# Message Passing Interfaces

**A. Sahu**

**Dept of CSE, IIT Guwahati**

# Outline

- Basic pf MPI
- MPI Constructs  and Example
- Running programming in IITG HPC system
- Reference and Other Resources

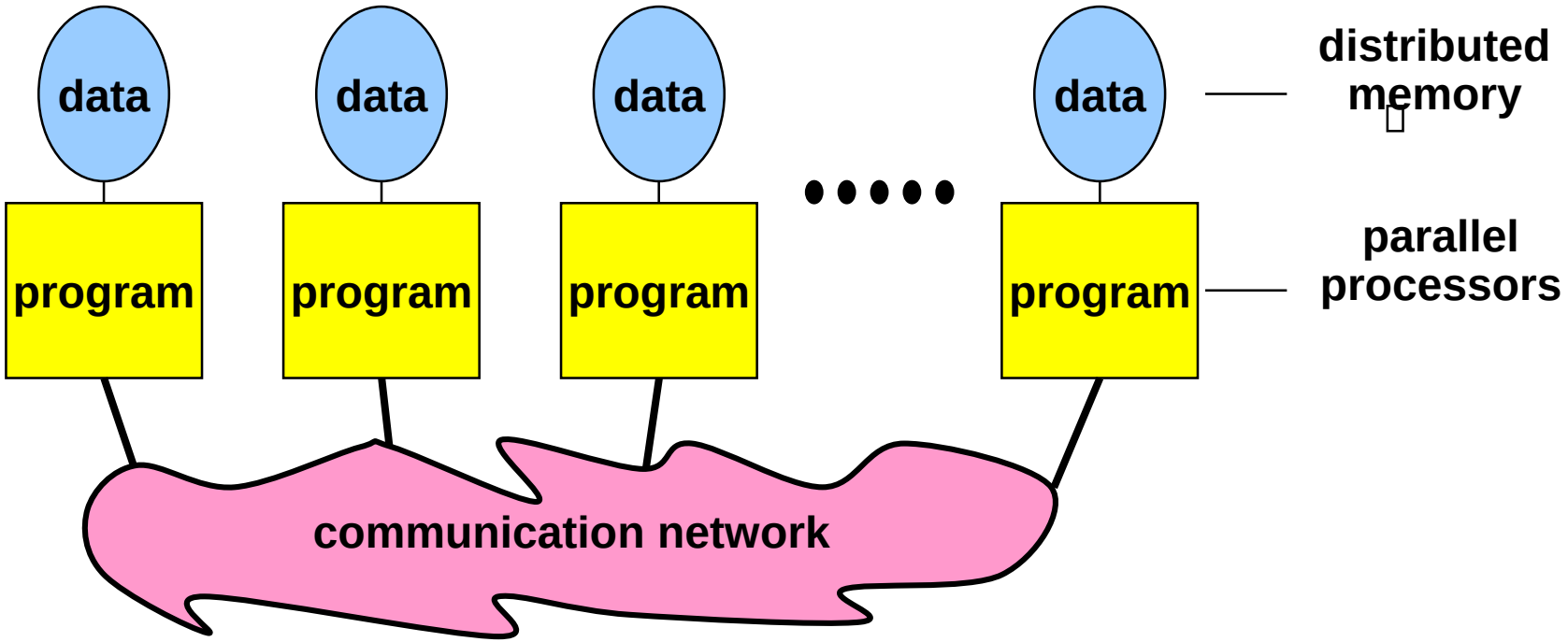
# How to compile and run on a Linux Machine

```
$mpicc hello_mpi.c –o hello_mpi
$mpirun –np 4 ./hello_mpi
```

**4 copies of hello_mpi process will run**

# The Message-Passing Programming Paradigm

- **Message-Passing Programming Paradigm**



# The Message-Passing Programming Paradigm

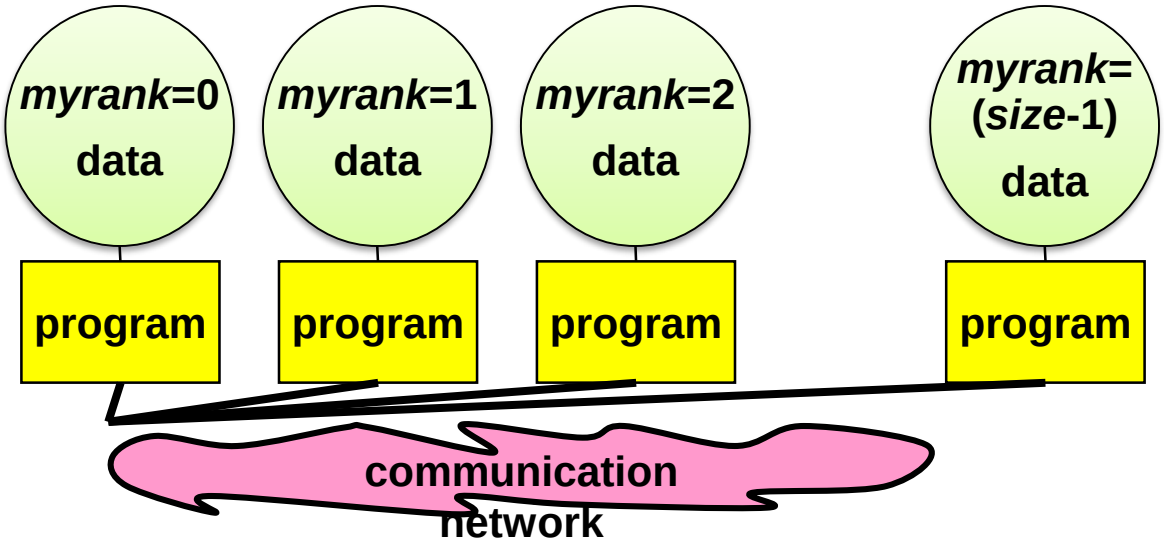
- Typically a single program operating of multiple dataset
- The variables of each sub-program have
  - **The same name**
  - **But different locations (distributed memory) and different data!**
  - **i.e., all variables are local to a process**
- Communicate via special send & receive routines (***message passing***)

# Every process of MPI are different

- Hi : single person : you do
  - Touch you nose by left hand
  - Hi : Touch you head by right hand
- Hi: all persons of this hall do:
  - Touch your nose

# Data and Work Distribution

- To communicate together mpi-processes need identifiers: **rank = identifying number**
- all distribution decisions are based on the ***rank***
  - i.e., which process works on which data



# What is SPMD

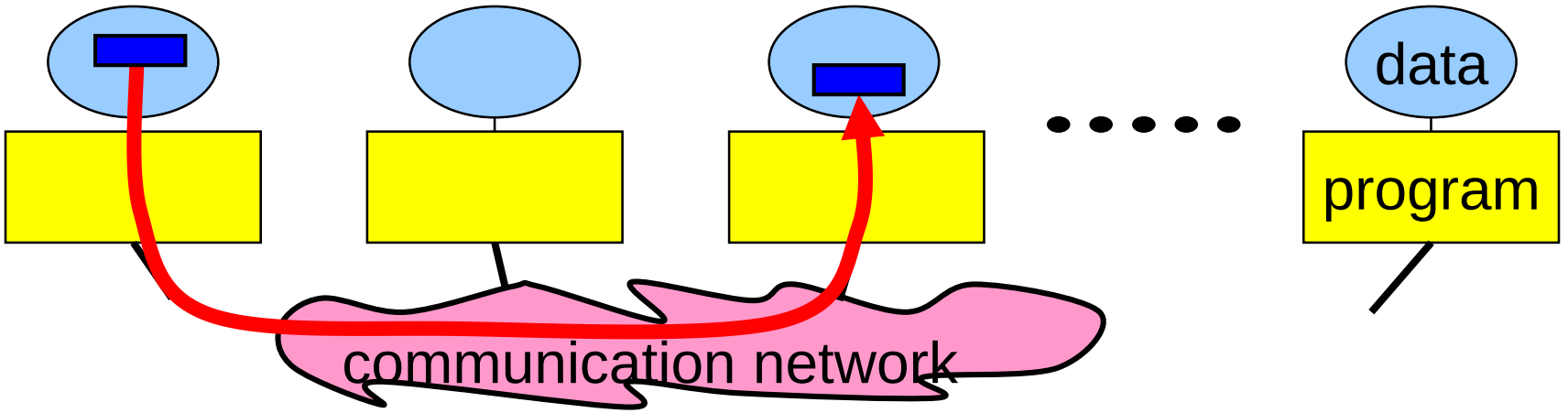
- **S**ingle **P**rogram, **M**ultiple **D**ata
- Same (sub-)program runs on each processor
- MPI allows also MPMD, i.e., **Multiple** Program, ...
  - but some vendors may be restricted to SPMD
  - MPMD can be emulated with SPMD

