MPI: Message Passing Interface
An Introduction

S. Lakshmivarahan
School of Computer Science

- MPI: A specification for message passing libraries designed to be a standard for distributed memory message passing, parallel computing

- Released in the summer 1994 - FORTRAN and C versions

- Not a language, a collection of subroutines

- Knowledge of MPI *DOES NOT* imply knowledge of parallel programming

- Precursor to MPI: PVM, EXPRESS, PARMACS, p4

Basic concepts:

•Processor vs. process

• Processor is the stand alone computer

• Process is a task represented by a piece of program

•One process per processor – else loss of parallelism

•In MPI need to allocate a fixed number of processors

•Cannot dynamically spawn processes

Programming Model

SPMD
Single Program
Multiple Data

Each processor does the same computation on different data sets

- DNA matching
- IRS agent

MPMD
Multiple Program
Multiple Data

Different processors doing different computations on different data set

- University-academic/administration

Parallel Programming requires knowledge of

•Processors – network topology

•Algorithms

•Data structures

•Possible patterns of communications

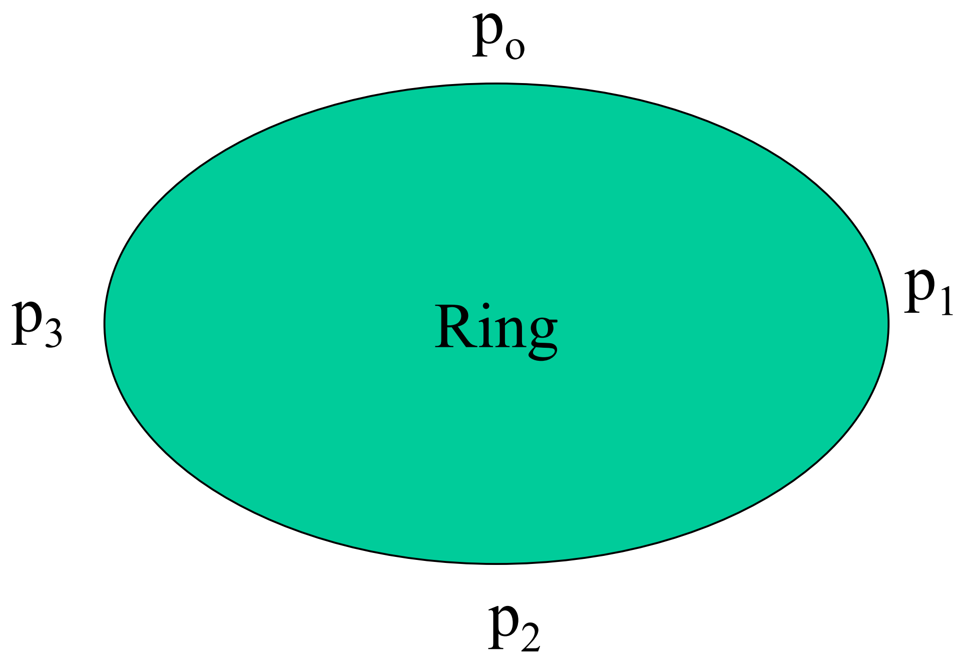
- A functional view of the parallel processor is essential for successful parallel programming experience

- To drive a car a functional view would help - on average automobile has over 2000 parts

- Functional view is – engine, transmission, throttle, break, steering, gas tank, lights, wiper, heater/air conditioner, etc

