PARALLEL PROGRAMMING...

Parallel Programming: Overview

SESSION 2/6

GOAL

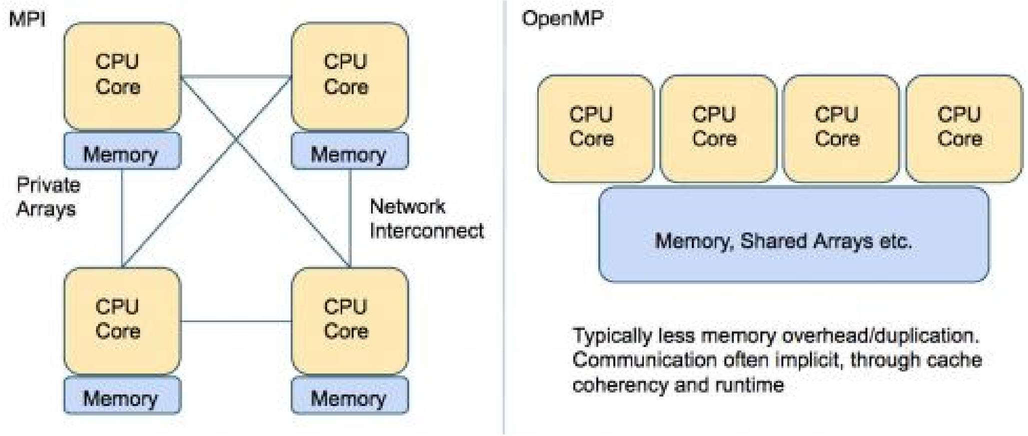
Programming Interface for parallel computing

MPI (Message Passing Interface)

Programming interface…

Remember

.



.

MPI (Message Passing Interface) is a multi-process model whose mode of communication between the processes is **explicit.**

==> communication management is the responsibility of the user.

.

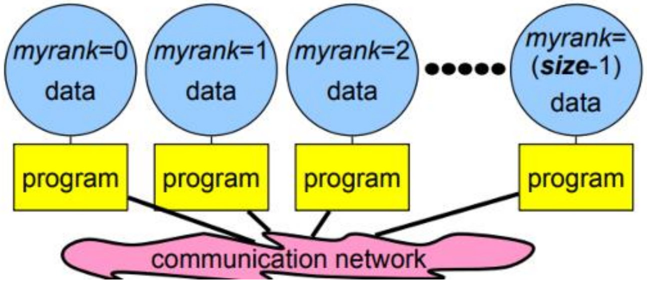
OpenMP (Open Multi-Processing) is a multitasking model whose mode of communication between tasks is **implicit**

==> communications is the responsibility of the compiler.

MPI (Message Passing Interface)

MPI (Message Passing Interface)

- MPI is a library of subroutines (in Fortran,C, and C++)
- Allows the coordination of a program running as multiple processes in a distributed-memory environment.
- Flexible enough to also be used in a shared-memory environment.
- Can be used and compiled on a wide variety of single platforms or (homogeneous or heterogeneous) clusters
- of computers over a network.
- The scalability of MPI **is not limited** by the number of processors/cores on one computation node,
- as opposed to shared memory parallel models.
- MPI library is standardized



MPI: Basic Environment

MPI programs **start** with a function call, which initializes **the message passing library.**

```
MPI_Init(&argc, &argv)
```

Initializes MPI environment
Must be called in every MPI program
Must be first MPI call
Can be used to pass command line arguments to all

```
MPI_Finalize()
```

Terminates MPI environment
Last MPI function call

MPI: Basic Environment

```
MPI_Comm_rank(comm, &rank)
```

Returns the rank of the calling MPI process
Within the communicator, comm
MPI_COMM_WORLD is set during Init(...)
Other communicators can be created if needed

```
MPI_Comm_size(comm, &size)
```

Returns the total number of processes
Within the communicator, comm

MPI : Communicators

- A communicator is an identifier associated with a group of processes
  - Each process has a unique rank within a specific communicator
    (the rank starts from 0 and has a maximum value of (nprocesses-1)).
  - Internal mapping of processes to processing units
  - Always required when initiating a communication by calling an MPI function or routine.

- Default communicator MPI_COMM_WORLD, which contains all available processes.