# Emulation of MPMD

```
main(int argc, char **argv){
      if (myrank < XX){
         ocean( /* arguments */ );
      }else{
        weather( /* arguments */ );
      }
   }
```

# Message passing

- Messages are packets of data moving between sub-programs
- Necessary information for the message passing system:
  - sending process – receiving process i.e., the ranks
  - source location – destination location
  - source data type – destination data type
  - source data size – destination buffer size



# Access

- A sub-program needs to be connected to a message passing system
- A message passing system is similar to:
  - phone line, mail box, fax machine, etc.
- MPI:
  - program must be linked with an MPI library
  - program must be started with the MPI startup tool

# What is message passing?

- Data transfer
- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

# Blocking vs. Non-Blocking

- Blocking
  - The program will not continue
  - Until the communication is completed.
- Non-Blocking
  - The program will continue
  - Without waiting for the communication to be completed.

# Features of MPI

- General
  - Communications combine context and group for message security.
  - Thread safety can't be assumed for MPI programs.

# Features that are NOT part of MPI

- Process Management
- Remote memory transfer
- Threads
- Virtual shared memory

# Why to use MPI?

- MPI provides a
  - Powerful, efficient, and portable way to express parallel programs.

# Why to use MPI?

- MPI provides a
  - Powerful, efficient, and portable way to express parallel programs.
- MPI was explicitly designed
  - To enable libraries which eliminate the need for many users to learn **much inside of MPI.**

# Why to use MPI?

- MPI provides a
  - Powerful, efficient, and portable way to express parallel programs.
- MPI was explicitly designed
  - To enable libraries which eliminate the need for many users to learn **much inside of MPI.**
- Portable !!!!!!!!!!!!!!!!!!!!!!!!!!!
- Good way to learn about subtle
  - Issues in parallel computing

# How big is the MPI library?

- Huge ( 125 Functions ).
- Basic ( 6 Functions ).

# How to install MPI in Linux Cluster

**MPI Library**

- **LAM : Local Area Multiprocessor**
- **MPI CH: Argon National Laboratory**

# Skeleton MPI Program

```
#include <mpi.h>

void main( int argc, char **argv )
{
    MPI_Init( &argc, &argv );

  /* main part of the program */

  /*
   Use MPI function call depend on
  your data partitioning and the
  parallelization architecture
  */
    MPI_Finalize();
}
```

# Initializing MPI

- The initialization routine MPI_INIT is the first MPI routine called.
- MPI_INIT is called once

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

# A minimal MPI program

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  printf("Hello, world!\n");
  MPI_Finalize();
  return 0;
}
```

# How to compile and run on a Linux Machine

```
$mpicc hello_mpi.c –o hello_mpi
$mpirun –np 4 ./hello_mpi
```

**4 copies of hello_mpi process will run**

# A minimal MPI program cont.

- #include <mpi.h>
  - Provides basic MPI definitions and types.
- MPI_Init starts MPI
- MPI_Finalize exits MPI
- **Note that all non-MPI routines are local; thus "printf" run on each process**
- MPI functions return
  - Error codes or MPI_SUCCESS

# Improved Hello.c

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
   int rank, size;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   printf("I am %d of %d\n", rank, size);
   MPI_Finalize();
   return 0;
}
```

# MPI Basic Communication Constructs

# MPI Concepts

- The default communicator is the `MPI_COMM_WORLD`
- A process is identified
  - By its rank in the group associated with a communicator.

# Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?

# Finding Out About the Environment

- MPI provides functions to answer these questions:
  - **`MPI_Comm_size`** reports the number of processes.
  - **`MPI_Comm_rank`** reports the *rank*, a number between 0 and size-1, identifying the calling process

# MPI: Data Types

- The data message which is sent or received is described by a triple
  - Address, count, data type
- The following data types are supported
  - Predefined data types
  - Arrays, sub blocks of a matrix and user defined

# MPI blocking send

```
MPI_Send(void *start, int count,
         MPI_DATATYPE datatype,
         int Dest, int Tag,
         MPI_COMM comm );
```

# MPI blocking send

```
MPI_Send(void *start, int count,
         MPI_DATATYPE datatype,
         int Dest, int Tag,
         MPI_COMM comm );
```

- The message buffer is described by (start, count, datatype).
- Dest is the rank of the target process in the defined communicator
- Tag is the message identification number.

# MPI blocking receive

```
MPI_Recv(void *start, int count,
      MPI_DATATYPE datatype,
      int Source, int Tag,
      MPI_COMM comm, MPI_COMM *S);
```

# MPI blocking receive

```
MPI_Recv(void *start, int count,
      MPI_DATATYPE datatype,
      int Source, int Tag,
      MPI_COMM comm, MPI_COMM *S);
```

- **Source** is the rank of the sender in the communicator.
- The receiver can specify
  - A wildcard value for source (MPI_ANY_SOURCE)
  - A wildcard value for tag (MPI_ANY_TAG),
  - Indicating that any source and/or tag are acceptable

# MPI blocking receive

```
MPI_Recv(void *start, int count,
      MPI_DATATYPE datatype,
      int Source, int Tag,
      MPI_COMM comm, MPI_COMM *S);
```

- **Status** is used for exrtra information
  - About the received message if a wildcard receive mode is used
- If the count of the message received is <= described by the MPI receive command
  - Message is successfully received
  - Else it is considered as a buffer overflow error.

# Sources of Deadlocks

- Send a large message from proc 0 to proc 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with

| Process 0 | Process 1 |
| --- | --- |
| **Send(1)** | **Send(0)** |
| **Recv(1)** | **Recv(0)** |

  - This is called "unsafe" because it depends on the availability of system buffers

