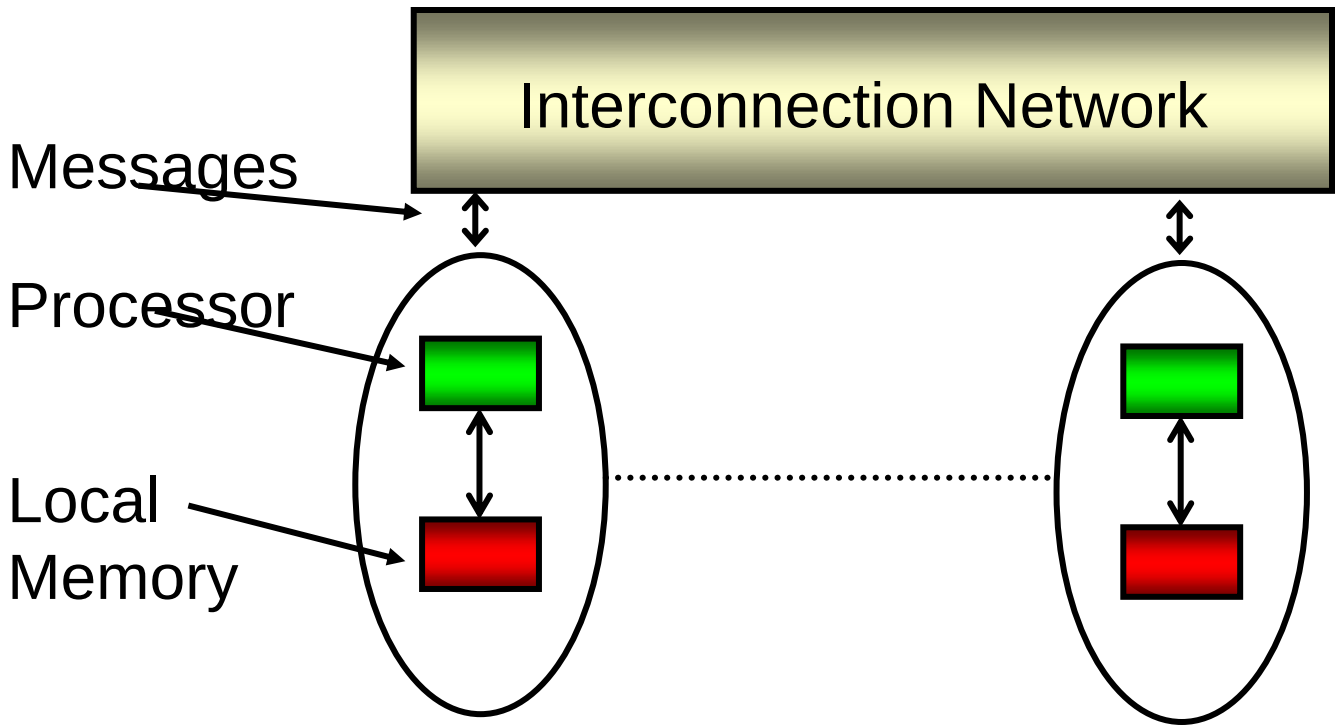
Message-Passing Computing for Distributed Multi-Computers

A review of basic concepts

Message-Passing Multicomputer

Complete computers connected through an interconnection network:



Programming

Programming a message-passing multicomputer can be achieved by

- Designing a special parallel programming language
- Extending the syntax/reserved words of an existing sequential high-level language to handle message passing
- Using an existing sequential high-level language and providing a ***library of external procedures for message passing***

Programming

Involves dividing problem into parts (domain or functional) that are intended to be executed simultaneously to solve the problem

Each part executed by separate computers

Parts (processes) communicate by sending messages - the only way to distribute data and collect result

Message Passing Parallel Programming Software Tools

Parallel Virtual Machine (PVM) - developed in late 1980's. Became very popular.

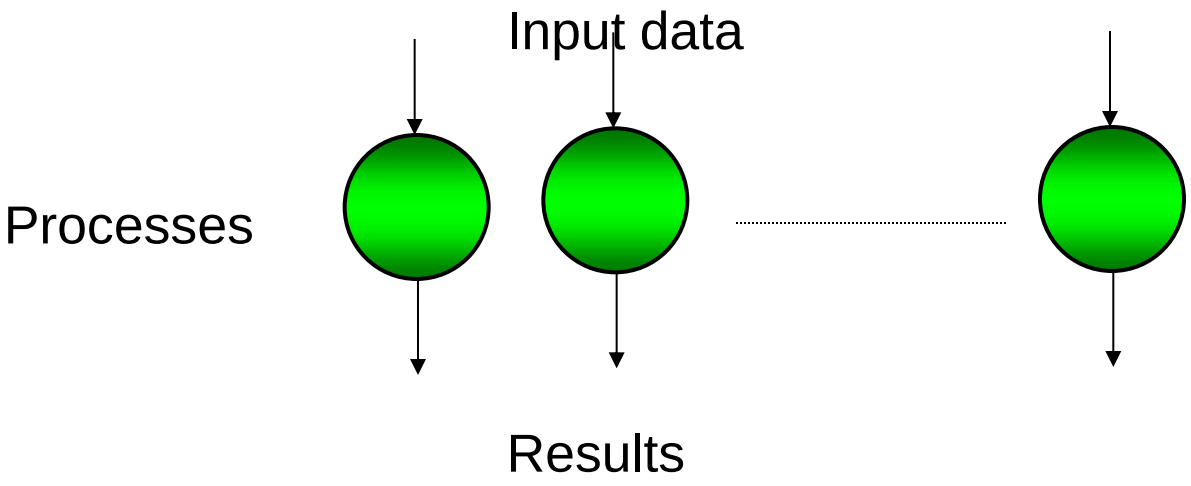
Message-Passing Interface (MPI) - standard defined in 1990s.

Both provide a set of user-level libraries for message passing. Use with regular programming languages (FORTRAN, C, C++, ...).

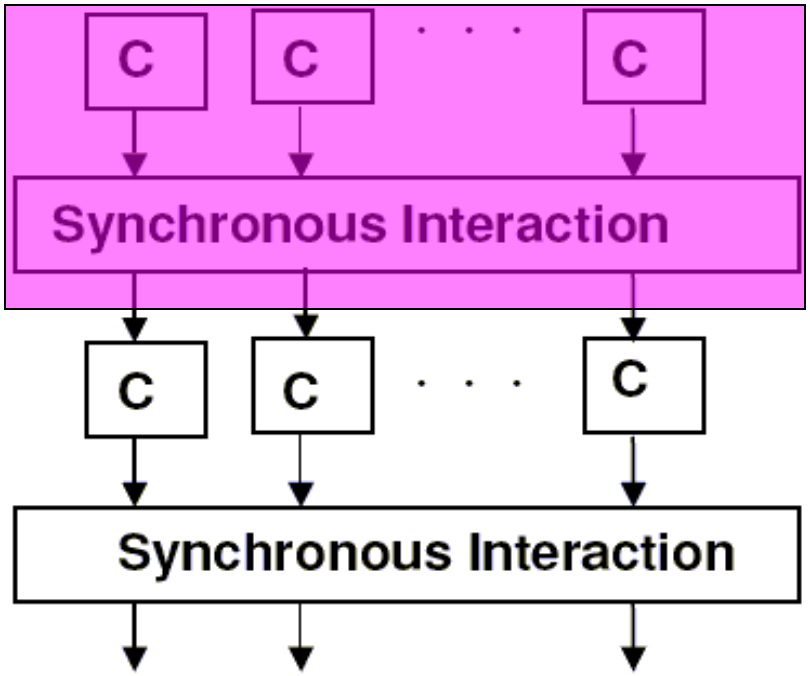
Embarrassingly Parallel Computations

A computation that can be divided into a number of completely independent parts, each of which can be executed by a separate process(or).

No communication or very little communication between processes



Phase Parallel Model

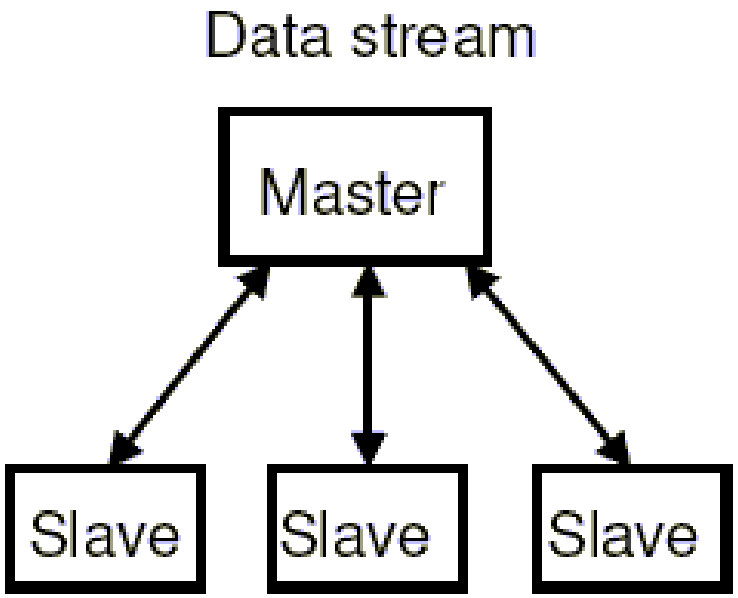


Nearly embarrassing

One can see how close it is to BSP

- The phase-parallel model offers a paradigm that is widely used in parallel programming.
- The parallel program consists of a number of super steps, and each has two phases.
- In a computation phase, multiple processes each perform an independent computation C.
- In the subsequent interaction phase, the processes perform one or more synchronous interaction
- operations, such as a barrier or a blocking communication.
- Then next super step is executed.

Process farm



- This paradigm is also known as the master-slave paradigm.
- A master process executes the essentially sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.
- When a slave finishes its workload, it informs the master which assigns a new workload to the slave.
- This is a very simple paradigm, where the coordination is done by the master.

MPI (Message Passing Interface)

Standard developed by group of academics and industrial partners to foster more widespread use and portability

Defines routines, not implementation

Several free implementations of MPI standard exist.

List of Few active products/projects

- CRI/EPCC
- Hitachi MPI
- HP MPI
- IBM Parallel Environment for AIX-MPI Library
- LAM/MPI (Supplier: Indiana University)
- MPI for UNICOS Systems
- MPICH (Supplier: Argonne National Laboratory)
- OS/390 Unix System Services Parallel
- Intel MPI
- SGI Message Passing Toolkit
- Sun MPI
- Open MPI

List of Few active products/projects

- CRI/EPCC
- Hitachi MPI
- HP MPI
- IBM Parallel Environment for AIX-MPI Library
- LAM/MPI (Supplier: Indiana University)
- MPI for UNICOS Systems
- **MPICH** (Supplier: Argonne National Laboratory)
- OS/390 Unix System Services Parallel
- Intel MPI
- SGI Message Passing Toolkit
- Sun MPI
- **Open MPI**

What is MPI

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

Where to use MPI?

- We need a portable parallel program
- We are writing a parallel Library

Why learn MPI?

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

- The attractiveness of the message-passing paradigm is its wide portability.

- Programs expressed this way may run on distributed-memory multiprocessors, networks of workstations, and combinations of all of these.

- In addition, shared-memory implementations are possible.

Basics of Message-Passing Programming

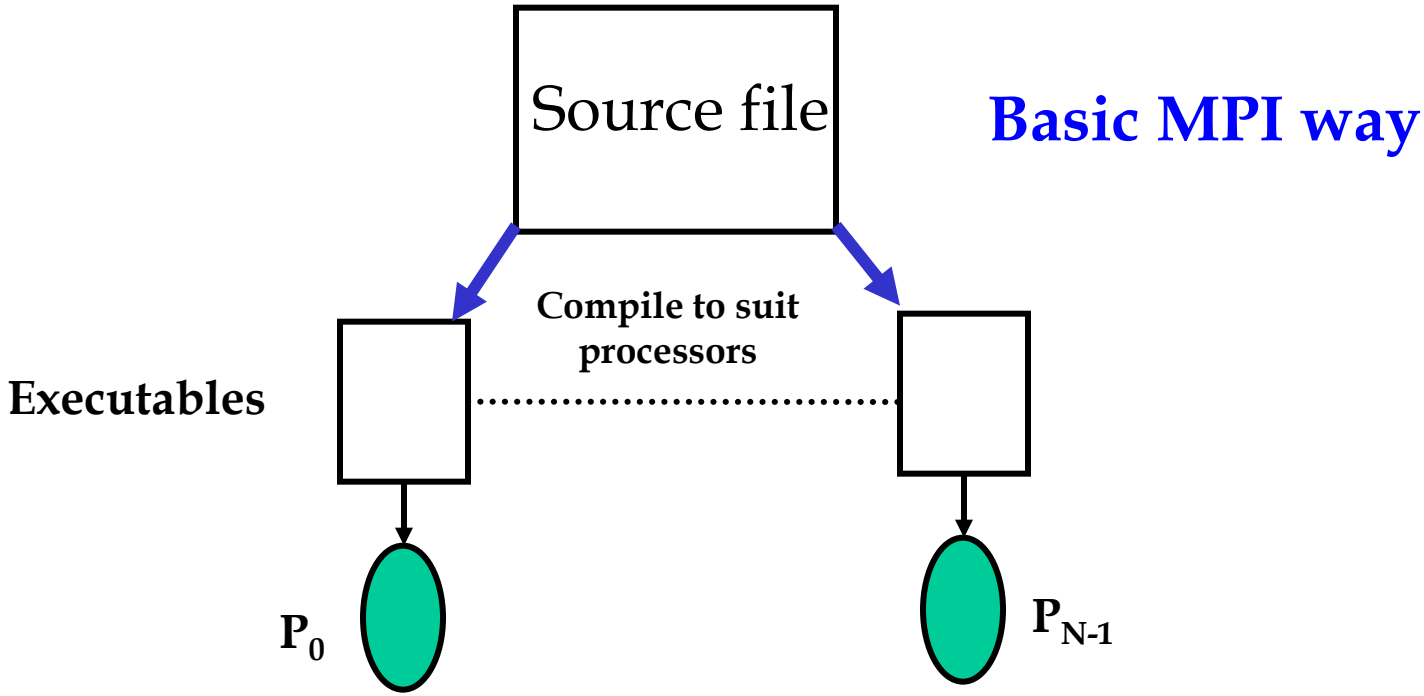
Two primary mechanisms needed:

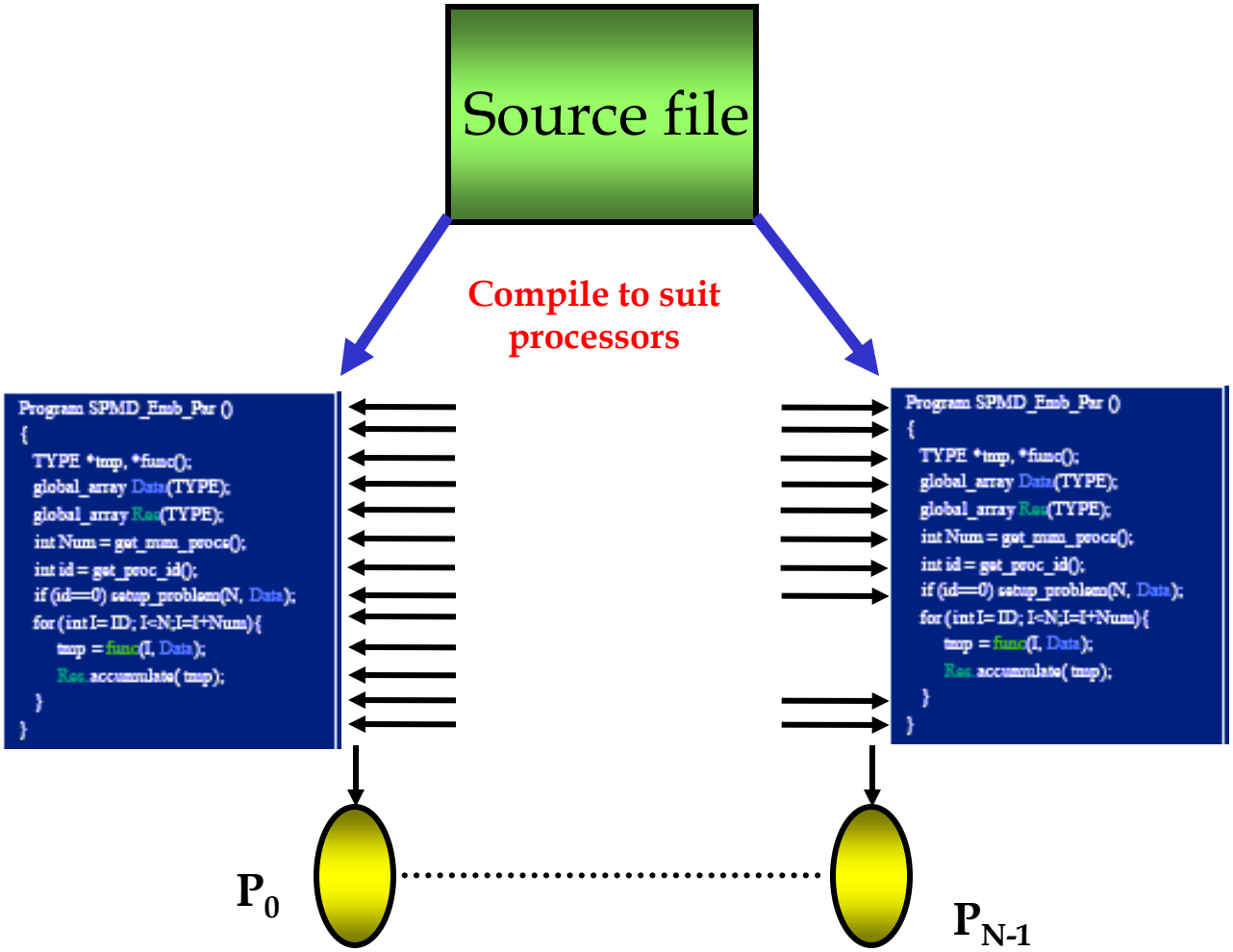
1. A method of creating separate processes for execution on different computers

2. A method of sending and receiving messages

Single Program Multiple Data (SPMD) model

Different processes merged into one program. Within program,control statements select different parts for each processor to execute. All executables start together - static process creation.