- Several communicators can coexist

  - A process can belong to different communicators at the same time,
    but has a unique rank in each communicator

MPI : Basic calls to exchange data

- Point-to-Point communications
  - Only 2 processes exchange data
  - It is the basic operation of all MPI calls

- Collective communications
  - A single call handles the communication between all the processes in a communicator
  - There are 3 types of collective communications
    - Data movement (e.g. **MPI_Bcast**)
    - Reduction (e.g. **MPI_Reduce**)
    - Synchronization: **MPI_Barrier**

MPI: Point-to-Point Communication

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

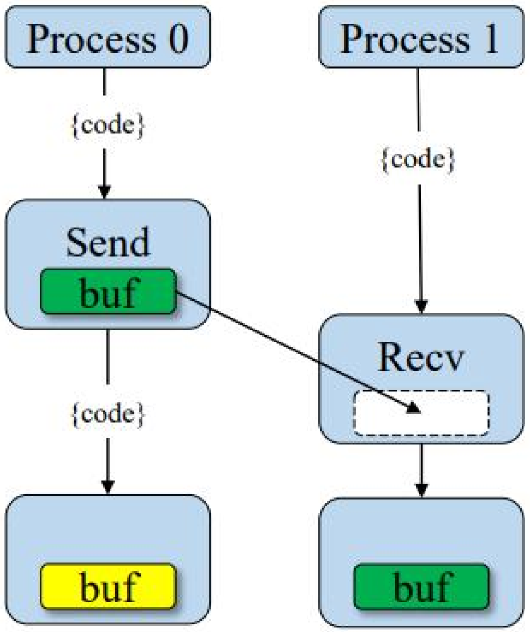
Send a message
Returns only after buffer is free for reuse
Blocking

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

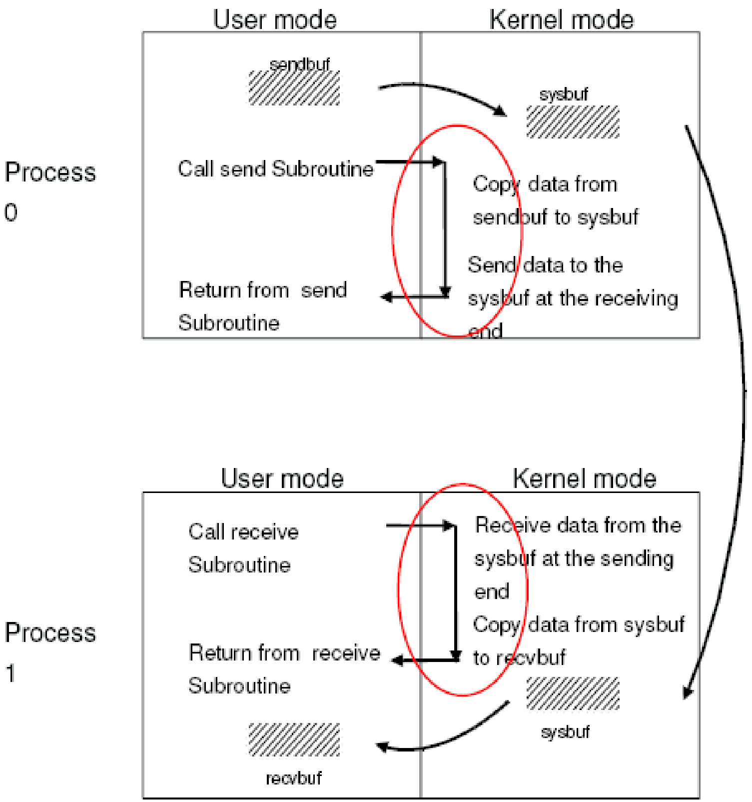
Received a message
Returns only when the data is avaible
Blocking

```
MPI_SendRecv(...)
```

