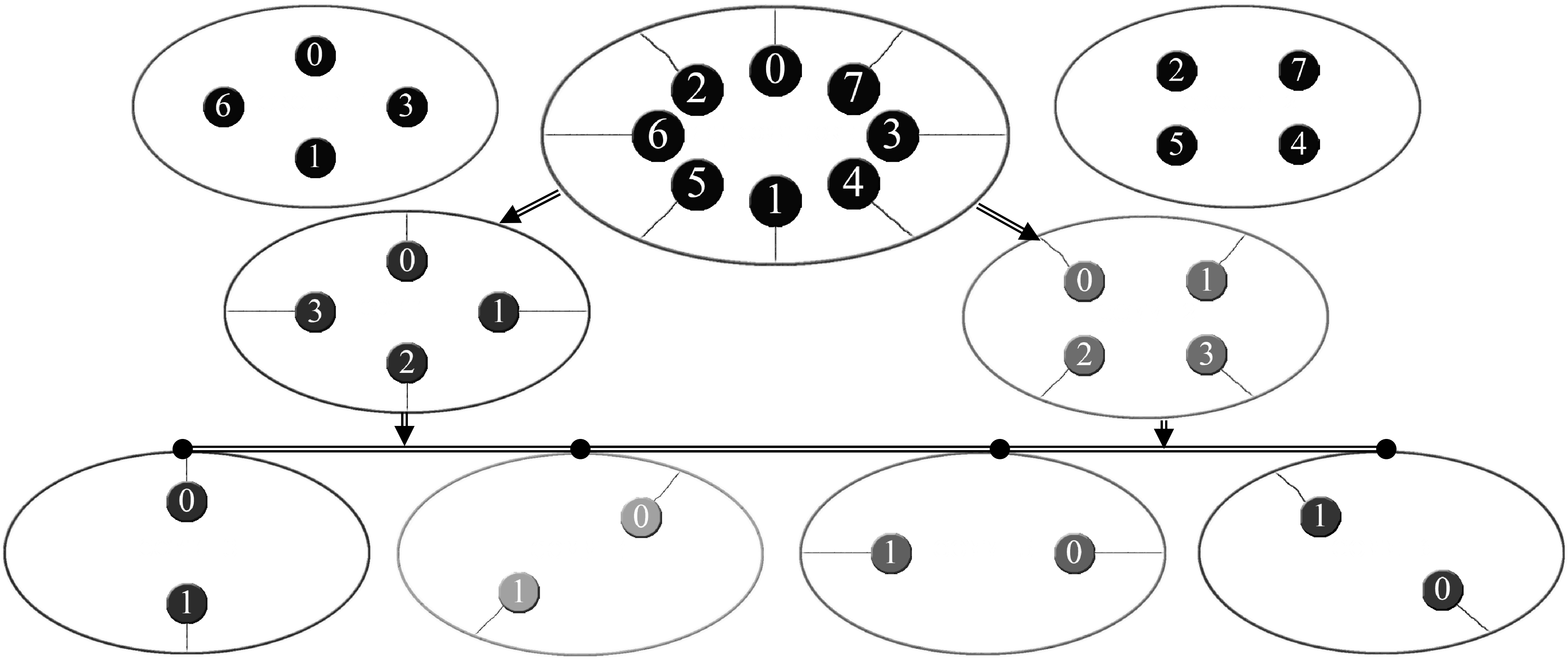
- `MPI_COMM_RANK`
  - Gets a process's rank (ID) within a process group
  - C : `int MPI_Comm_rank(MPI_Comm comm, int *rank)`
  - Fortran : `call MPI_COMM_RANK(mpi_comm, rank, ierror)`
  - C++ : `int MPI::Comm::Get_rank() const;`
- `MPI_COMM_SIZE`
  - Gets the number of processes within a process group
  - C : `int MPI_Comm_size(MPI_Comm comm, int *size)`
  - Fortran : `call MPI_COMM_SIZE(mpi_comm, size, ierror)`
  - C++ : `int MPI::Comm::Get_size() const;`
- `MPI_Comm` : Communicator

MPI_COMM_WORLD communicator

- MPI_COMM_WORLD is the default communicator setup by MPI_Init()
- It contains all the processes
- For simplicity just use it wherever a communicator is required!



MPI_COMM_WORLD example



WORLD, rank0	WORLD, rank3	WORLD, rank1	WORLD, rank6
COMM 1, rank0	COMM 1, rank1	COMM 1, rank2	COMM 1, rank3
COMM 3, rank0	COMM 5, rank0	COMM 3, rank1	COMM 5, rank1
WORLD, rank2	WORLD, rank7	WORLD, rank5	WORLD, rank4
COMM 2, rank0	COMM 2, rank1	COMM 2, rank2	COMM 2, rank3
COMM 6, rank1	COMM 4, rank0	COMM 4, rank1	COMM 6, rank0

Every process has three communicating groups and a distinct rank associated to it

Go to Menu

Example: “Hello From …” – C

```
#include <stdio.h>
#include “mpi.h”
int main(int argc, char** argv)
{
  int my_PE_num, err;
  err = MPI_Init(&argc, &argv);
  if (err != MPI_SUCCESS) {
      printf ("Error initializing MPI.\n");
      MPI_Abort(MPI_COMM_WORLD, err); }
  MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
  printf(“Hello from %d. \n”, my_PE_num);
  MPI_Finalize();
}
```

Example: “Hello From …” – Fortran

```
program Hello_From
include ‘mpif.h’
integer my_PE_num, ierror, rc
call MPI_INIT(ierror)
if (ierror .ne. MPI_SUCCESS) then
   print *,'Error initializing MPI.’
   call MPI_ABORT(MPI_COMM_WORLD, rc, ierror)
end if
call MPI_COMM_RANK(MPI_COMM_WORLD, my_PE_num, ierror)
print *, ‘Hello from’, my_PE_num
call MPI_FINALIZE(ierror)
end
```

Example: “Hello From …” – C++

```
#include "mpi.h"
#include <iostream>
int main(int argc, char** argv)
{
        int my_PE_num;
        MPI::Init(argc, argv);
        my_PE_num = MPI::COMM_WORLD.Get_rank();
        cout << "Hello from " << my_PE_num << endl;
        MPI::Finalize();
}
```

Outline: Point to Point Communications

- Overview
- Buffering the Messages
- Blocking Calls
- Non-Blocking Calls
- Order in MPI
- Fairness in MPI
- For a Communication to Succeed…
- MPI Basic Datatypes
- Communication Modes
- Communication Modes: Blocking Behavior
- Syntax of Blocking Send
- Syntax of Blocking Receive
- Example: Passing a Message
- Example: Deadlock Situation
- Communication Modes: Non-Blocking Behavior
- Syntax of Non-Blocking Calls
- Sendrecv
- Testing Communications for Completion
- Example: Ring (Blocking Communication)
- Example: Ring (Non-blocking Communication)
- Example: Simple Array
- Example: MPI Communication Timing Test
- Example: π -calculation
- Example: Simple Matrix Multiplication Algorithm
- Fox's Algorithm

Go to Menu

Overview

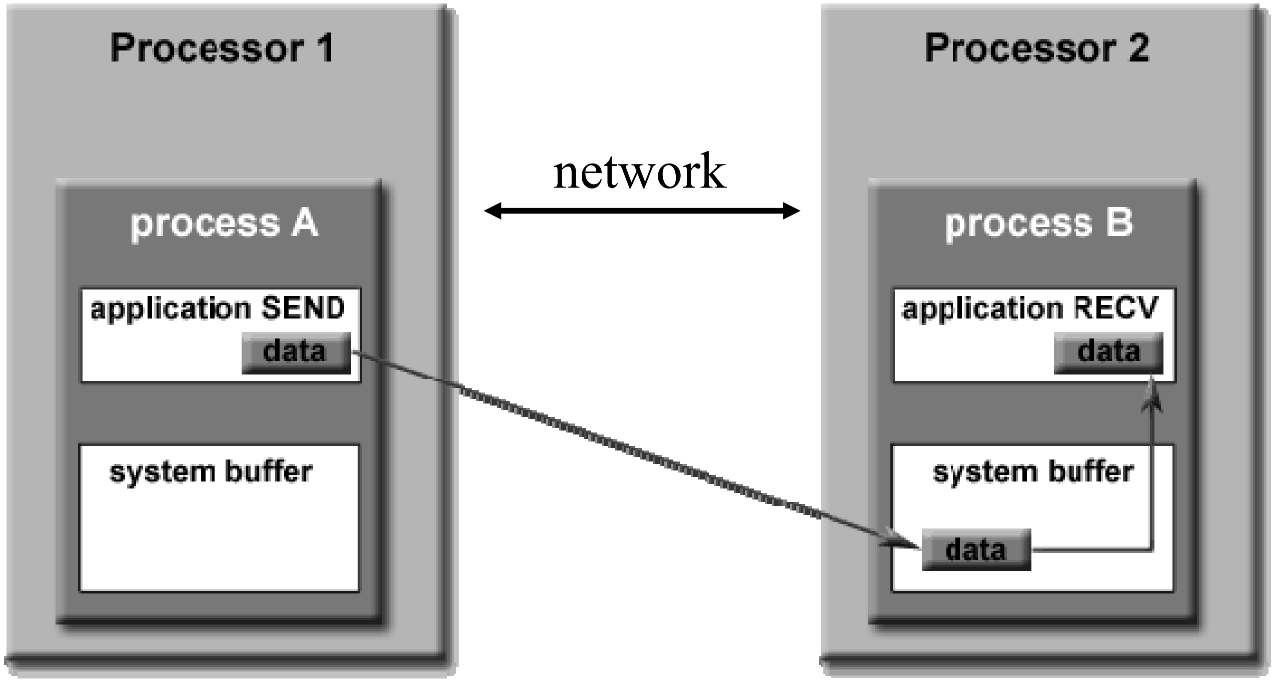
- Point to point is sending a message from one process to another, i.e. source process sends message to destination process
- Communication takes place within a given communicator
- MPI defines four communication modes for blocking and non-blocking send:
  - synchronous mode ("safest")
  - ready mode (lowest system overhead)
  - buffered mode (decouples sender from receiver)
  - standard mode (compromise)
- The receive call does not specify communication mode - it is simply blocking and non-blocking
  - Blocking call stops the program until the message buffer is safe to use
  - Non-blocking call separates communication from computation
- Two messages sent from one process to another will arrive in same relative order as they are sent.

Buffering the Messages

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. And there are cases when that may not be desirable. MPI implementation must be able to deal with storing data when the two tasks are out of sync.
- Consider the following two cases:
  - A send operation occurs 5 seconds before the receive is ready - where is the message while the receive is pending?
  - Multiple sends arrive at the same receiving task which can only accept one send at a time - what happens to the messages that are "backing up"?
- The MPI implementation (not the MPI standard) decides what happens to data in these two cases. Typically, a *system buffer* area is reserved to hold data in transit.

Buffering the Messages (2)

- System buffer space:
  - Memory in the processes
  - Opaque to the programmer and managed entirely by the MPI library
  - A finite resource that can be exhausted
  - May exist on a sending side, a receiving side, or both
  - May improve program performance because it allows send - receive operations to be asynchronous.



Path of a message buffered at the receiving process

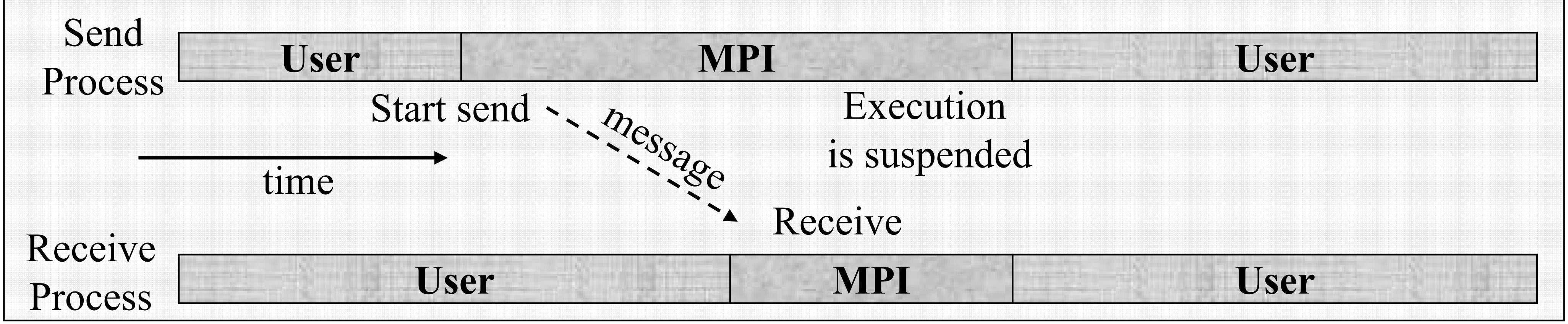
Blocking Calls

- A blocking send or receive call suspends execution of user's program until the message buffer being sent/received is safe to use.
- In case of a blocking send, this means the data to be sent have been copied out of the send buffer, but these data have not necessarily been received in the receiving task. The contents of the send buffer can be modified without affecting the message that was sent
- The blocking receive implies that the data in the receive buffer are valid.

Blocking Send and Receive

- A blocking MPI call means that the program execution will be suspended until the message buffer is safe to use. The MPI standards specify that a blocking SEND or RECV does not return until the send buffer is safe to reuse (for MPI_SEND), or the receive buffer is ready to use (for MPI_RECV).

Blocking Send/Receive Diagram:



Non-Blocking Calls

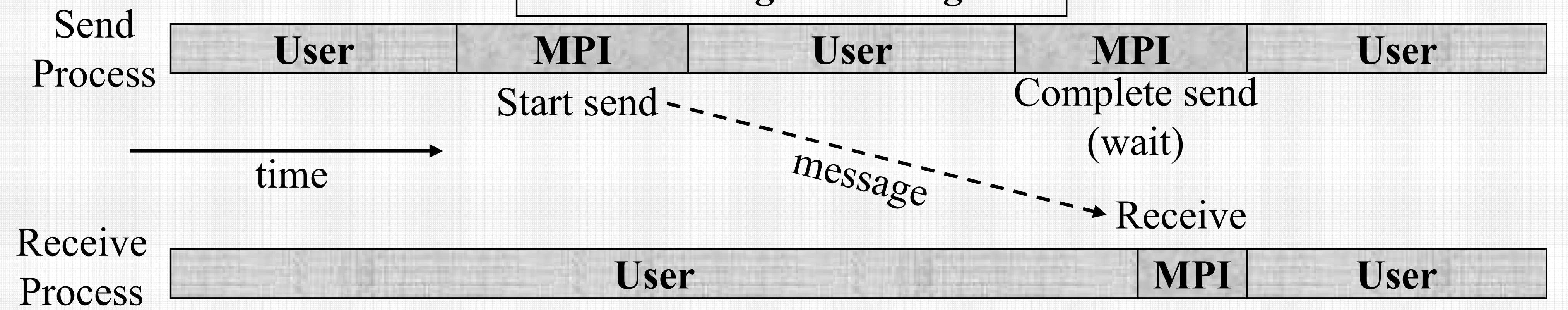
- Non-blocking calls return immediately after initiating the communication.
- In order to reuse the send message buffer, the programmer must check for its status.
- The programmer can choose to block before the message buffer is used or test for the status of the message buffer.
- A blocking or non-blocking send can be paired to a blocking or non-blocking receive

Go to Menu

Non-Blocking Send and Receive

- Separate Non-Blocking communication into three phases:
  - Initiate non-blocking communication.
  - Do some work (perhaps involving other communications?)
  - Wait for non-blocking communication to complete.