Source file
Compile to suit processors
Program SPMD_Emb_Par ()
{
TYPE *tmp, *func();
global_array Data(TYPE);
global_array Res(TYPE);
int Num = get_num_procs();
int id = get_proc_id();
if (id==0) setup_problem(N, Data);
for (int I= ID; I<N;I=I+Num){
tmp = func(I, Data);
Res.accumulate(tmp);
}
}
Program SPMD_Emb_Par ()
{
TYPE *tmp, *func();
global_array Data(TYPE);
global_array Res(TYPE);
int Num = get_num_procs();
int id = get_proc_id();
if (id==0) setup_problem(N, Data);
for (int I= ID; I<N;I=I+Num){
tmp = func(I, Data);
Res.accumulate(tmp);
}
}
P0
PN-1

Evaluating General Message

Message Passing SPMD : C program

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

A History of MPICH

- MPICH was developed during the MPI standards process to provide feedback to the MPI forum on implementation and usability issues.
- With the release of the MPI standard, MPICH was designed to provide an implementation of the MPI standard.
- It supported both MIMD programming and heterogeneous clusters from the very beginning.

MPICH

- MPICH is an open-source, portable implementation of the Message-Passing Interface Standard.
- Designed at Mathematics and Computer Science Division of **Argonne National Laboratory**.
- The “CH" in MPICH stands for Chameleon symbol of adaptability to one's environment and thus of portability.

MPICH

William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.

It contains a complete implementation of version 3.1 of the MPI Standard and also significant parts of MPI-2, particularly in the area of parallel I/O.

MPICH

- MPICH is the latest implementation of MPI. Present version in 3.4a.
- The features in MPICH also include support for one-side communication, dynamic processes, inter communicator collective operations, and expanded MPI-IO functionality.
- Clusters consisting of both single-processor and SMP nodes are supported.

MPICH Architecture

- Most code is completely portable
- An Abstract Device defines the communication layer
- The abstract device (The central mechanism for achieving the goals of portability and performance is a specification we call the abstract device interface (ADI)) can have widely varying instantiations, using:
  - sockets
  - shared memory
  - other special interfaces
    - e.g. Myrinet, Quadrics, InfiniBand, Grid protocols

MPICH implementations

MPICH is freely available and is distributed as open source.

- The Unix/Linux (all flavors) version of MPICH
- The Microsoft Windows version of MPICH (mpich-3.2.1)
- MVAPICH2-GDR, support for InfiniBand and NVIDIA CUDA GPUs

Is MPICH Large or Small?

MPICH is large (~396 Functions)

- MPICH's extensive functionality requires many functions
- Number of functions not necessarily a measure of complexity

MPICH is small (6 Functions)

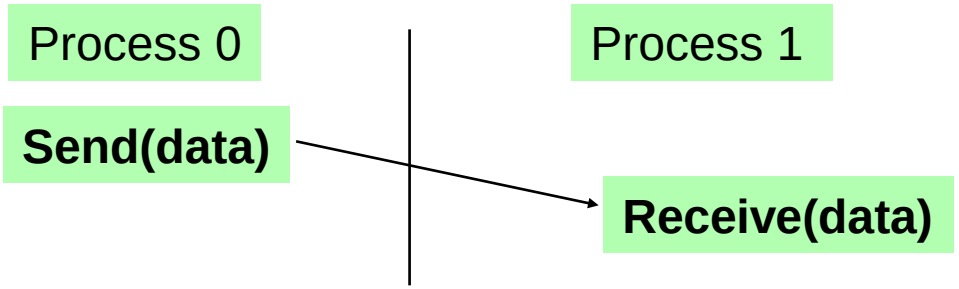
- Many parallel programs can be written with just 6 basic functions

MPICH is just **right** candidate for message passing

- One need not master all parts of MPICH to use it

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**

Begin programming with 6 MPI function calls

- MPI_INIT — *Initializes MPI*
- MPI_COMM_SIZE — *Determines number of processes*
- 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 Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays

C data types

- **MPI_CHAR**
  char
- **MPI_BYTE**
  like unsigned char
- **MPI_SHORT**
  short
- **MPI_INT**
  int
- **MPI_LONG**
  long
- **MPI_FLOAT**
  float
- **MPI_DOUBLE**
  double
- **MPI_UNSIGNED_CHAR**
  unsigned char
- **MPI_UNSIGNED_SHORT**
  unsigned short
- **MPI_UNSIGNED**
  unsigned int
- **MPI_UNSIGNED_LONG**
  unsigned long
- **MPI_LONG_DOUBLE**
  long double (some systems may not implement)
- **MPI_LONG_LONG_INT**
  long long (some systems may not implement)

FORTRAN data types

- **MPI_REAL**
  REAL
- **MPI_INTEGER**
  INTEGER
- **MPI_LOGICAL**
  LOGICAL
- **MPI_DOUBLE_PRECISION**
  DOUBLE PRECISION
- **MPI_COMPLEX**
  COMPLEX
- **MPI_DOUBLE_COMPLEX**
  complex*16 (or complex*32) where supported

The following data types are optional

- **MPI_INTEGER1**
  integer*1 if supported
- **MPI_INTEGER2**
  integer*2 if supported
- **MPI_INTEGER4**
  integer*4 if supported
- **MPI_REAL4**
  real*4 if supported
- **MPI_REAL8**
  real*8 if supported

Caution!

- Fortran types should only be used in Fortran programs,
- C types should only be used in C programs.

For example, it is in error to use MPI_INT for a Fortran INTEGER, which should be MPI_INTEGER.

MPI_Op Options (Collective Operation)

- MPI_BAND
- MPI_BOR
- MPI_BXOR
- MPI_LAND
- MPI_LOR
- MPI_LXOR
- MPI_MAX
- MPI_MAXLOC*
- MPI_MIN
- MPI_MINLOC
- MPI_PROD
- MPI_SUM

* Maximum and Location

Communicators

Defines *scope* of a communication operation.

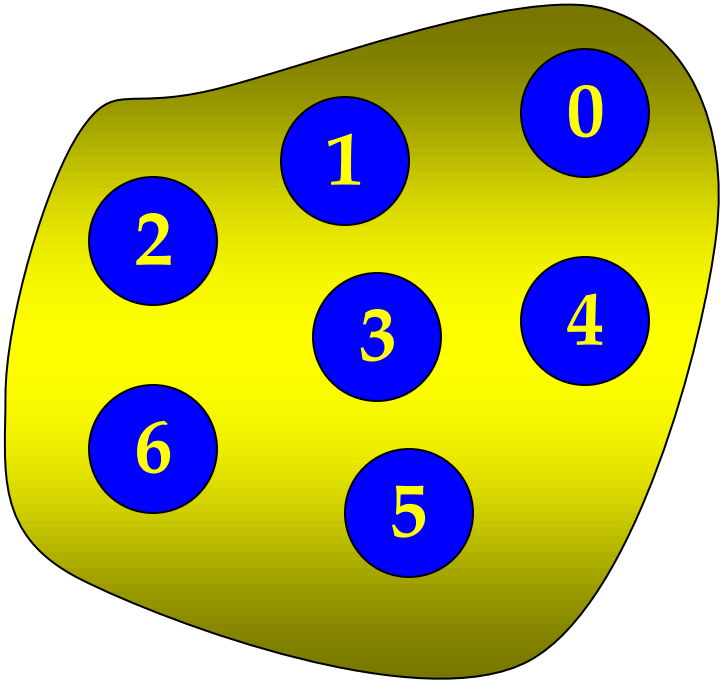
Processes have **ranks** associated with the communicator.

Initially, all processes enrolled in a **"universe"** called **MPI_COMM_WORLD** and each process is given a unique rank, a number from 0 to *n* - 1, where there are *n* processes.

Other communicators can be established for groups of processes.

MPI_COMM_WORLD communicator

This Default Communicator is MPI's mechanism for establishing individual communication universes



MPI_COMM_WORLD

MPI Messages

Message : data (3 parameters) + envelope (3 parameters)

Data: startbuf, count, datatype

- **Startbuf:** address where the data starts
- **Count:** number of elements (items) of data in the message

Envelope: dest, tag, comm

- **Destination or Source:** Sending or Receiving processes
- **Tag:** Integer to distinguish messages

Communicator:

The communicator is communication “universe.”

Messages are sent or received within a given “universe.”

Synchronous Message Passing (Blocking)

Routines that actually return when message transfer completed.

Synchronous send routine Waits until complete message can be accepted by the receiving process before sending the next message.

Synchronous receive routine Waits until the message it is expecting arrives.

Synchronous routines fundamentally perform two actions: They transfer data and they synchronize processes.

Asynchronous Message Passing (Non-Blocking)

Routines that do not wait for actions to complete before returning. Usually require local storage for messages.

More than one version depending upon the actual semantics for returning.

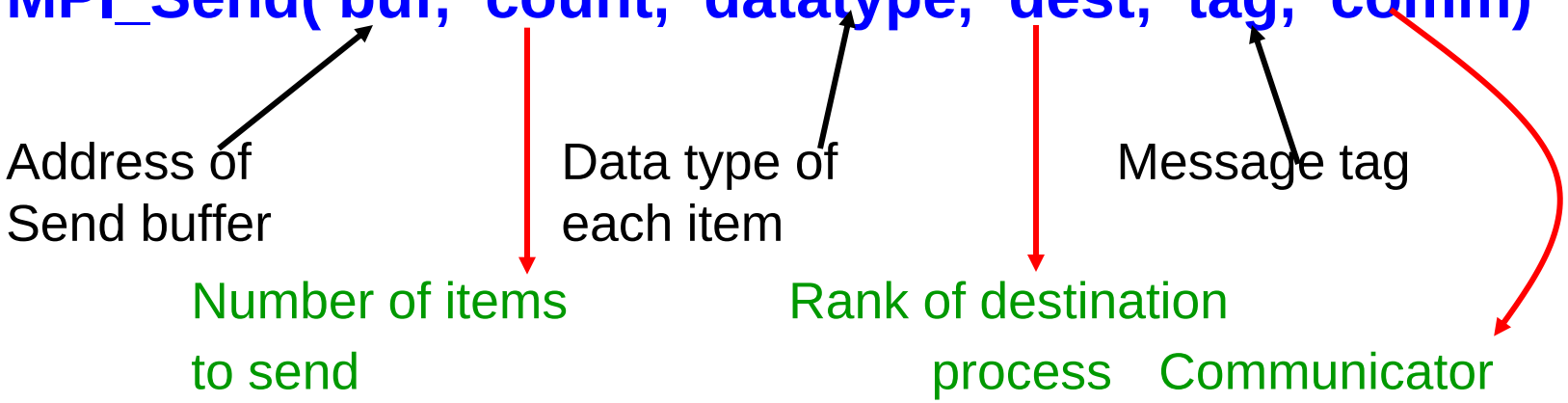
In general, they do not synchronize processes but allow processes to move forward sooner. Must be used with care.

MPICH Send and Recv

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

Parameters of the blocking send

MPI_Send(buf, count, datatype, dest, tag, comm)

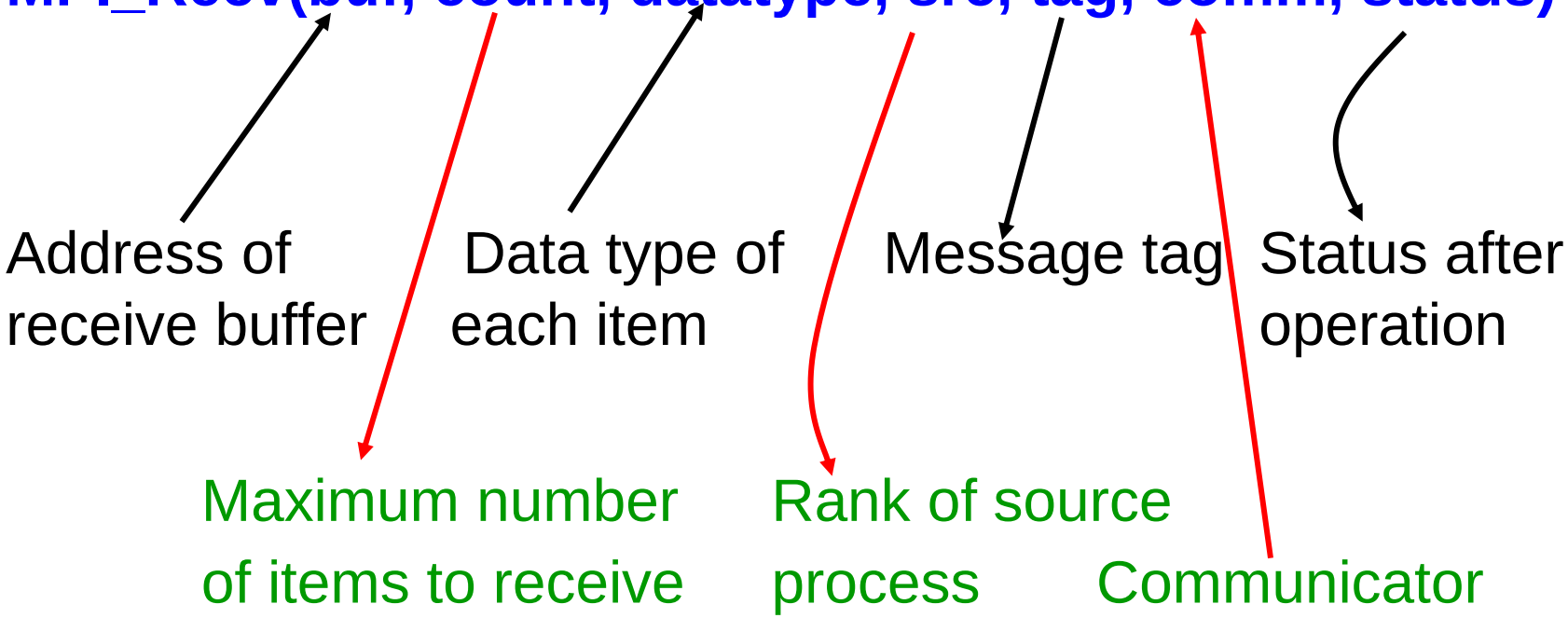


MPI Basic (Blocking) Send

- When this function returns, the data has been delivered to the system and the buffer can be reused.

