PARALLEL PROGRAMMING...

Parallel Programming: Overview

SESSION 2/6

GOAL

Programming Interface for parallel computing

MPI (Message Passing Interface)

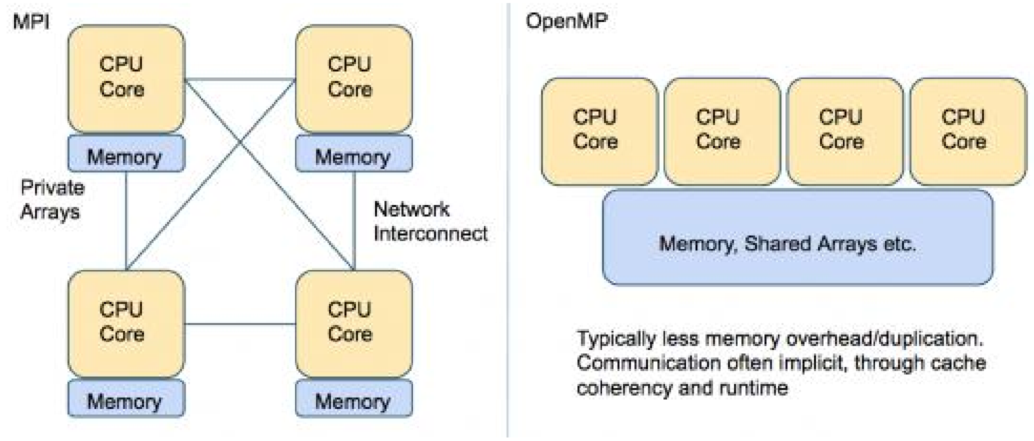
Programming interface for parallel computing

병렬 컴퓨팅을 위한 프로그래밍 인터페이스

Programming interface...

Remember

.


MPI
CPU Core
CPU Core
Memory
Memory
Private Arrays
Network Interconnect
CPU Core
CPU Core
Memory
Memory
OpenMP
CPU Core
CPU Core
CPU Core
CPU Core
Memory, Shared Arrays etc.
Typically less memory overhead/duplication. Communication often implicit, through cache coherency and runtime

.

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, 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, should work (without further changes!)
on any machine on which the MPI library is installed.

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

```
int my_rank, size;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

MPI: Point-to-Point Communication

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

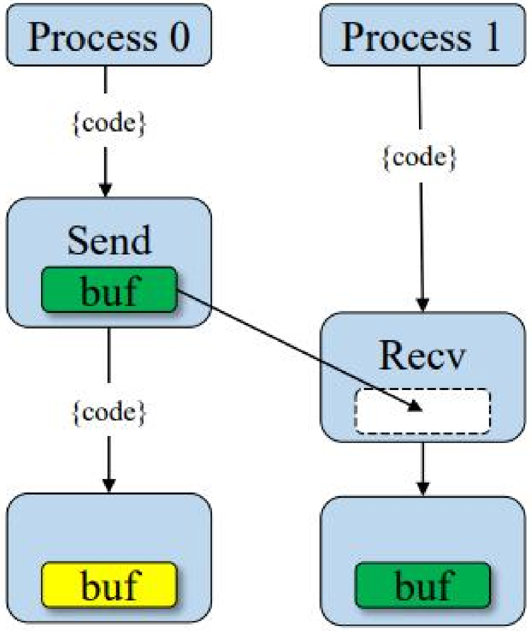
Send a message
Returns only after buffer is free for reuse
Blocking

```
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: Point-to-Point Communication

Blocking

- Only returns after completed
  - Received: data has arrived and ready to use
  - Send: safe to reuse sent buffer
- Be aware of deadlocks !!!
- Tip: use when possible