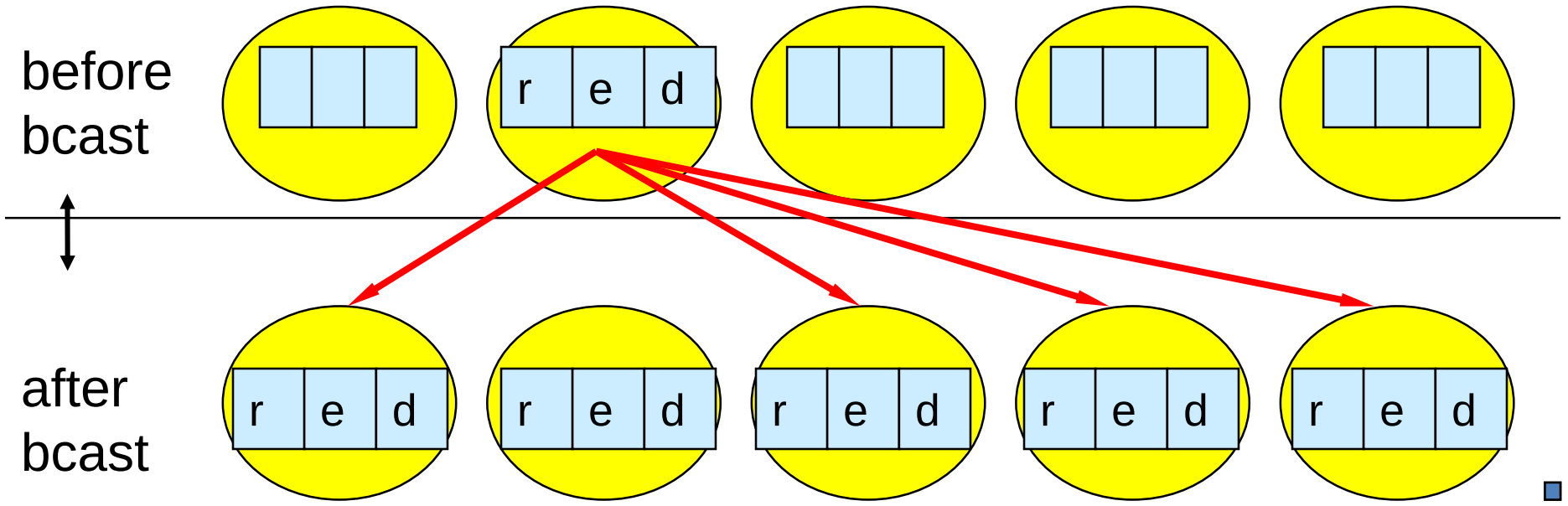
# MPI Collective Communication Constructs

# Collective Communication

- **Optimised Communication routines involving a group of processes**
- Collective action over a communicator
  - i.e. all processes must call the collective routine
- Synchronization may or may not occur
- All collective operations are blocking
  - and No tags
- Receive buffers must have exactly the same size as send buffers.

# Broadcast

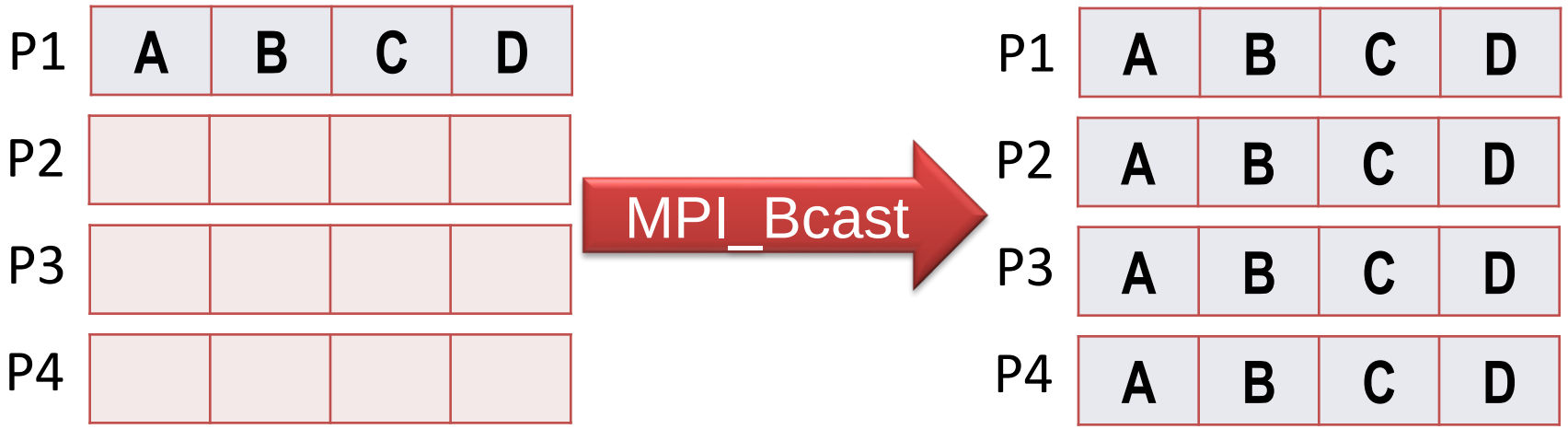
```
int MPI_Bcast(void *buf, int count,
              MPI_Datatype datatype,
              int root, MPI_Comm comm);
```




e.g., root=1

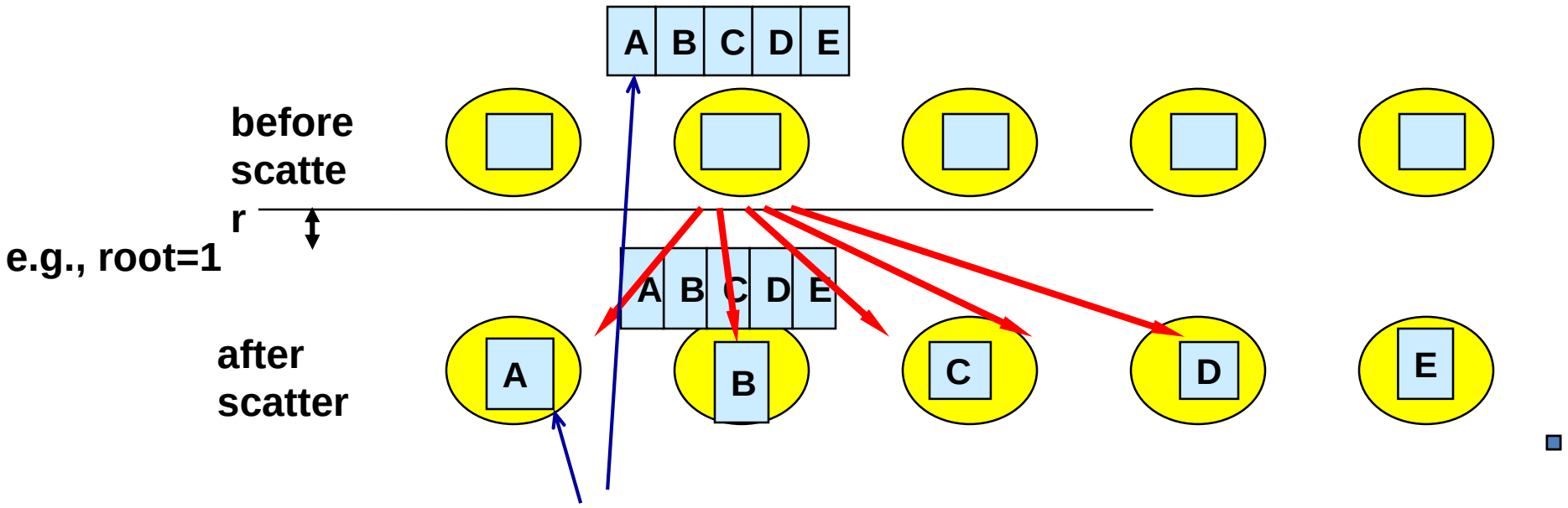
- rank of the sending process (i.e., root process)
- must be given identically by all processes