Two way communication
Blocking



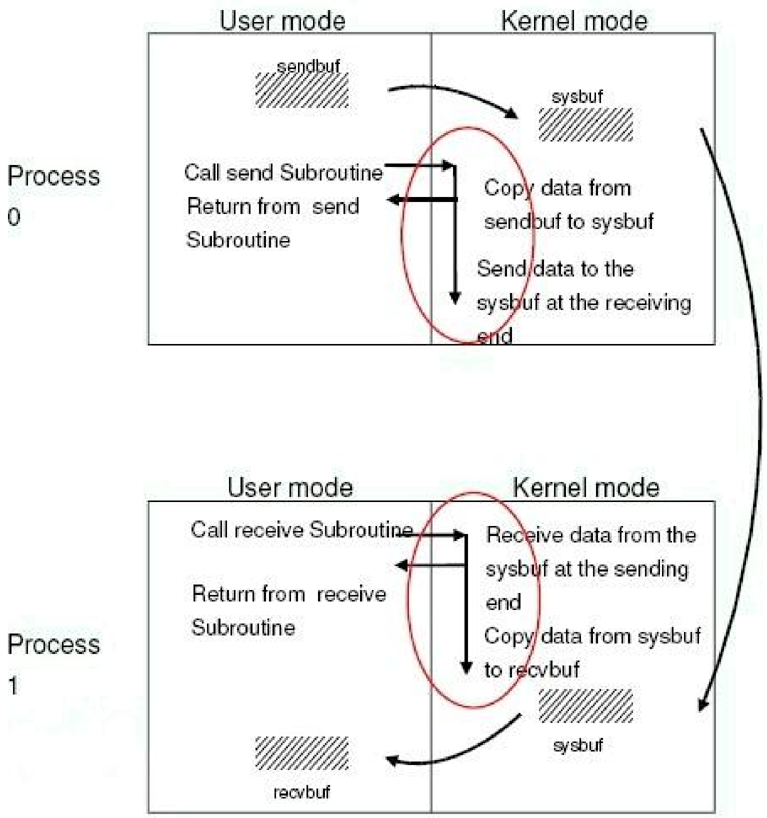
MPI: Blocking communications



- The call waits until the data transfer is done.
  - The sending process waits until all data are transferred to the system buffer.
  - The receiving process waits until all data are transferred from the system buffer to the receive buffer.

- All collective communications are blocking

MPI: Non-blocking communications



- Returns immediately after the data transferred is initiated
- Allows to overlap computation with communication
- Need to be careful though
  - When send and receive buffers are updated before the transfer is over, the result will be wrong

MPI: Non-blocking send and received

Point to point communication

MPI_Isend (buf,count,datatype,dest,tag,comm,request,ierr)

MPI_Irecv (buf,count,datatype,source,tag,comm,request,ierr)

The functions MPI_Wait and MPI_Test are used to complete a nonblocking communication

MPI_Wait (request,status,ierr)

MPI_Test (request,flag,status,ierr)

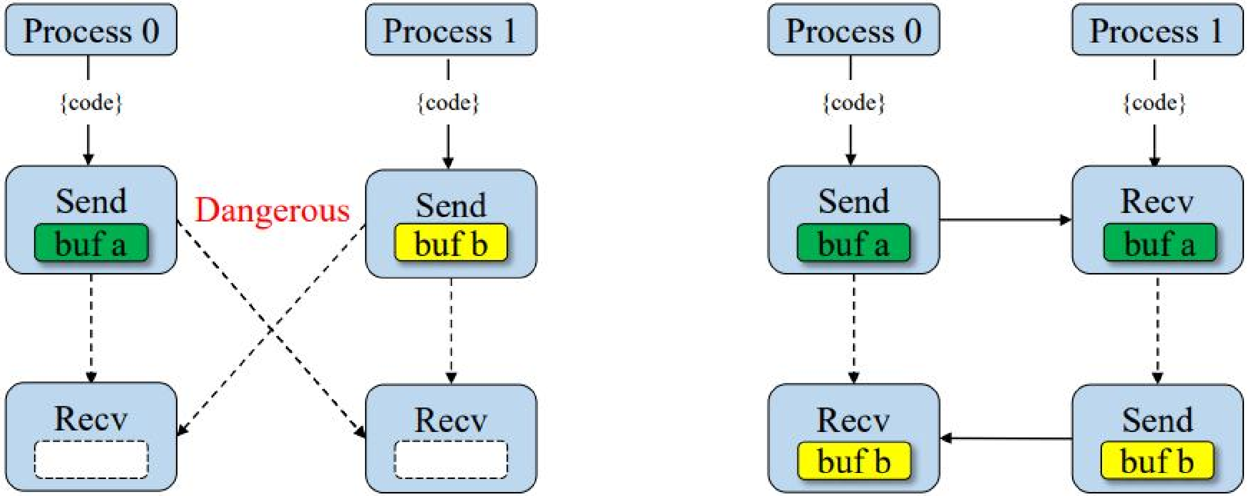
MPI_Wait returns when the operation identified by “request” is complete. This is a non-local operation.