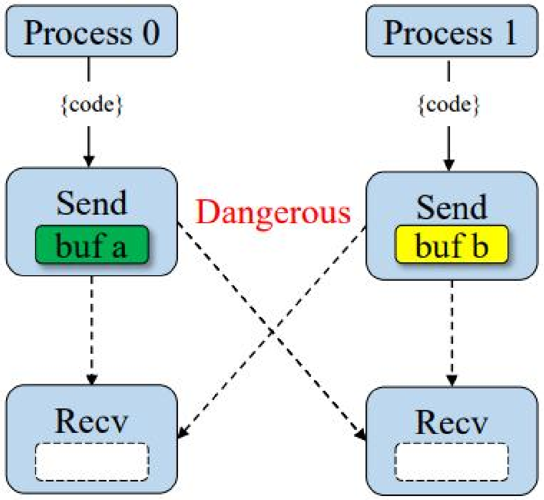
Non-Blocking

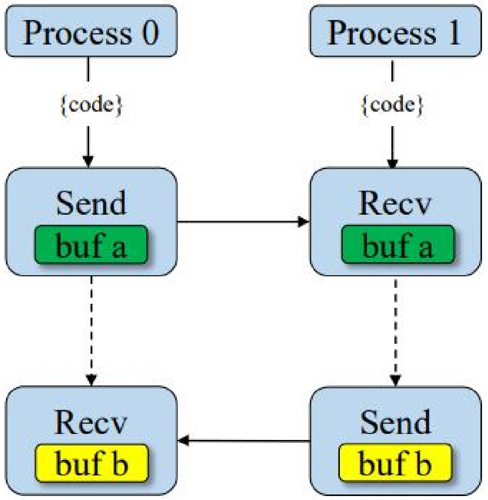
- returns immediately
  - Unsafe to modify buffers until operation is known to be complete
- Allows computation and communication to overlap
- Tip: Use only when needed

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

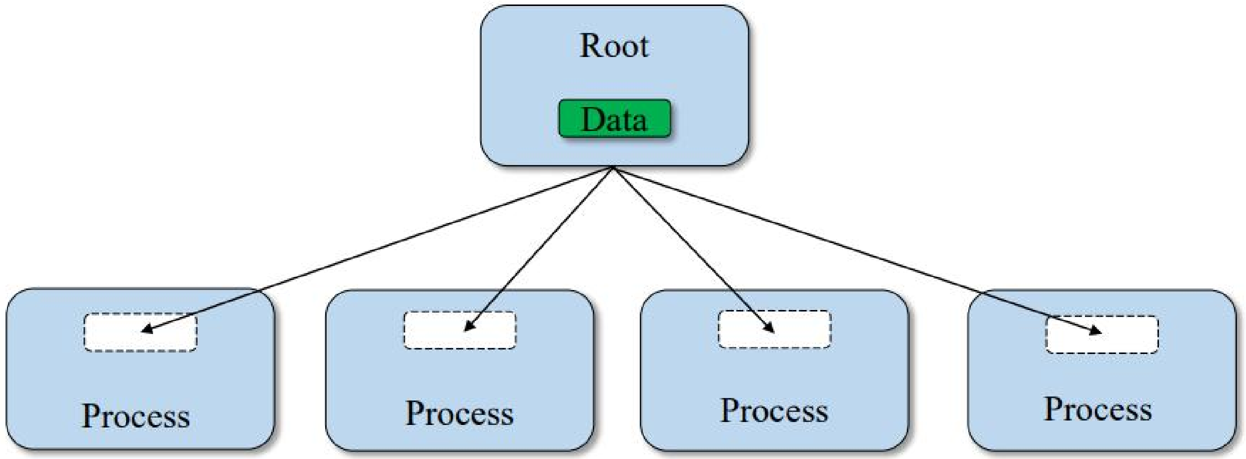




MPI: Coolective Communication (BCast)