# Broadcast

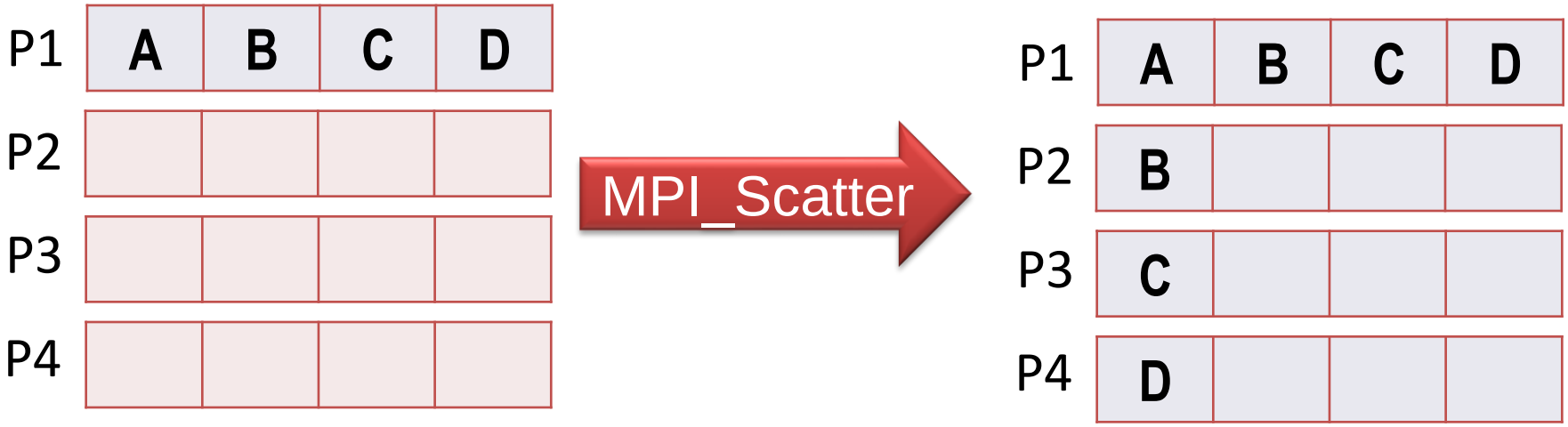


# Scatter

```
int MPI_Scatter(void *sendbuf, int
    sendcnt, MPI_Datatype  sendtype,
    void *recvbuf, int recvcount,
    MPI_Datatype recvtype,
    int root, MPI_Comm comm);
```

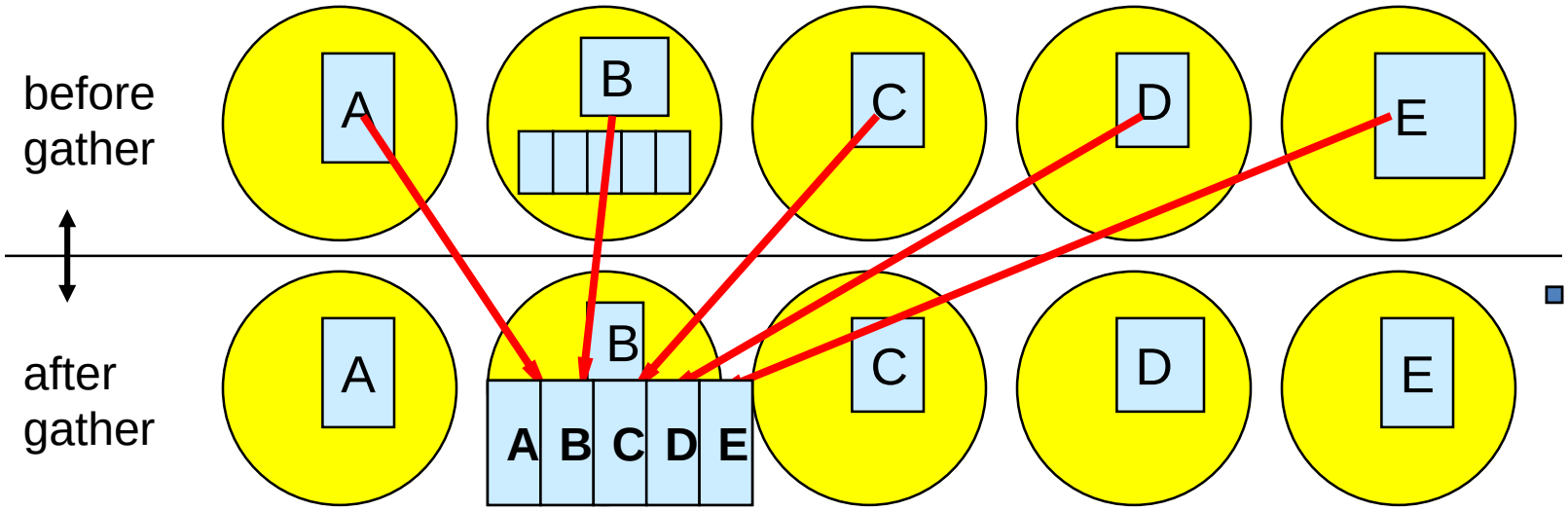


# Scatter



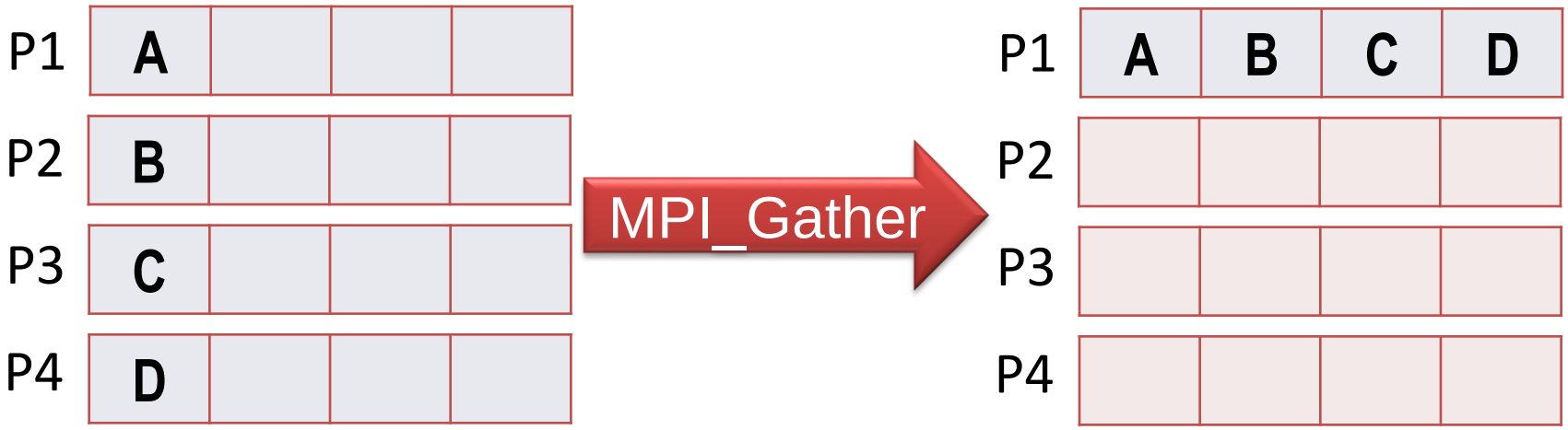
# Gather

```
int MPI_Gather(void *sendbuf,
  int sendcnt, MPI_Datatype
 sendtype, void *recvbuf,
 int recvcnt,  MPI_Datatype
 recvtype, int root, MPI_Comm comm)
```