MPI_Test returns “flag = true” if the operation identified by “request” is complete. Otherwise it returns “flag = false”. This is a local operation.

MPI: Deadlock

Blocking calls can **results in deadlock**

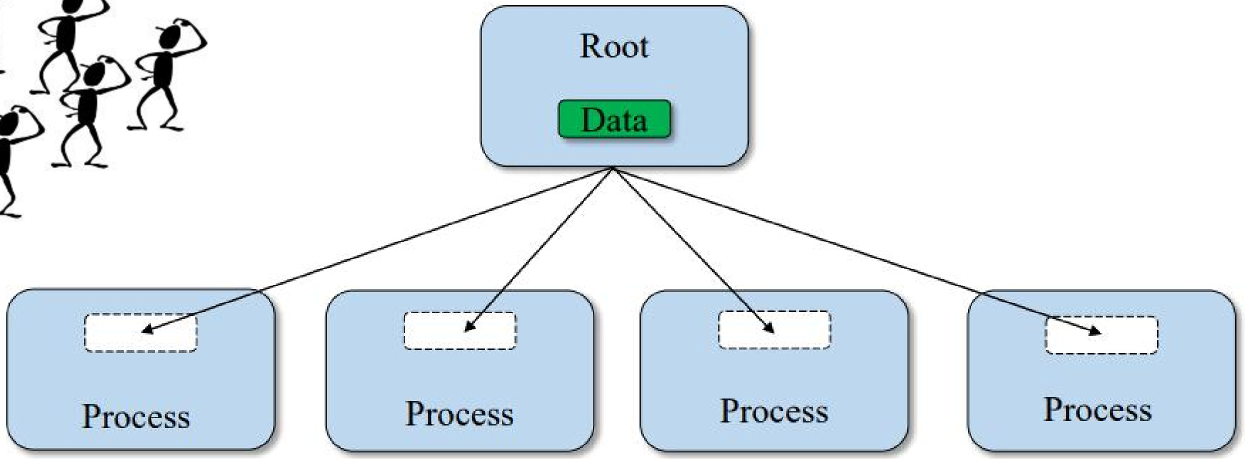
- One process is waiting for message that will never arrive
- Only option is to abort the interrupt/kill the code (CTRL-c) **:-(**
- Might not always deadlock - depends on size of system buffer



MPI: Collective Communication (BroadCast)

```
MPI_Bcast(&buffer, count, datatype, root, comm)
```

One process (called “root”) sends data to all the other processes in the same communicator
Must be called by **ALL processes** with the same arguments
Useful when reading in input parameters from file.

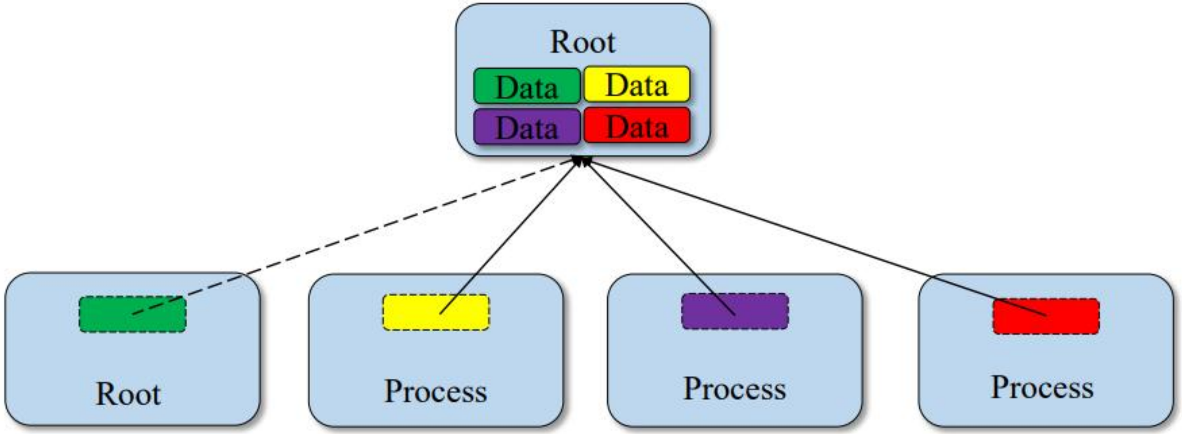


MPI: Collective Communication (Gather)

```
MPI_Gather(&sendbuf, sendcnt, sendtype, &recvbuf,
           recvcnt, recvtype, root, comm)
```