Parameters of the blocking receive

MPI_Recv(buf, count, datatype, src, tag, comm, status)



MPI Basic (Blocking) Receive

- Waits until a matching (both **source** and **tag**) message is received from the system, and the buffer can be used
- **source** is rank in communicator specified by **comm**, or **MPI_ANY_SOURCE**
- **tag** is a tag to be matched on or **MPI_ANY_TAG**
- receiving fewer than **count** occurrences of **datatype** is OK, but **receiving more is an error**
- **status** contains further information (e.g. size of message)

Message Tag

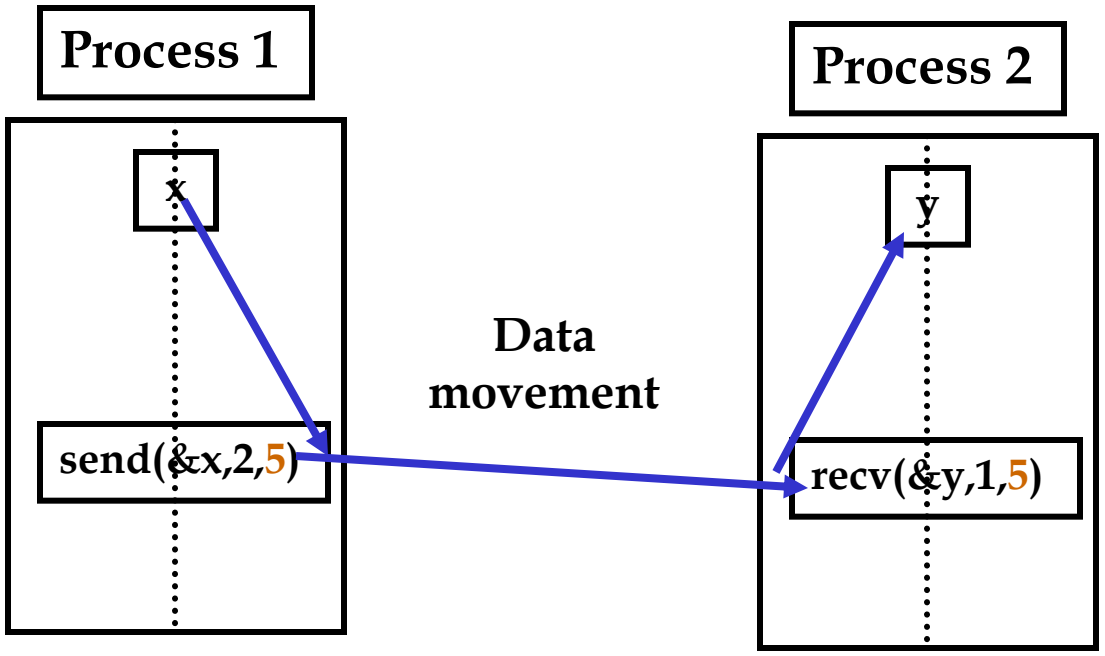
Used to differentiate between different types of messages being sent.

Message tag is carried within message.

If special type matching is not required, a *wild card* message tag is used, so that the recv() will match with any send().

Message Tag Example

To send a message **x** with message tag 5 from a source process 1 to a destination process 2 and assign to **y**:



Waits for a message from process 1 with a tag of 5

Initializing MPICH

- Must be first routine called
- int MPI_Init(int *argc, char **argv);

What makes an MPICH Program?

Include files

- mpi.h (c)
- mpif.h (Fortran)

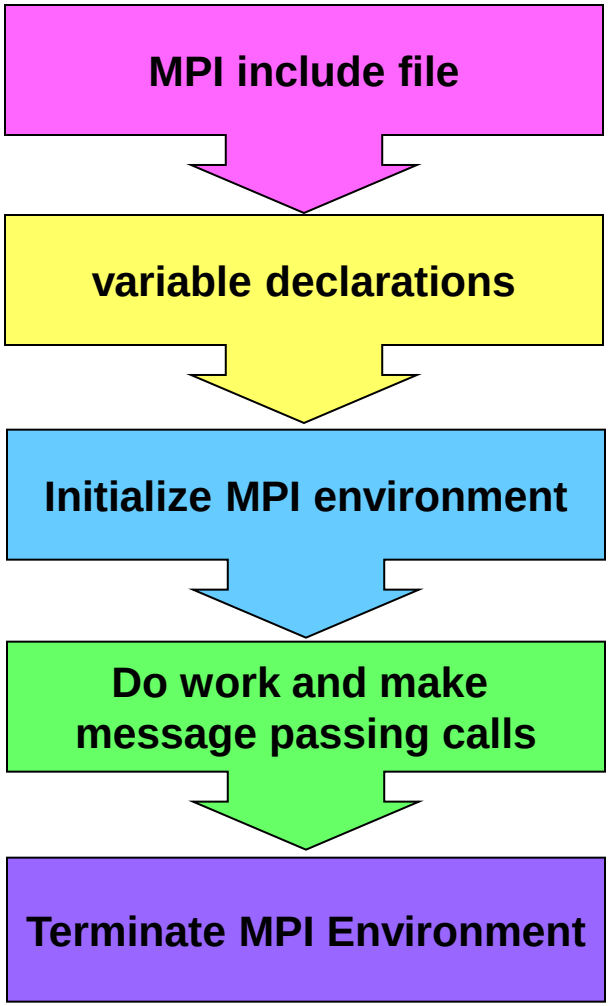
Initiation of MPI

- MPI_INIT

Completion of MPI

- MPI_FINALIZE

General MPI Program Structure




```
#include <mpi.h>
void main (int argc, char **argv)
{
int np, rank;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&np);
/*      Do Some Works              */
MPI_Finalize();
}
```

MPI_Init

- The command line arguments are provided to MPI_Init to allow an MPI implementation to use them in initializing *the MPI environment*.

- They are passed by reference to allow an MPI implementation to provide them in environments where the command-line arguments are not provided to **main function**.

MPI_Init

- At least one process has access to `stdin`, `stdout`, and `stderr`
- The user can find out which process this is by querying the attribute MPI_IO on MPI_COM WORLD
- In MPICH all processes have access to `stdin`, `stdout`, and `stderr` and on networks these I/O streams are routed back to the process with rank 0 in MPI_COMM_WORLD.

MPI_Init

- On most systems, these streams also can be redirected through `mpirun`, as follows

```
mpirun –np 64 myprog -myarg 13 <data.in>
results.out
```

- Here we assume that –myarg 13 are command-line arguments processed by the application myprog. After MPI_Init, each process will have these arguments in its `argv`

Example

To send an integer *x* from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank); /* find rank */
if (myrank == 0) {
        int x;
            MPI_Send(&x, 1, MPI_INT, 1, msgtag,
                                MPI_COMM_WORLD);
            } else if (myrank == 1) {
            int x;
            MPI_Recv(&x, 1, MPI_INT, 0,msgtag,
                                MPI_COMM_WORLD, status);
}
```

MPI_Init

- MPI_Init is called prior to any calls to other MPI routines. Its purpose is to initialize the MPI environment.
- Calling MPI_Init more than once during the execution of a program **will lead to an error**.
- MPI_Finalize is called at the end of the computation, and it performs various clean-up tasks to terminate the MPI environment.
- No MPI calls may be performed after MPI_Finalize has been called, not even MPI_Init.

MPI_Init

- Both MPI_Init and MPI_Finalize must be called by all the processes, otherwise MPI's behavior will be undefined.
- The exact calling sequences of these two routines for C are as follows:
  - int MPI_Init(int *argc, char ***argv)
  - int MPI_Finalize()

MPI_Init

- The arguments argc and argv of MPI_Init are the command-line arguments of the C program.
- An MPI implementation is expected to remove from the argv array any command-line arguments that should be processed by the implementation before returning back to the program, and to decrement argc accordingly.

MPI_Init

- **Thus, command-line processing should be performed only after MPI_Init has been called.**

- Upon successful execution, MPI_Init and MPI_Finalize return MPI_SUCCESS; otherwise they return an implementation-defined error code.

Let us write the first Complete C/MPICH program

Write a simple parallel program in which every process with rank greater than 0 sends a message "Hello-Participants" to a process with rank 0. The processes with rank 0 receives the message and prints it.