Non-Blocking Send Diagram:



Send Process

User | MPI | User | MPI | User

Start send

Complete send (wait)

time

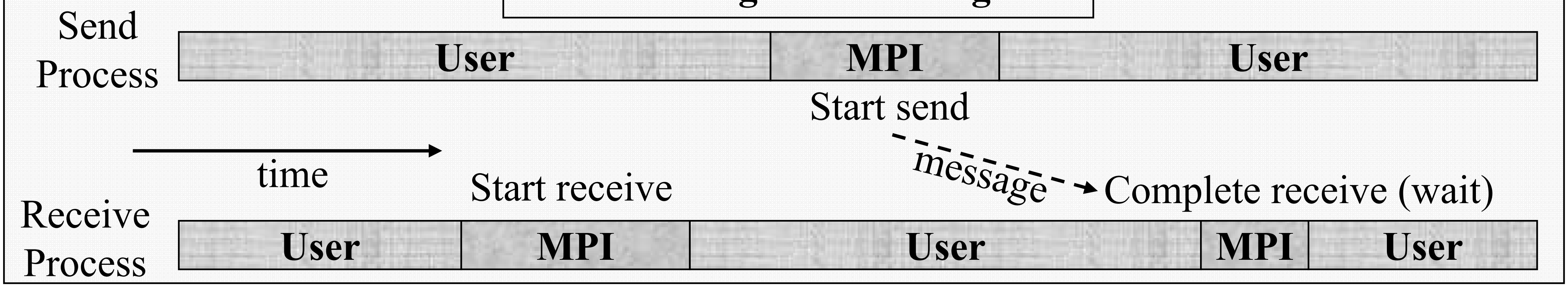
message

Receive

Receive Process

User | MPI | User

Non-Blocking Receive Diagram:



Send Process

User | MPI | User

Start send

time

Start receive

message

Complete receive (wait)

Receive Process

User | MPI | User | MPI | User

Order in MPI

- Order:
  - MPI guarantees that messages from same process will not overtake each other.
    - If a sender sends two messages (Message 1 and Message 2) in succession to the same destination, and both match the same receive, the receive operation will receive Message 1 before Message 2.
    - If a receiver posts two receives (Receive 1 and Receive 2), in succession, and both are looking for the same message, Receive 1 will receive the message before Receive 2.
    - Order rules do not apply if there are multiple threads participating in the communication operations.

Fairness in MPI

- Fairness:
  - A parallel algorithm is *fair* if no process is effectively ignored
  - In the example below processes with low rank (like process zero) may be the only one whose messages are received.
  - MPI does not guarantee fairness - it's up to the programmer to prevent "operation starvation".
  - Another example: task 0 sends a message to task 2. However, task 1 sends a competing message that matches task 2's receive. Only one of the sends will complete.

```
if (rank == 0) {
  for (i=0; i<100*(size-1); i++) {
    MPI_Recv(buf,1,MPI_INT,ANY_SOURCE,ANY_TAG,MPI_COMM_WORLD,&status);
    printf("Msg from %d with tag %d\n",status.MPI_SOURCE,status.MPI_TAG);}}
else {
  for (i=0; i<100; i++)
    MPI_Send( buf, 1, MPI_INT, 0, i, MPI_COMM_WORLD ); }
```

For a Communication to Succeed…

- Sender must specify a valid destination rank
- Receiver must specify a valid source rank
  - may use wildcard : `MPI_ANY_SOURCE`
- The communicator must be the same
- Tags must match
  - may use wildcard : `MPI_ANY_TAG`
- Message types must match
- Receiver's buffer must be large enough

Go to Menu

MPI Basic Datatypes for C

MPI Datatypes	C Datatypes
MPI_CHAR	signed char
MPI_INT	signed int
MPI_LONG	signed long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	--------
MPI_SHORT	signed short int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED	unsigned int
MPI_PACKED	--------

MPI Basic Datatypes for Fortran

MPI Datatypes	Fortran Datatypes
`MPI_INTEGER`	`integer`
`MPI_REAL`	`real`
`MPI_DOUBLE_PRECISION`	`double precision`
`MPI_COMPLEX`	`complex`
`MPI_DOUBLE_COMPLEX`	`double complex`
`MPI_LOGICAL`	`logical`
`MPI_CHARACTER`	`character(1)`
`MPI_BYTE`	`--------`
`MPI_PACKED`	`--------`

Go to Menu

MPI Basic Datatypes for C++

MPI Datatypes	C++ Datatypes
MPI::CHAR	char
MPI::WCHAR	wchar_t
MPI::SHORT	signed short
MPI::INT	signed int
MPI::LONG	signed long
MPI::SIGNED_CHAR	signed char
MPI::UNSIGNED_CHAR	unsigned char
MPI::UNSIGNED_SHORT	unsigned short
MPI::UNSIGNED	unsigned int
MPI::UNSIGNED_LONG	unsigned long int
MPI::FLOAT	float
MPI::DOUBLE	double
MPI::LONG_DOUBLE	long double
MPI::BOOL	bool
MPI::COMPLEX	Complex<float>
MPI::DOUBLE_COMPLEX	Complex<double>
MPI::LONG_DOUBLE_COMPLEX	Complex<long double>
MPI::BYTE	--------
MPI::PACKED	--------

Go to Menu

Communication Modes

- MPI has 8 different types of Send
- The non-blocking send has an extra argument of ***request handle***

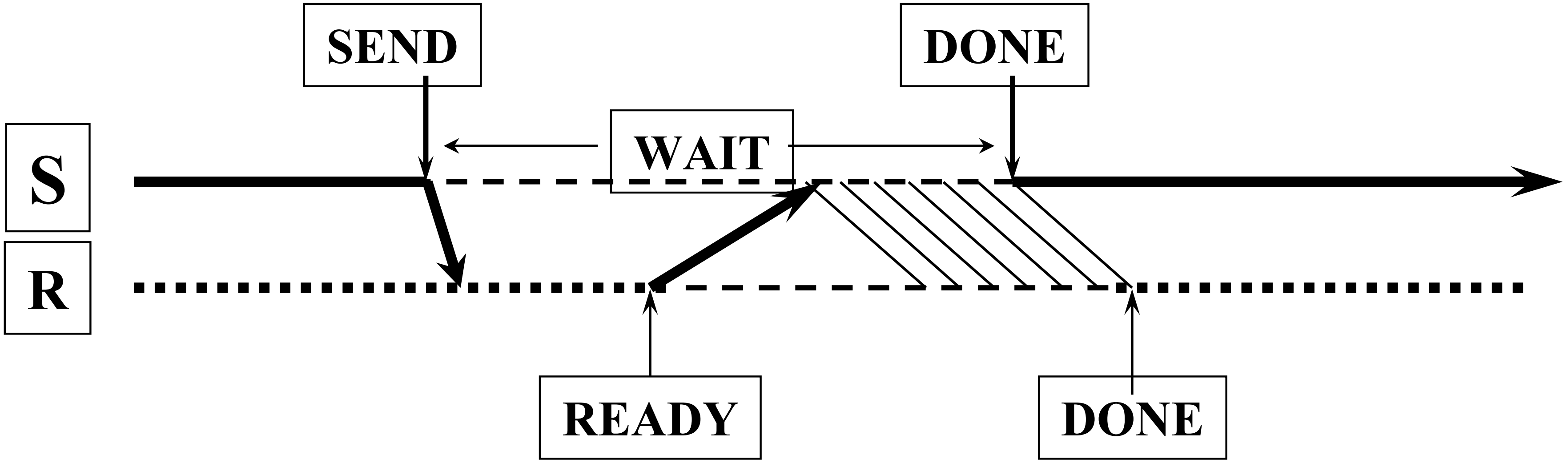
	Blocking	**Non-Blocking**
Standard	`MPI_Send`	`MPI_Isend`
Synchronous	`MPI_Ssend`	`MPI_Issend`
Buffer	`MPI_Bsend`	`MPI_Ibsend`
Ready	`MPI_Rsend`	`MPI_Iresend`
	`MPI_RECV`	`MPI_IRECV`
	`MPI_SENDRECV`	

Communication Modes: Blocking Behavior

Synchronous Send MPI_SSEND	Return when the message buffer can be safely reused. Can be started whether or not a matching receive was posted. However, the send will complete successfully only if a matching receive is posted.
Buffered Send MPI_BSEND	Return when message is copied to the system buffer
Ready Send MPI_RSEND	Complete if matching receive is already waiting
Standard Send MPI_SEND	Either synchronous or buffered, implemented by vendor to give good performance for most programs

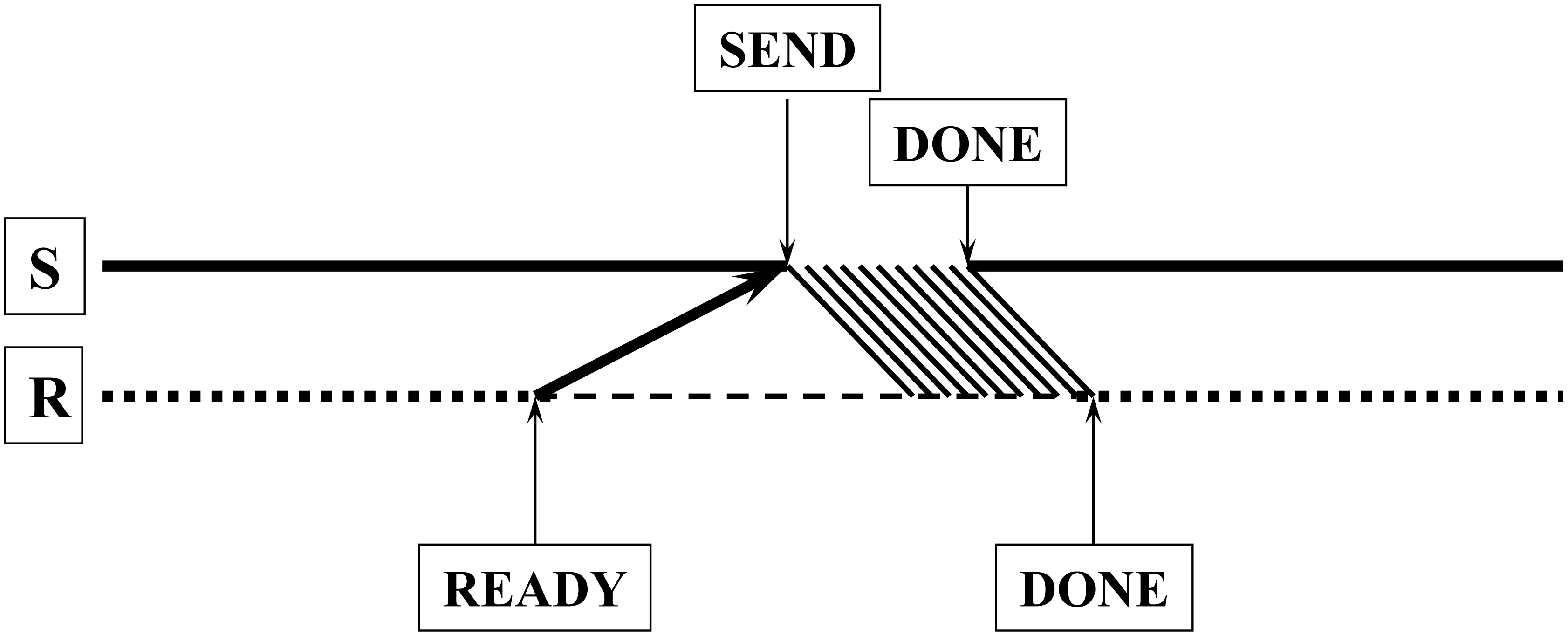
Blocking Synchronous Send: `MPI_SSEND`

- the sending task tells the receiver that a message is ready for it and waits for the receiver to acknowledge
- system overhead : buffer to network and vice versa
- synchronization overhead : handshake + waiting
- safest , most portable



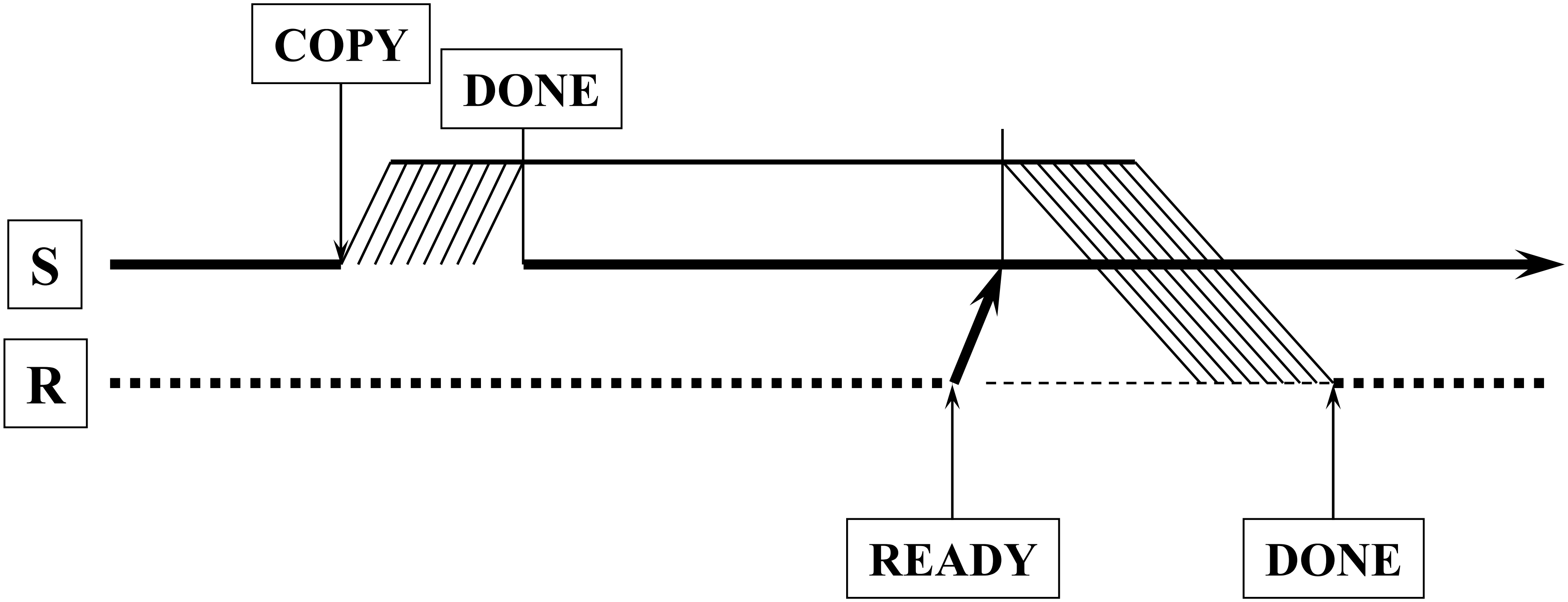
Blocking Ready Send: `MPI_RSEND`

- requires a “ready to receive” notification, if not => error, exit
- sends message out over network
- minimize overhead



Blocking Buffered Send: `MPI_BSEND`

- Blocking buffer send - user-supply buffer on send node
- Buffer can be statically or dynamically allocated
- Send : data copy to buffer and return
- Receive : when notify, data are copied from buffer



Blocking Standard Send: `MPI_Send`

- Implemented by vendors to give good performance for most programs.
- Simple and easy to use
- Either synchronous or buffered
- CRAY XT :
  - Based on MPICH2
  - Use a portals device for MPICH2
  - Support MPI2-RMA (one-sided)
  - Full MPI-IO support
  - No dynamic process management (NO Spawn process!!)
  - `man intro_mpi` for more information

Syntax of Blocking Send

C :

```
int MPI_Send(&buf, count, datatype, dest, tag, comm)
```

`&buf`:	pointer of object to be sent
`count`:	the number of items to be sent, e.g. 10
`datatype`:	the type of object to be sent, e.g. `MPI_INT`
`dest`:	destination of message (rank of receiver), e.g. 6
`tag`:	message tag, e.g. 78
`comm`:	communicator, e.g. `MPI_COMM_WORLD`

Fortran :

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

C++ :

```
void MPI::Comm::Send(const void* buf, int count, const
  MPI::Datatype& datatype, int dest, int tag) const
```

Syntax of Blocking Receive

C :

```
int MPI_Recv(&buf, count, datatype, source, tag, comm, &status)
```

`source`: the node to receive from, e.g. 0

`&status`: a structure which contains three fields, the source, tag, and error code of the incoming message.