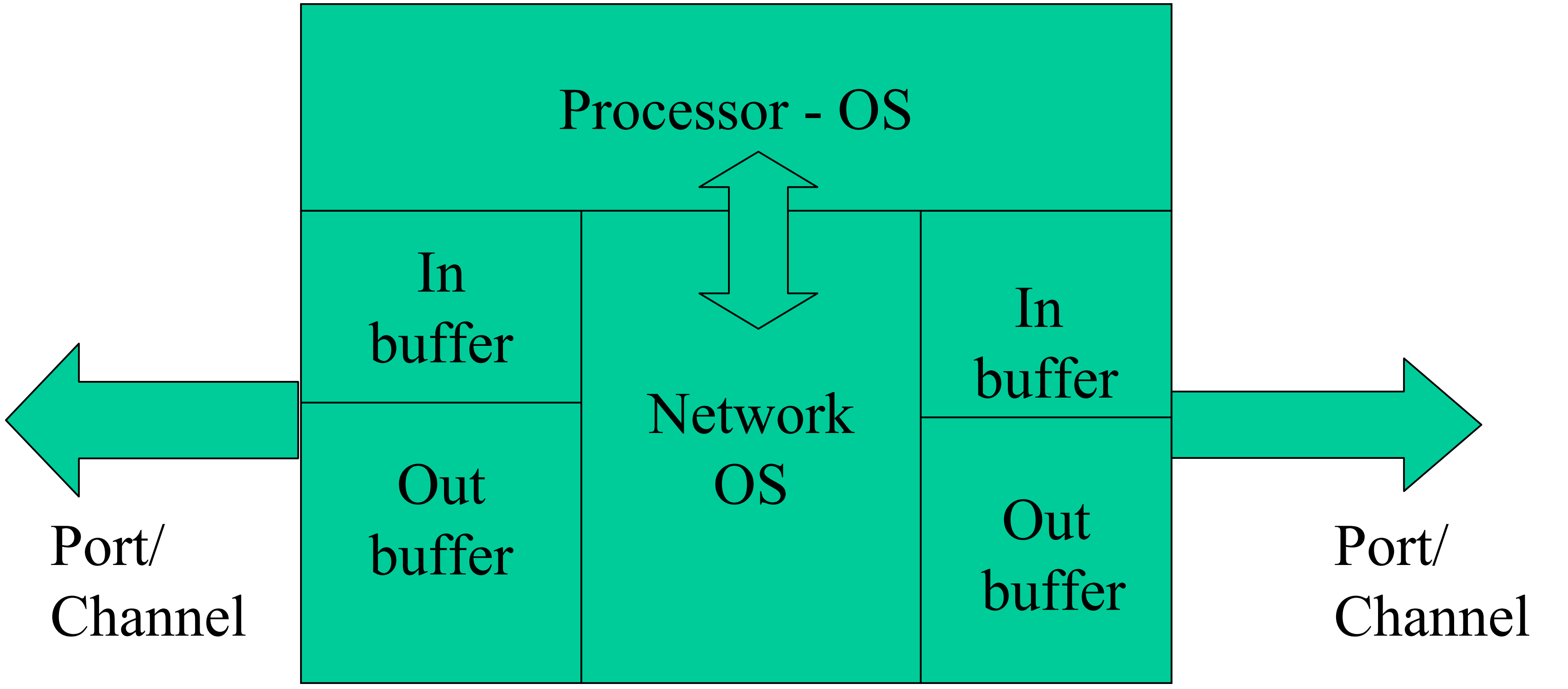
Processors are connected in a network

Physical network topology – Ring, Star, Hypercube, Toroid



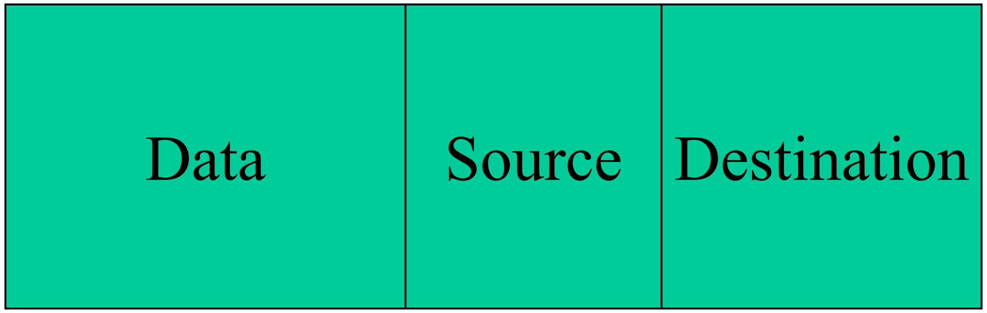
- •Each processor has two neighbors – two ports
- •One processor is designated as the master – handles I/O
- •Programs/data to all the other processors via master

A View of Processor



Processor and NOS communicate by interrupt

Packets



Packet size system specified – 4Kbytes

Packets enter the networks via the in/out buffers

In all MPI has 129 functions - subroutines

Many of these require a handful of parameters

Classification

- Point to Point
- Collective communication within a group

A few – about a dozen are sufficient to get started

include “mpif.h”

- Not an MPI command but every program needs this and is the first line in each program

- Makes the MPI subroutine library available to the given program

Structure of MPI commands:

MPI_ command-name (parameters)

These are names of subroutines to be called using the FORTRAN Call statement

MPI standard does not specify the implementation details

First command: MPI_INIT (error)

- initiates an MPI session

Last command: MPI_FINALIZE (error)

- terminates an MPI session

error – integer variable

MPI_INITIALIZED (flag, error)