```
#include "mpi.h"
main (int argc, char **argv) {
int MyRank, Numprocs, tag, ierror, i;
MPI_Status status;
char send_message[20], recv_message[20];
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &Numprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &MyRank);
tag = 100;
strcpy (send_message, "Hello-Participants");
if (MyRank==0) {
        for (i=1; i<Numprocs; i++) {
          MPI_Recv (recv_message,20, MPI_CHAR, i, tag, MPI_COMM_WORLD,&status);
          printf ("node %d : %s \n", i, recv_message);
         }
} else
         MPI_Send(send_message, 20, MPI_CHAR,0, tag, MPI_COMM_WORLD);
         MPI_Finalize();
}
```

Basic instructions for compiling/executing MPICH programs

Preliminaries

- Set up paths
- Create required directory structure
- Modify makefile to match your source file
- Create a file (hostfile) listing machines to be used (required)

Compiling/executing (SPMD) C/MPICH program

To compile MPI programs:

mpicc -o file file.c

Or

mpiCC -o file file.cpp

To execute MPI program: mpirun -np no_processes file

Basic instructions for compiling/executing MPI programs

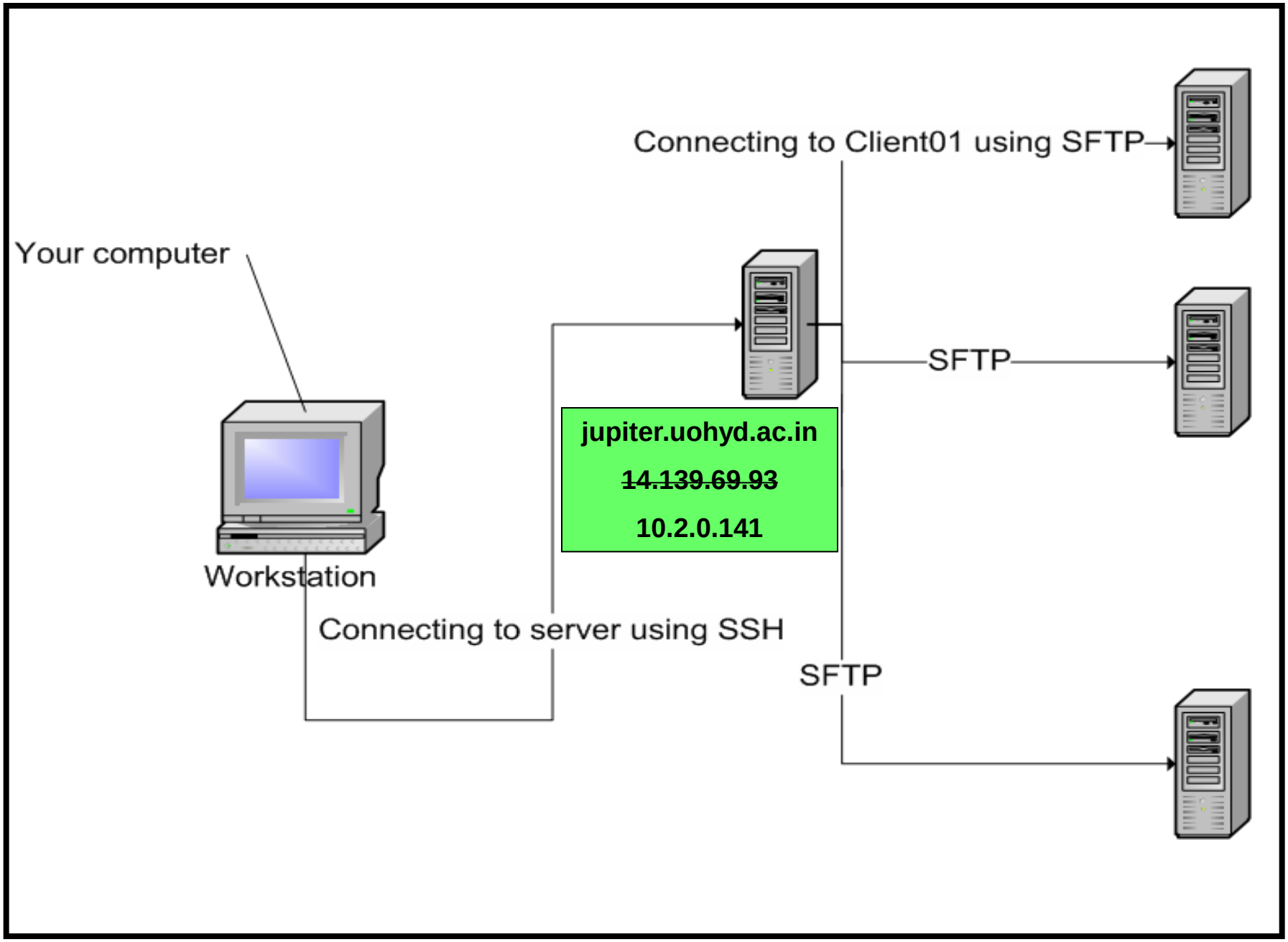
Depends on your implementation of MPI

❑For Mpich/OpenMPI:

- mpirun -np2 a001 # run MPI program

❑ For lam:

- lamboot -v lamhosts # starts LAM
- mpirun -v -np 2 a001 # run MPI program
- lamclean -v # rm all user processes
- mpirun … # run another program
- lamclean …
- lamhalt # stop LAM


Connecting to Client01 using SFTP
Your computer
SFTP
jupiter.uohyd.ac.in
14.139.69.93
10.2.0.141
Workstation
Connecting to server using SSH
SFTP

Procedure Speciation

MPI procedures are specified using a language independent notation. The arguments of

procedure calls are marked as IN, OUT or INOUT. The meanings of these are:

- the call uses but does not update an argument marked IN,
- the call may update an argument marked OUT,
- the call both uses and updates an argument marked INOUT.

Procedure Speciation

- There is one special case: if an argument is a handle to an opaque object and the object is updated by the procedure call, then the argument is marked OUT.

- It is marked this way even though the handle itself is not modified we use the OUT attribute to denote that the handle references is updated.

Opaque objects

- MPI manages system memory that is used for buffering messages and for storing internal representations of various MPI objects such as groups, communicators, datatypes, etc.
- This memory is not directly accessible to the user, and objects stored there are opaque: their size and shape is not visible to the user. Opaque objects are accessed via handles, which exist in user space.
- In Fortran, all handles have type INTEGER. In C, a different handle type is defined for each category of objects.

Collective Communications

The sending and/or receiving of messages to/from **groups of processes**. A collective communication implies that all processes need to participate in the communication.

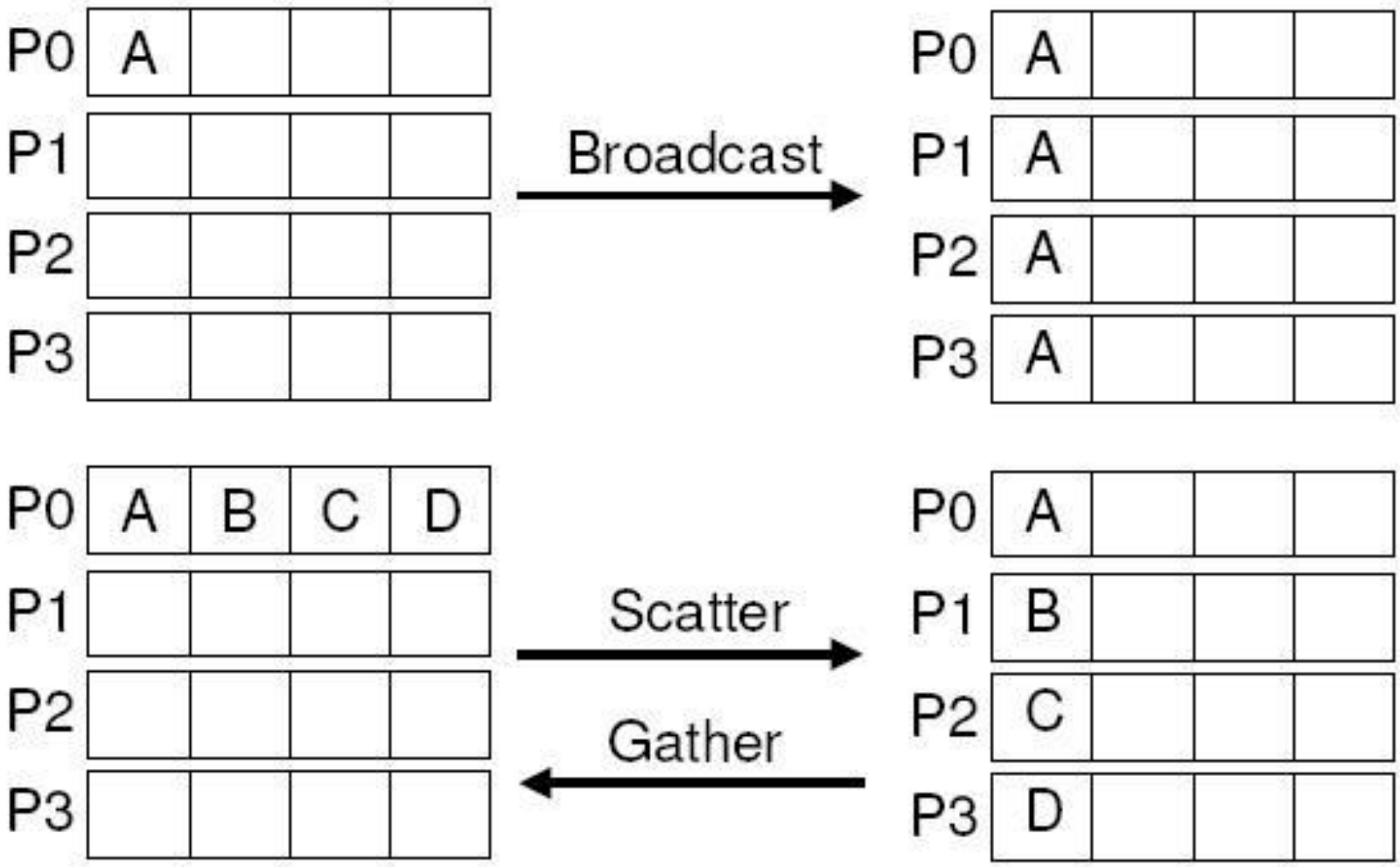
- Involves coordinated communication within a group of processes
- No message tags used
- All collective routines block until they are locally complete

Collective Communication

- Two broad classes:
  - Data movement routines
  - Global computation routines
- Called by all processes in a communicator

Examples:

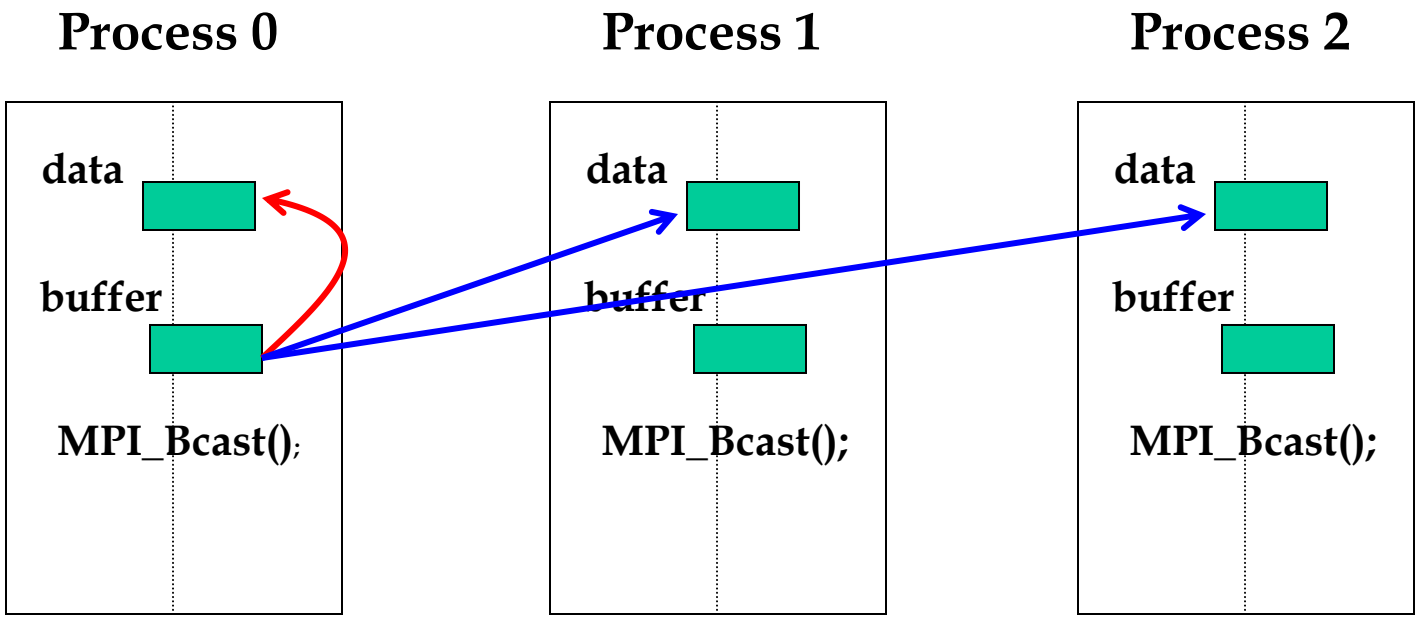
- Barrier synchronization
- Broadcast, scatter, gather
- Global sum, global maximum, etc.



Representation of collective data movement in MPI

Broadcast

Sending same message to all processes concerned with problem.
Multicast - sending same message to defined group of processes.



MPI form

Broadcast

A broadcast sends data from one processor to all other processors, including itself.

C:

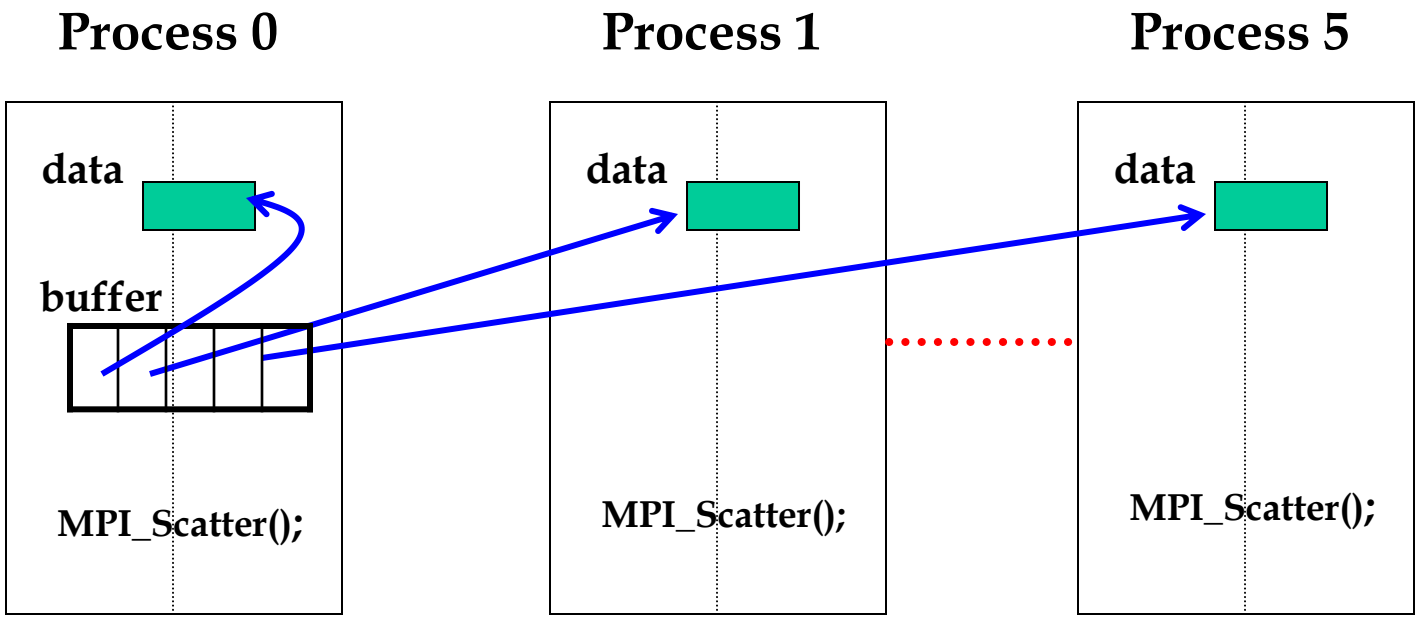
int **MPI_Bcast** (void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

Input/output Parameters

INOUT	**buffer**	starting address of buffer
IN	**count**	number of entries in buffer
IN	**Datatype**	data type of buffer
IN	**root**	rank of broadcast root
IN	**comm**	communicator

Scatter

In its simplest form sending each element of an array in ***root*** process to a separate process. Contents of i^{th} location of array sent to i^{th} process.



MPI form

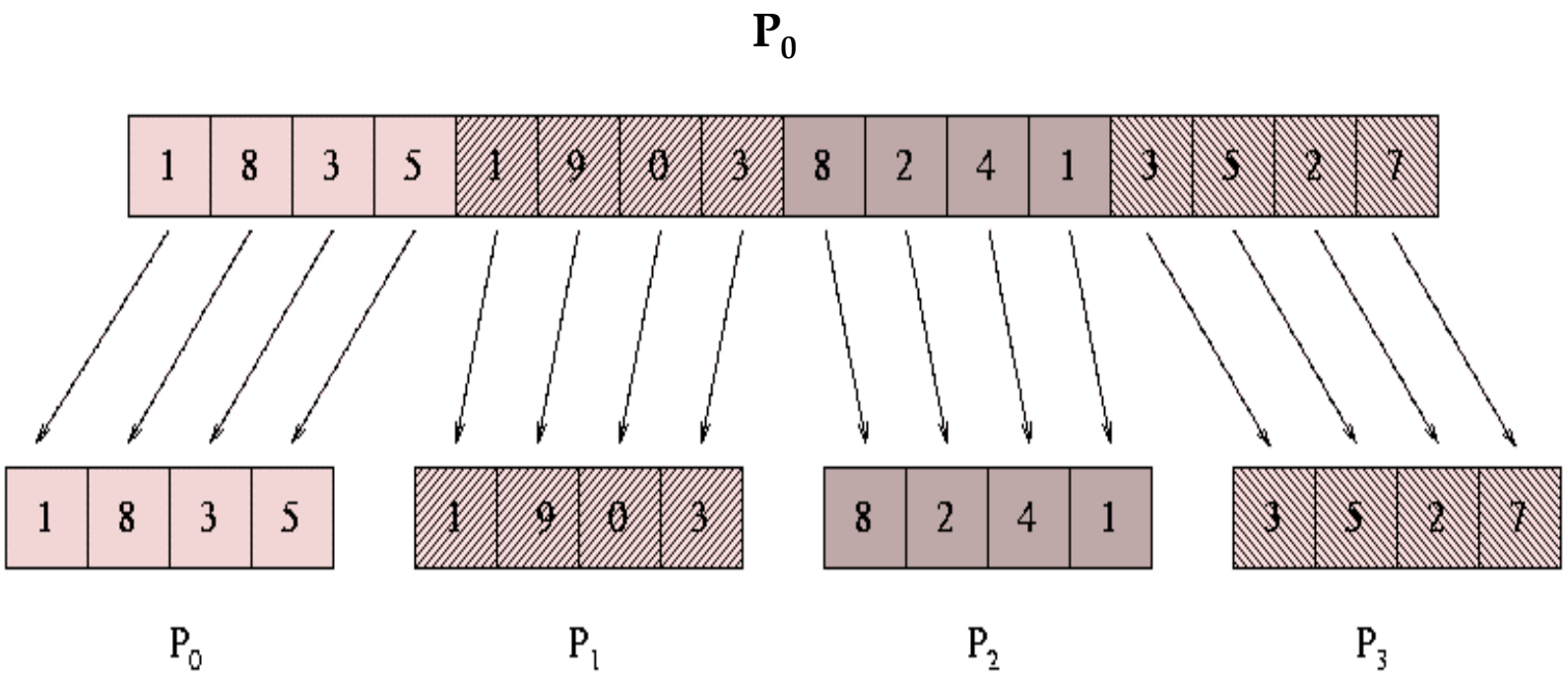
Scatter

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:

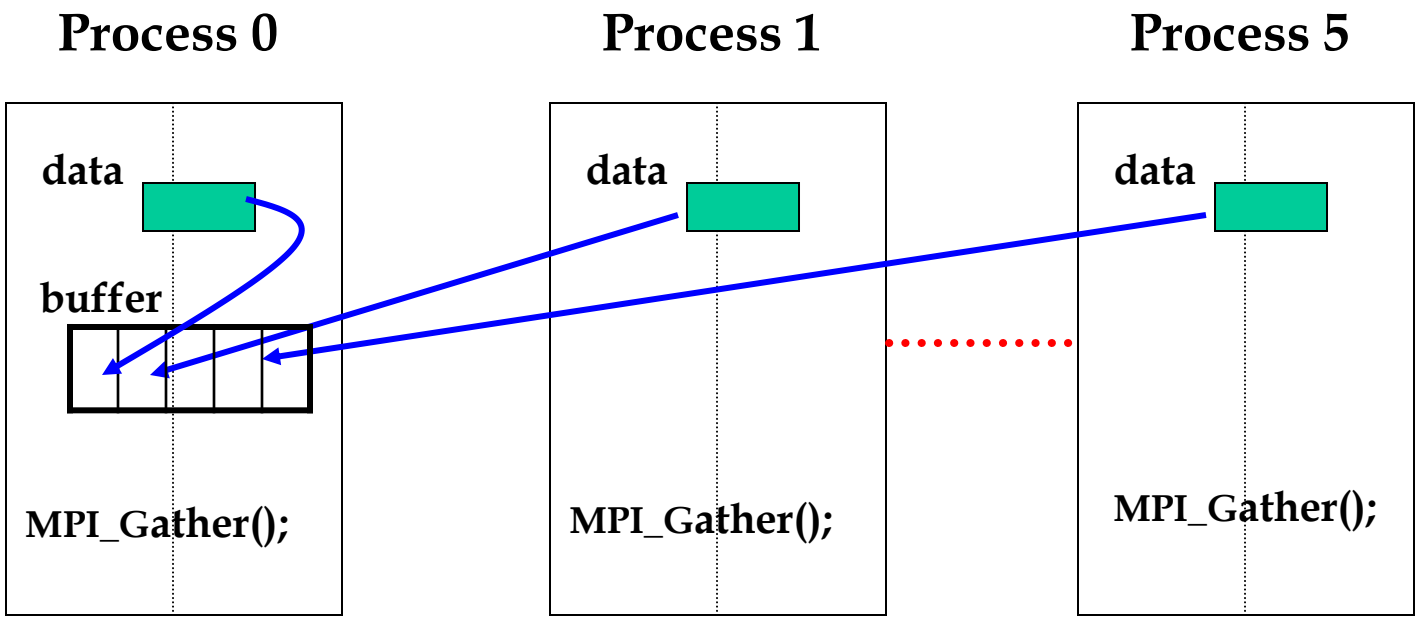
int **MPI_Scatter**(void *send_buffer, int send_count, MPI_Datatype send_type, void *recv_buffer, int recv_count, MPI_Datatype recv_type, **int root**, MPI_Comm comm);

Scatter



Gather

Having one process collect individual values from set of processes.



MPI form

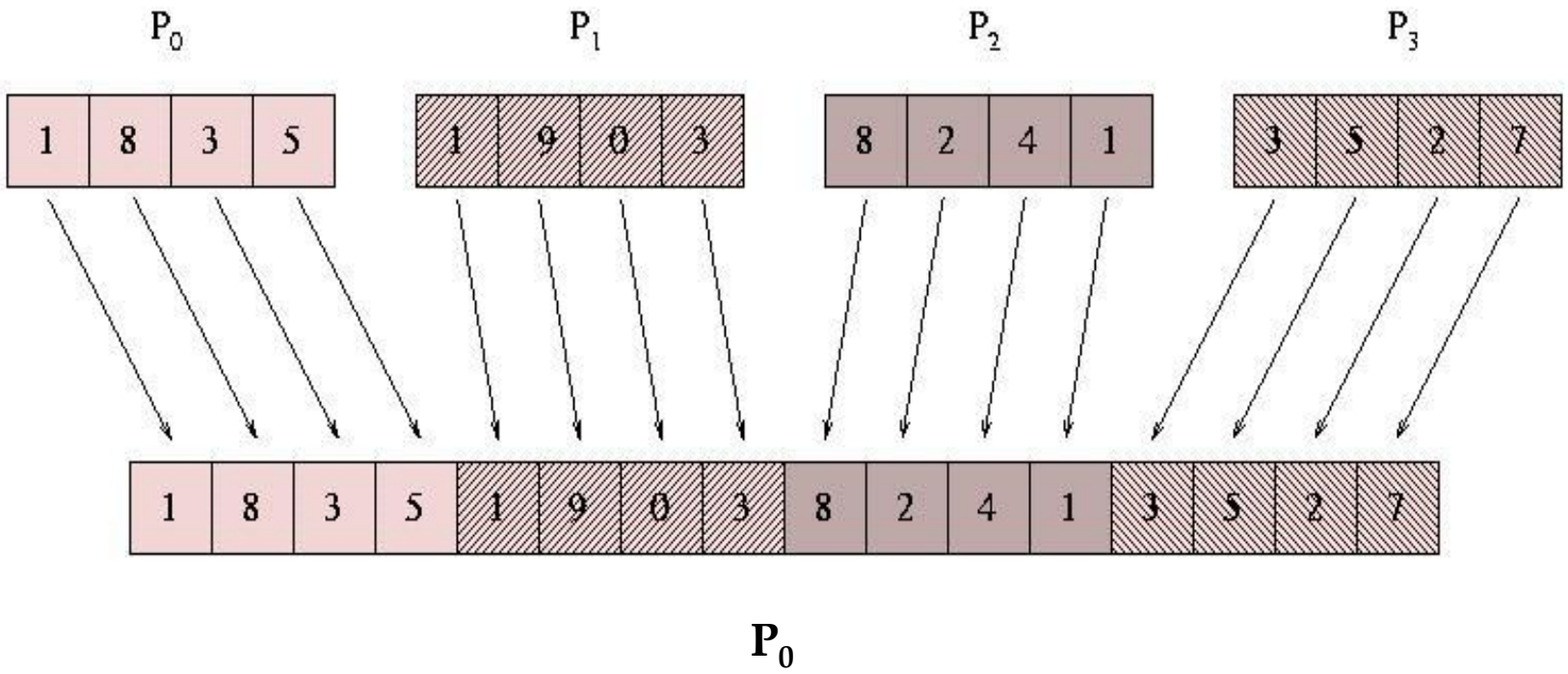
Gather

Each process in *comm* sends 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 argument *recv_count* indicates the number of items received from each process – not the total number received.

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);

Gather

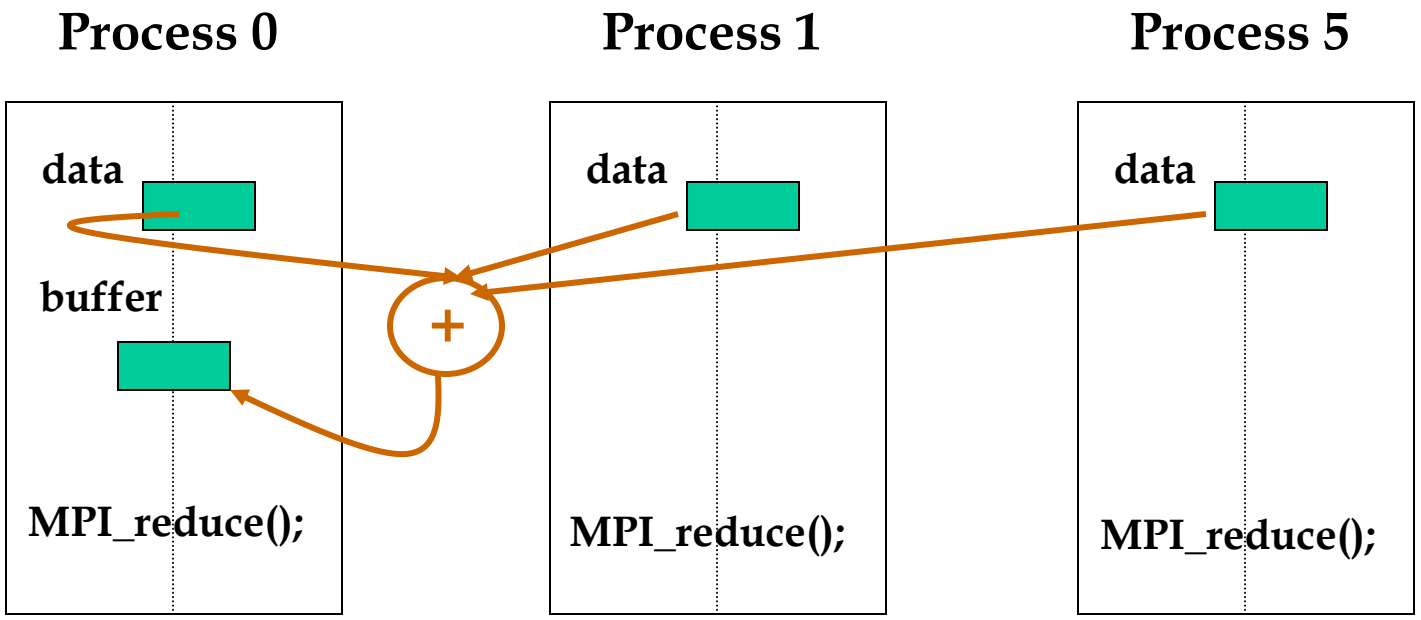


- Your ID on jupiter is your registration number
- Password is test123, change immediately
- Press “Enter” if it asks any key to enter
- jupiter.uohyd.ac.in
- 14.139.69.93 (from outside) **shh only on port 8223**
- 10.2.0.141 (from inside UoH)
- Browser: http://jupiter.uohyd.ac.in/wordpress/

Reduce

It is a Gather operation combined with specified arithmetic/ logical operation.

Values could be gathered and then added together by root:



MPI form

MPI_Reduce

int **MPI_Reduce** (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Input Parameters

sendbuf	address of send buffer (choice)
count	number of elements in send buffer
datatype	data type of elements of send buffer (handle)
op	reduce operation
root	rank of root process (integer)
comm	communicator (handle)

Output Parameter

recvbuf address of receive buffer (significant only at root)

Predefined reduce operations

[MPI_Max]	maximum
[MPI_Min]	minimum
[MPI_Sum]	sum
[MPI_Prod]	product
[MPI_Land]	logical and
[MPI_Band]	bit-wise and
[MPI_Lor]	logical or
[MPI_Bor]	bit-wise or
[MPI_Lxor]	logical xor
[MPI_Bxor]	bit-wise xor
[MPI_Maxloc]	max value and location
[MPI_Minloc]	min value and location

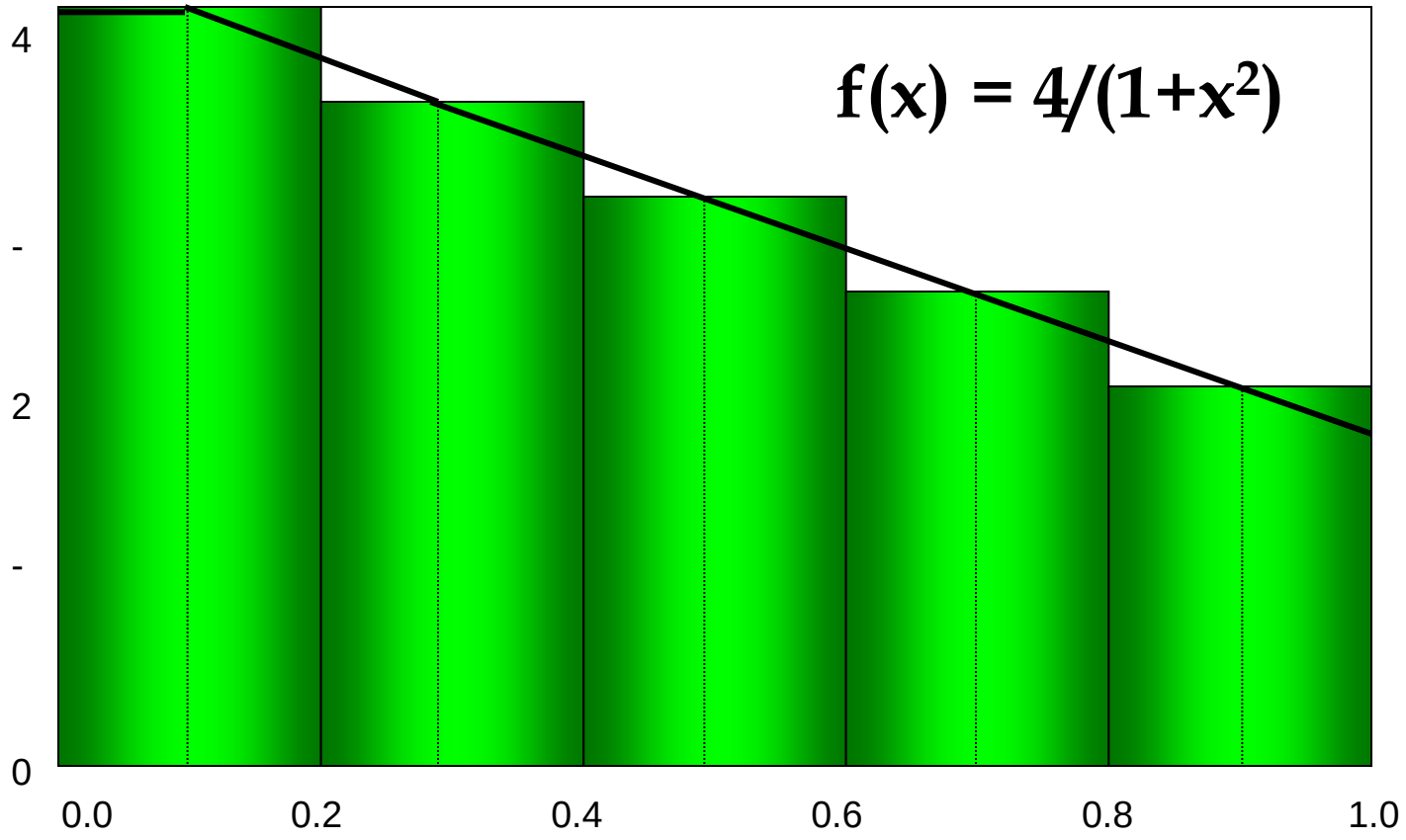
Example

Write a simple parallel program to calculate value of **pi** by numerical integration. Since

$$\int_0^1 \frac{1}{1+x^2} dx = \tan^{-1}(x)\,|_0^1 = \tan^{-1}(1) - \tan^{-1}(0) = \tan^{-1}(1) = \frac{\pi}{4}$$

We will integrate the function $f(x) = 4/(1 + x^2)$

Graph