One root process collects data from all the other processes in the same communicator
Must be called by all the processes in the communicator with the same arguments

Make sure that you have enough space in your receiving buffer!

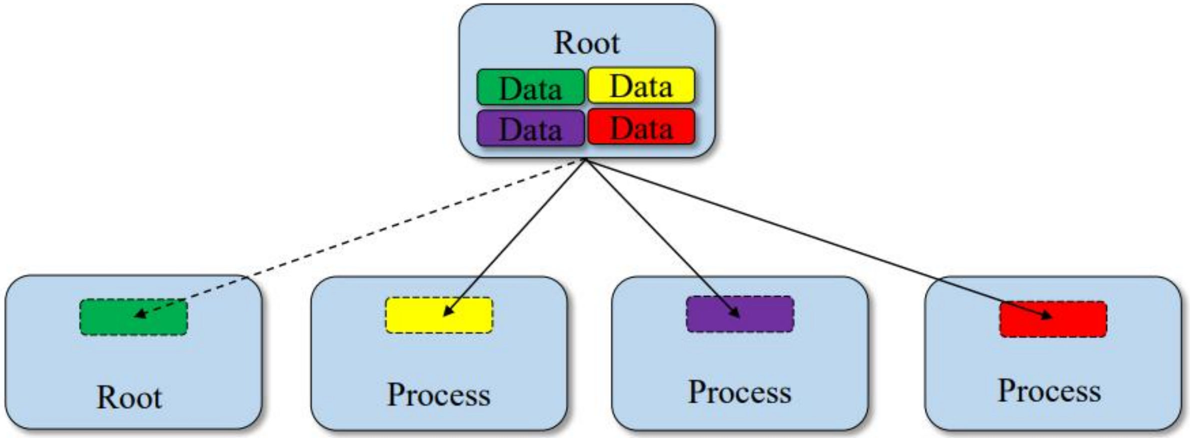


Opposite of **Scatter.**

MPI: Collective Communication (Scatter)

```
MPI_Scatter(&sendbuf, sendcnt, sendtype, &recvbuf,
            recvcnt, recvtype, root, comm)
```

One “root” process send a different piece of the data to each one of the other Processes (inverse of gather)

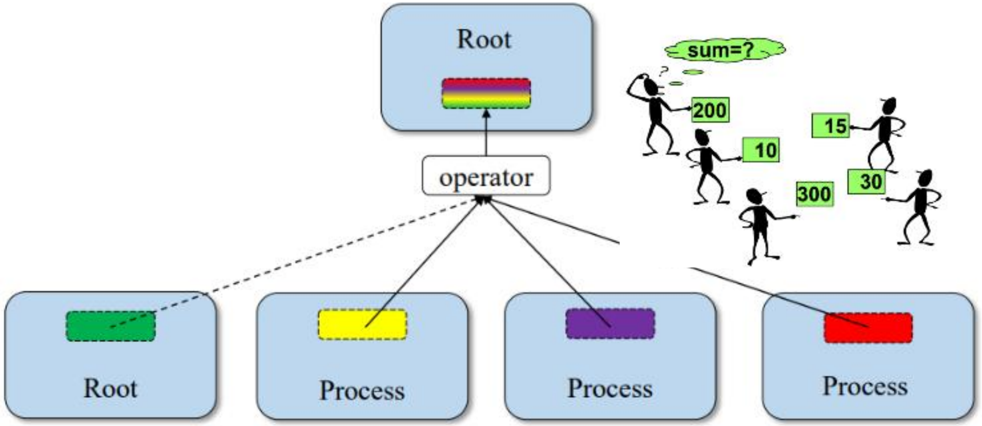


MPI: Collective Communication (Reduce)

```
MPI_Reduce(&sendbuf, &recvbuf, count, datatype,
           mpi_operation, root, comm)
```

One root process collects data from all the other processes in the same communicator
and performs an operation on the received data.