•Checks initialization of an MPI session
flag – logical variable
error – integer variable

MPI_COMM_SIZE (MPI_COMM_WORLD, nprocs, error)

$\longleftrightarrow$

Communicator/
Handle. Integer

•Determines the total number of processes in a session
nprocs, error - integer

MPI_COMM_RANK (MPI_COMM_WORLD, myrank,error)

Determines the rank of each processor (assigned by the system) involved in a given MPI session

After this command each processor can print their rank

myrank, error - integer

A sample program:

```
/* This program uses the five MPI commands*/
Program Sample_1
include “mpif.h”
Integer myrank, nprocs, error

call MPI_INIT (error)
call MPI_COMM_SIZE ( MPI_COMM_WORLD, nprocs, error)
call MPI_COMM_RANK (MPI_COMM_WORLD, myrank, error)

Print *,  “My rank is  “, myrank”
If ( myrank .eq. 0) “Total number of processes =“, nprocs

call MPI_FINALIZE (error)

stop
end
```

How many processors to use in a given session?
How to run an MPI program?

Example: Let myprog be the name of the executable for our sample program

```
 myprog –n 4  /*Executes this program on 4 processors*/
```

```
Output:   My rank is 2    /* output order is not defined*/
          My rank is 1
          My rank is 0
          My rank is 3
```

Point to point communication:

MPI_SEND – to send data – uses 7 parameters

- variable name denoting the data items to be sent – real, array
- count denoting the number of data items being sent- integer
- MPI_INTEGER
- destination – processor id
- tag – integer to indicate the assigned type
- MPI_COMM_WORLD
- error

MPI_RECV – to receive data- uses 8 parameters

- data item to be received –real array

- count denoting the number of data items being sent- integer

- MPI_INTEGER

- source – processor id

- tag – integer to indicate the assigned type

- MPI_COMM_WORLD

- status to indicate if the receive action is complete

- error

There are two modes: **block vs. non-block**

In **block send/receive** control is returned is returned to the

calling program when it is safe to use the sending/receiving buffer

In **non-block mode** control is returned even before it is safe to use

the buffer

Problem statement:

There are two processors labeled 0 and 1 – master is 0

Master sends a data item (say the value of the year 2002) to processor 1

This is done using the SPMD mode of programming where all the processors have the same copy of the program.

$$\begin{tikzcd} p_0 \arrow[r] & p_1 \end{tikzcd}$$

A logical view of the algorithm

```
Program Sample_2
/* A sample of SPMD program – next 4 slides*/

include “mpif.h”
integer myrank, nprocs, error, source, dest, count, tag, year
integer status (MPI_STATUS_SIZE)

call MPI_INT(error)
call MPI_COMM_SIZE (MPI_COMM_WORLD, nprocs, error)
call MPICOMM_RANK (MPI_COMM_WORLD, myrank, error)

If(nprocs .ne. 2) then    /* can use only two processors*/
If (myrank.eq.0) write(6,*) “Nprocs .ne. 2, stop”
call MPI_FINALIZE(error)
Stop
End if
```

```
If ( myrank .eq.0) then
Year  = 2002   /* the data being sent*/
Dest = 1          /* Destination is processor 1*/
Count =1        /* Number of data items*/
Tag=99         /*  This message is asigned a tag value of 99 */
Write (6,*) “My rank is, “ myrank ,  “year=“ year
Call MPI_SEND(Year, count, MPI_INTEGER, dest, tag,
1                                MPI_COMM_WORLD, error)
Endif
```

/* Note that identical copy of the program resides in each processor and what one does at what time depends on the rank of the processor. Tag values range from 0 to 32767 */

```
If ( myrank .eq. 1) then
Source = 0
Count = 1
Tag = 99
 call MPI_RECV (year, count, MPI_INTEGER, source, tag,
1                          MPI_COMM_WORLD, Status, error)

Write(6,*) “Myrank,”, myrank, “year = “, year

Call MPI_GET_COUNT(status, MPI_Integer, count,error)
Write(6,*) “No of Integers Received = “, count
Endif
```

```
/* Need to complete the program Sample_2 */

Call MPI_FINALIZE (error)
Stop
End Program Sample_2
```

System decides which of the two processors you get
for this session - may depend on other active jobs

The actual communication time depends on if the
assigned processors are physically neighbors in the network.

If they are, then communication overhead is a minimum, else
there is a penalty for mismatch.

