# Introduction to MPI

Mesfin Diro Chaka
Computational Science
Addis Ababa University
mesfin.diro@aau.edu.et

Addis Ababa, Ethiopia

June 06, 2019

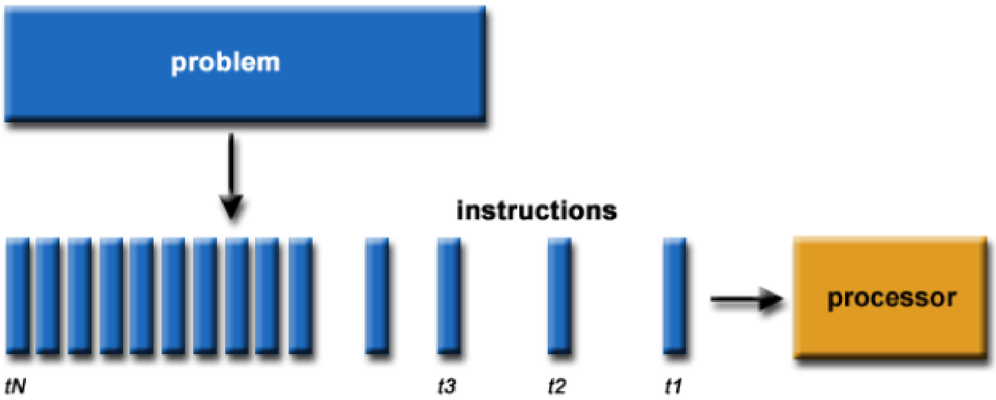
# What is Parallel Computing?

### Serial Computing:

- Traditionally, software has been written for serial computation:
  - A problem is broken into a discrete series of instructions
  - Instructions are executed sequentially one after another
  - Executed on a single processor
  - Only one instruction may execute at any moment in time

# What is Parallel Computing? . . .

## Serial Computing:



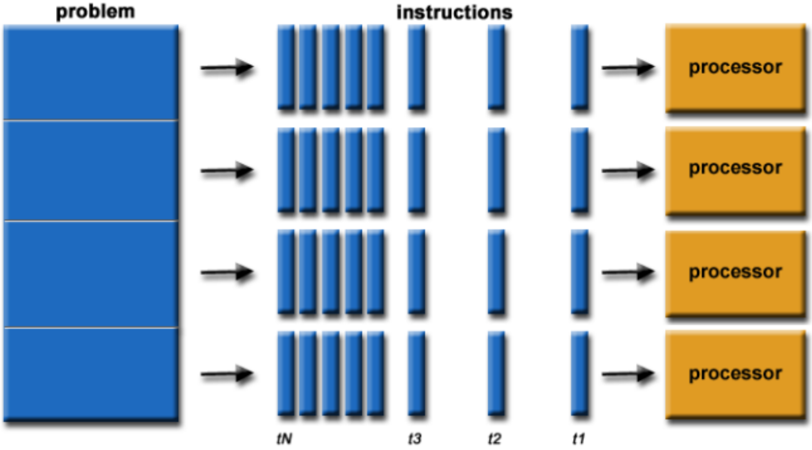
# What is Parallel Computing . . .

**Parallel Computing:**

- In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
  - Instructions from each part execute simultaneously on different processors
  - An overall control/coordination mechanism is employed

# What is Parallel Computing . . .

## Parallel Computing:



# What is Parallel Computing? . . .

- The computational problem should be able to:
  - Be broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Be solved in less time with multiple compute resources than with a single compute resource.
- The compute resources are typically:
  - A single computer with multiple processors/cores
  - An arbitrary number of such computers connected by a network

# Parallel Computer Memory Architectures

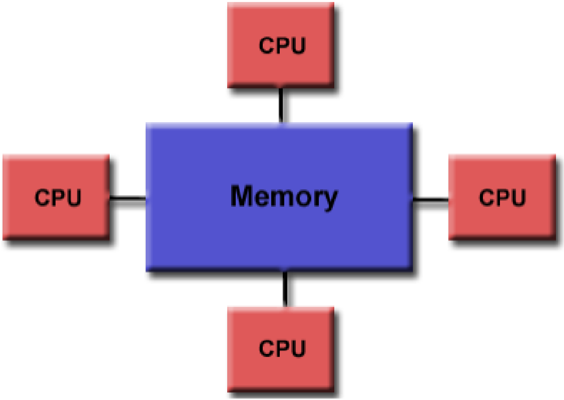
## Shared Memory

**General Characteristics:**

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Historically, shared memory machines have been classified as UMA and NUMA, based upon memory access times.