```
#include "mpi.h"
#include <math.h>
double f(a)
double a;
{
    return (4.0 / (1.0 + a*a));
}
int main(argc,argv)
int argc;
char *argv[];
{
    int n, myid, numprocs, i, rc;
    double PI25DT = 3.14159265;
    double mypi, pi, h, sum, x, a;
    double startwtime, endwtime;
```

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
if (myid == 0)
    {
     printf("Enter the number of intervals:”);
     scanf("%d",&n);
     startwtime = MPI_Wtime();
    }
```

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
h   = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs)
    {
      x = h * ((double)i - 0.5);
      sum += f(x);
    }
    mypi = h * sum;
```

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
          if (myid == 0)
            {
             printf("pi is approximately
               %.16f, Error is %.16f\n",pi, fabs(pi
               - PI25DT));
               endwtime = MPI_Wtime();
               printf("wall clock time = %f\n",
                      endwtime-startwtime);
            }
        }
    }
    MPI_Finalize();
}
```

```
#include "mpi.h"
#include <math.h>
double f(a)
double a;
{
    return (4.0 / (1.0 + a*a));
}
int main(argc,argv)
int argc;
char *argv[];
{
    int n, myid, numprocs, i, rc;
    double PI25DT = 3.14159265;
    double mypi, pi, h, sum, x, a;
    double startwtime, endwtime;
```

Some variable declarations:

myid, mypi, sum, x, i are local to process.

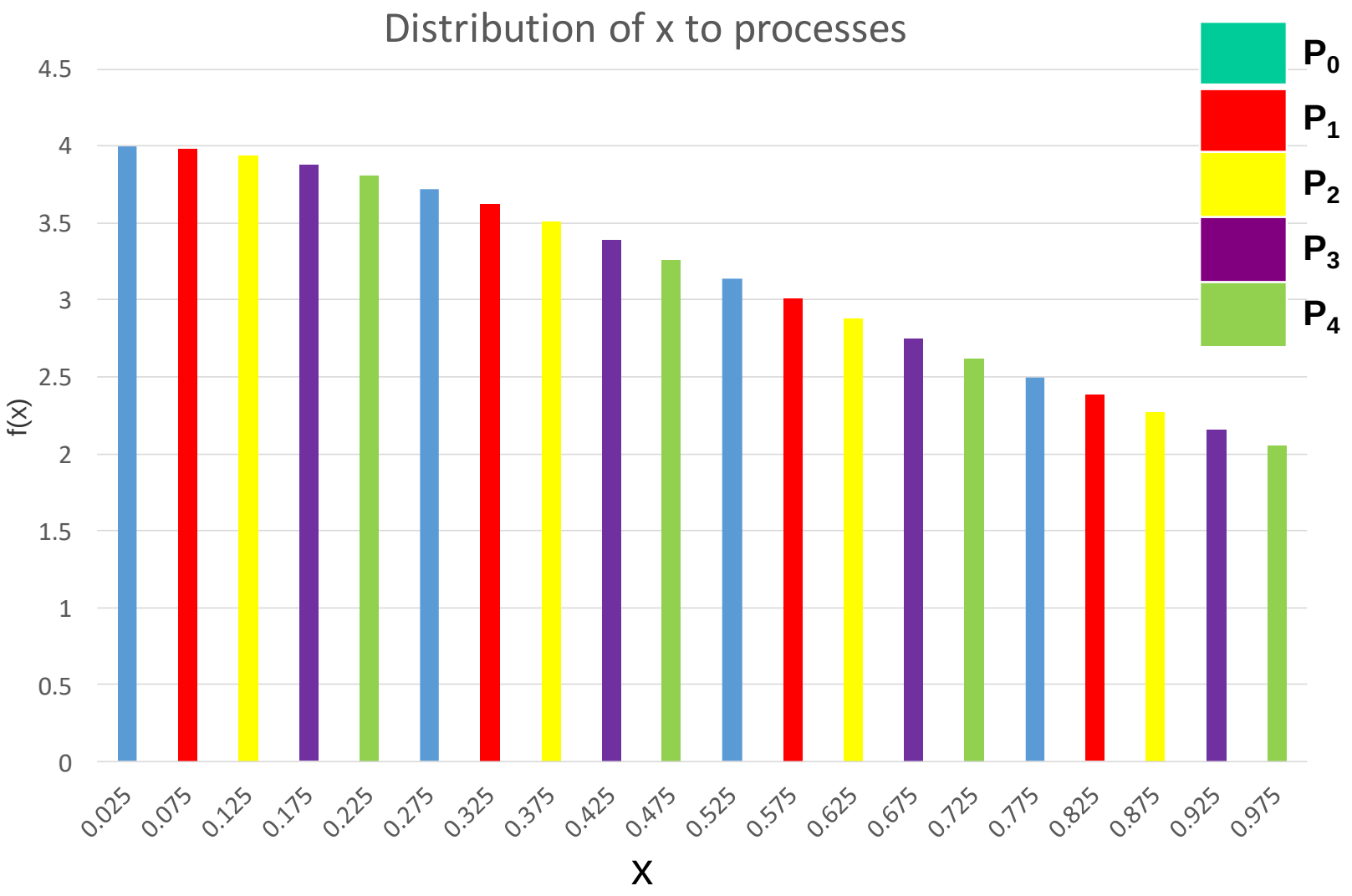
n, numproc are global

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
if (myid == 0)
    {
     printf("Enter the number of intervals:”);
     scanf("%d",&n);
     startwtime = MPI_Wtime();
    }
```

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
h   = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs)
    {
      x = h * ((double)i - 0.5);
      sum += f(x);
    }
    mypi = h * sum;     /area of the rectangle formula
```

Suppose n = 20, numproc = 5 then it will be broadcasted to all processes, h = 1/20 = .05