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

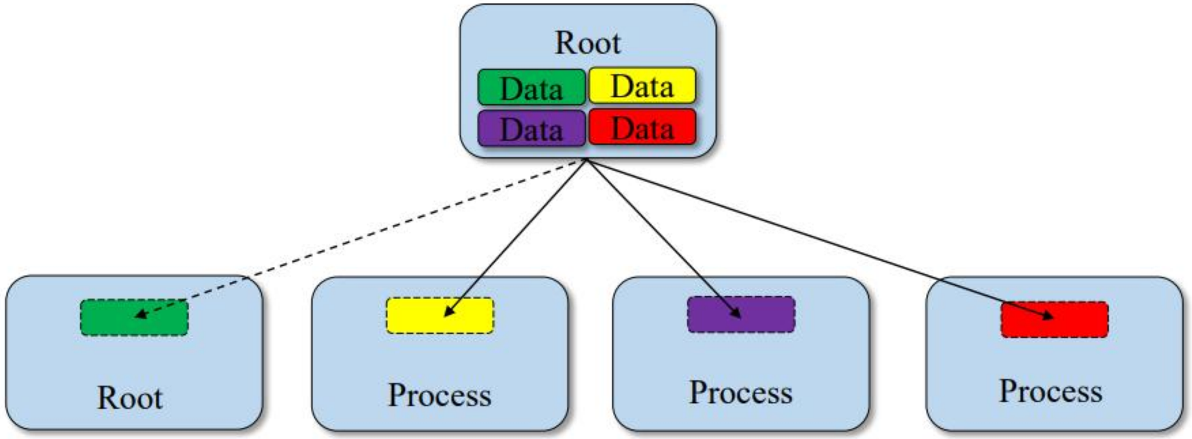
Broadcast a message from the root process to all other processes.
Useful when reading in input parameters from file.



MPI: Collective Communication (Scatter)

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

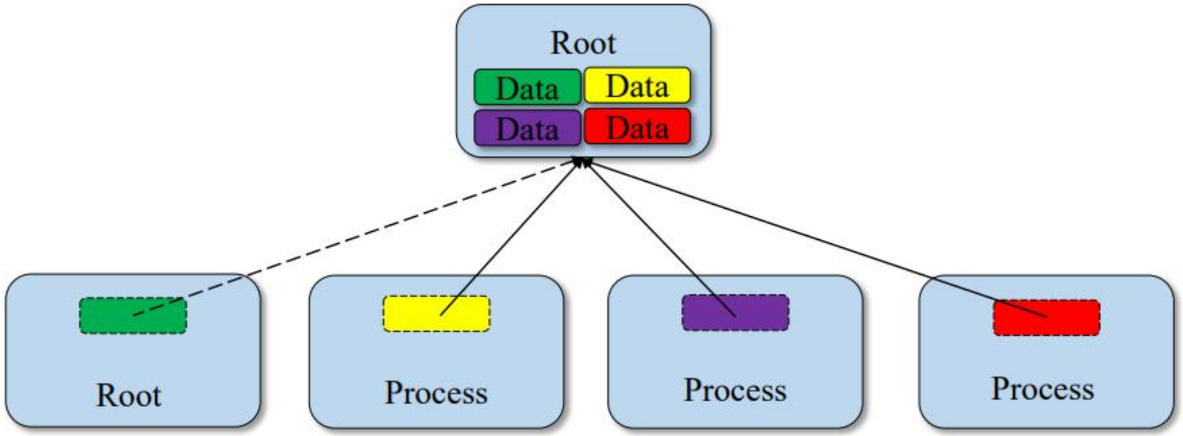
Sends individual messages from the root process to all other processes.



MPI: Collective Communication (Gather)

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

Opposite of **Scatter.**



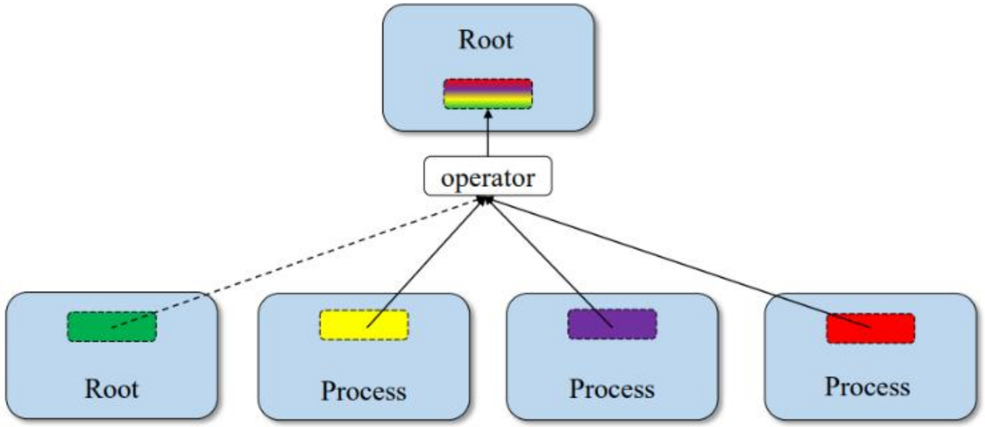
MPI: Collective Communication (Reduce)