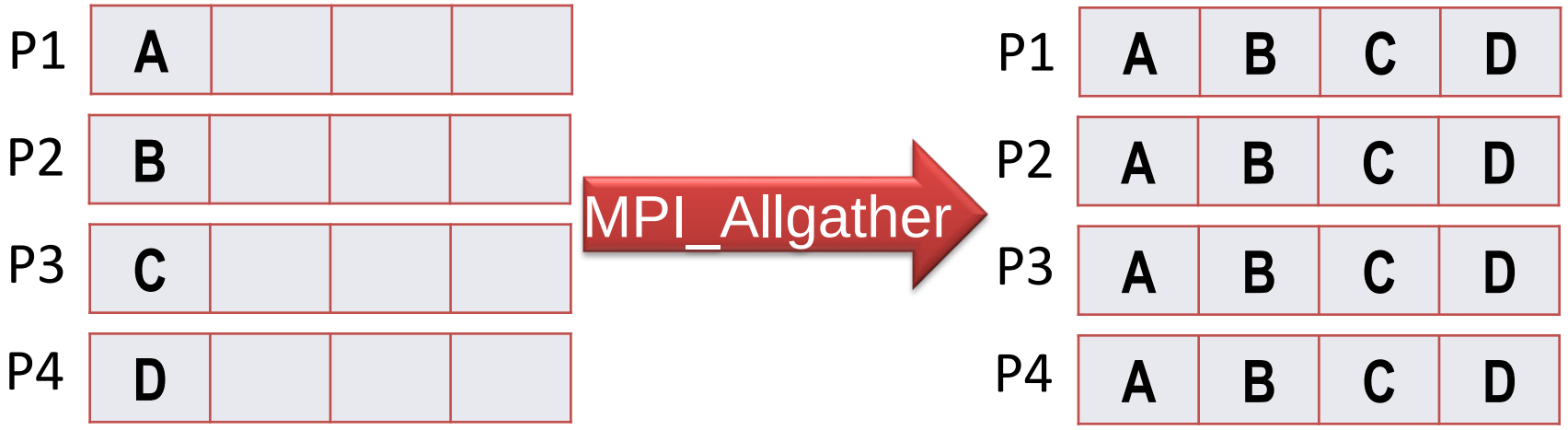



e.g., root=1

# Gather



# Gather to All



# Global Reduction Operations

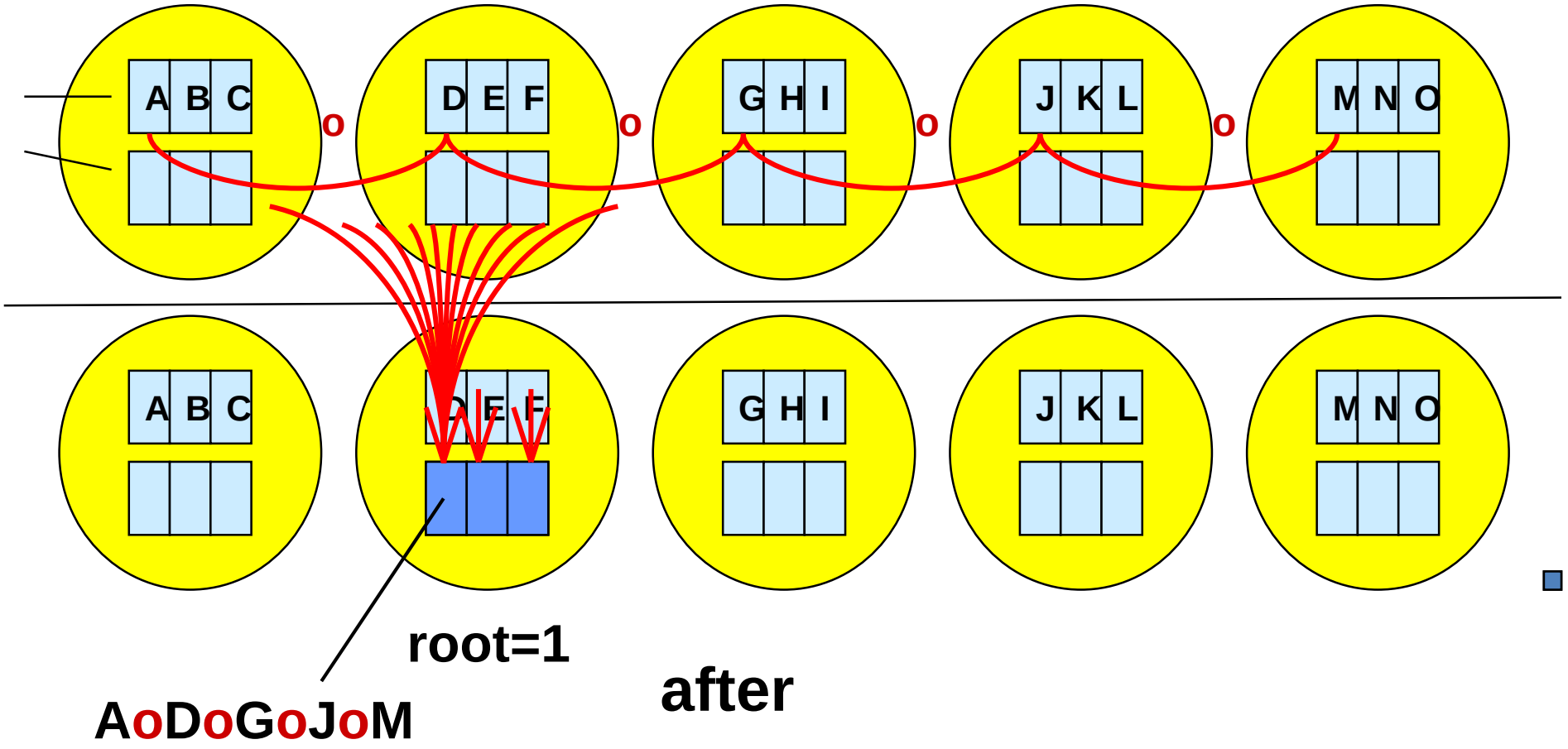
- Perform a global reduce operation across all members of a group.

$$d_0 \; o \; d_1 \; o \; d_2 \; o \; d_3 \; o \; \ldots \; o \; d_{s-2} \; o \; d_{s-1}$$

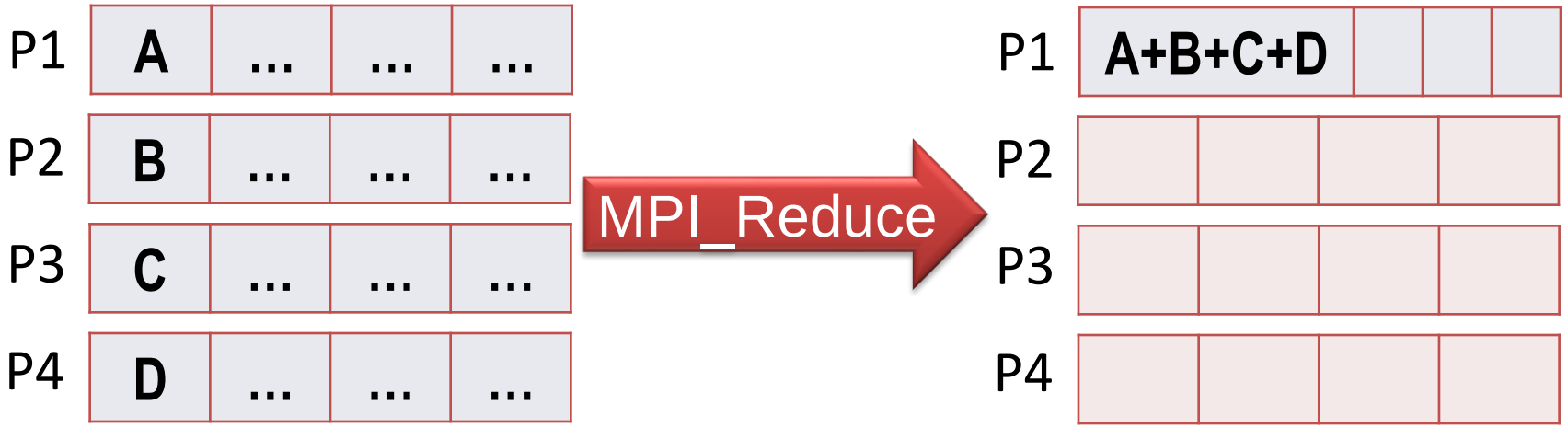
- $d_i$ = data in process rank i
  - –single variable, or vector
- **o** = associative operation
- Example:
  - –global sum or product
  - –global maximum or minimum
  - –global user-defined operation