Operations are: *MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more*
User can define own operation with MPI_Op_create()



Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD

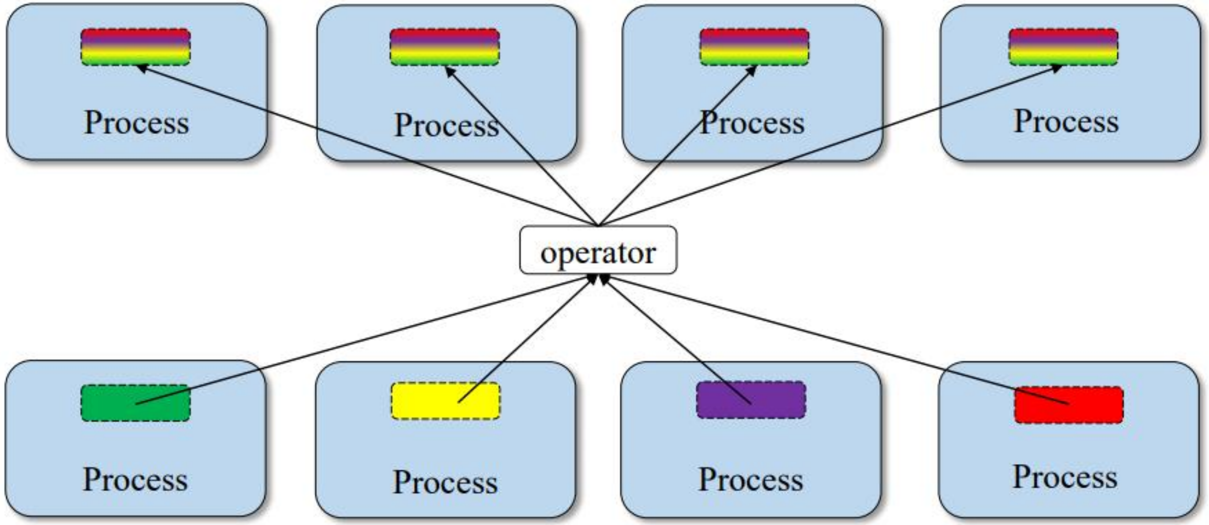
MPI: Collective Communication (Allreduce)

```
MPI_Allreduce(&sendbuf, &recvbuf, count,
              datatype, mpi_operation, comm)
```

Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD

Applies **reduction** operation on data **from all processes**.

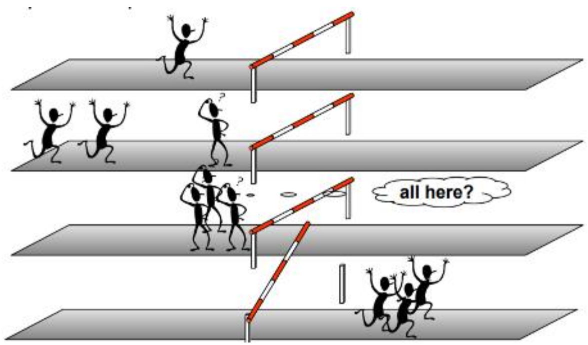
Store results on all processes.



MPI: Collective Communication (Barrier)

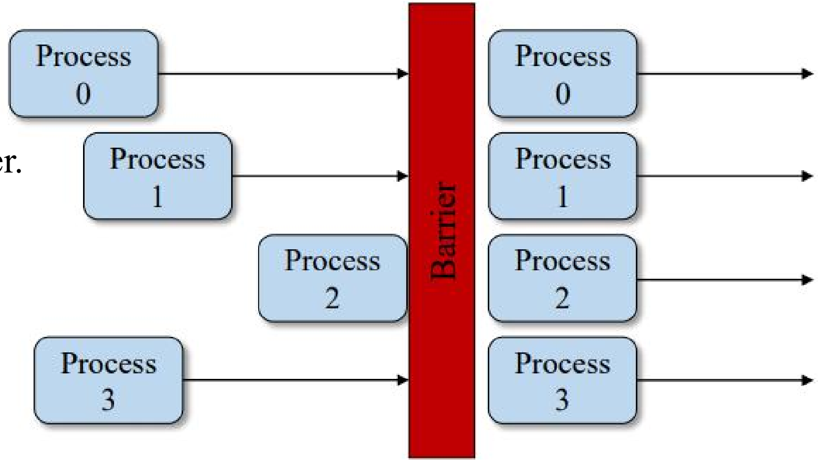
```
MPI_Barrier(comm)
```

When necessary, all the processes within a communicator can be forced to wait for each other although this operation can be expensive