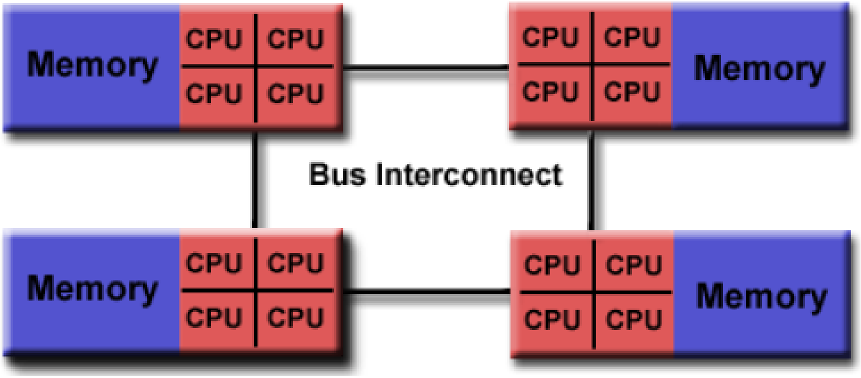
# Parallel Computer Memory Architectures . . .

## Uniform Memory Access (UMA) Shared Memory



# Parallel Computer Memory Architectures . . .

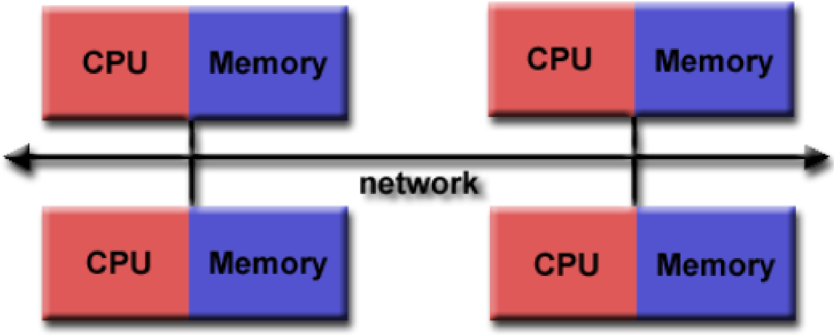
## Non-Uniform Memory Access (NUMA) Shared Memory



# Parallel Computer Memory Architectures

- **General Characteristics:**
  - Distributed memory systems require a communication network to connect inter-processor memory.
  - Processors have their own local memory
  - Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
  - The network “fabric” used for data transfer varies widely, though it can be as simple as Ethernet.

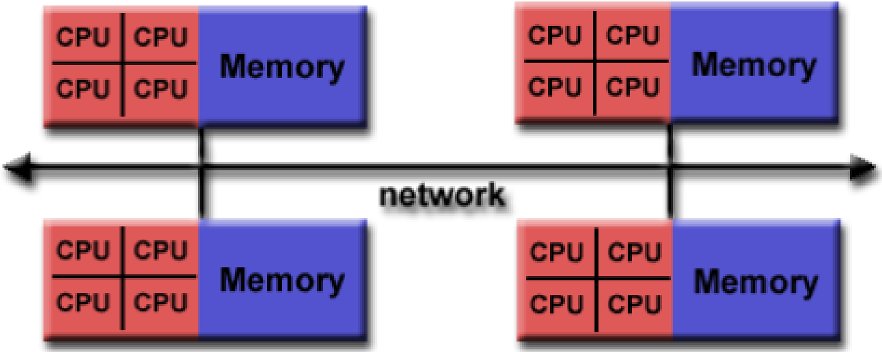
# Parallel Computer Memory Architectures . . .