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

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

Puts results on root process.

Operator
MPI_SUM
MPI_MAX
MPI_MIN
MPI_PROD



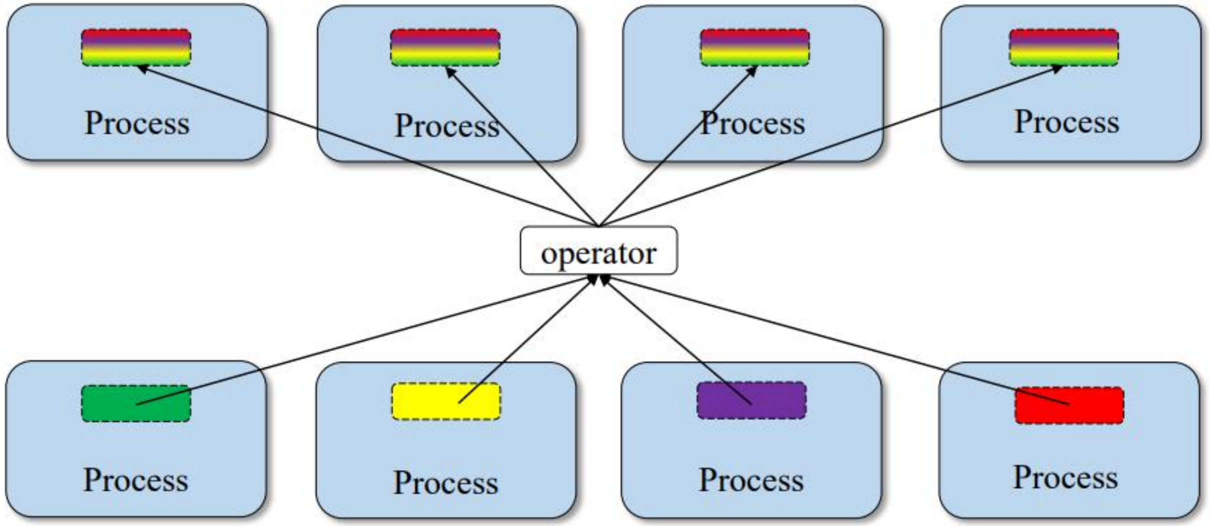
MPI: Collective Communication (Allreduce)

```
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)

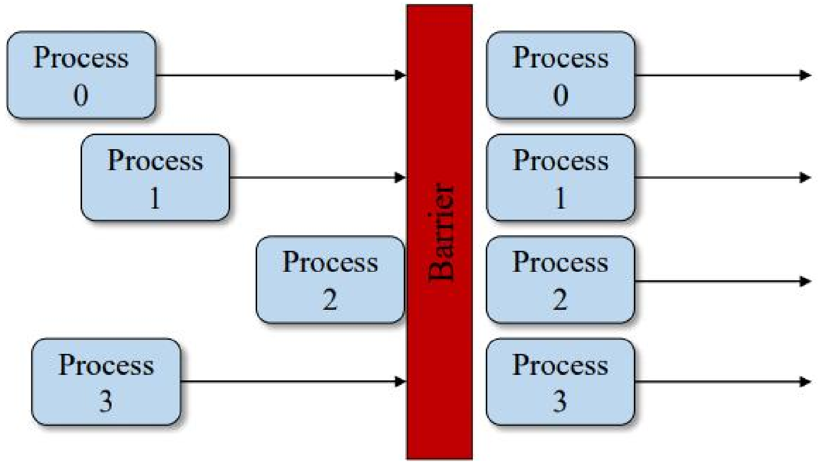
```
MPI_Barrier(comm)
```

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

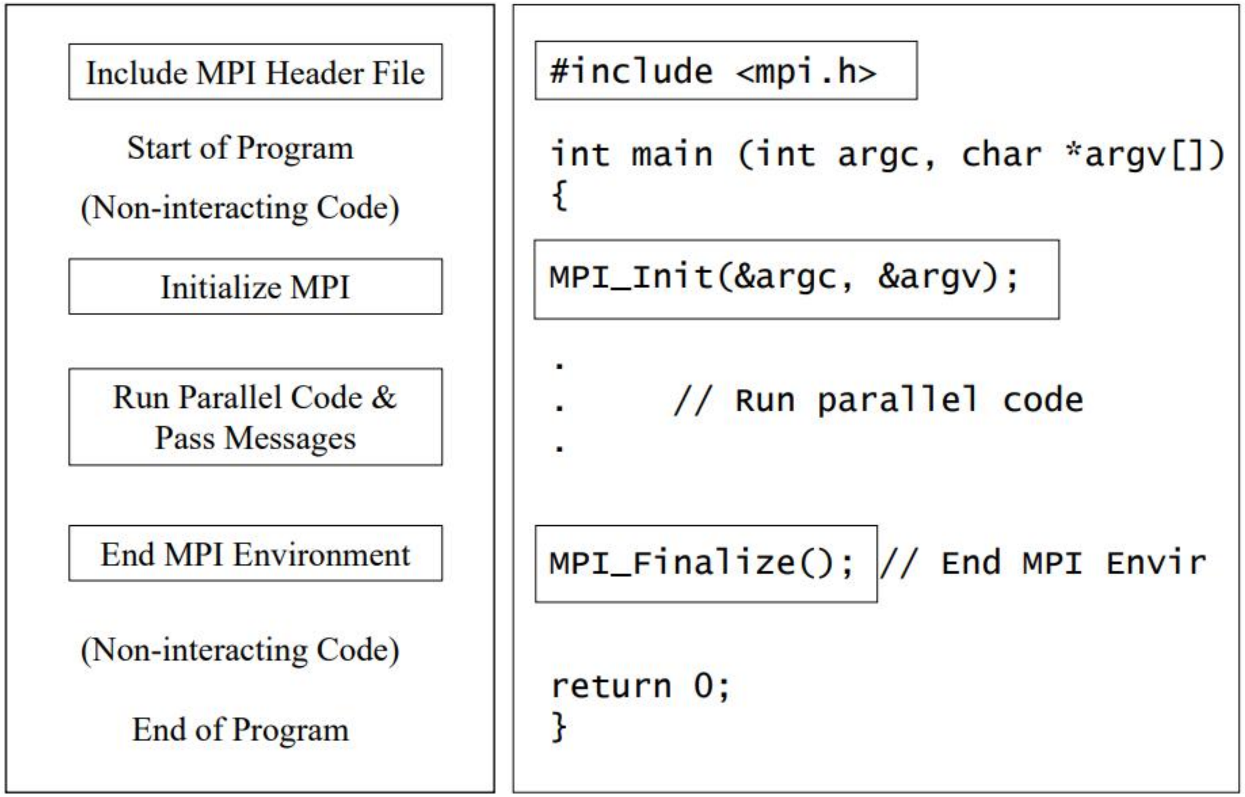
6.1 Explicit addresses
• MPI File read at: Reading
• MPI File read at all: Collective reading
• MPI File write at: Writing

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.h>

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

MPI_Init(&argc, &argv);

.
.     // Run parallel code
.


MPI_Finalize(); // End MPI Envir


return 0;
}
```

MPI: Example

```
#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 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);
        }
```

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

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

```
...
#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);
        }
```

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 !