Process synchronization (blocking).
All processes forced to wait for each other.

Use only where necessary.
Will reduce parallelism.



MPI: keywords

1 environment
- MPI Init: Initialization of the MPI environment
- MPI Comm rank: Rank of the process
- MPI Comm size: Number of processes
- MPI Finalize: Deactivation of the MPI environment
- MPI Abort: Stopping of an MPI program
- MPI Wtime: Time taking

2 Point-to-point communications
- MPI Send: Send message
- MPI Isend: Non-blocking message sending
- MPI Recv: Message received
- MPI Irecv: Non-blocking message reception
- MPI Sendrecv and MPI Sendrecv replace: Sending and receiving messages
- MPI Wait: Waiting for the end of a non-blocking communication
- MPI Wait all: Wait for the end of all non-blocking communications

3 Collective communications
- MPI Bcast: General broadcast
- MPI Scatter: Selective spread
- MPI Gather and MPI Allgather: Collecting
- MPI Alltoall: Collection and distribution
- MPI Reduce and MPI Allreduce: Reduction
- MPI Barrier: Global synchronization

4 Derived Types
- MPI Contiguous type: Contiguous types
- MPI Type vector and MPI Type create hvector: Types with a con-standing
- MPI Type indexed: Variable pitch types
- MPI Type create subarray: Sub-array types
- MPI Type create struct: H and erogenous types
- MPI Type commit: Type commit
- MPI Type get extent: Recover the extent
- MPI Type create resized: Change of scope
- MPI Type size: Size of a type
- MPI Type free: Release of a type

MPI: Keywords

5 Communicator
• MPI Comm split: Partitioning of a communicator
• MPI Dims create: Distribution of processes
• MPI Cart create: Creation of a Cart´esian topology
• MPI Cart rank: Rank of a process in the Cart´esian topology
• MPI Cart coordinates: Coordinates of a process in the Cart´esian topology
• MPI Cart shift: Rank of the neighbors in the Cart´esian topology
• MPI Comm free: Release of a communicator

6 MPI-IO
• MPI File open: Opening a file
• MPI File set view: Changing the view
• MPI File close: Closing a file

6.1 Explicit addresses
• MPI File read at: Reading
• MPI File read at all: Collective reading
• MPI File write at: Writing

6.2 Individual pointers
• MPI File read: Reading
• MPI File read all: collective reading
• MPI File write: Writing
• MPI File write all: collective writing
• MPI File seek: Pointer positioning

6.3 Shared pointers
• MPI File read shared: Read
• MPI File read ordered: Collective reading
• MPI File seek shared: Pointer positioning

7.0 Symbolic constants
• MPI COMM WORLD, MPI SUCCESS
• MPI STATUS IGNORE, MPI PROC NULL
• MPI INTEGER, MPI REAL, MPI DOUBLE PRECISION
• MPI ORDER FORTRAN, MPI ORDER C
• MPI MODE CREATE,MPI MODE RONLY,MPI MODE WRONLY

MPI: Program Basics

Include MPI Header File

Start of Program

(Non-interacting Code)

Initialize MPI

Run Parallel Code &
Pass Messages

End MPI Environment

(Non-interacting Code)

End of Program

```
#include <mpi.h>

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

.
.     // Run parallel code
.

MPI_Finalize(); // End MPI Envir

return 0;
}
```

MPI: Example

```c
#include <mpi.h>
#include <stdio.h>


int main (int argc, char *argv[]) {

  int rank, size;

  MPI_Init (&argc, &argv);  //initialize MPI library

  MPI_Comm_size(MPI_COMM_WORLD, &size); //get number of processes
  MPI_Comm_rank(MPI_COMM_WORLD, &rank); //get my process id

  //do something
  printf ("Hello World from rank %d\n", rank);
  if (rank == 0) printf("MPI World size = %d processes\n", size);

  MPI_Finalize(); //MPI cleanup

  return 0;
}
```