Fortran :

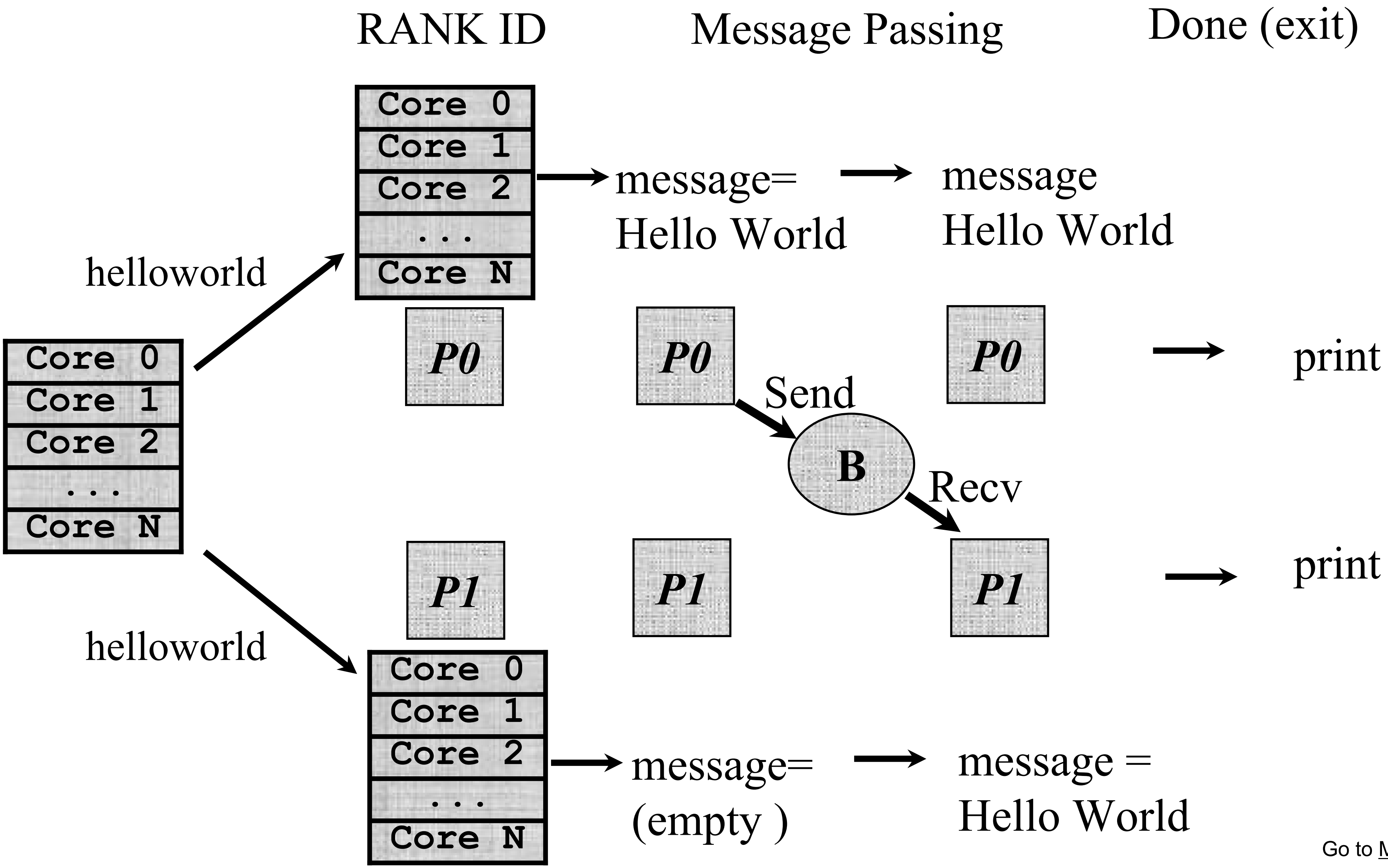
```
call MPI_RECV(buf, count, datatype, source, tag, comm,
         status(MPI_STATUS_SIZE), ierror)
```

`status`: an array of integers of size MPI_STATUS_SIZE

C++ :

```
void MPI::Comm::Recv(void* buf, int count, const MPI::Datatype&
  datatype, int source, int tag, MPI::Status& status) const
```

Example: Passing a Message – Schematic



RANK ID

Message Passing

Done (exit)

Core 0
Core 1
Core 2
...
Core N

helloworld

Core 0
Core 1
Core 2
...
Core N

message=
Hello World

message
Hello World

P0

P0

Send

B

Recv

P0

print

helloworld

P1

P1

P1

print

Core 0
Core 1
Core 2
...
Core N

message=
(empty)

message =
Hello World

Example: Passing a Message – Hello World Again!

C Example

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char ** argv)
  {
     int my_PE_num, ntag = 100;
     char message[12] = "Hello, world";
     MPI_Status status;
     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

     if ( my_PE_num == 0 )
       MPI_Send(&message, 12,MPI_CHAR,1,ntag,MPI_COMM_WORLD);
     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();
  }
```

Example: Passing a Message – Hello World Again!

Fortran Example

```
program Hello_World
include ‘mpif.h’
integer me, ierror, ntag, status(MPI_STATUS_SIZE)
character(12) message
call MPI_INIT(ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, me, ierror)
ntag = 100
if ( me .eq . 0) then
     message = ‘Hello, World’
     call MPI_Send(message, 12, MPI_CHARACTER, 1, ntag,
   MPI_COMM_WORLD, ierror)
else if ( me .eq . 1) then
     call MPI_Recv(message, 12, MPI_CHARACTER, 0, ntag,
   MPI_COMM_WORLD, status, ierror)
     print *, ‘Node’,me, ‘:’, message
endif
call MPI_FINALIZE(ierror)
end
```

Example: Passing a Message – Hello World Again!

C++ Example

```
#include "mpi.h"
#include <iostream>
#include <string>
int main(int argc, char ** argv)
  {
        int my_PE_num, ntag = 100;
        char message[13] = "Hello, world";
        MPI::Status status;
        MPI::Init(argc, argv);
        my_PE_num = MPI::COMM_WORLD.Get_rank();

        if ( my_PE_num == 0 )
          MPI::COMM_WORLD.Send(message,12,MPI::CHAR,1,ntag);
        else if ( my_PE_num == 1 ) {
          MPI::COMM_WORLD.Recv(message,12,MPI::CHAR,0,ntag, status);
          cout << "Node " << my_PE_num <<" : " << message << endl; }

        MPI::Finalize();
  }
```

Example: Deadlock Situation

- You should be careful with your communications pattern to avoid getting into a deadlock. A *deadlock* is a situation that arises when a process cannot proceed because it is waiting on another process that is, in turn, waiting on the first process.
- Deadlock
  - All tasks are waiting for events that haven't been initiated
  - common to SPMD program with blocking communication, e.g. every task sends, but none receives
  - insufficient system buffer space is available
- Remedies :
  - arrange one task to receive
  - use `MPI_Ssendrecv`
  - use non-blocking communication

Example: Deadlock – Fortran

```
c  Improper use of blocking calls results in deadlock run on two
   nodes
c  author : Roslyn Leibensperger, (CTC)

   program deadlock
   implicit none
   include ‘mpif.h’
   integer MSGLEN, ITAG_A, ITAG_B
   parameter (MSGLEN = 2048, ITAG_A = 100, ITAG_B = 200)
   real rmsg1(MSGLEN) , rmsg2(MSGLEN)
   integer irank, idest, isrc, istag, iretag,
   istatus(MPI_STATUS_SIZE), ierr, I

   call MPI_Init (ierr)
   call MPI_Comm_rank( MPI_COMM_WORLD, irank, ierr)
   do I = 1, MSGLEN
        rmsg1(I) = 100
        rmsg2(I) = -100
   end do
```

Example: Deadlock – Fortran (Cont’d)

```
if ( irank .eq. 0 ) then
     idest = 1
     isrc = 1
     istag = ITAG_A
     iretag = ITAG_B
else if ( irank .eq. 1 ) then
     idest = 0
     isrc = 0
     istag = ITAG_B
     iretag = ITAG_A
end if
print *, “Task “, irank, “ has sent the message “
call MPI_Ssend (rmsg1,MSGLEN, MPI_REAL, idest, istag,
MPI_COMM_WORLD, ierr)
call MPI_Recv (rmsg2, MSGLEN, MPI_REAL, isrc, iretag,
MPI_COMM_WORLD,istatus, ierr)
print*, “Task”, irank, “ has received the message “
call MPI_Finalize (ierr)
end
```

Example: Deadlock – Fortran (fixed)

```
c  Solution program showing the use of a non-blocking send to
   eliminate deadlock
c  author : Roslyn Leibensperger (CTC)
   program fixed
   implicit none
   include 'mpif.h‘
   integer MSGLEN, ITAG_A, ITAG_B
   parameter (MSGLEN = 2048, ITAG_A = 100, ITAG_B = 200)
   real rmsg1(MSGLEN), rmsg2(MSGLEN)
   integer irank, idest, isrc, istag, iretag
   integer ierr, I, request, irstatus(MPI_STATUS_SIZE)
   integer isstatus(MPI_STATUS_SIZE)
   ------------------//--------------------
   print *, “ Task”, irank, “ has started the send”
   call MPI_Isend ( rmsg1, MSGLEN, MPI_REAL, idest, istag,
   MPI_COMM_WORLD,  request, ierr)
   call MPI_Recv (rmsg2, MSGLEN, MPI_REAL, isrc, iretag,
   MPI_COMM_WORLD, irstatus, ierr)
   call MPI_Wait (request, isstatus, ierr)
   print *, “Task “, irank, “has completed the send”
   call MPI_Finalize (ierr)
   end
```

Example: Deadlock – C

```
#include <stdio.h>
#include "mpi.h‟
int main(int argc, char *argv[])
{
#define MSGLEN 2048
int ITAG_A = 100,ITAG_B = 200;
int irank, i, idest, isrc, istag, iretag;
float rmsg1[MSGLEN];
float rmsg2[MSGLEN];
MPI_Status recv_status;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &irank);

for (i = 1; i <= MSGLEN; i++)
    {
      rmsg1[i] = 100;
      rmsg2[i] = -100;
    }
```

Example: Deadlock – C (Cont’d)

```
if ( irank == 0 )
  {
    idest  = 1;
    isrc   = 1;
    istag  = ITAG_A;
    iretag = ITAG_B;
  }
else if ( irank == 1 )
  {
    idest  = 0;
    isrc   = 0;
    istag  = ITAG_B;
    iretag = ITAG_A;
  }
printf("Task %d has sent the message\n", irank);
MPI_Ssend(&rmsg1, MSGLEN, MPI_FLOAT, idest, istag,
  MPI_COMM_WORLD);
MPI_Recv(&rmsg2, MSGLEN, MPI_FLOAT, isrc, iretag,
  MPI_COMM_WORLD, &recv_status);
printf("Task %d has received the message\n", irank);
MPI_Finalize();}
```

Go to Menu

Example: Deadlock – C (fixed)

```
//Solution program showing the use of a non-blocking send to
    eliminate deadlock
#include <stdio.h>
#include "mpi.h“
int main(int argc, char *argv[])
{
#define MSGLEN 2048
  int ITAG_A = 100,ITAG_B = 200;
  int irank, i, idest, isrc, istag, iretag;
  float rmsg1[MSGLEN];
  float rmsg2[MSGLEN];
  MPI_Status irstatus, isstatus;
  MPI_Request request;
    ------------------//--------------------
  printf("Task %d has sent the message\n", irank);
  MPI_Isend(&rmsg1, MSGLEN, MPI_FLOAT, idest, istag,
    MPI_COMM_WORLD, &request);
  MPI_Recv(&rmsg2, MSGLEN, MPI_FLOAT, isrc, iretag,
    MPI_COMM_WORLD, &irstatus);
  MPI_Wait(&request, &isstatus);
  printf("Task %d has received the message\n", irank);
  MPI_Finalize();}
```

Go to Menu

Example: Deadlock – C++

```
#include "mpi.h"
#include <iostream>

int main(int argc, char *argv[])
{
#define MSGLEN 2048
  int ITAG_A = 100,ITAG_B = 200;
  int irank, i, idest, isrc, istag, iretag;
  float rmsg1[MSGLEN];
  float rmsg2[MSGLEN];
  MPI::Status recv_status;

  MPI::Init(argc, argv);
  irank = MPI::COMM_WORLD.Get_rank();

  for (i = 1; i <= MSGLEN; i++)
    {
      rmsg1[i] = 100;
      rmsg2[i] = -100;
    }
```

Example: Deadlock – C++ (Cont’d)

```
if ( irank == 0 )
   {
     idest  = 1;
     isrc   = 1;
     istag  = ITAG_A;
     iretag = ITAG_B;
   }
 else if ( irank == 1 )
   {
     idest  = 0;
     isrc   = 0;
     istag  = ITAG_B;
     iretag = ITAG_A;
   }

 cout << "Task "  << irank << " has sent the message" << endl;
 MPI::COMM_WORLD.Ssend(rmsg1, MSGLEN, MPI::FLOAT, idest, istag);
 MPI::COMM_WORLD.Recv(rmsg2, MSGLEN, MPI::FLOAT, isrc, iretag,
   recv_status);
 cout << "Task " << irank << " has received the message" << endl;
 MPI::Finalize();
```

Example: Deadlock – C++ (fixed)

```
#include "mpi.h"
#include <iostream>

int main(int argc, char *argv[])
{
#define MSGLEN 2048
  int ITAG_A = 100,ITAG_B = 200;
  int irank, i, idest, isrc, istag, iretag;
  float rmsg1[MSGLEN];
  float rmsg2[MSGLEN];
  MPI::Status irstatus, isstatus;
  MPI::Request request;
    ------------------//--------------------
  cout << "Task "  << irank << " has sent the message" << endl;
  request = MPI::COMM_WORLD.Isend(rmsg1, MSGLEN, MPI::FLOAT, idest,
    istag);
  MPI::COMM_WORLD.Recv(rmsg2, MSGLEN, MPI::FLOAT, isrc, iretag,
    irstatus);
  MPI_Wait(request, isstatus);
  cout << "Task " << irank << " has received the message" << endl;
  MPI::Finalize();
}
```

Communication Modes: Non-Blocking Behavior

- The non-blocking calls have the same syntax as the blocking calls with two exceptions:
  - Each call has an “I” immediately following the “_”
  - The last argument is a handle to an opaque request object that contains information about the message
- Non-blocking call returns immediately after initiating the communication
- The programmer can block or check for the status of the message buffer : `MPI_Wait` or `MPI_Test`.

Syntax of Non-Blocking Calls

Fortran :

```
call MPI_Isend(buf,count,datatype,dest,tag,comm,handle,ierr)
call MPI_Irecv(buff,count,datatype,src,tag,comm,handle,ierr)
call MPI_Test(handle, flag, status, ierr)
call MPI_Wait(handle, status, ierr)
```

C :