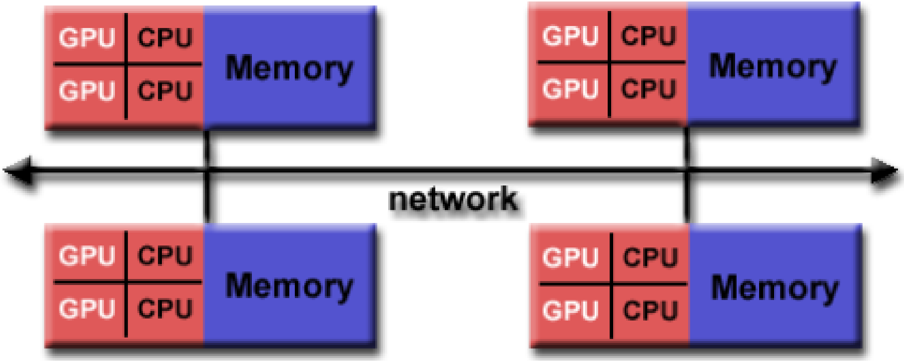
# Hybrid Distributed-Shared Memory

- General Characteristics:
  - The largest and fastest computers in the world today employ both shared and distributed memory architectures.
  - The shared memory component can be a shared memory machine and/or graphics processing units (GPU).
  - The distributed memory component is the networking of multiple shared memory/GPU machines
  - Network communications are required to move data from one machine to another.
  - Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.

# Hybrid Distributed-Shared Memory . . .



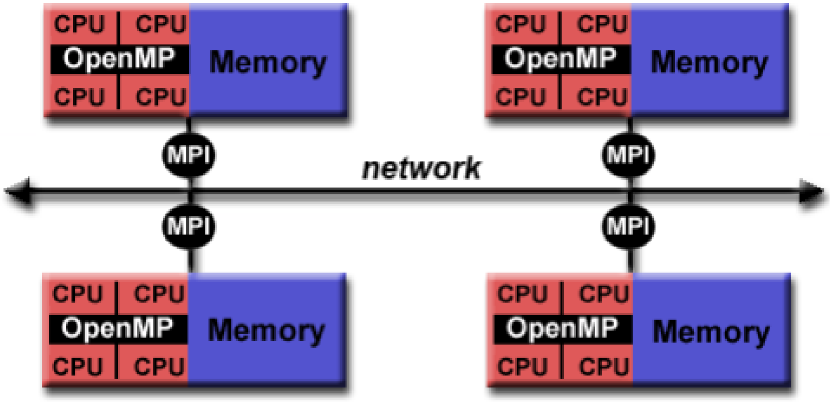
# Hybrid Distributed-Shared Memory . . .



# Hybrid Parallel Programming Models

- A hybrid model combines more than one of the previously described programming models.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP).
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI
- This hybrid model lends itself well to the most popular hardware environment of clustered multi/many-core machines.
- Another similar and increasingly popular example of a hybrid model is using MPI with CPU-GPU
  - MPI tasks run on CPUs using local memory and communicating with each other over a network.
  - Computationally intensive kernels are off-loaded to GPUs on-node.
  - Data exchange between node-local memory and GPUs uses CUDA

# Hybrid Parallel Programming Models . . .



# Message Passing Interface (MPI)

- An Interface Specification:
- MPI = Message Passing Interface
- MPI primarily addresses the message-passing parallel programming model
- Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be:
- The MPI standard has gone through a number of revisions, with the most recent version being MPI-4.x
- Programming Model:
  - Originally, MPI was designed for distributed memory architectures
  - Today, MPI runs on virtually any hardware platform:
    - Distributed Memory
    - Shared Memory
    - Hybrid

# Message Passing Interface (MPI)...

## Implementation

- MPI is a library of function/subroutine calls
- MPI is not a language
- There is no such thing as an MPI compiler
- The C or Fortran compiler you invoke knows nothing about what MPI actually does
  - only knows prototype/interface of the function/subroutine calls

# Message Passing Interface (MPI)...

### Reasons for Using MPI:

- Standardization - MPI is the only message passing library that can be considered a standard.
- Portability - There is little or no need to modify your source code when you port your application to a different platform that supports the MPI standard.
- Functionality - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
- Availability - A variety of implementations are available, both vendor and public domain.