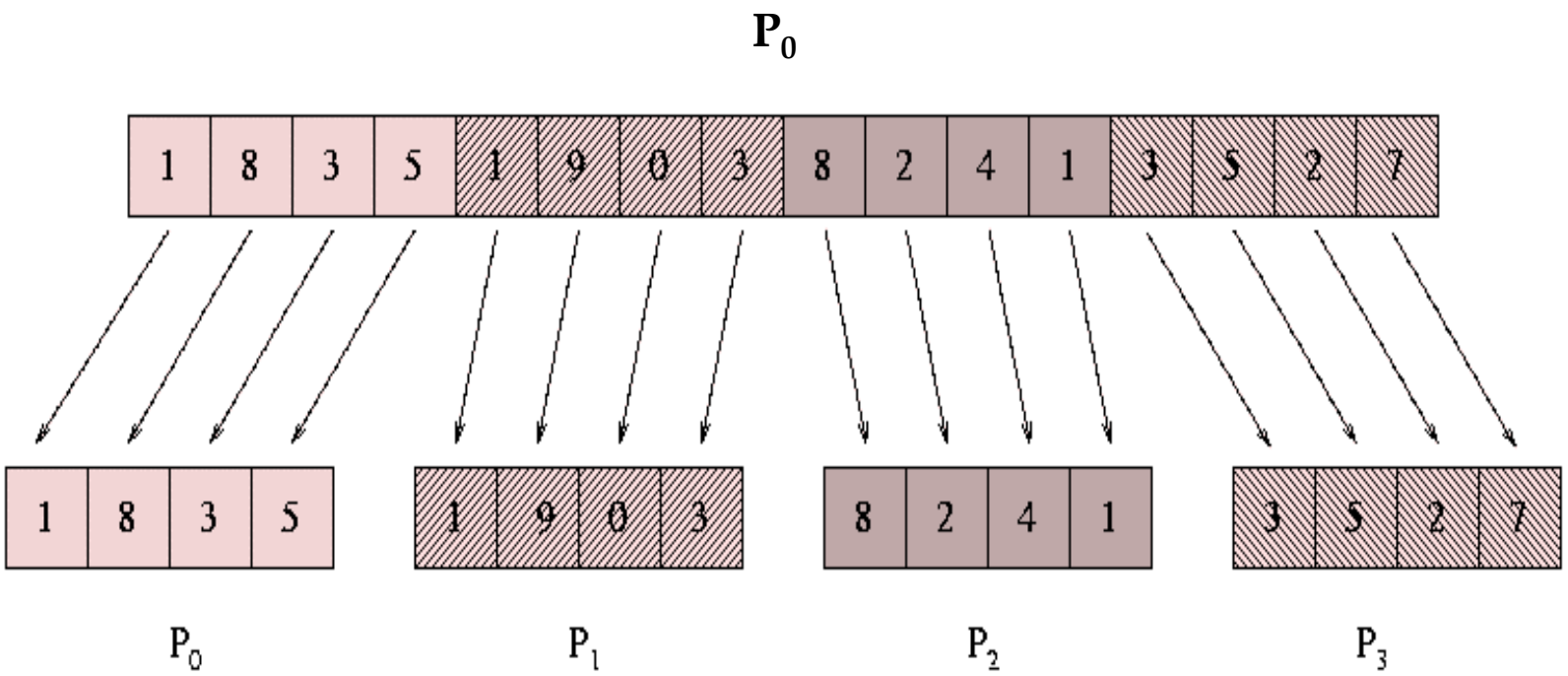
P_0	P_1	P_2	P_3	P_4
sum = 0.0	sum = 0.0	sum = 0.0	sum = 0.0	sum = 0.0
i x f(x)	i x f(x)	i x f(x)	i x f(x)	i x f(x)
1 0.025 3.99	2 0 .075 3.98	3 0.125 3.93	4 0.175 3.88	5 0.225 3.81
6 0.275 3.71	7 0.325 3.62	8 0.375 3.50	9 0.425 3.39	10 0.475 3.26
11 0.525 3.13	12 0.575 3.01	13 0.625 2.87	14 0.675 2.75	15 0.725 2.62
16 0.775 2.49	17 0.825 2.38	18 0.875 2.26	19 0.925 2.16	20 0.975 2.05
sum = 13.35	**sum = 12.99**	**sum = 12.58**	**sum = 12.18**	**sum = 11.74**


Distribution of x to processes
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
f(x)
0.025
0.075
0.125
0.175
0.225
0.275
0.325
0.375
0.425
0.475
0.525
0.575
0.625
0.675
0.725
0.775
0.825
0.875
0.925
0.975
X
P0
P1
P2
P3
P4

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
          if (myid == 0)
            {
             printf("pi is approximately
               %.16f, Error is %.16f\n",pi, fabs(pi
               - PI25DT));
               endwtime = MPI_Wtime();
               printf("wall clock time = %f\n",
                      endwtime-startwtime);
            }
        }
    }
    MPI_Finalize();
}
```

Simple Scatter Program

revisiting Scatter



Simple Scatter program

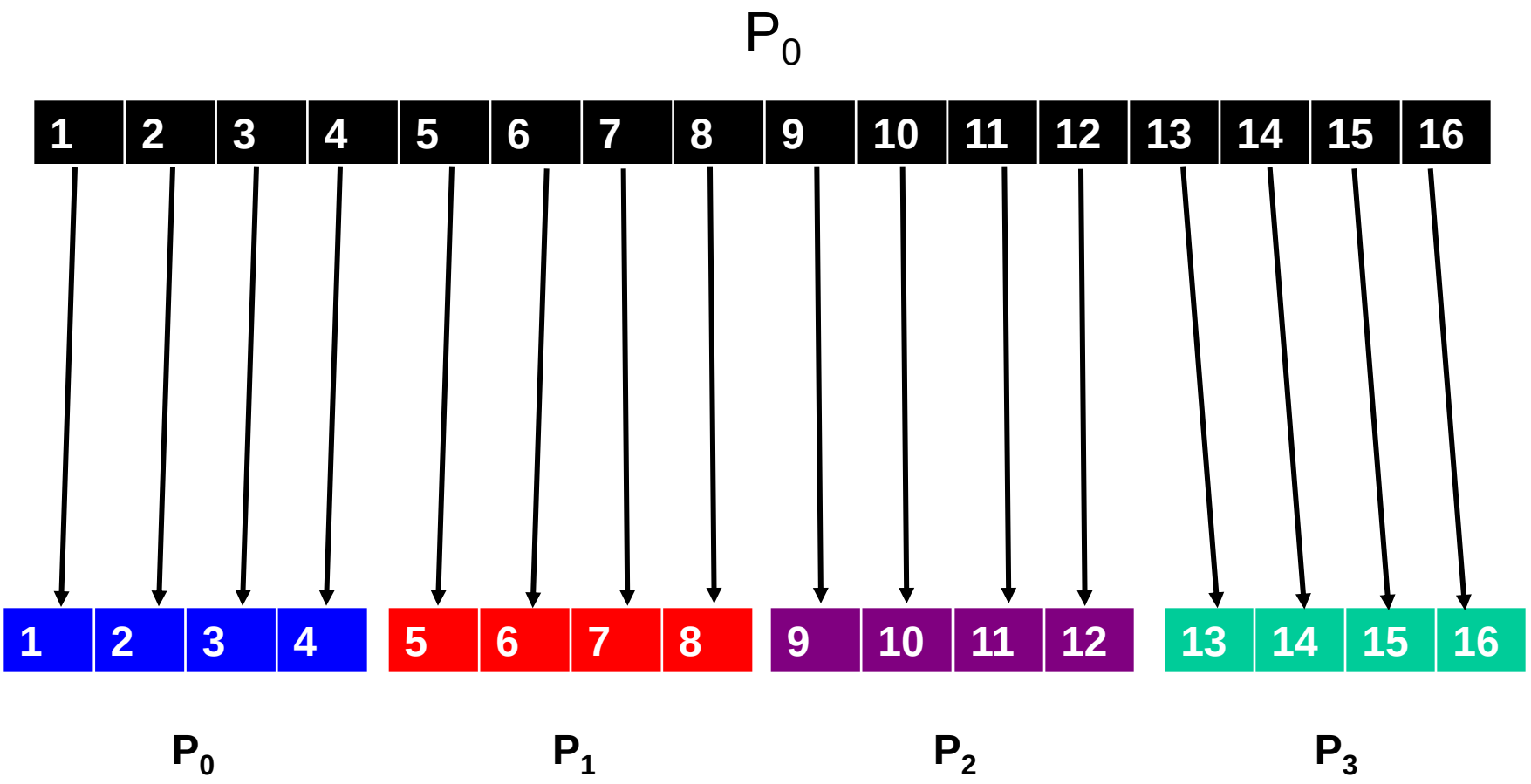
```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
//on 14.139.69.97 program name is scatter.c
int main (int argc, char *argv[])
{
int numtasks, rank, sendcount, recvcount, source;
  float sendbuf[SIZE][SIZE] = {
    {1.0, 2.0, 3.0, 4.0},
    {5.0, 6.0, 7.0, 8.0},
    {9.0, 10.0, 11.0, 12.0},
    {13.0, 14.0, 15.0, 16.0}  };
```

Simple Scatter program

```
float recvbuf[SIZE];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

  if (numtasks == SIZE) {
    source = 1; //it can be any source
    sendcount = SIZE;
    recvcount = SIZE;

MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,r
ecvcount, MPI_FLOAT,source,MPI_COMM_WORLD);
```

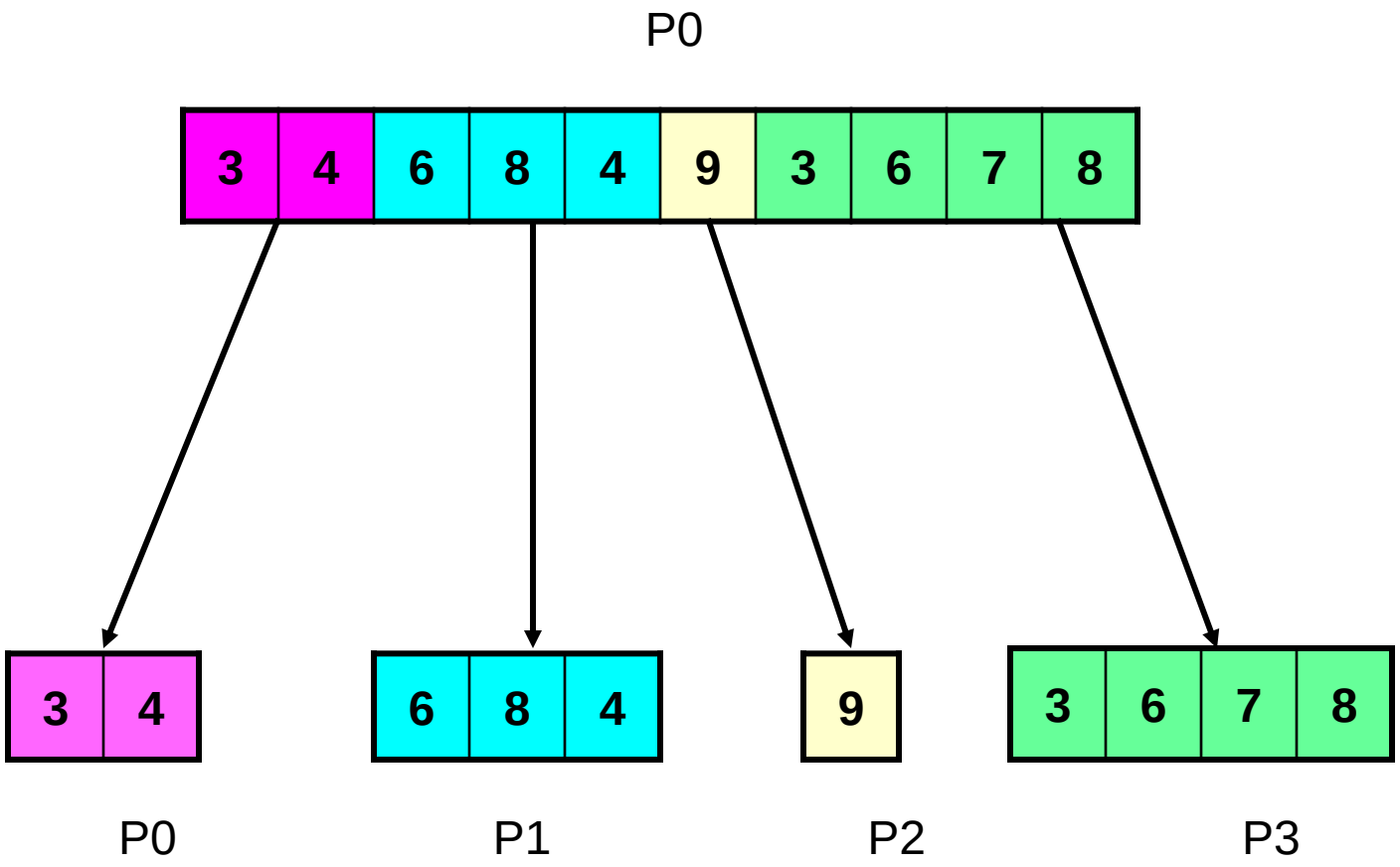
Simple Scatter program

```
printf("rank= %d  Results: %f %f %f
%f\n",rank,recvbuf[0],
recvbuf[1],recvbuf[2],recvbuf[3]);

  }

  else

    printf("Must specify %d processors.
Terminating.\n",SIZE);


  MPI_Finalize();


}
```


P0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
P0
P1
P2
P3

For n = 16, P = 4

Scatterv Operation



MPI SCATTERV(sendbuf, sendcounts, displs, sendtype, recvbuf, recvcount, recvtype, root, comm)

IN	sendbuf	address of send buffer (choice, significant only at root)
IN	sendcounts	integer array (of length group size) specifying the number of elements to send to each processor
IN	displs	integer array (of length group size). Entry i species the displacement (relative to sendbuf from which to take the outgoing data to process i
IN	sendtype	data type of send buffer elements
OUT	recvbuf	address of receive buffer
IN	recvcount	number of elements in receive buffer (integer)
IN	recvtype	data type of receive buffer elements
IN	root	rank of sending process (integer)
IN	comm	communicator

Header for MPI_Scatterv

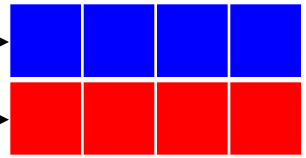
```
int MPI_Scatterv (
    void          *send_buffer,
    int           *send_cnt,
    int           *send_disp,
    MPI_Datatype   send_type,
    void          *receive_buffer,
    int            receive_cnt,
    MPI_Datatype   receive_type,
    int            root,
    MPI_Comm       communicator)
```

Scatterv Example

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
//program name on hpc is scattervv.c
int main (int argc, char *argv[])
{
  int numproc, i, rank, recvcount, source;
  float sendbuf[SIZE][SIZE] = {
    {1.0, 2.0, 3.0, 4.0},
    {5.0, 6.0, 7.0, 8.0},
    {9.0, 10.0, 11.0, 12.0},
    {13.0, 14.0, 15.0, 16.0}  };
```

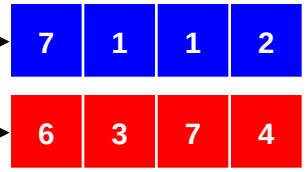
Scatterv Example

```
int sendcount[SIZE];
int displ[SIZE];
```



```
float recvbuf[SIZE*SIZE];
```

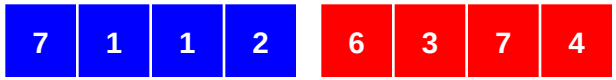
```
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numproc);

if (numproc == SIZE) {
  source = 0;
  for (i = 0; i < SIZE; i++){
      sendcount[i] = rand() % (2*SIZE);
          displ[i] = rand() % (2*SIZE);
  }
```



Scatterv Example

```
recvcount = SIZE*SIZE;

    for (i = 0; i < SIZE*SIZE; i++)
        recvbuf[i] = 0;
```


```
    if (rank==0){
    for (i = 0; i < SIZE; i++){
        printf ("sendcount[%d] = %d, displ[%d] = %d\n", i, sendcount[i], i,
displ[i]);
      }
    }
    printf("\n");
```



Scatterv Example

MPI_Scatterv(sendbuf,sendcount,displ,
MPI_FLOAT,recvbuf,recvcount, MPI_FLOAT,source,
MPI_COMM_WORLD);