```
MPI_Isend(&buf, count, datatype, dest, tag, comm, &handle)
MPI_Irecv(&buff, count, datatype, src, tag, comm, &handle)
MPI_Test(&handle, &flag, &status)
MPI_Wait(&handle, &status)
```

C++ :

```
MPI::Request MPI::Comm::Isend(const void *buf, int count,
  const MPI::Datatype& datatype, int dest, int tag) const;
MPI::Request MPI::Comm::Irecv(void *buf, int count, const
  MPI::Datatype& datatype, int source, int tag) const;
bool MPI::Request::Test(MPI::Status& status);
void MPI::Request::Wait(MPI::Status& status);
```

Sendrecv

- useful for executing a shift operation across a chain of processes
- system takes care of possible deadlock due to blocking calls

```
MPI_Sendrecv(sbuf, scount, stype, dest, stag, rbuf,
  rcount, rtype, source, rtag, comm, status)
```

`sbuf (rbuf):`	initial address of send (receive)buffer
`scount (rcount):`	number of elements in send (receive) buffer
`stype (rtype):`	type of elements in send (receive) buffer
`stag (rtag):`	send (receive) tag
`dest:`	rank of destination
`source:`	rank of source
`comm:`	communicator
`status:`	status object

Testing Communications for Completion

- `MPI_Wait(request, status)`
  - These routines block until the communication has completed. They are useful when the data from the communication buffer is about to be re-used

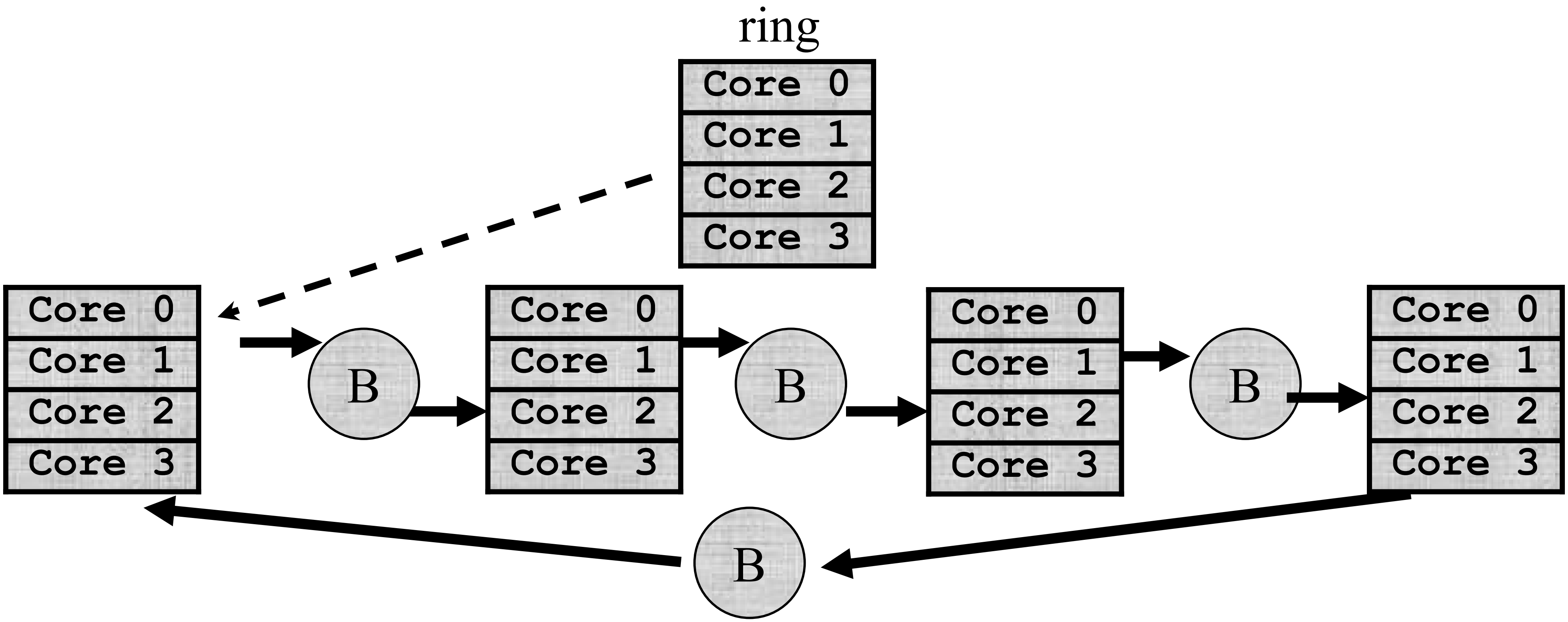
- `MPI_Test(request, flag, status)`
  - This routine blocks until the communication specified by the handle ***request*** has completed. The ***request*** handle will have been returned by an earlier call to a non-blocking communication routine. The routine queries completion of the communication and the result (TRUE of FALSE) is returned in ***flag***

Timer: `MPI_Wtime`

C :	`double MPI_Wtime(void)`
Fortran :	`double precision MPI_Wtime()`
C++:	`double MPI::Wtime();`

- Time is measured in seconds.
- Time to perform a task is measured by consulting the timer before and after.
- Modify your program to measure its execution time and print it out.

Example: Ring (Blocking Communication) – Schematic



myrank=0	myrank=1	myrank=2	myrank=3
size=4	size=4	size=4	size=4
leftid=3	leftid=0	leftid=1	leftid=2
rightid=1	rightid=2	rightid=3	rightid=0
Val=3,2,1,0	Val=0,3,2,1	Val=1,0,3,2	Val=2,1,0,3

Go to Menu

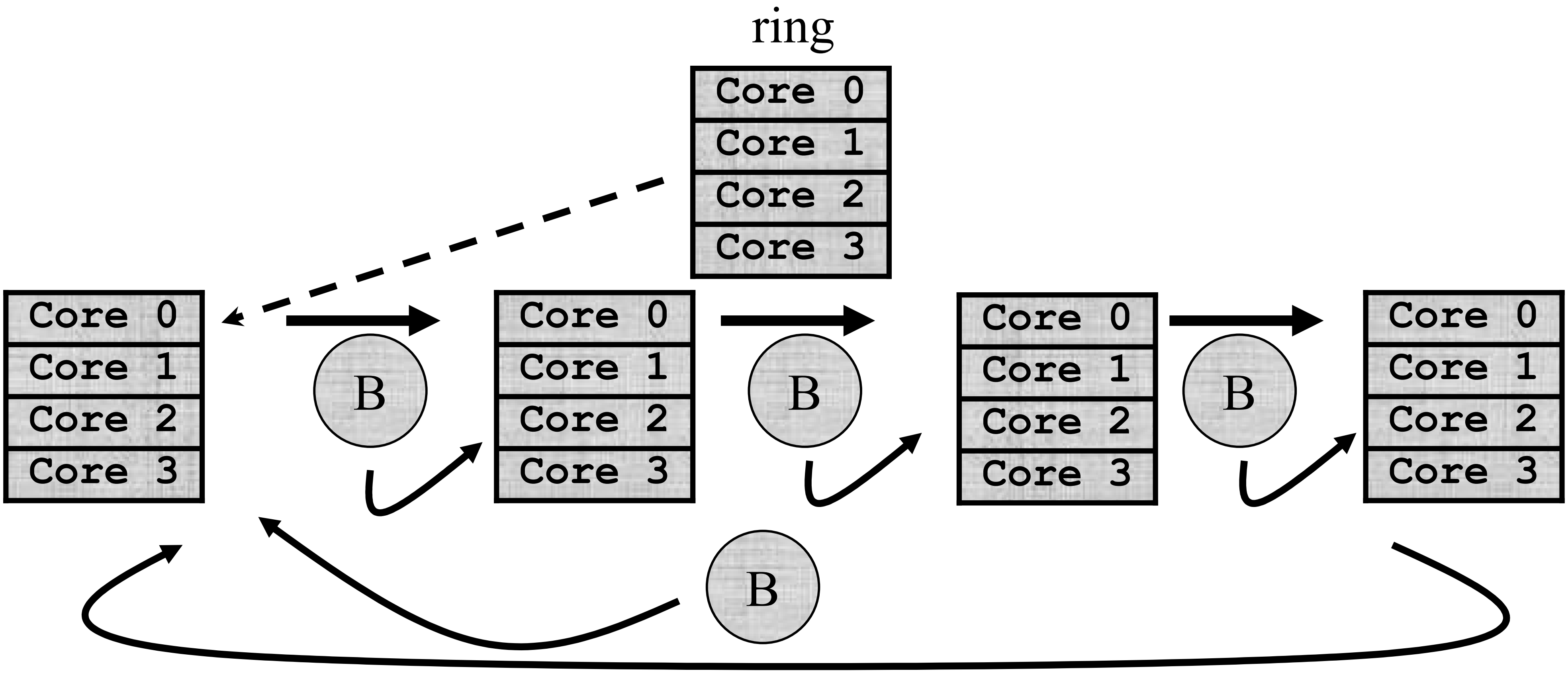
Example: Ring (Blocking Communication) – C

```
#include <stdio.h>
#include “mpi.h”
int main(int argc, char *argv[])             {
  int myrank, nprocs, leftid, rightid, val, sum, tmp;
  MPI_Status recv_status;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  if((leftid=(myrank-1)) < 0) leftid = nprocs -1;
  if((rightid=(myrank+1)) == nprocs) rightid = 0;
  val = myrank;
  sum = 0;
 do {
   MPI_Send(&val,1,MPI_INT,rightid,99, MPI_COMM_WORLD);
   MPI_Recv(&tmp,1, MPI_INT, leftid, 99, MPI_COMM_WORLD,
   &recv_status);
   val = tmp;
   sum += val;
  } while (val != myrank);
  printf(“proc %d sum = %d \n”, myrank, sum);
  MPI_Finalize();                    }
```

Example: Ring (Blocking Communication) – Fortran

```
     PROGRAM ring
     IMPLICIT NONE
     include "mpif.h‟
     INTEGER ierror, val, my_rank, nprocs, rightid, leftid, tmp, sum, request
     INTEGER send_status(MPI_STATUS_SIZE), recv_status(MPI_STATUS_SIZE)
     CALL MPI_INIT(ierror)
     CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierror)
     CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierror)
     rightid = my_rank + 1
     IF (rightid .EQ. nprocs) rightid = 0
     leftid = my_rank - 1
     IF (leftid .EQ. -1) leftid = nprocs-1
     sum = 0
     val = my_rank
100  CONTINUE
     CALL MPI_SEND(val, 1, MPI_INTEGER, rightid, 99,
    $     MPI_COMM_WORLD, request, ierror)
     CALL MPI_RECV(tmp, 1, MPI_INTEGER, leftid, 99,
    $     MPI_COMM_WORLD, recv_status, ierror)
     sum = sum + tmp
     val = tmp
     IF(tmp  .NE. my_rank) GOTO 100
     PRINT *, 'Proc ', my_rank, ' Sum = ',sum
     CALL MPI_FINALIZE(ierror)
     STOP
     END
```

Example: Ring (Non-blocking Communication) – Schematic



```
myrank=0
size=4
leftid=3
rightid=1
```

```
myrank=1
size=4
leftid=0
rightid=2
```

```
myrank=2
size=4
leftid=1
rightid=3
```

```
myrank=3
size=4
leftid=2
rightid=0
```

Example: Ring (Non-blocking Communication) – C

```
#include <stdio.h>
#include “mpi.h”
main(int argc, char *argv[]) {
  int myrank, nprocs, leftid, rightid, val, sum, tmp;
  MPI_Status recv_status, send_status;
  MPI_request send_request;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  if((leftid=(myrank-1)) < 0) leftid = nprocs -1;
  if((rightid=(myrank+1) == nprocs) rightid = 0;
  val = myrank
  sum = 0;
 do {
   MPI_Issend(&val,1,MPI_INT,right,99, MPI_COMM_WORLD,&send_request);
   MPI_Recv(&tmp,1, MPI_INT, left, 99, MPI_COMM_WORLD, &recv_status);
   MPI_Wait(&send_request,&send_status);
   val = tmp;
   sum += val;
  } while (val != myrank);
  printf(“proc %d sum = %d \n”, myrank, sum);
  MPI_Finalize();             }
```

Example: Ring (Non-blocking Communication) – Fortran