Notice that the program will still work and give results but
may take a longer time - affecting the performance

In the previous example we have specified that processor 1 receive data from a specified source. We can instead use MPI_ANY_SOURCE to receive data from any source

We can also use MPI_ANY_TAG can be used to receive data with the specific tag as was used in the above example

Global Communication primitives:

MPI_Broadcast: master sends the **same** data items to all others

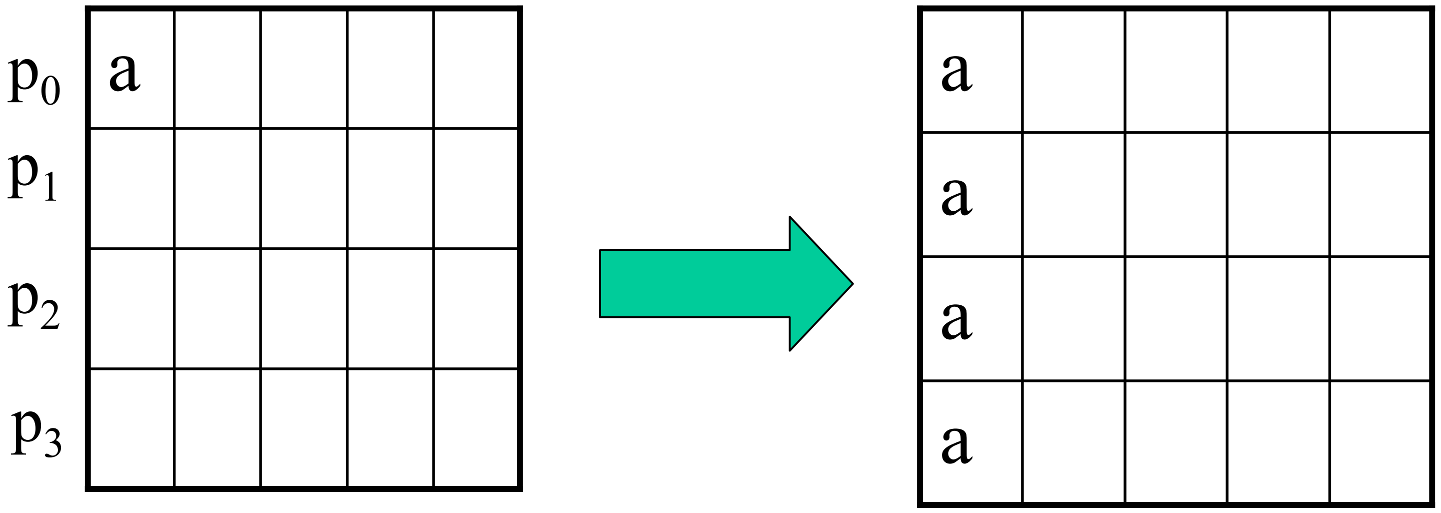
MPI_Scatter: master sends **different** data items to all others
(This is known as personalized communication)

MPI_Gather: all the processors send their results to the master

MPI_Send/Recv: all neighbors communicate – used in finite difference calculations