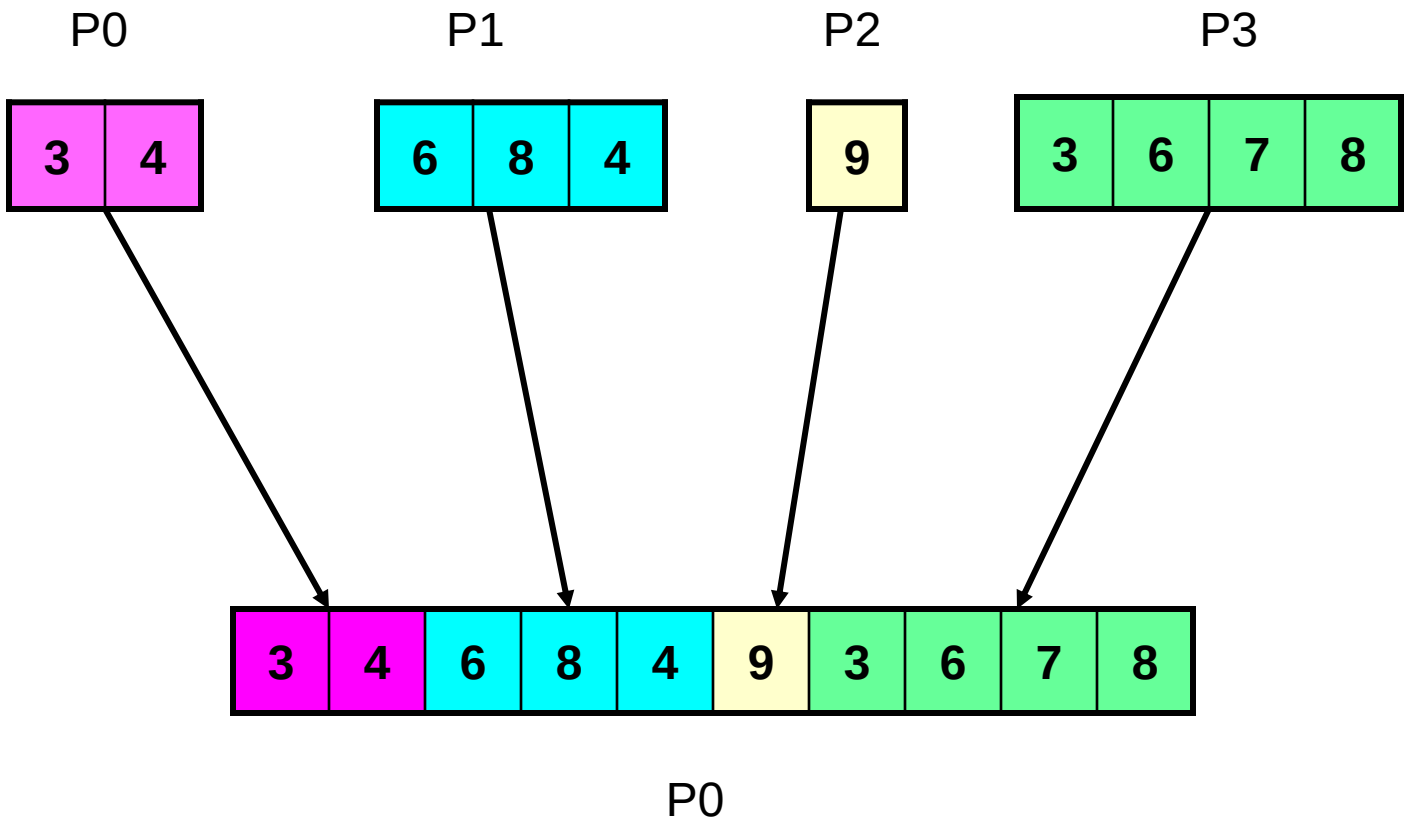
```
    printf("rank= %d\n", rank);
    for (i=0; i<SIZE*SIZE; i++){
    printf("%.1f ",recvbuf[i]);
    }

    printf("\n");
  }
  else
 printf("Must specify %d processors. Terminating.\n",SIZE);
  MPI_Finalize();
}
```

7.0	8.0	9.0	10.0	11.0	12.0	13.0	0	0	0	0	0	0	0	0	0
4.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5.0	6.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Steps

- Download Putty software, free
- Connect to 14.139.69.93
- **ssh** port number **8223** and ***not 22***
- Use any editor (Ex. **vi or nano)** to write c program
- mpicc -o scattervv scattervv.c
- mpirun -np 4 scattervv

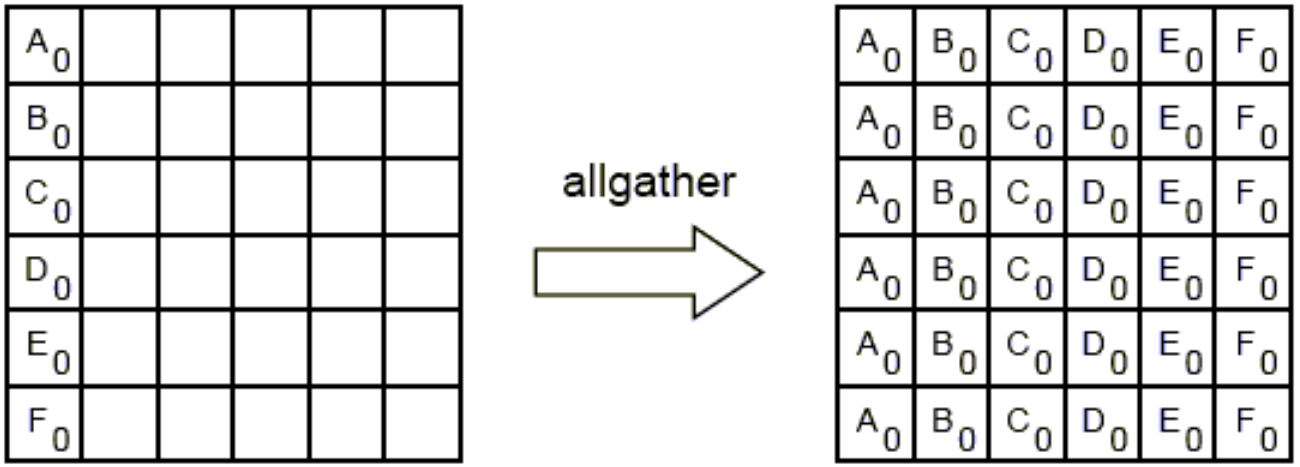
Gatherv Operation



Header for MPI_Gatherv

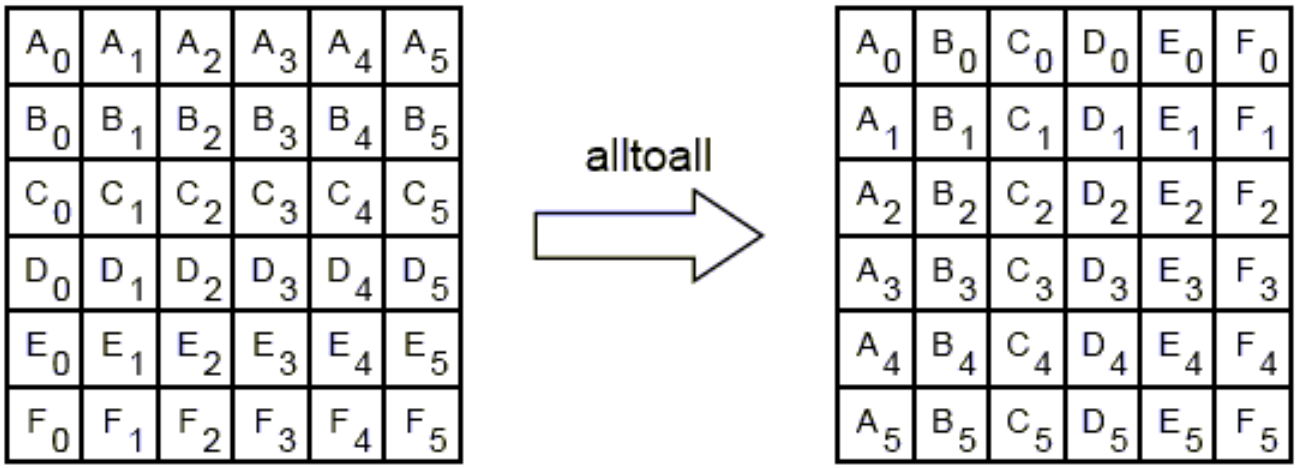
```
int MPI_Gatherv (
    void          *send_buffer,
    int            send_cnt,
    MPI_Datatype   send_type,
    void          *receive_buffer,
    int           *receive_cnt,
    int           *receive_disp,
    MPI_Datatype   receive_type,
    int            root,
    MPI_Comm       communicator)
```

Allgather



Gathers data from all tasks and distribute it to all

Alltoall



- Sends data from all to all processes
- Useful in getting transpose of a matrix

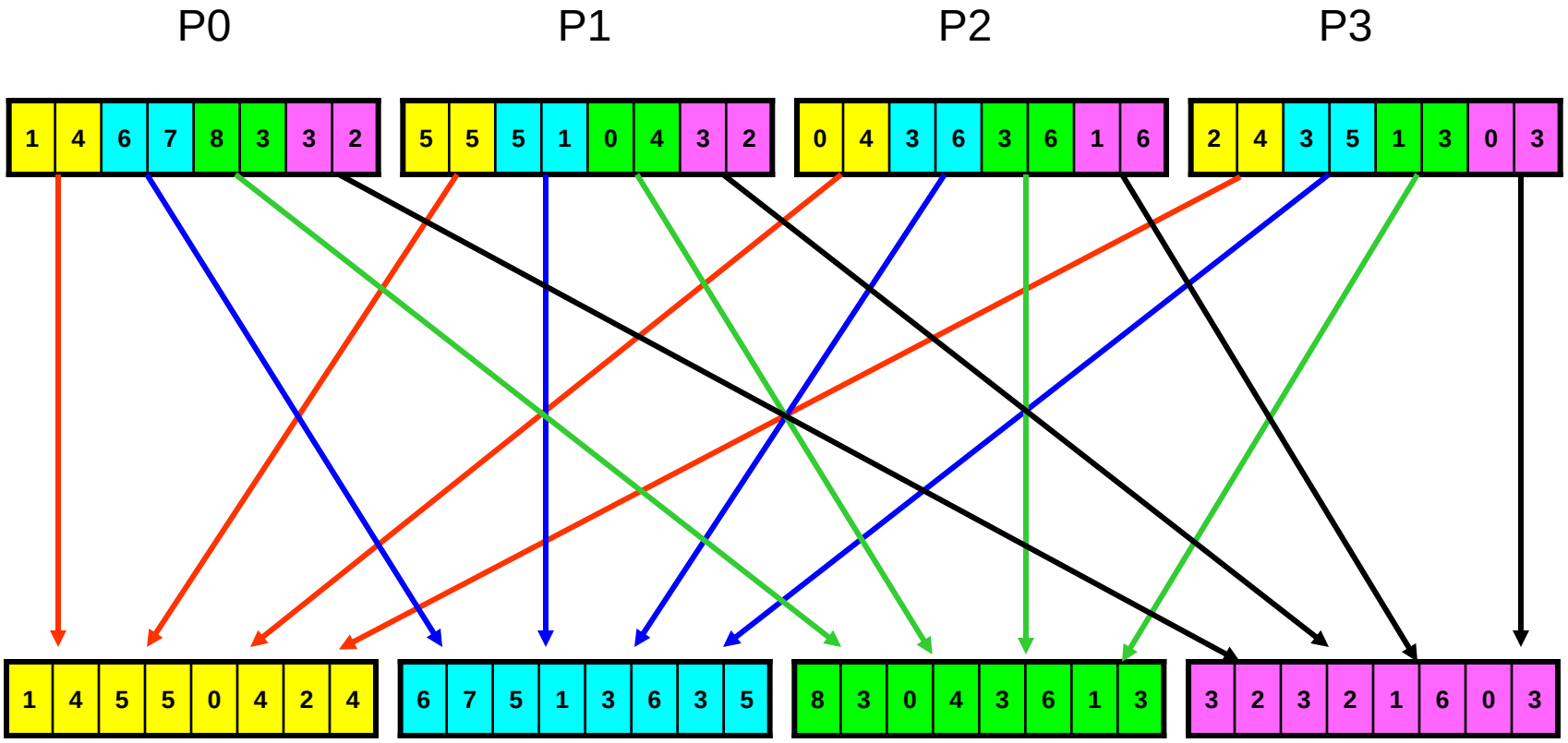
MPI_Alltoall

int **MPI_Alltoall**(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)

Input Parameters

sendbuf	starting address of send buffer
sendcount	number of elements to send to each process (integer)
sendtype	data type of send buffer elements
recvcount	number of elements received from any process (integer)
recvtype	data type of receive buffer elements
comm	communicator

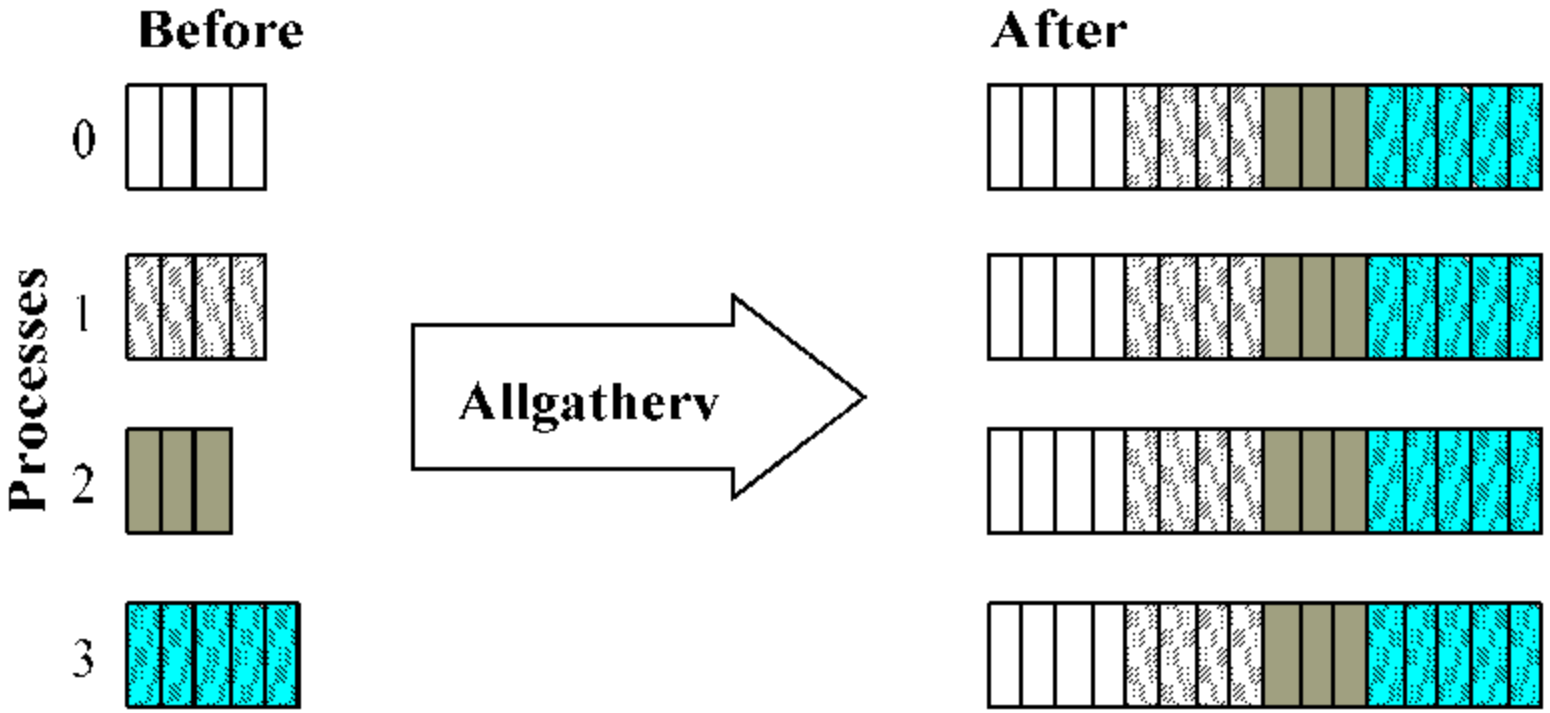
Alltoall Operation



Sends data from all to all processes

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

MPI_Allgatherv



MPI_Allgatherv

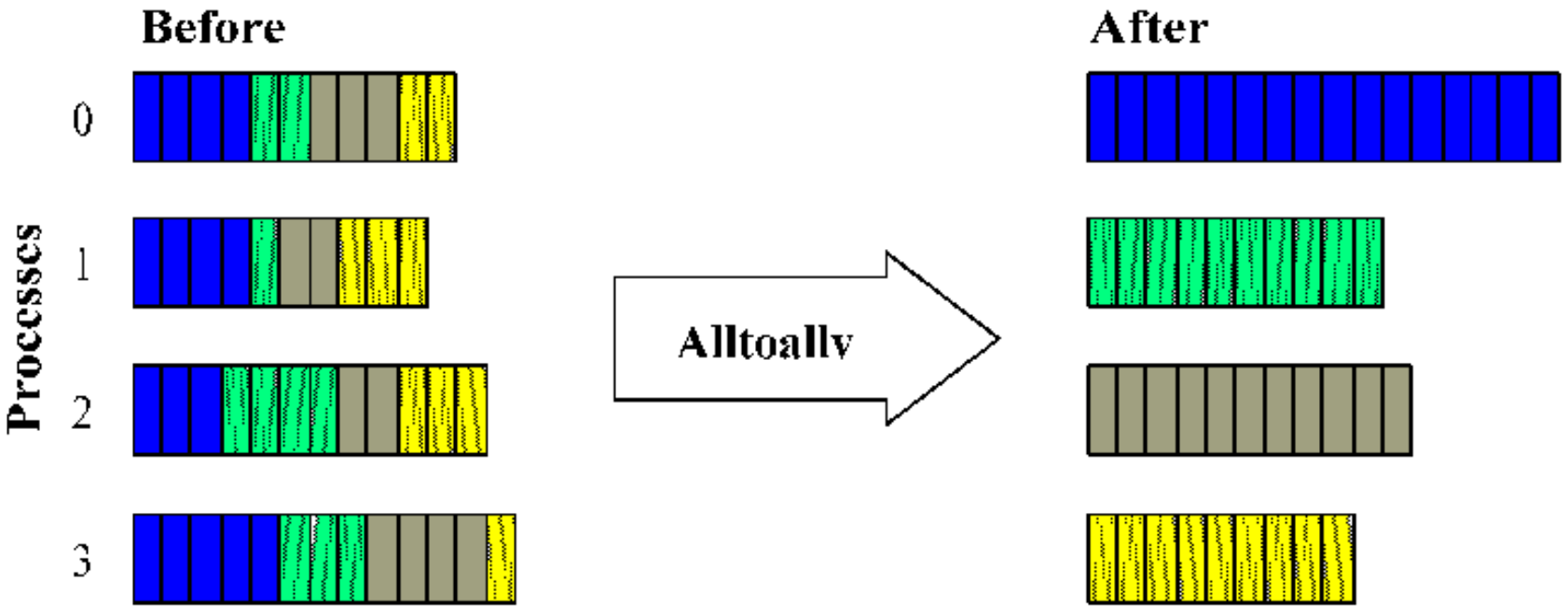
```
int MPI_Allgatherv (
    void            *send_buffer,
    int              send_cnt,
    MPI_Datatype     send_type,
    void            *receive_buffer,
    int             *receive_cnt,
    int             *receive_disp,
    MPI_Datatype     receive_type,
    MPI_Comm         communicator)
```

Input Parameters (MPI_Allgatherv)

sendbuf	starting address of send buffer (choice)
sendcount	number of elements in send buffer (integer)
sendtype	data type of send buffer elements
recvcounts	integer array containing the number of elements that are received from each process
displs	integer array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i
recvtype	data type of receive buffer elements
comm	communicator
recvbuf	address of receive buffer (choice)

The block of data sent from the jth process is received by every process and placed in the jth block of the buffer recvbuf.