```
      PROGRAM ring
      IMPLICIT NONE
      include "mpif.h“
      INTEGER ierror, val, my_rank, nprocs, rightid, leftid, tmp, sum
      INTEGER send_status(MPI_STATUS_SIZE), recv_status(MPI_STATUS_SIZE)
      INTEGER request
      CALL MPI_INIT(ierror)
      CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierror)
      CALL MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierror)
      rightid = my_rank + 1
      IF (rightid .EQ. nprocs) rightid = 0
      leftid = my_rank - 1
      IF (leftid .EQ. -1) leftid = nprocs-1
      sum = 0
      val = my_rank
100   CONTINUE
      CALL MPI_ISSEND(val, 1, MPI_INTEGER, rightid, 99,
     $     MPI_COMM_WORLD, request, ierror)
      CALL MPI_RECV(tmp, 1, MPI_INTEGER, leftid, 99,
     $     MPI_COMM_WORLD, recv_status, ierror)
      CALL MPI_WAIT(request, send_status, ierror)
      sum = sum + tmp
      val = tmp
      IF(tmp .NE. my_rank) GOTO 100
      PRINT *, 'Proc ', my_rank, ' Sum = ',sum
      CALL MPI_FINALIZE(ierror)
      STOP
      END
```

Example: Simple Array

- This is a simple array assignment used to demonstrate the distribution of data among multiple tasks and the communications required to accomplish that distribution.
- The master distributes an equal portion of the array to each worker. Each worker receives its portion of the array and performs a simple value assignment to each of its elements. Each worker then sends its portion of the array back to the master. As the master receives a portion of the array from each worker, selected elements are displayed.
- Note: For this example, the number of processes should be set to an odd number, to ensure even distribution of the array to numtasks-1 worker tasks.

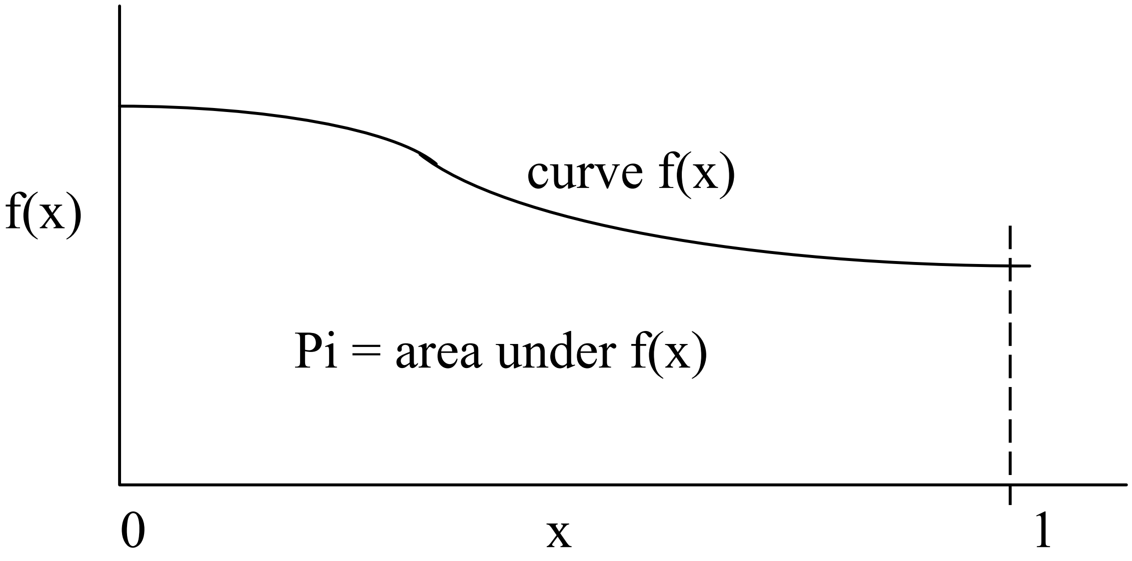
Go to Menu

Example: MPI Communication Timing Test

- The objective of this exercise is to investigate the amount of time required for message passing between two processes, i.e. an MPI communication timing test is performed.
- In this exercise different size messages are sent back and forth between two processes a number of times. Timings are made for each message before it is sent and after it has been received. The difference is computed to obtain the actual communication time. Finally, the average communication time and the bandwidth are calculated and output to the screen.
- For example, one can run this code on two nodes (one process on each node) passing messages of length 1, 100, 10,000, and 1,000,000 and record the results in a table.

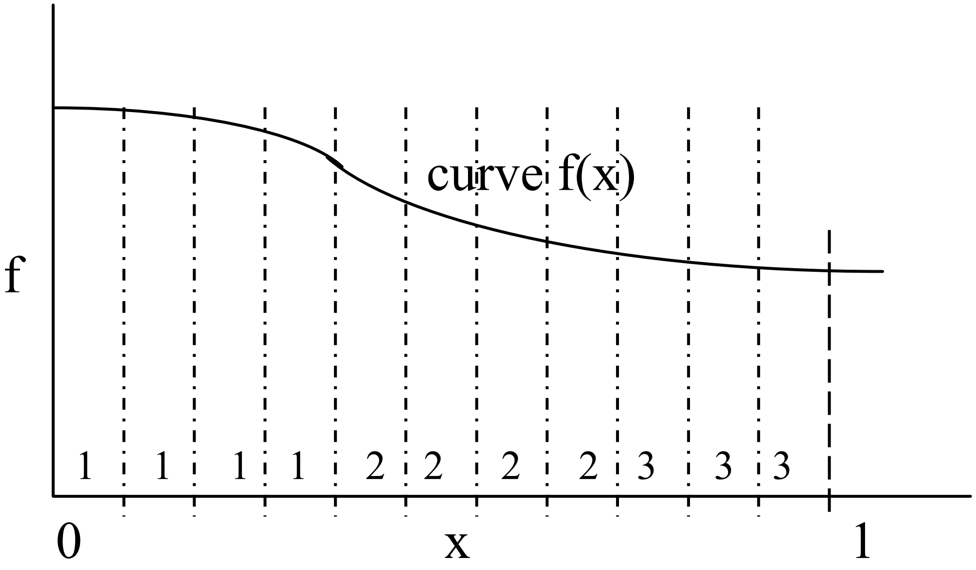
Example: π -calculation

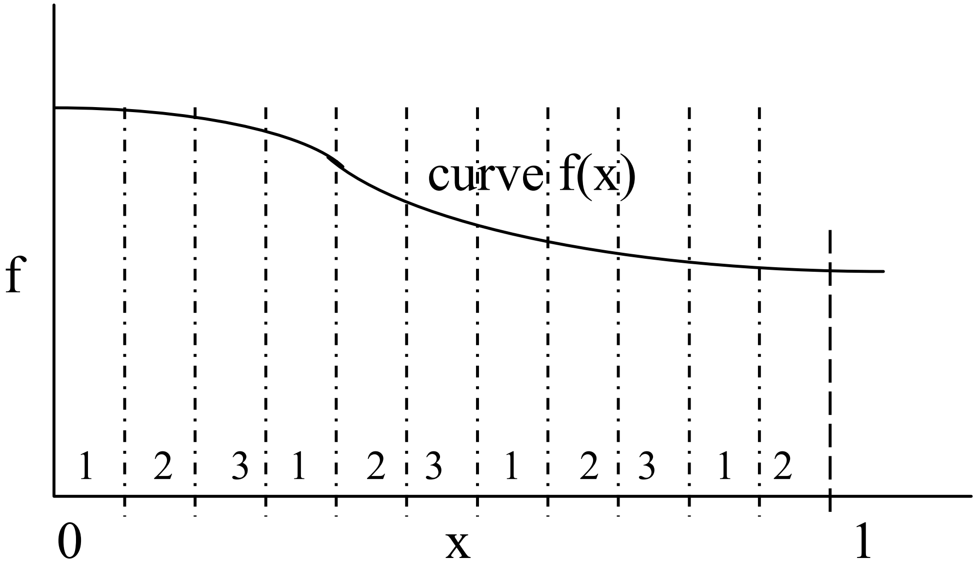
- This program calculates π -number by integrating $f(x) = 4 / (1+x^2)$
- Area under the curve is divided into rectangles and the rectangles are distributed to the processors.
- Let $f(x) = 4 / (1+x^2)$ then integrate f(x) from x = 0 to 1



Example: π -calculation – Using Rectangles

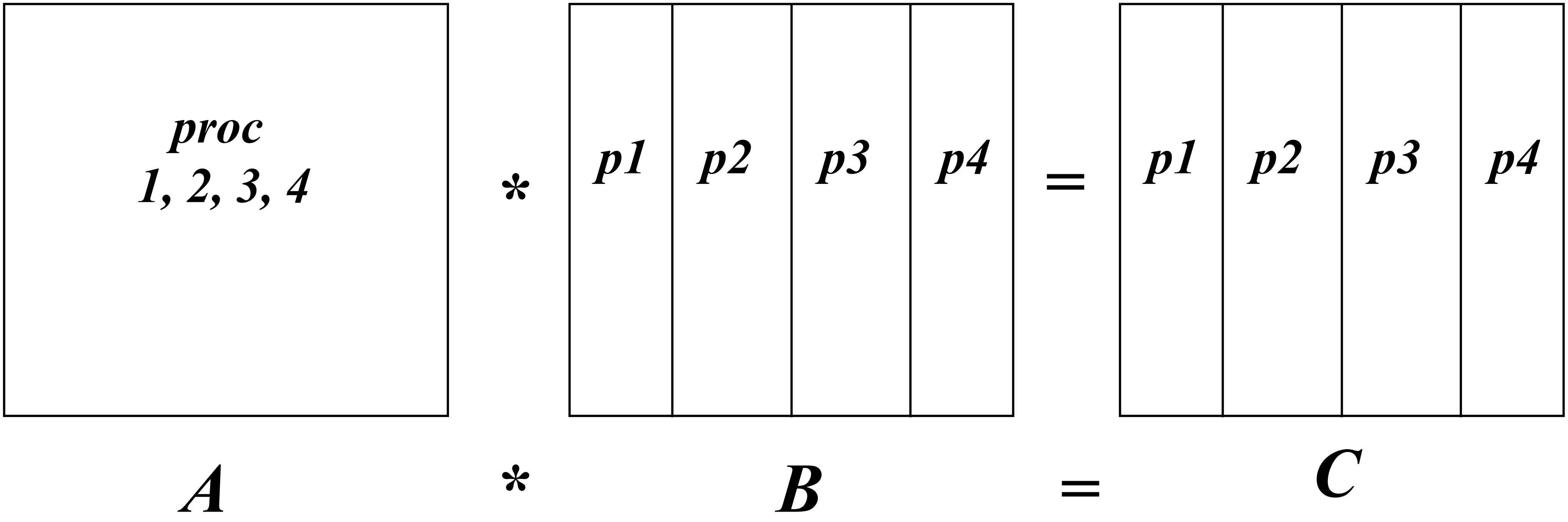
- Method: Divide area under curve into rectangles and distribute the rectangles to the processors
- Suppose there are 3 processors, how should the distribution be done?



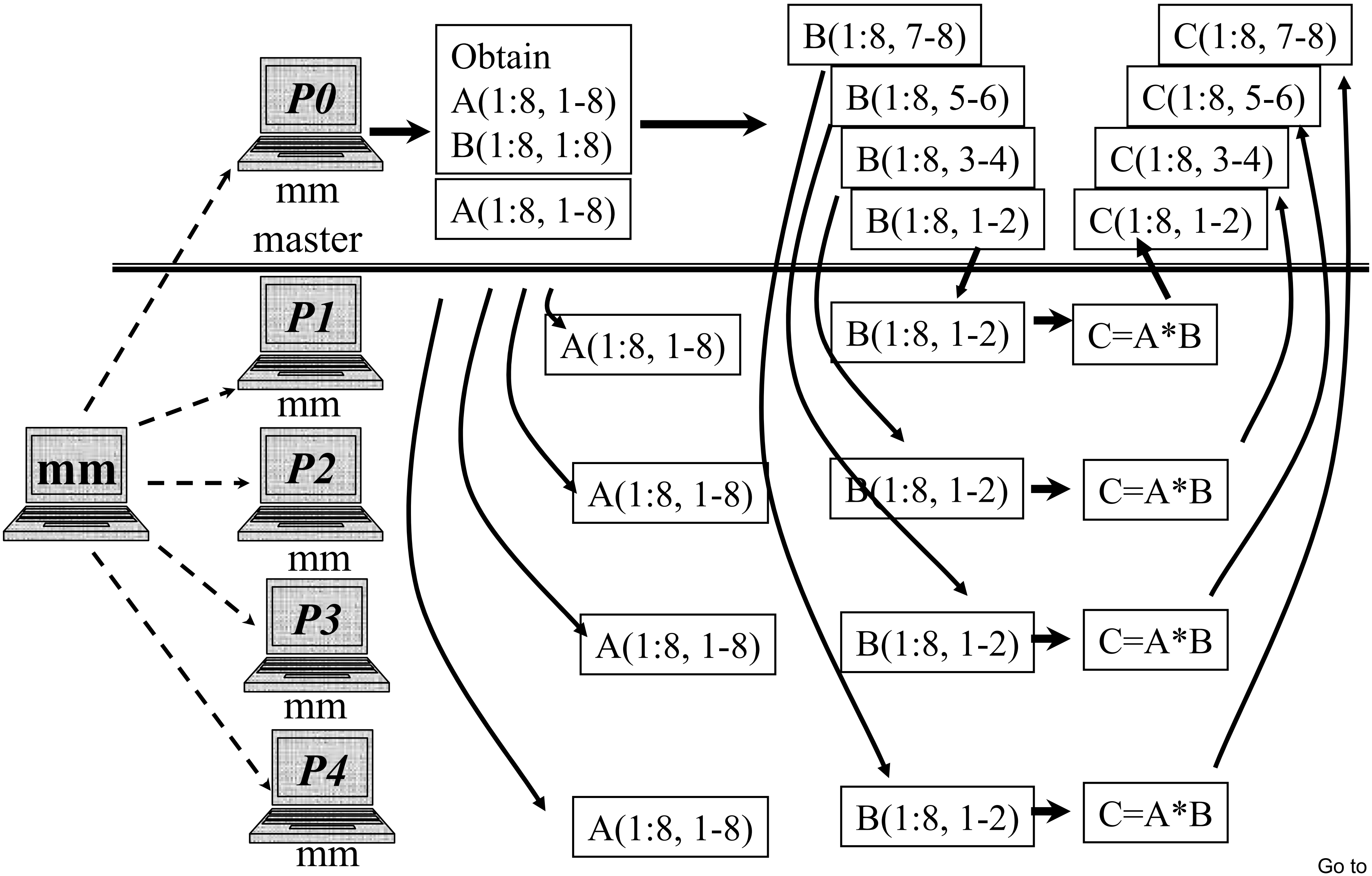


Example: Simple Matrix Multiplication Algorithm

- Matrix A is copied to every processor (FORTRAN)
- Matrix B is divided into blocks and distributed among processors
- Perform matrix multiplication simultaneously
- Output solutions

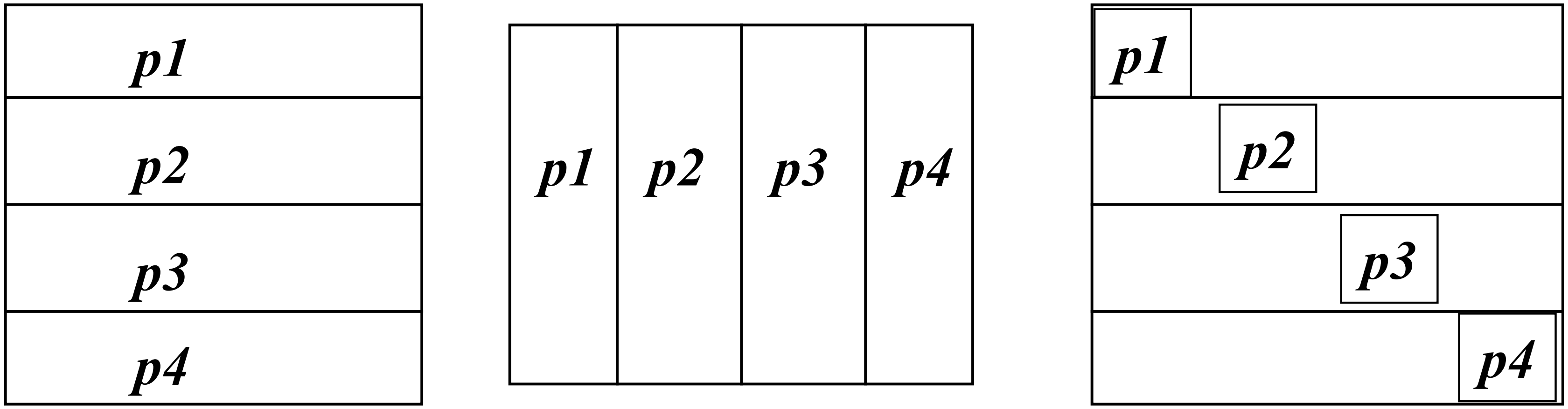


Example: Parallel Processing, matrix dimension n=8

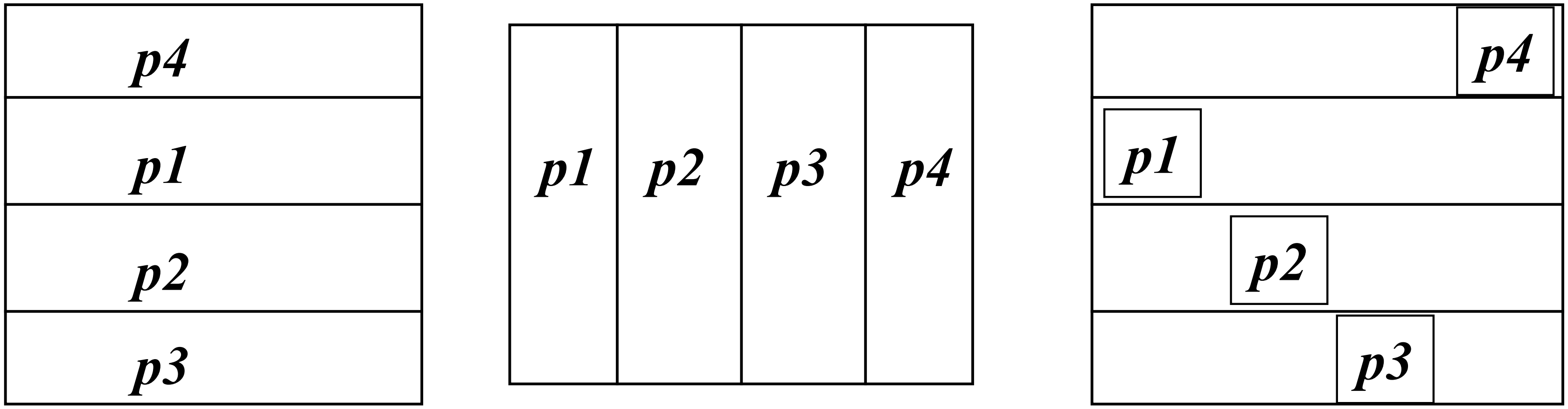


Example: Matrix Multiplication, steps 1-2

step 1



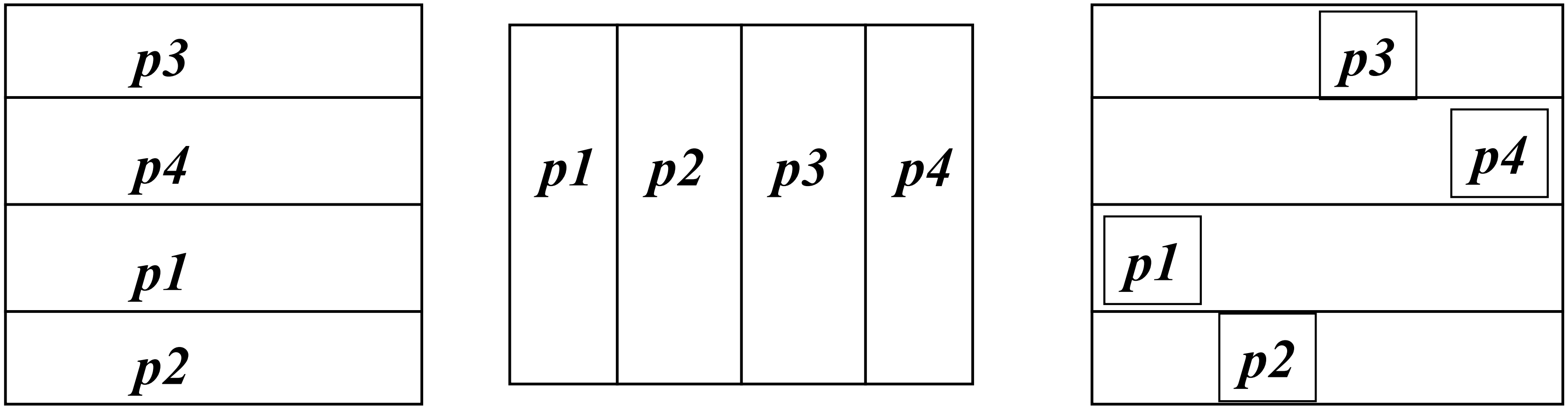
step 2



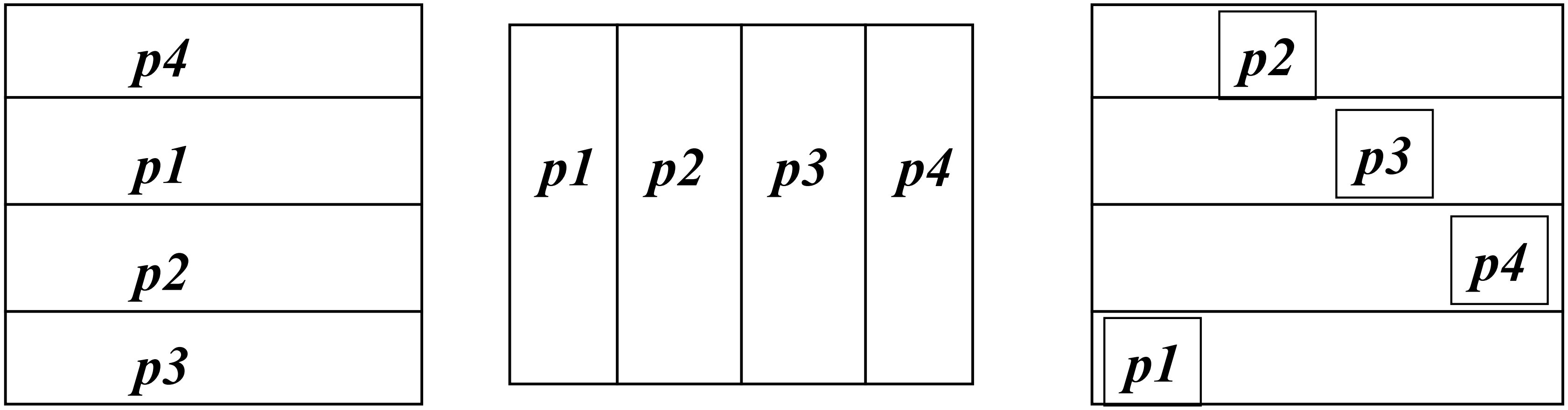
Go to Menu

Example: Matrix Multiplication, steps 3-4

step 3



step 4



Fox’s Algorithm (1)

Broadcast the diagonal element of block A in rows, perform multiplication.

A(0,0)	A(0,0)	A(0,0)
A(1,1)	A(1,1)	A(1,1)
A(2,2)	A(2,2)	A(2,2)

X

B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)

=

C(0,0)		
	C(1,1)	
		C(2,2)

C(0,0) = A(0,0)*B(0,0) + A(0,1)*B(1,0) + A(0,2)*B(2,0)
C(0,1) = A(0,0)*B(0,1) + A(0,1)*B(1,1) + A(0,2)*B(2,1)
C(0,2) = A(0,0)*B(0,2) + A(0,1)*B(1,2) + A(0,2)*B(2,2)
C(1,0) = A(1,0)*B(0,0) + A(1,1)*B(1,0) + A(1,2)*B(2,0)
C(1,1) = A(1,0)*B(0,1) + A(1,1)*B(1,1) + A(1,2)*B(2,1)
C(1,2) = A(1,0)*B(0,2) + A(1,1)*B(1,2) + A(1,2)*B(2,2)
C(2,0) = A(2,0)*B(0,0) + A(2,1)*B(1,0) + A(2,2)*B(2,0)
C(2,1) = A(2,0)*B(0,1) + A(2,1)*B(1,1) + A(2,2)*B(2,1)
C(2,2) = A(2,0)*B(0,2) + A(2,1)*B(1,2) + A(2,2)*B(2,2)

Fox’s Algorithm (2)

Broadcast next element of block A in rows, shift Bij in column, perform multiplication

A(0,1)	A(0,1)	A(0,1)
A(1,2)	A(1,2)	A(1,2)
A(2,0)	A(2,0)	A(2,0)

X

B(1,0)	B(1,1)	B(1,2)
B(2,0)	B(2,1)	B(2,2)
B(0,0)	B(0,1)	B(0,2)

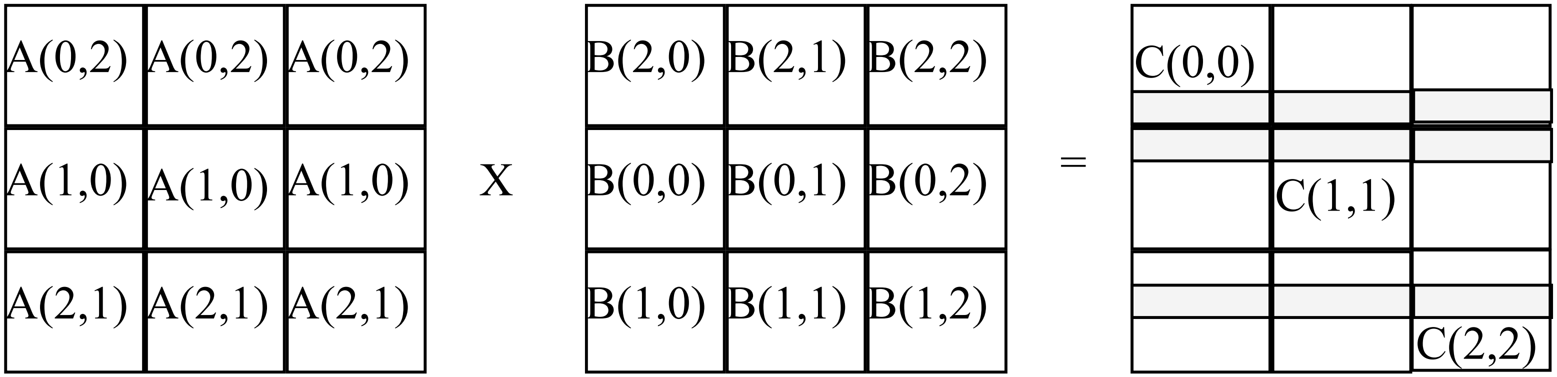
=

C(0,0)		
	C(1,1)	
		C(2,2)

C(0,0) = A(0,0)*B(0,0) + A(0,1)*B(1,0) + A(0,2)*B(2,0)
C(0,1) = A(0,0)*B(0,1) + A(0,1)*B(1,1) + A(0,2)*B(2,1)
C(0,2) = A(0,0)*B(0,2) + A(0,1)*B(1,2) + A(0,2)*B(2,2)
C(1,0) = A(1,0)*B(0,0) + A(1,1)*B(1,0) + A(1,2)*B(2,0)
C(1,1) = A(1,0)*B(0,1) + A(1,1)*B(1,1) + A(1,2)*B(2,1)
C(1,2) = A(1,0)*B(0,2) + A(1,1)*B(1,2) + A(1,2)*B(2,2)
C(2,0) = A(2,0)*B(0,0) + A(2,1)*B(1,0) + A(2,2)*B(2,0)
C(2,1) = A(2,0)*B(0,1) + A(2,1)*B(1,1) + A(2,2)*B(2,1)
C(2,2) = A(2,0)*B(0,2) + A(2,1)*B(1,2) + A(2,2)*B(2,2)

Fox’s Algorithm (3)

Broadcast next element of block A in rows, shift Bij in column, perform multiplication.



A(0,2)	A(0,2)	A(0,2)
A(1,0)	A(1,0)	A(1,0)
A(2,1)	A(2,1)	A(2,1)

X

B(2,0)	B(2,1)	B(2,2)
B(0,0)	B(0,1)	B(0,2)
B(1,0)	B(1,1)	B(1,2)

=

C(0,0)		
	C(1,1)	
		C(2,2)

$$
\begin{aligned}
C(0,0) &= A(0,0)*B(0,0) + A(0,1)*B(1,0) + A(0,2)*B(2,0) \\
C(0,1) &= A(0,0)*B(0,1) + A(0,1)*B(1,1) + A(0,2)*B(2,1) \\
C(0,2) &= A(0,0)*B(0,2) + A(0,1)*B(1,2) + A(0,2)*B(2,2) \\
C(1,0) &= A(1,0)*B(0,0) + A(1,1)*B(1,0) + A(1,2)*B(2,0) \\
C(1,1) &= A(1,0)*B(0,1) + A(1,1)*B(1,1) + A(1,2)*B(2,1) \\
C(1,2) &= A(1,0)*B(0,2) + A(1,1)*B(1,2) + A(1,2)*B(2,2) \\
C(2,0) &= A(2,0)*B(0,0) + A(2,1)*B(1,0) + A(2,2)*B(2,0) \\
C(2,1) &= A(2,0)*B(0,1) + A(2,1)*B(1,1) + A(2,2)*B(2,1) \\
C(2,2) &= A(2,0)*B(0,2) + A(2,1)*B(1,2) + A(2,2)*B(2,2)
\end{aligned}
$$

Outline: Collective Communications

- Overview
- Barrier Synchronization Routines
- Broadcast Routines
- MPI_Scatterv and MPI_Gatherv
- MPI_Allgather
- MPI_Alltoall
- Global Reduction Routines
- Reduce and Allreduce
- Predefined Reduce Operations

Overview

- Substitutes for a more complex sequence of point-to-point calls
- Involve all the processes in a process group
- Called by all processes in a communicator
- All routines block until they are locally complete
- Receive buffers must be exactly the right size
- No message tags are needed
- Divided into three subsets :
  - synchronization
  - data movement
  - global computation

Barrier Synchronization Routines

- To synchronize all processes within a communicator
- A node calling it will be blocked until all nodes within the group have called it.
- C:

```
 ierr = MPI_Barrier(comm)
```

- Fortran:

```
 call MPI_Barrier(comm, ierr)
```

- C++:

```
void MPI::Comm::Barrier() const;
```

Broadcast Routines

- One processor sends some data to all processors in a group

C:

```
ierr = MPI_Bcast(buffer,count,datatype,root,comm)
```

Fortran:

```
call MPI_Bcast(buffer,count,datatype,root,comm,ierr)
```

C++:

```
void MPI::Comm::Bcast(void* buffer, int count, const
MPI::Datatype& datatype, int root) const;
```

- The `MPI_Bcast` must be called by each node in a group, specifying the same communicator and root. The message is sent from the root process to all processes in the group, including the root process.

Scatter

- Data are distributed into n equal segments, where the i^{th} segment is sent to the i^{th} process in the group which has n processes.

C :

```
ierr = MPI_Scatter(&sbuff, scount, sdatatype,
&rbuf, rcount, rdatatype, root, comm)
```

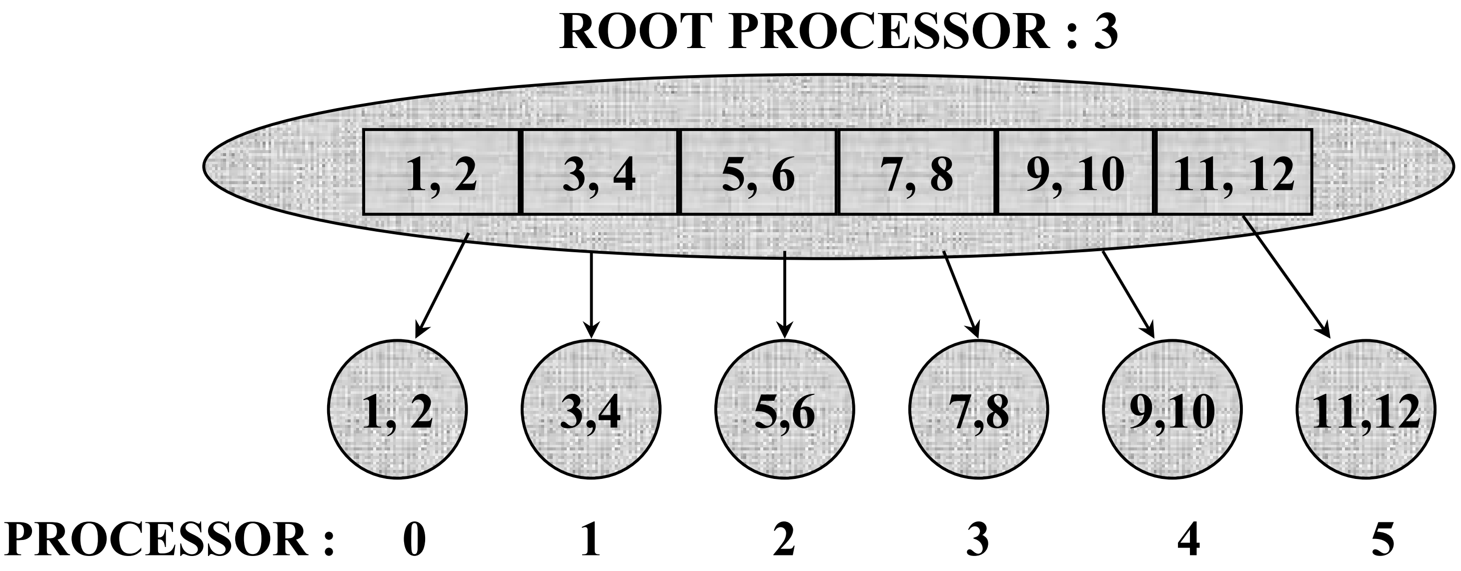
Fortran :