# MPI_Reduce

before MPI_REDUCE
- inbuf and result



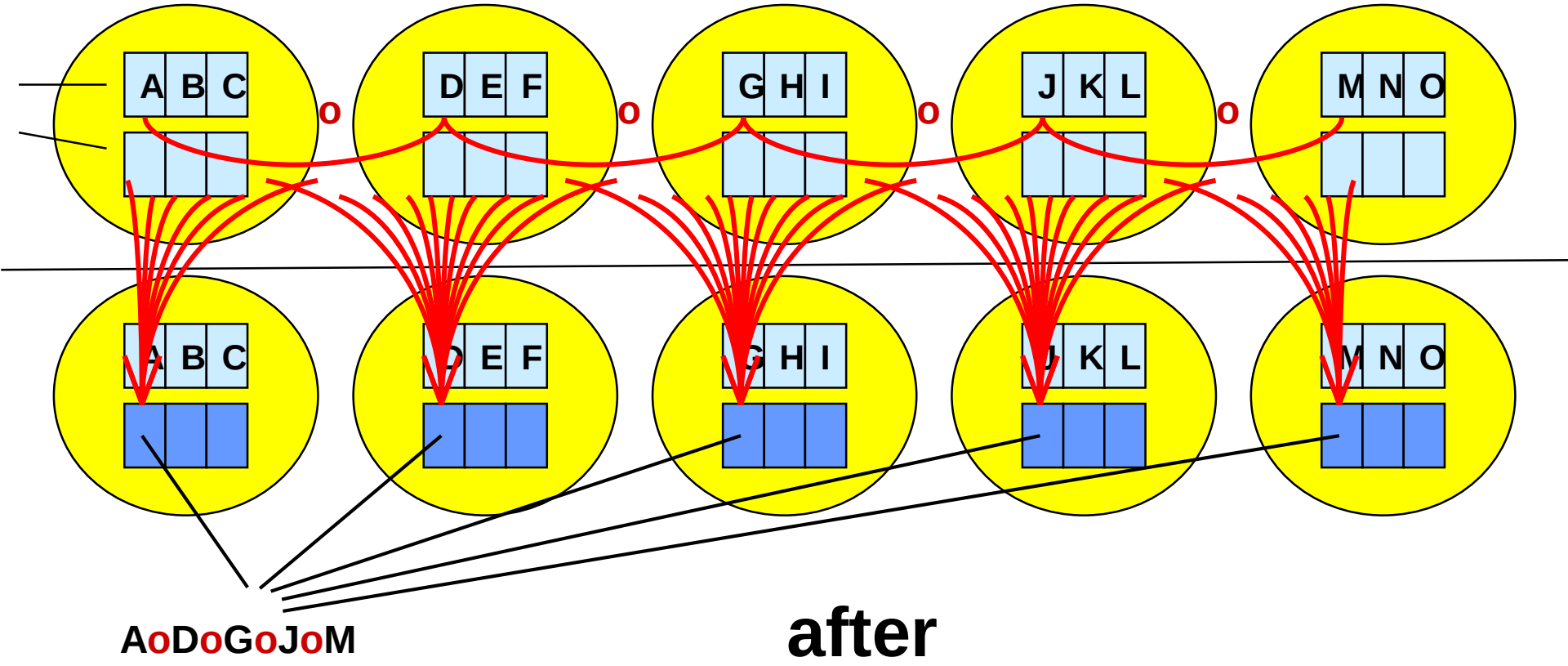
# MPI_Reduce



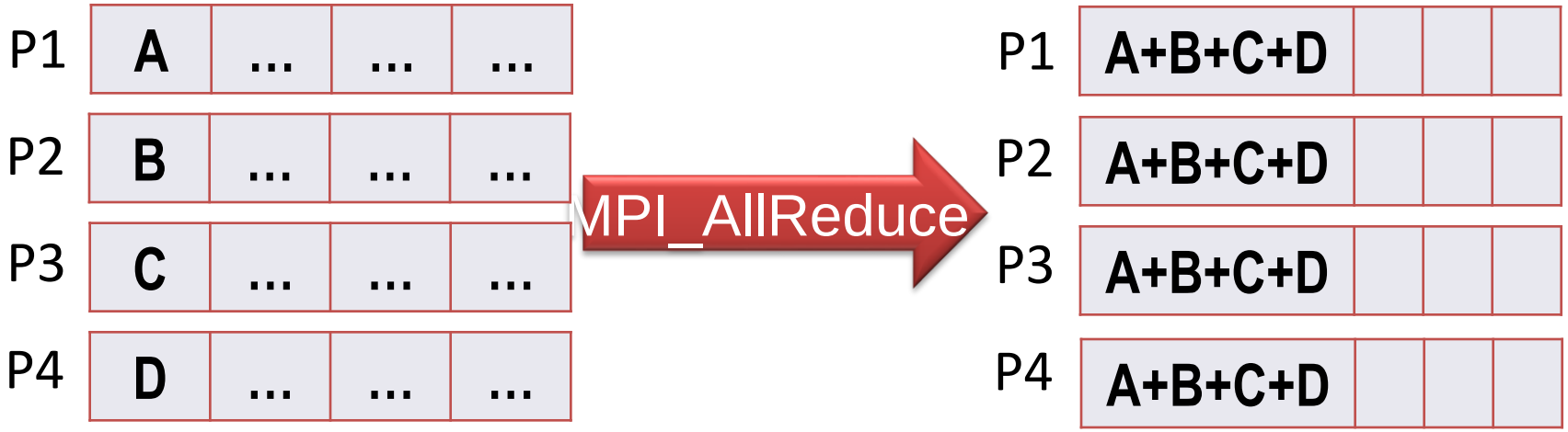
# MPI_AllReduce

before MPI_ALLREDUCE

- inbuf and result



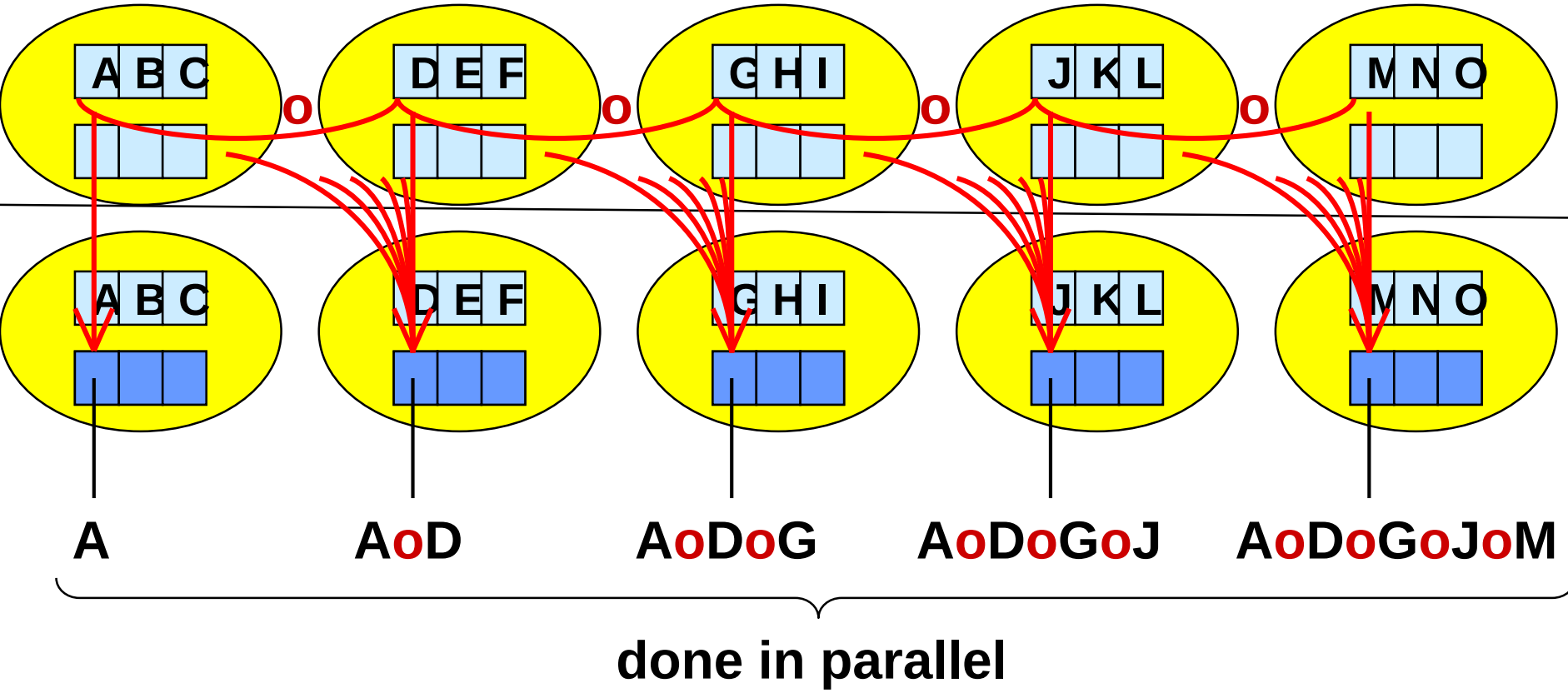
# MPI_AllReduce



# MPI_Scan

**before MPI_SCAN**

- **inbuf and result**




after

# MPI Examples

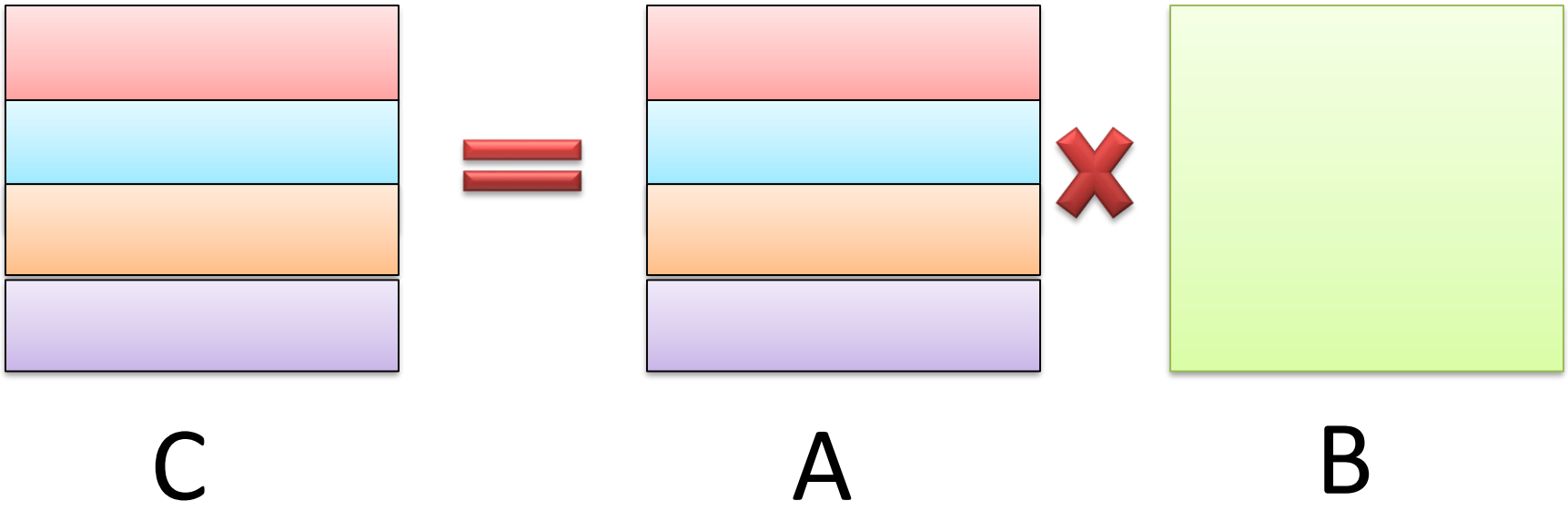
# Example: Sum of N data

- Master Process
  - Data to be read by process 0 or MASTER
  - Divide the data in to N/M chunk size (N %M==0)
  - **SEND** respective chunk of data to other process
  - **Do local sum on each process (in master also)**
  - **RECV** sum of other process and calculate final sum
- Other Process
  - **RECV** data from Mater
  - **Do local sum on each process**
  - **SEND** local sum to MASTER

# See the Code

# Example: Matrix MUL

- c=axb: a[NRA][NCA], b[NCA][NCB], c[NRA][NCB]
- Work get divided: Based on Rows



# Example: Matrix MUL

- c=axb: a[NRA][NCA], b[NCA][NCB], c[NRA][NCB]
- One Master Processor
- Many Workers, Assume NRA % NumWorker==0
  - Master divide the work between worker
  - Send respective rows of A and whole B to workers
  - RECV array C from all worker
- Every Worker
  - get some Row of A, Whole of B
  - calculate part of C
  - Send calculated C to Master