Function MPI_Alltoallv



Header for MPI_Alltoallv

```
int MPI_Gatherv (
    void            *send_buffer,
    int             *send_cnt,
    int             *send_disp,
    MPI_Datatype     send_type,
    void            *receive_buffer,
    int             *receive_cnt,
    int             *receive_disp,
    MPI_Datatype     receive_type,
    MPI_Comm         communicator)
```

Matrix-vector Multiplication

Outline

- At least three parallel implementation are possible
  - Rowwise block striped
  - Columnwise block striped
  - Block decomposition

Storing Vectors

- Divide vector elements among processes
- Replicate vector elements
- Vector replication acceptable because vectors have only n elements, versus n^2 elements in matrices

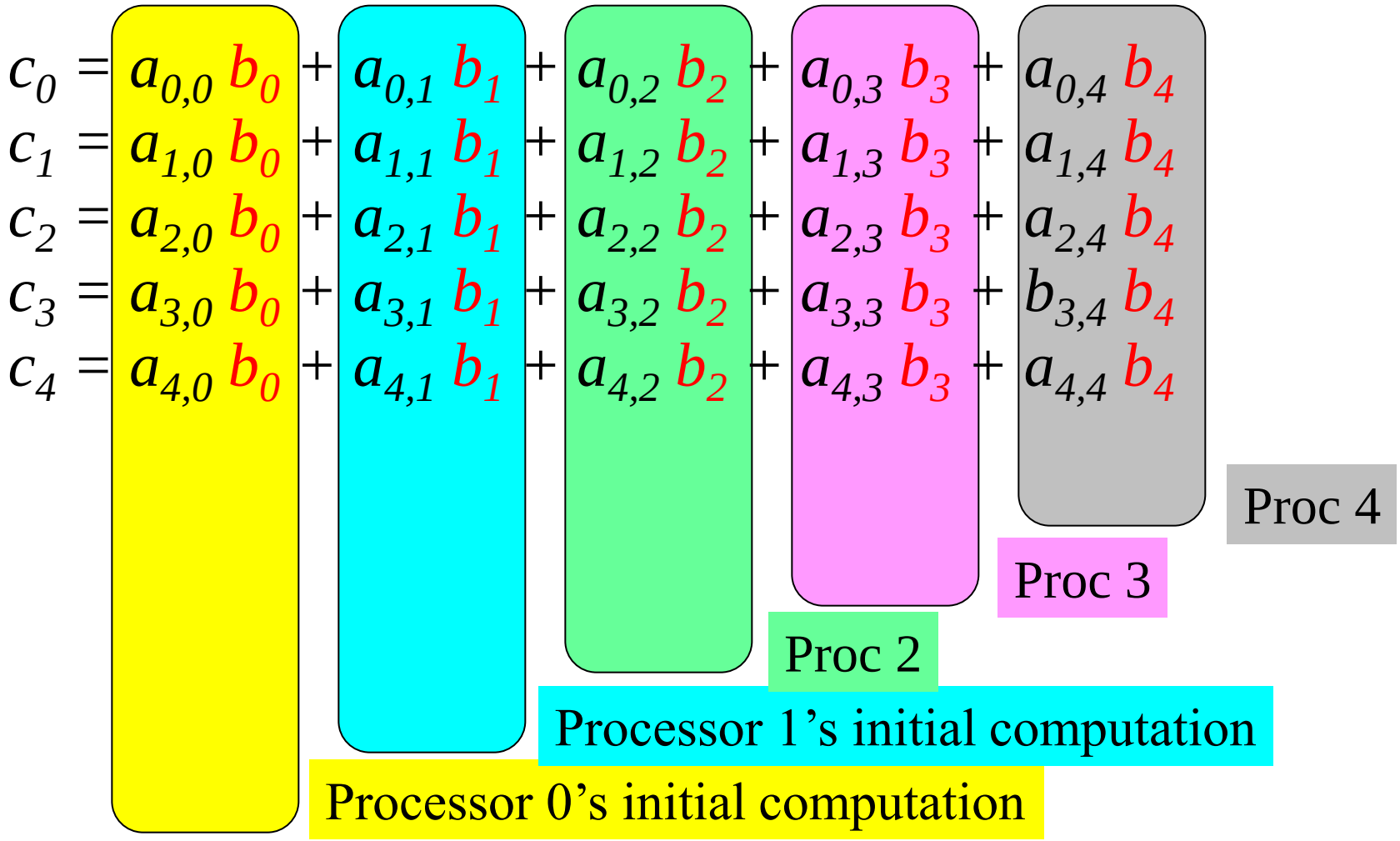
Rowwise Block Striped Matrix

- Partitioning through domain decomposition
- Primitive task associated with
  - Row of matrix
  - Entire vector

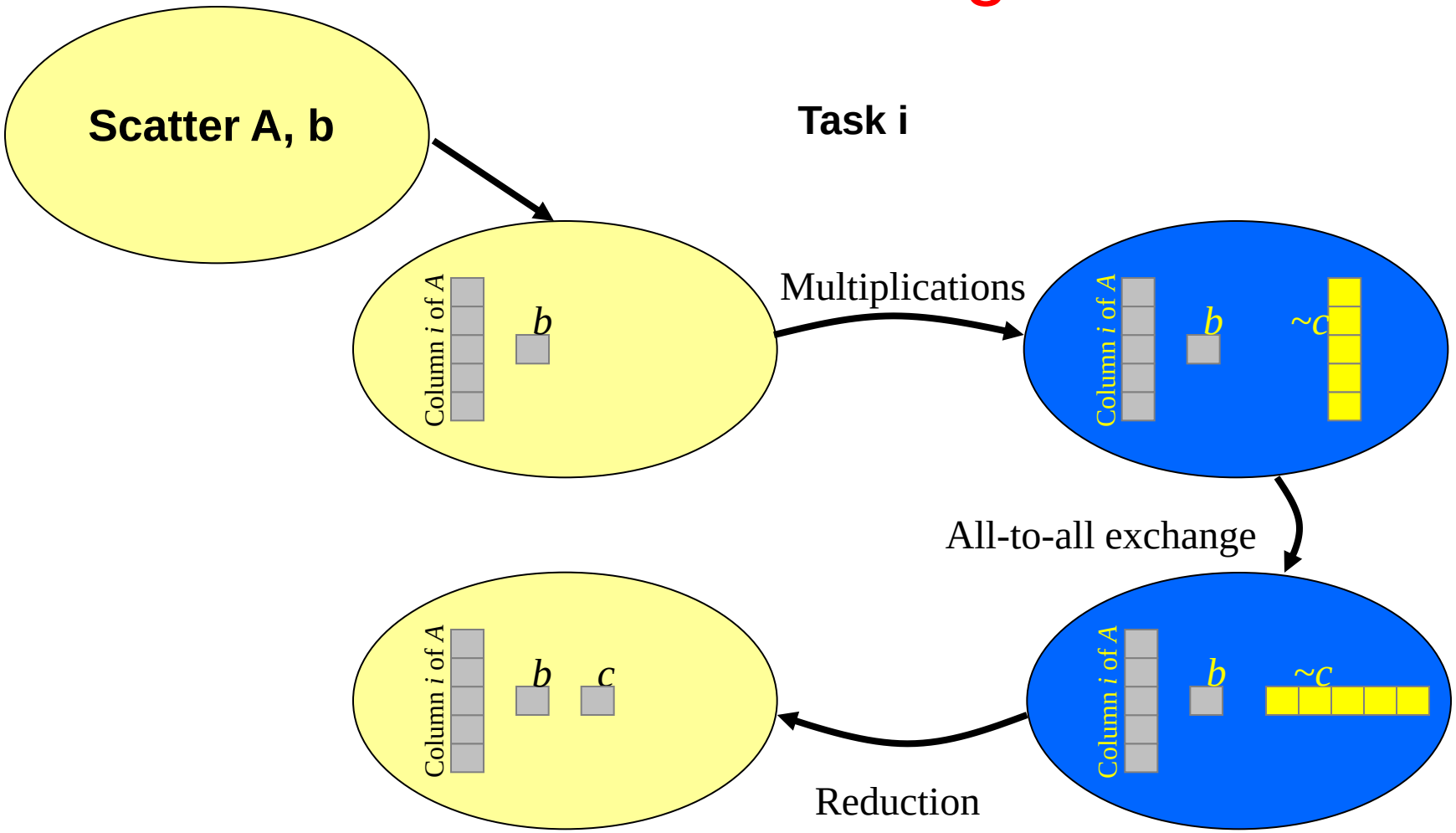
Columnwise Block Striped Matrix

- Partitioning through domain decomposition
- Task associated with
  - Column of matrix
  - Vector element
- MPICH function used are MPI_Scatter, MPI_Gather, MPI_alltoall

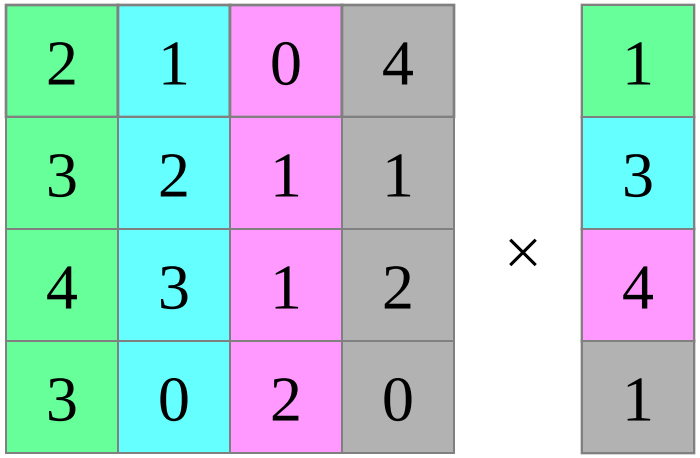
Matrix-Vector Multiplication

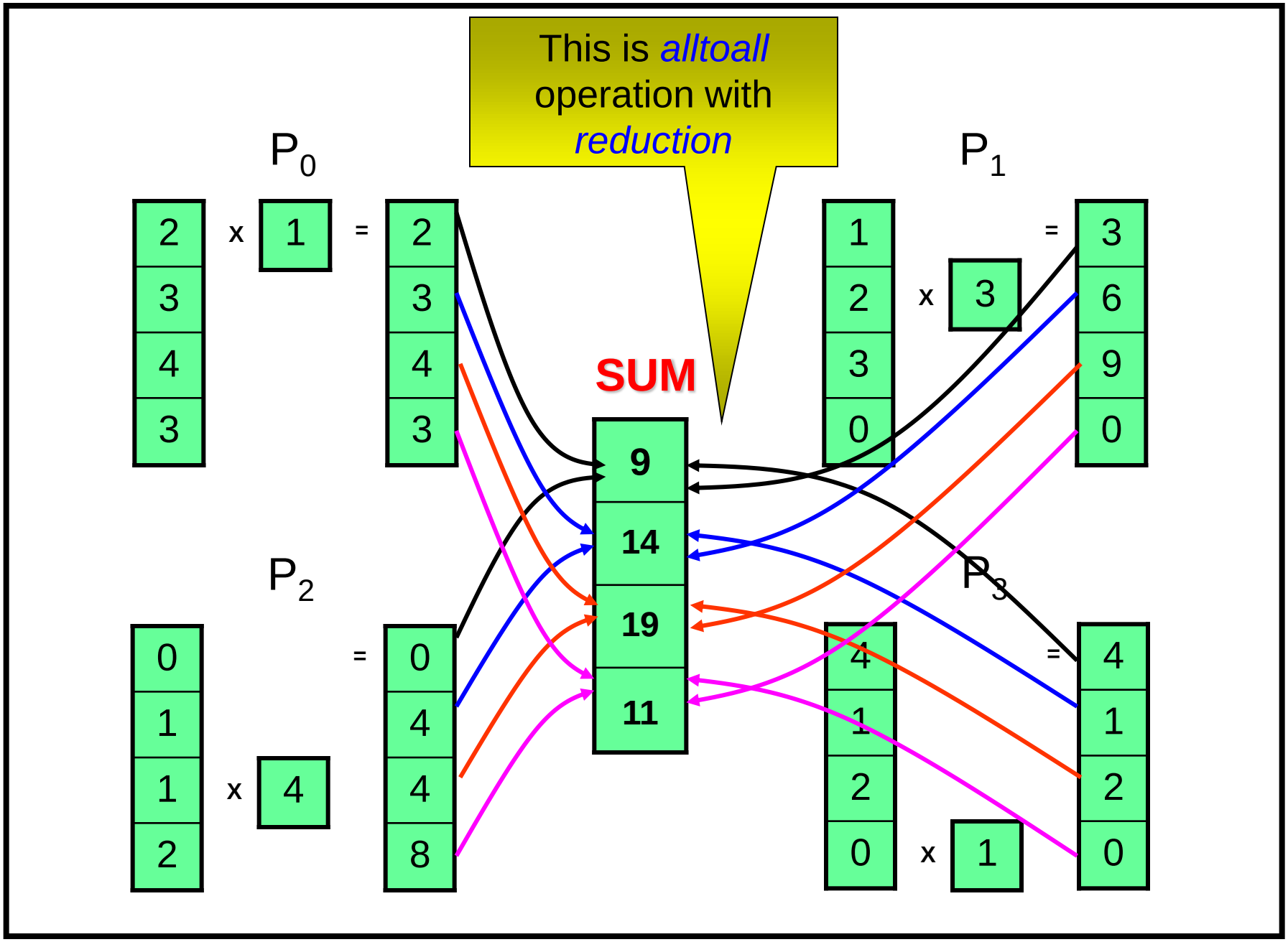


Phases of Parallel Algorithm



Matrix-Vector




This is alltoall operation with reduction
P0
P1
P2
P3
SUM
2
3
4
3
x
1
=
2
3
4
3
1
2
3
0
x
3
=
3
6
9
0
0
1
1
2
x
4
=
0
4
4
8
4
1
2
0
x
1
=
4
1
2
0
9
14
19
11

Evaluating Programs Empirically

Measuring Execution Time

To measure the execution time between point L1 and point L2 in the code, we might have a construction such as.

```
L1:     time(&t1);      /* start timer */..

L2:     time(&t2);      /* stop timer */.

elapsed_time = difftime(t2, t1); /* elapsed_time = t2 - t1 */

printf("Elapsed time = %5.2f seconds", elapsed_time);
```

MPICH provides the routine MPI_Wtime() for returning time (in seconds).

References

- William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high performance, portable implementation of the MPI Message-Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.
- William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.
- William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

References

- Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.
- Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufman, 1997.
- Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core,* 2nd edition. MIT Press, Cambridge, MA, 1998.
- http://www.mcs.anl.gov/mpi/mpich

References

- William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.
- Michael J Quinn. *Parallel Programming in C with MPI and OpenMP*, Tata-McGraw-Hill Edition, 2003.
- Barry Wilkinson And Michael Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, Upper Saddle River, NJ, 1999.