- 4 processes

```
Hello World from rank 3
Hello World from rank 0
MPI World size = 4 processes
Hello World from rank 2
Hello World from rank 1
```

- Code ran on each process independently
- MPI Processes have *private* variables
- Processes can be on completely different machines

MPI: Example Point-to-Point communication

```
#include<iostream>
#include<mpi.h>
using namespace std;
int main (int argc, char *argv[])
{
        int numprocs,myid;
        MPI_Init(&argc,&argv);

        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);

        MPI_Status status;
        int small=myid;
        cout<<"Before " <<myid<<" of "<<,numprocs<<" small = "<<small,<<endl;

        If (myid==0) { MPI_Send(&small,1,MPI_INT,3,10,MPI_COMM_WORLD); }

        If (myid==3) { MPI_Recv(&small,1,MPI_INT,0,10,MPI_COMM_WORLD,&status) }

        MPI_Barrier( MPI_COMM_WORLD);

        cout<<"After " <<myid<<" of "<<numprocs<<" small = "<<small<<endl;

        MPI_Finalize();
        }
```

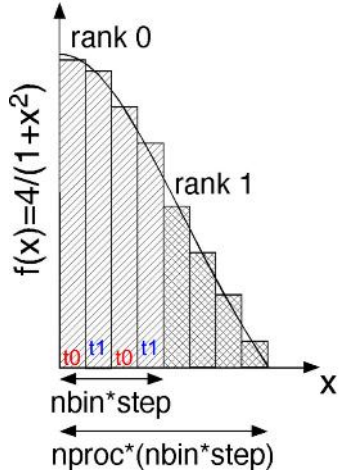
MPI: Example Reduction

```
...
#include<mpi.h>
using namespace std;
double f( double a ) {return (4.0 / (1.0 + a*a));}
int main (int argc, char *argv[])
{
        int myid, numprocs;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        int n = 1000000000;
        double pi,sum=0.0;
        double startwtime = 0.0;
        if (myid == 0) { startwtime = MPI_Wtime(); }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        for (int i = myid + 1; i <= n; i += numprocs) { sum += f((i-0.5)/(double) n); }
        sum/= (double) n;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        if (myid == 0)
        {
          cout<<"pi is approximately equal "<<setprecision(16) << pi <<" Error is"<<fabs(pi - M_PI)<<endl;
          cout<<"Wall clock time = "<<MPI_Wtime()-startwtime<<endl;
        }
        MPI_Finalize();
        Exit(0);
        }
```

GOAL : The following code computes the π number by using a numerical evaluation of an integral by a rectangle method.

Each virtual core computes a part of the loop and a reduction instruction is performed

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx \cong \Delta \sum_{i=0}^{N-1} \frac{4}{1+x_i^2}$$



MPI: Example Broadcast

Broadcast a message from the root process to all other processes.
Useful when reading in input parameters from file.

```
#include<iostream>
#include<mpi.h>
using namespace std;
int main (int argc, char *argv[])
{
        int numprocs,myid,namelen;
        char processor_name[MPI_MAX_PROCESSOR_NAME];
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        MPI_Get_processor_name(processor_name,&namelen);

        double reel=(double) myid;
        cout<<"Before " <<myid<<" of "<<numprocs<<" on "<<processor_name<<" integervalue "<<reel<<endl;

        MPI_Bcast(&reel,1, MPI_DOUBLE,3,MPI_COMM_WORLD);
        MPI_Barrier( MPI_COMM_WORLD);

        cout<<"After " <<myid<<" of "<<numprocs<<" on "<<processor_name<<" integervalue "<<reel<<endl;

        MPI_Finalize();
        exit(0);
        }
```

COMPILING an MPI Program

- **Compiling a program** for MPI is almost just like compiling a regular C or C++ program
  - The C compiler is **mpicc** and the **C++** compiler is **mpic++**.
    - For example, to compile **MyProg.c** you would use a command like
      - **mpicc** - O2 -o **MyProg MyProg . c**

Thank you for your attention !