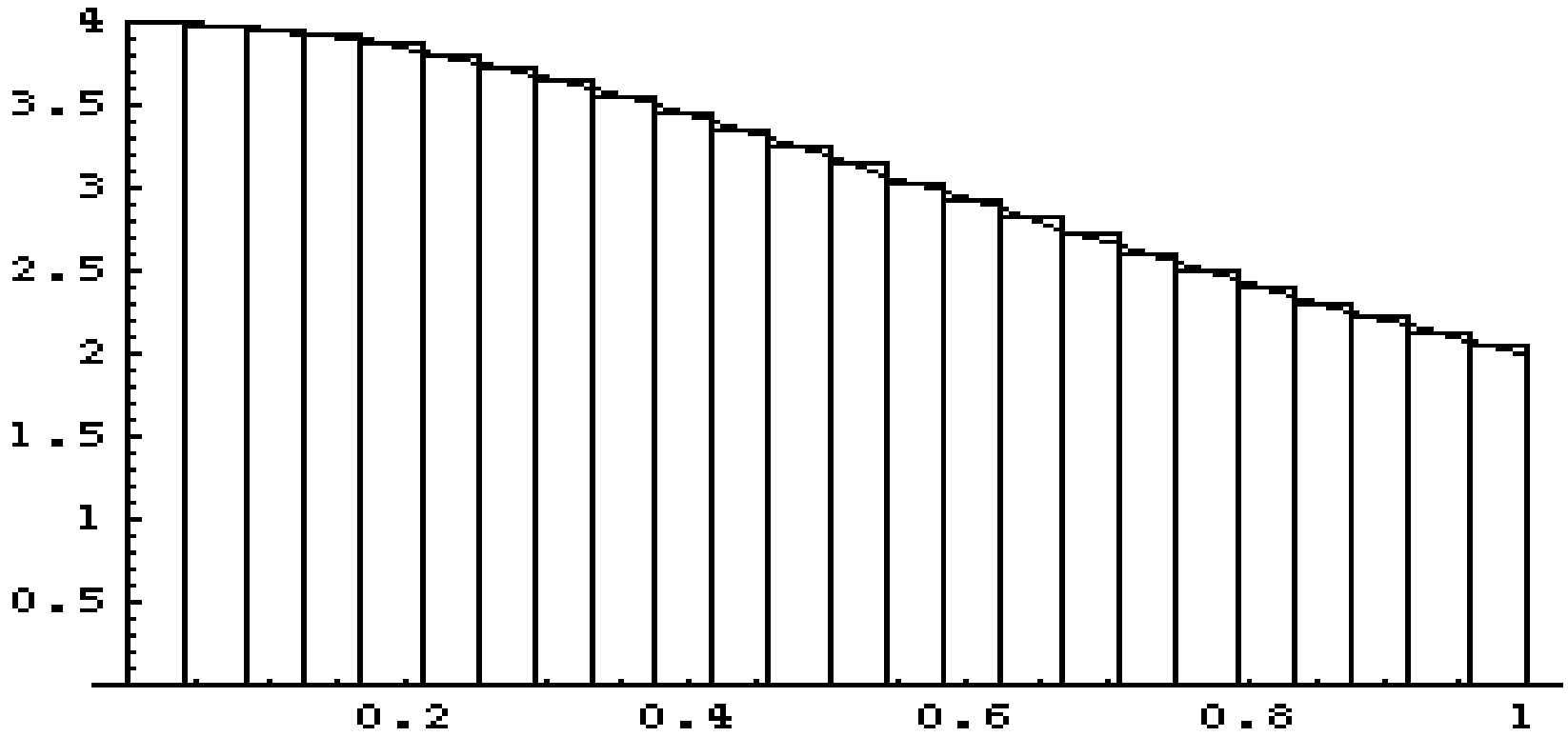
# See the Code

# Example: Compute PI

$$\pi = \int_0^1 \frac{4}{1+x^2} \, dx$$

# Example: Compute PI

$$\pi = \int_0^1 \frac{4}{1+x^2}\, dx$$



# How to write Program?

- Divide the range in to N interval/piece
  - Piece of size h = Range/N;
- Calculate area under each piece
  - Calculate the function value at piece X and multiply with piece size
  - h * F(X)
- Sum all the piece
  - $\sum_{i=1}^{n} h*F(X_i)$ with $X_i = R_{min}+i*h$

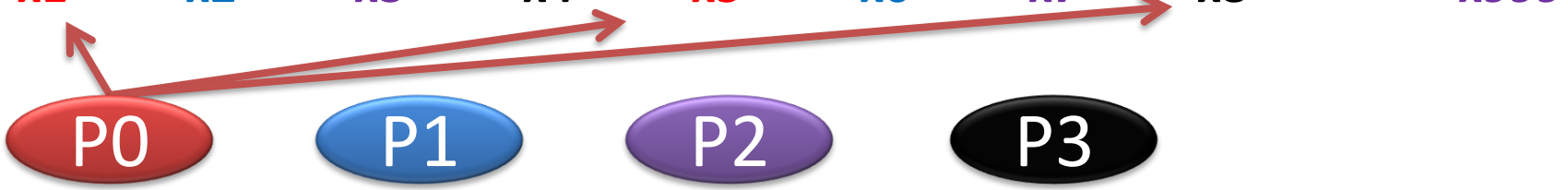
# How to write Program?

```
printf("Enter Num intervals: ");
scanf("%d", &n);
h = 1.0 / (double)n;
sum = 0.0;
for (i=1; i<n; i++) {
  x = h*(i-0.5); Fx=4.0/(1.0+ x*x);
  sum = sum + Fx;
}
pi = h*sum;
printf("pi is approx %.16f", pi);
```

# How to write Parallel Program?

- Divide the range in to N interval/piece
  - Piece of size h = Range/N;
  - **Suppose N =1000, NumProcessor =4**
- In Parallel: Calculate area under each piece

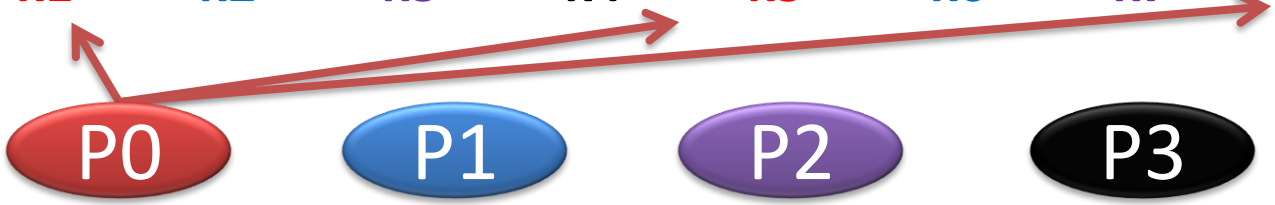
$$hF_{x1}+hF_{x2}+hF_{x3}+hF_{x4}+hF_{x5}+hF_{x6}+hF_{x7}+hF_{x8}+..+hF_{x999}$$



# How to write Parallel Program?

- Divide the range in to N interval/piece
  - Piece of size h = Range/N;
  - **Suppose N =1000, NumProcessor =4**
- In Parallel: Calculate area under each piece

$$hF_{x1}+hF_{x2}+hF_{x3}+hF_{x4}+hF_{x5}+hF_{x6}+hF_{x7}+hF_{x8}+..+hF_{x999}$$




- $(hF_{x1}+hF_{x5+}+..+hF_{x997}) + (hF_{x2}+hF_{x6}+..+hF_{x998})+ (hF_{x3}+hF_{x7}+..+ hF_{x999})+ (hF_{x4}+hF_{x8}+..+hF_{x996})$

# Example: Compute PI

```
#include <mpi.h>
#include <math.h>
int main(int argc, char *argv[]){
  int n, myid, Nproc, i;
  double lsum, pi, h, sum, x, a;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &Nproc);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid == 0) {
   printf("Enter Num intervals: \n");
   scanf("%d", &n);
  }
  MPI_Bcast(&n, 1, MPI_INT,0,
                MPI_COMM_WORLD);
```

# Example: Compute PI

```
h = 1.0 / (double)n; sum = 0.0;
for (i=myid+1; i<=n; i+= Nproc) {
    x = h*((double)i - 0.5);
    sum += 4.0 / (1.0 + x * x);
}
lsum = h*sum;
MPI_Reduce(&lsum, &pi, 1, MPI_DOUBLE,
      MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0)
   printf("pi is approx %.16f\n", pi);
MPI_Finalize();
return 0;
}
```

# IITG HPC clusters: Spec

- 4 login nodes
- 126 compute node
- 16 GPU compute nodes
- 16 Phi compute nodes
- Total 126+16+16= 158 nodes
  - Each node 12 cores * 2 threaded
  - Effective 24*158 =3792 cores

# Running MPI program on IITG HPC clusters

- Logic to one login nodes : non GPU/PHI
  - param.-ishan.iitg.ernet.in (172.17.0.7)
- Compile MPI-code

# Running MPI program on IITG HPC clusters

- Logic to one login nodes : non GPU/PHI
  - param.-ishan.iitg.ernet.in (172.17.0.7)
- Compile MPI-code
- Run using srun or sbatch
  - In s batch specify number of node, task per node
  - Total process
- SLURM : Simple Linux Util for Resce Mngt
  - Scheduler the JOB efficiently, user need not to worry where it is scheduling

# Resources

- https://computing.llnl.gov/tutorials/mpi/
- V. Kumar, A. Grama, A. Gupta, and G. Karypis. ***Introduction to Parallel Computing****: Design and Analysis of Algorithms*. Benjamin-Cummings Publ. Co, 1994 **[metis software]**
- Michael J. Quinn. ***Parallel Programming in C with MPI and OpenMP*****.** McGraw-Hill Education Group. 2003.
- Joseph JáJá. ***An Introduction to Parallel Algorithms***. Addison Wesley Longman Publishing Co., Inc.,, USA. 1992