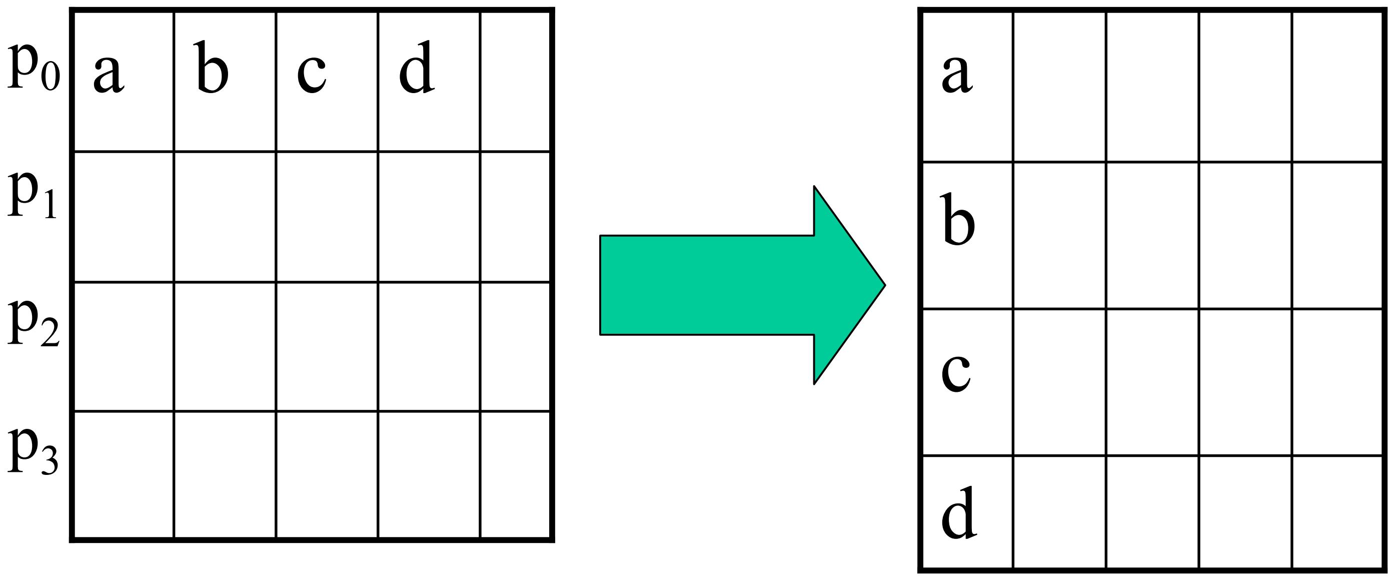
MPI_BARRIER

One to all broadcast: master broadcast its rank to all others



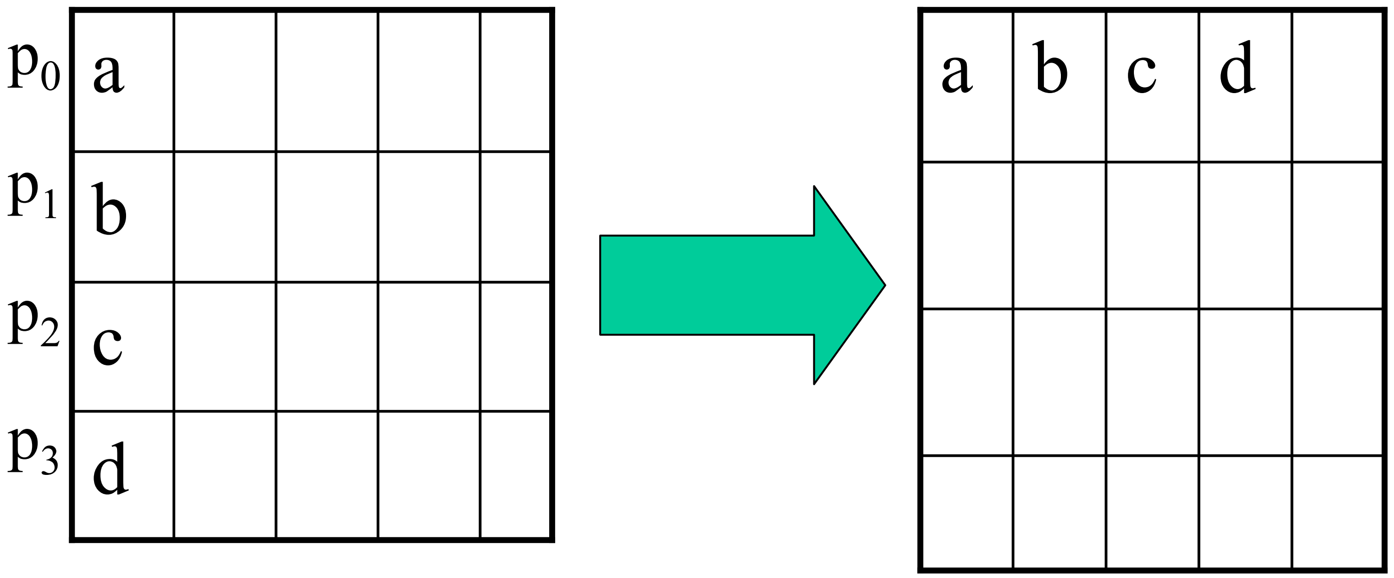
```
MPI_BCAST(myrank, 1, MPI_INTEGER, 0,
                              MPI_COMM_WORLD, error)
```

Scatter: Personalized communication



MPI_SCATTER (has 9 arguments)

GATHER: all the processors send to master



MPI_Gather (has 9 arguments)

Writing a parallel is like writing music for an ensemble

References:

W.Gropp, E. Lusk, and A. Skjellum (1995) Using MPI: Portable Parallel Programming with the Message Passing Interface, **MIT Press**

Ian Foster (1995) Design and Building Parallel Programs, **Addison Wesley**

S.Lakshmivarahan and S. K. Dhall (2002) Programming in FORTRAN 90/95, **Pearson Publishing**