```
call MPI_Scatter(sbuff, scount, sdatatype, rbuf,
rcount, rdatatype, root , comm, ierr)
```

C++:

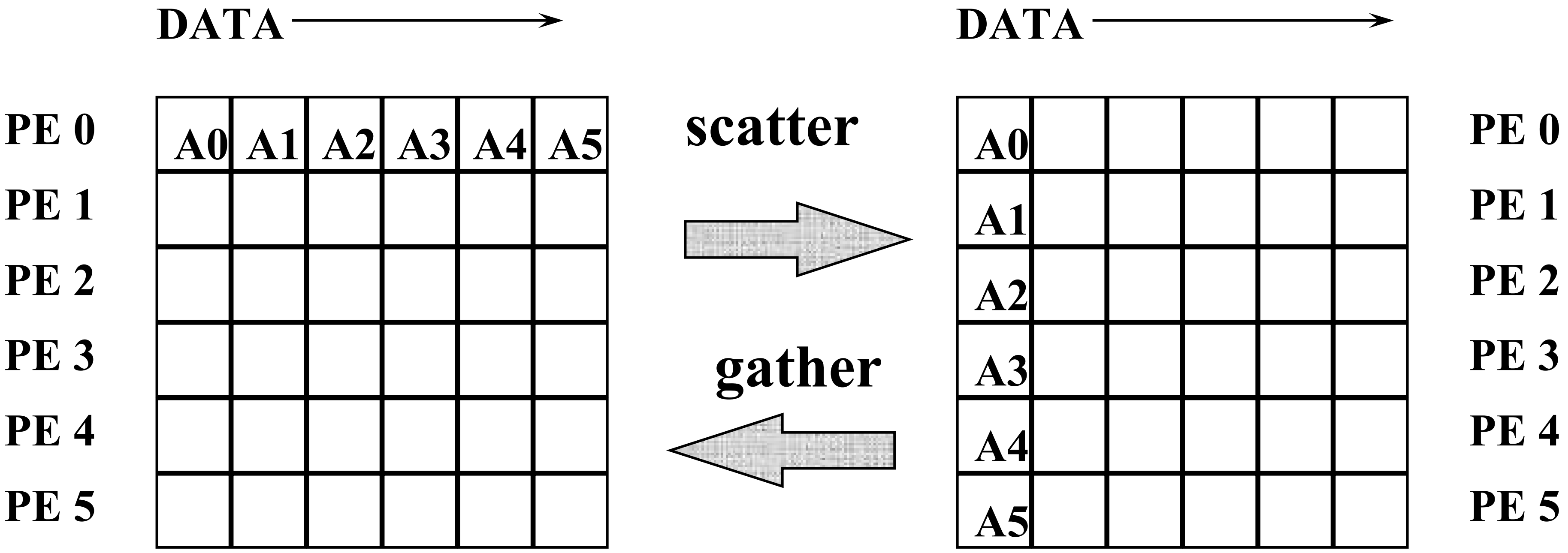
```
void MPI::Comm::Scatter(const void* sendbuf, int
sendcount, const MPI::Datatype& sendtype, void*
recvbuf, int recvcount, const MPI::Datatype&
recvtype, int root) const;
```

Example : `MPI_Scatter`



```
real sbuf(12), rbuf(2)
call MPI_Scatter(sbuf, 2, MPI_INT, rbuf, 2, MPI_INT, 3,
MPI_COMM_WORLD, ierr)
```

Scatter and Gather



DATA →

PE 0	A0	A1	A2	A3	A4	A5
PE 1						
PE 2						
PE 3						
PE 4						
PE 5						

scatter →

← gather

DATA →

A0						PE 0
A1						PE 1
A2						PE 2
A3						PE 3
A4						PE 4
A5						PE 5

Gather

- Data are collected into a specified process in the order of process rank, reverse process of scatter.

- C :

```
ierr = MPI_Gather(&sbuf, scount, sdtatatype,
&rbuf, rcount, rdatatype, root, comm)
```

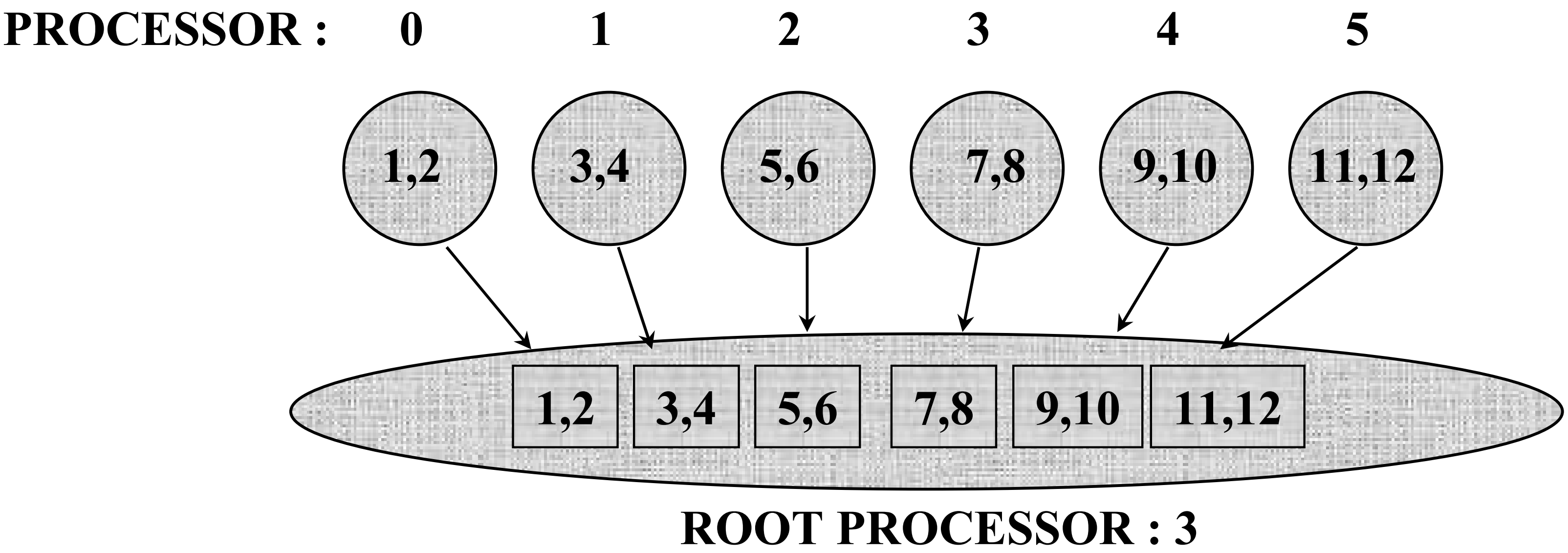
- Fortran :

```
call MPI_Gather(sbuff, scount, sdatatype,
rbuff, rcount, rdtatatype, root, comm, ierr)
```

- C++:

```
void MPI::Comm::Gather(const void* sendbuf, int
sendcount, const MPI::Datatype& sendtype, void*
recvbuf, int recvcount, const MPI::Datatype&
recvtype, int root) const;
```

Example : MPI_Gather



```
real sbuf(2),rbuf(12)
call MPI_Gather(sbuf,2,MPI_INT, rbuf, 2, MPI_INT, 3,
MPI_COMM_WORLD, ierr)
```

MPI_Scatterv and MPI_Gatherv

- allow varying count of data and flexibility for data placement

- C :

```
ierr = MPI_Scatterv( &sbuf, &scount, &displace,
sdatatype, &rbuf, rcount, rdatatype, root, comm)
```

```
ierr = MPI_Gatherv(&sbuf, scount, sdatatype, &rbuf,
&rcount, &displace, rdatatype, root, comm)
```

- Fortran :

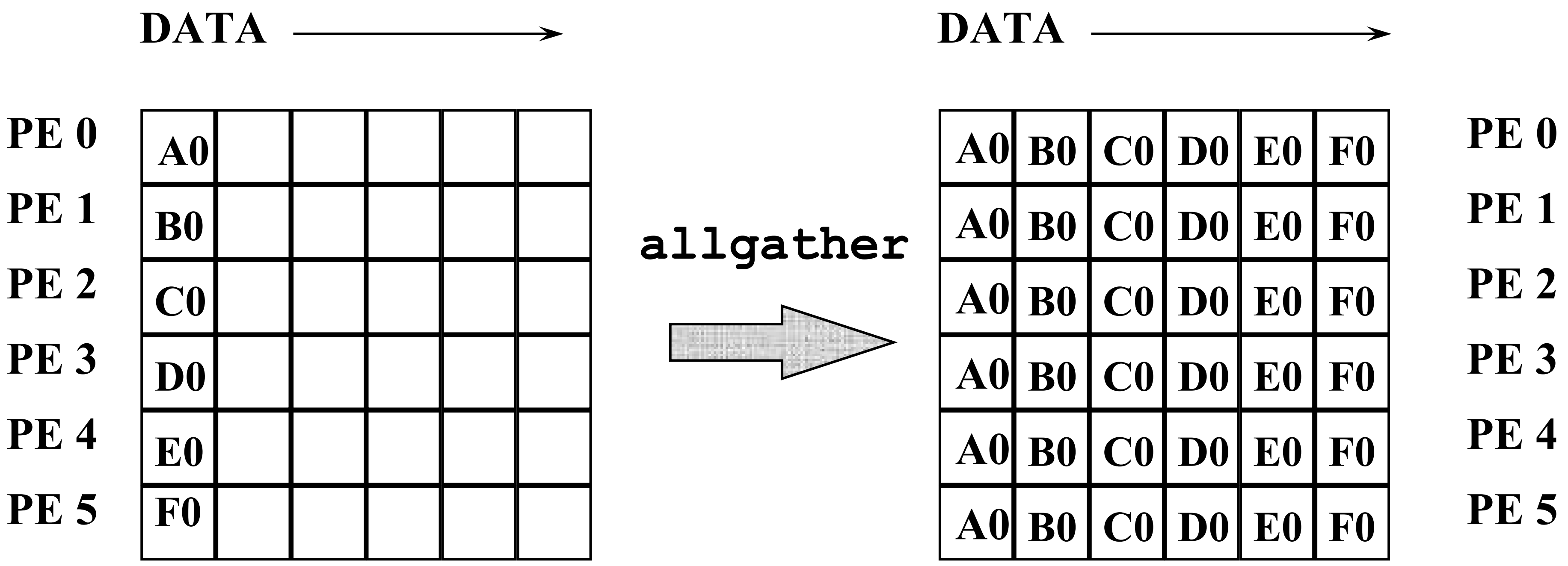
```
call MPI_Scatterv(sbuf,scount,displace,sdatatype, rbuf,
rcount, rdatatype, root, comm, ierr)
```

- C++:

```
void MPI::Comm::Scatterv(const void* sendbuf, const
int sendcounts[], const int displs[], const
MPI::Datatype& sendtype, void* recvbuf, int
recvcount, const MPI::Datatype& recvtype, int root)
const;
```

MPI_Allgather

```
ierr = MPI_Allgather(&sbuf, scount, stype, &rbuf,
         rcount, rtype, comm)
```



	DATA →					
PE 0	A0					
PE 1	B0					
PE 2	C0					
PE 3	D0					
PE 4	E0					
PE 5	F0					

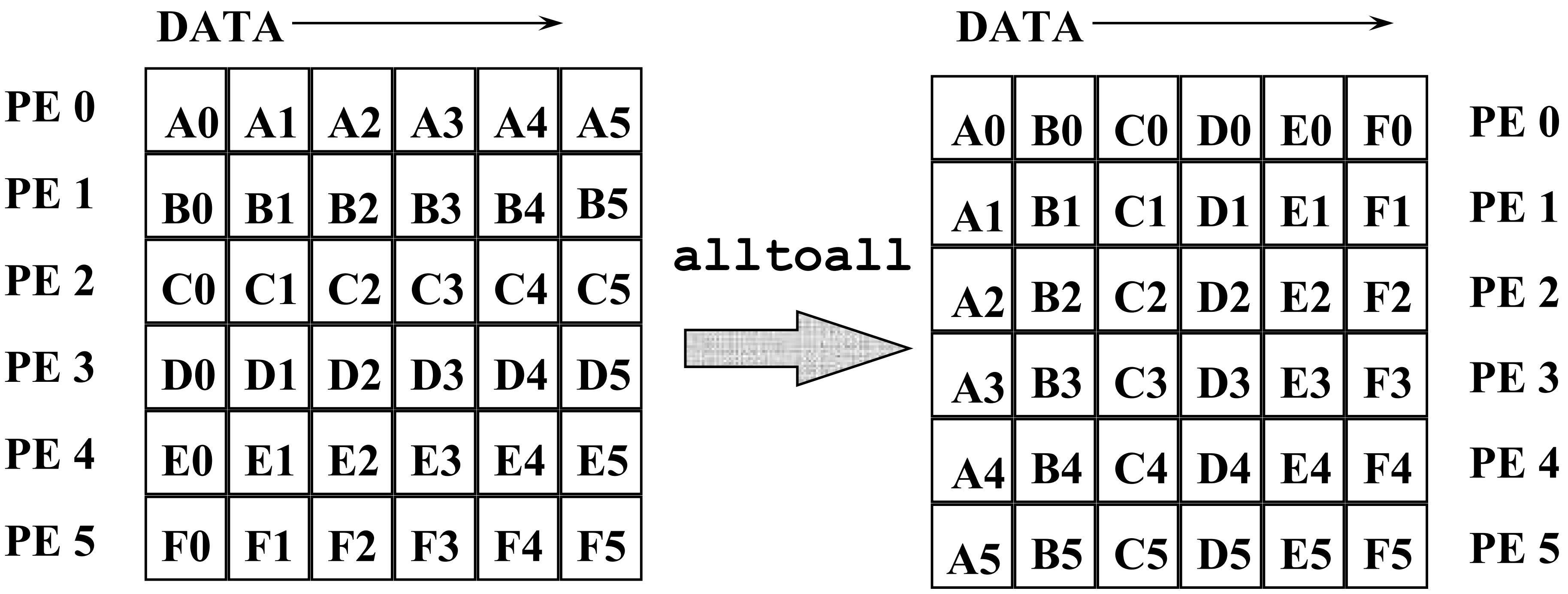
DATA →						
A0	B0	C0	D0	E0	F0	PE 0
A0	B0	C0	D0	E0	F0	PE 1
A0	B0	C0	D0	E0	F0	PE 2
A0	B0	C0	D0	E0	F0	PE 3
A0	B0	C0	D0	E0	F0	PE 4
A0	B0	C0	D0	E0	F0	PE 5

MPI_Alltoall

```
MPI_Alltoall(sbuf,scount,stype,rbuf,rcount,rtype,comm)
```

`sbuf:`	starting address of send buffer (*)
`scount:`	number of elements sent to each process
`stype:`	data type of send buffer
`rbuff:`	address of receive buffer (*)
`rcount:`	number of elements received from any process
`rtype:`	data type of receive buffer elements
`comm:`	communicator

All to All



	DATA →					
PE 0	A0	A1	A2	A3	A4	A5
PE 1	B0	B1	B2	B3	B4	B5
PE 2	C0	C1	C2	C3	C4	C5
PE 3	D0	D1	D2	D3	D4	D5
PE 4	E0	E1	E2	E3	E4	E5
PE 5	F0	F1	F2	F3	F4	F5

`alltoall`

DATA →						
A0	B0	C0	D0	E0	F0	PE 0
A1	B1	C1	D1	E1	F1	PE 1
A2	B2	C2	D2	E2	F2	PE 2
A3	B3	C3	D3	E3	F3	PE 3
A4	B4	C4	D4	E4	F4	PE 4
A5	B5	C5	D5	E5	F5	PE 5

Global Reduction Routines

- The partial result in each process in the group is combined together using some desired function.
- The operation function passed to a global computation routine is either a predefined MPI function or a user supplied function
- Examples :
  - global sum or product
  - global maximum or minimum
  - global user-defined operation

Reduce and Allreduce

```
MPI_Reduce(sbuf, rbuf, count, stype, op, root, comm)
MPI_Allreduce(sbuf, rbuf, count, stype, op, comm)
```

`sbuf`:	address of send buffer
`rbuf`:	address of receive buffer
`count`:	the number of elements in the send buffer
`stype`:	the datatype of elements of send buffer
`op`:	the reduce operation function, predefined or user-defined
`root`:	the rank of the root process
`comm`:	communicator

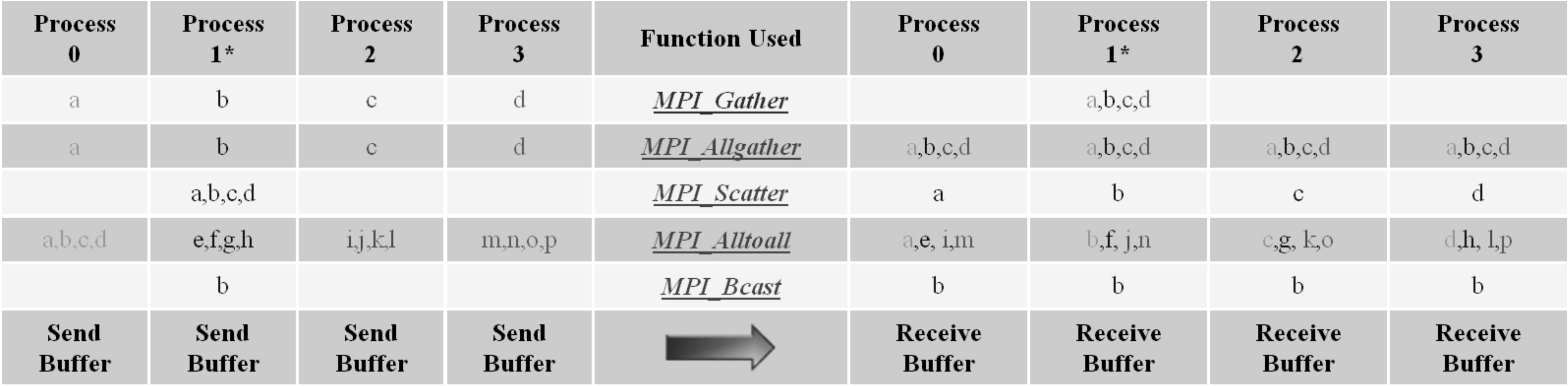
`MPI_Reduce`	returns results to single process
`MPI_Allreduce`	returns results to all processes in the group

Predefined Reduce Operations

MPI NAME	FUNCTION	MPI NAME	FUNCTION
`MPI_MAX`	Maximum	`MPI_LOR`	Logical OR
`MPI_MIN`	Minimum	`MPI_BOR`	Bitwise OR
`MPI_SUM`	Sum	`MPI_LXOR`	Logical exclusive OR
`MPI_PROD`	Product	`MPI_BXOR`	Bitwise exclusive OR
`MPI_LAND`	Logical AND	`MPI_MAXLOC`	Maximum and location
`MPI_LOR`	Bitwise AND	`MPI_MINLOC`	Minimum and location

Example: MPI Collective Communication Functions

- Collective communication routines are a group of MPI message passing routines to perform one (processor)-to-many (processors) and many-to-one communications.
- The first four columns on the left denote the contents of respective send buffers (*e.g.,* arrays) of four processes. The content of each buffer, shown here as alphabets, is assigned a unique color to identify its origin. For instance, the alphabets in blue indicate that they originated from process 1. The middle column shows the MPI routines with which the send buffers are operated on. The four columns on the right represent the contents of the processes' receive buffers resulting from the MPI operations.



Process 0	**Process 1***	**Process 2**	**Process 3**	**Function Used**	**Process 0**	**Process 1***	**Process 2**	**Process 3**
a	b	c	d	*MPI_Gather*		a,b,c,d		
a	b	c	d	*MPI_Allgather*	a,b,c,d	a,b,c,d	a,b,c,d	a,b,c,d
	a,b,c,d			*MPI_Scatter*	a	b	c	d
a,b,c,d	e,f,g,h	i,j,k,l	m,n,o,p	*MPI_Alltoall*	a,e, i,m	b,f, j,n	c,g, k,o	d,h, l,p
	b			*MPI_Bcast*	b	b	b	b
Send Buffer	**Send Buffer**	**Send Buffer**	**Send Buffer**	➡	**Receive Buffer**	**Receive Buffer**	**Receive Buffer**	**Receive Buffer**

Outline: Derived Datatypes

- – Overview
- – Datatypes
- – Defining Datatypes
- – MPI_Type_vector
- – MPI_Type_struct

Overview

- To provide a portable and efficient way of communicating mixed types, or non-contiguous types in a single message
  - Datatypes are built from the basic MPI datatypes
  - MPI datatypes are created at run-time through calls to MPI library
- Steps required
  - construct the datatype : define shapes and handle
  - allocate the datatype : commit types
  - use the datatype : use constructors
  - deallocate the datatype : free space

Go to Menu

Datatypes

- Basic datatypes :

  `MPI_INT, MPI_REAL, MPI_DOUBLE, MPI_COMPLEX, MPI_LOGICAL, MPI_CHARACTER, MPI_BYTE,…`

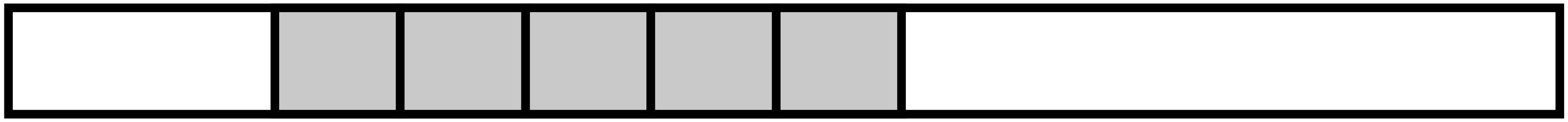
- MPI also supports array sections and structures through general datatypes. A general datatypes is a sequence of basic datatypes and integer byte displacements. These displacements are taken to be relative to the buffer that the basic datatype is describing. ==> *typemap*

Datatype = { (type0, disp0) , (type1, disp1) , ….., (typeN, dispN)}

Defining Datatypes

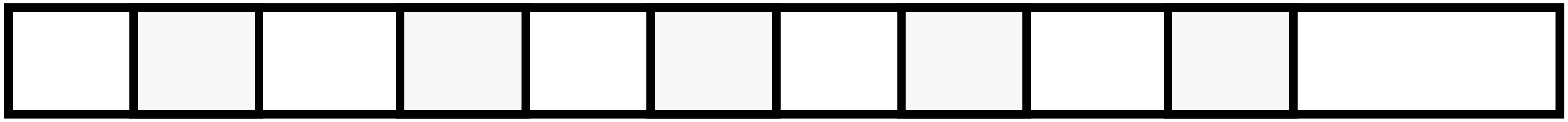
```
MPI_Type_contiguous(count,oldtype,newtype,ierr)
```

‘`count`’ copies of ‘`oldtype`’ are concatenated



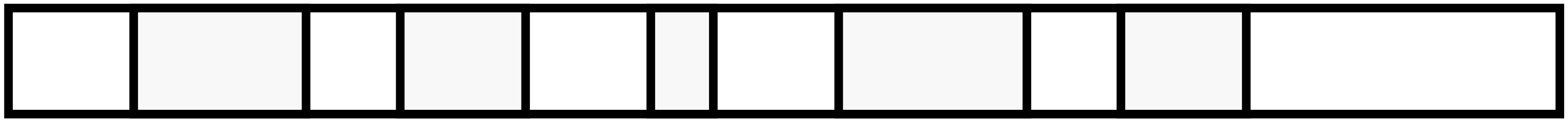
```
MPI_Type_vector(count,buffer,strides,oldtype,newtype,ierr)
```

‘`count`’ blocks with ‘`blen`’ elements of ‘`oldtype`’ spaced by ‘`stride`’



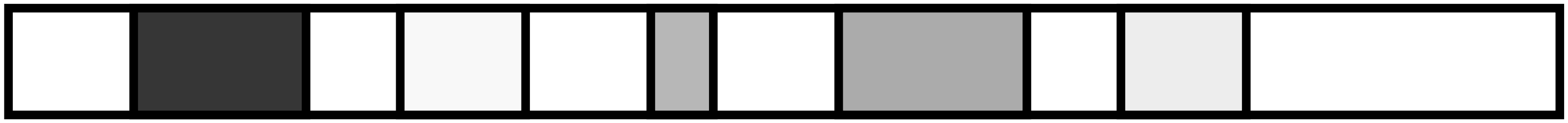
```
MPI_Type_indexed(count, buffer, strides, oldtype,newtype,ierr)
```

Extension of vector: varying ‘`blens`’ and ‘`strides`’



```
MPI_Type_struct(count,buffer, strides,oldtype,newtype,ierr)
```

extension of indexed: varying data types allowed



MPI_Type_vector

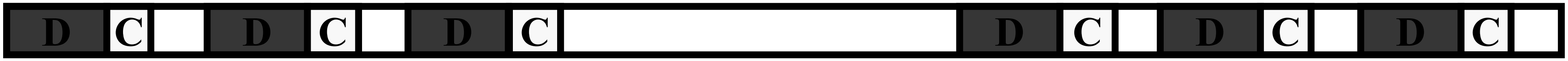
- It replicates a datatype, taking blocks at fixed offsets.

```
MPI_Type_vector(count,blocklen,stride,oldtype,newtype)
```

- The new datatype consists of :
  - `count` : number of blocks
  - each block is a repetition of `blocklen` items of `oldtype`
  - the start of successive blocks is offset by `stride` items of `oldtype`

If `count` = 2, `stride` = 4, `blocklen` = 3, `oldtype` = {`(double,0),(char,8)`}

```
   newtype =  {(double,0),(char,8),(double,16),(char,24),
              (double,32),(char,40),(double,64),
              (char,72),(double,80),(char,88),
              (double,96),(char,104)}
```



Example: Datatypes

```
#include <mpi.h>
{
   float mesh[10][20];
   int dest, tag;
   MPI_Datatype newtype;
   /* Do this once */
   MPI_Type_vector(    10,     /* # column elements */
                        1,     /* 1 column only */
                       20,     /* skip 20 elements */
                MPI_FLOAT,     /* elements are float */
               &newtype);      /* MPI derived datatype */
MPI_Type_commit(&newtype);     /* Do this for every new message */
MPI_Send(&mesh[0][19], 1, newtype, dest, tag, MPI_COMM_WORLD);
}
```

MPI_Type_struct

- To gather a mix of different datatypes scattered at many locations in space into one datatype

  `MPI_Type_struct(count, array_of_blocklength, array_of_displacements, array_of_types, newtype, ierr)`

  - **`count` : number of blocks**
  - **`array_of_blocklength (B)` : number of elements in each block**
  - **`array_of_displacements (I)` : byte of displacement of each block**
  - **`array_of_type (T)` : type of elements in each block**

```
If count = 3
   T = {MPI_FLOAT, type1, MPI_CHAR}
   I = {0,16,26}
   B = {2 , 1 , 3}
   type1 = {(double,0),(char,8)}
   newtype = {(float,0),(float,4),(double,16),(char,24),
        (char,26),(char,27),(char,28)}
```

Example

```
Struct{ char    display[50];
        int     maxiter;
        double  xmin, ymin, xmax, ymax;
        int     width, height;         } cmdline;

/* set up 4 blocks */
int            blockcounts[4] = {50, 1, 4, 2};
MPI_Datatype   types[4];
MPI_Aint       displs[4];
MPI_Datatype   cmdtype;

/* initialize types and displacements with addresses of items */

MPI_Address(&cmdline.display, &displs[0];
MPI_Address(&cmdline.maxiter, &displs[1];
MPI_Address(&cmdline.xmin, &displs[2];
MPI_Address(&cmdline.width, &displs[3];
types[0] = MPI_CHAR; types[1]=MPI_INT;
types[2]=MPI_DOUBLE; types[3]=MPI_INT;

for ( i = 3 ; i >= 0; i--)  displs[i] -= displs[0]
MPI_Type_struct(4, blockcounts, displs, types, &cmdtype);
MPI_Type_commit(&cmdtype);
```

Outline: Resources for Users

- Getting Started
- Advanced Topics
- More Information
- man pages and MPI web-sites
- MPI Books

Resources for Users: Getting Started

- About Jaguar

  http://www.nccs.gov/computing-resources/jaguar/

- Quad Core AMD Opteron Processor Overview

  http://www.nccs.gov/wp-content/uploads/2008/04/amd_craywkshp_apr2008.pdf

- PGI Compilers for XT5

  http://www.nccs.gov/wp-content/uploads/2008/04/compilers.ppt

- NCCS Training & Education – archives of NCCS workshops and seminar series, HPC/parallel computing references

  http://www.nccs.gov/user-support/training-education/

- 2009 Cray XT5 Quad-core Workshop

  http://www.nccs.gov/user-support/training-education/workshops/2008-cray-xt5-quad-core-workshop/

Resources for Users: Advanced Topics

- Debugging Applications Using TotalView

  http://www.nccs.gov/user-support/general-support/software/totalview

- Using Cray Performance Tools - CrayPat

  http://www.nccs.gov/computing-resources/jaguar/debugging-optimization/cray-pat/

- I/O Tips for Cray XT4

  http://www.nccs.gov/computing-resources/jaguar/debugging-optimization/io-tips/

- NCCS Software

  http://www.nccs.gov/computing-resources/jaguar/software/

Resources for Users: More Information

- NCCS website

  http://www.nccs.gov/

- Cray Documentation

  http://docs.cray.com/

- Contact us

  help@nccs.gov

Resources for Users: `man` pages and MPI web-sites

- There are man pages available for MPI which should be installed in your MANPATH. The following man pages have some introductory information about MPI.

```
% man MPI
% man cc
% man ftn
% man qsub
% man MPI_Init
% man MPI_Finalize
```

- MPI man pages are also available online.
  http://www.mcs.anl.gov/mpi/www/
- Main MPI web page at Argonne National Laboratory
  http://www-unix.mcs.anl.gov/mpi
- Set of guided exercises
  http://www-unix.mcs.anl.gov/mpi/tutorial/mpiexmpl
- MPI tutorial at Lawrence Livermore National Laboratory
  https://computing.llnl.gov/tutorials/mpi/
- MPI Forum home page contains the official copies of the MPI standard.
  http://www.mpi-forum.org/

Resources for Users: MPI Books

- Books on and about MPI
  - Using MPI, 2nd Edition by William Gropp, Ewing Lusk, and Anthony Skjellum, published by MIT Press ISBN 0-262-57132-3. The example programs from this book are available at ftp://ftp.mcs.anl.gov/pub/mpi/using/UsingMPI.tar.gz.
    The Table of Contents is also available. An errata for the book is available. Information on the first edition of Using MPI is also available, including the errata. Also of interest may be The LAM companion to ``Using MPI..." by Zdzislaw Meglicki (gustav@arp.anu.edu.au).
  - Designing and Building Parallel Programs is Ian Foster's online book that includes a chapter on MPI. It provides a succinct introduction to an MPI subset. (ISBN 0-201-57594-9; Published by Addison-Wesley>)
  - **MPI: The Complete Reference**, by Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra, The MIT Press .
  - MPI: The Complete Reference - 2nd Edition: Volume 2 - The MPI-2 Extensions, by William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir, The MIT Press.
  - Parallel Programming With MPI, by Peter S. Pacheco, published by Morgan Kaufmann.
  - RS/6000 SP: Practical MPI Programming, by Yukiya Aoyama and Jun Nakano (IBM Japan), and available as an IBM Redbook.
  - **Supercomputing Simplified: The Bare Necessities for Parallel C Programming with MPI**, by William B. Levy and Andrew G. Howe, ISBN: 978-0-9802-4210-2. See the website for more information.