# Message Passing Interface (MPI). . .

## MPI Implementations and Compilers

- Although the MPI programming interface has been standardized, actual library implementations will differ.
- MPI library implementations on LC systems vary, as do the compilers they are built for. These are summarized in the table below:

| MPI Library | Where? | Compilers |
|---|---|---|
| MPICH | Linux clusters | GNU, Intel, PGI, Clang |
| Open MPI | Linux clusters | GNU, Intel, PGI, Clang |
| Intel MPI | Linux clusters | Intel, GNU |
| IBM BG/Q MPI | BG/Q clusters | IBM, GNU |
| IBM Spectrum MP | Coral Early Access and Sierra clusters | IBM, GNU, PGI, Clang |

# Message Passing Interface (MPI)...

## Message Passing Model

- The message passing model is based on the notion of processes
  - an think of a process as an instance of a running program, together with the program's data
- In the message passing model, parallelism is achieved by having many processes co-operate on the same task
- Each process has access only to its own data
  - i.e all variables are private
- Processes communicate with each other by sending and receiving messages
  - typically library calls from a conventional sequential language

# Message Passing Interface (MPI). . .

## Single-Program-Multiple-Data(SPMD)

- Most message passing programs use the Single-Program-Multiple-Data (SPMD) model
- All processes run (their own copy of) the same program
- Each process has a separate copy of the data
- To make this useful, each process has a unique identifier
- Processes can follow different control paths through the program, depending on their process ID
- Usually run one process per processor / core

# Message Passing Interface (MPI)...

## Writing MPI programs

```
//hello.c program
#include "mpi.h"
#include <stdio.h>

int main(int argc , char argv){
 MPI_Init(&argc, &argv);
 print("Hell world!\n");
 MPI_Finalize();
 return 0;
}
```

# Message Passing Interface (MPI)...

### Writing MPI programs

- #include "mpi.h" provides basic MPI definitions and type
- MPI_Init starts MPI
- MPI_Finalize exits MPI
- Note that all non MPI routines are local;thus the printf run on each process

### Compiling, linking & Running MPI programs

- Both opemmpi & MPICH implementation provides the commands `mpicc` for compiling and linking.
  - `mpicc -o hello hello.c`
- mpirun is common with several MPI implementations to run the progrma.
  - `mpirun -np 4 hello`

# Message Passing Interface (MPI)...

## Finding out about the environment

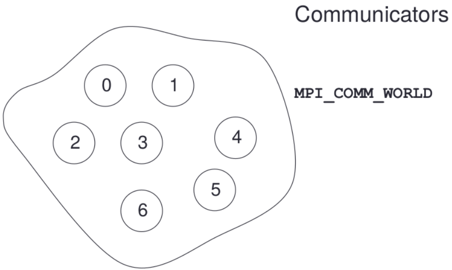
- Two of the first questions asked in a parallel program are:
  1. How many processes are there? and
  2. Who am I?
- How many is answered with `MPI_Comm_size`
- Who am I is answered with `MPI_Comm_rank`.
- The rank is a number between zero and size-1.

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

# Message Passing Interface (MPI)...

## Finding out about the environment ...

- MPI_COMM_WORLD is a communicator.
- Communicators are used to separate the communication of different modules of the application.
- Communicators are essential for writing reusable libraries.



## Emulating General Message Passing (C)

```
main (int argc, char **argv)
{
if (controller_process)
{
Controller( /* Arguments */ );
}
else
{
Worker
( /* Arguments */ );
}
}
```

# Message Passing Interface (MPI). . .

## Messages

- A message transfers a number of data items of a certain type from the memory of one process to the memory of another process
- A message typically contains:
  - the ID of the sending processor
  - the ID of the receiving processor
  - the type of the data items
  - the number of data items
  - the data itself
  - a message type identifier

# Message Passing Interface (MPI)...

## Blocking Point-to-Point Communications

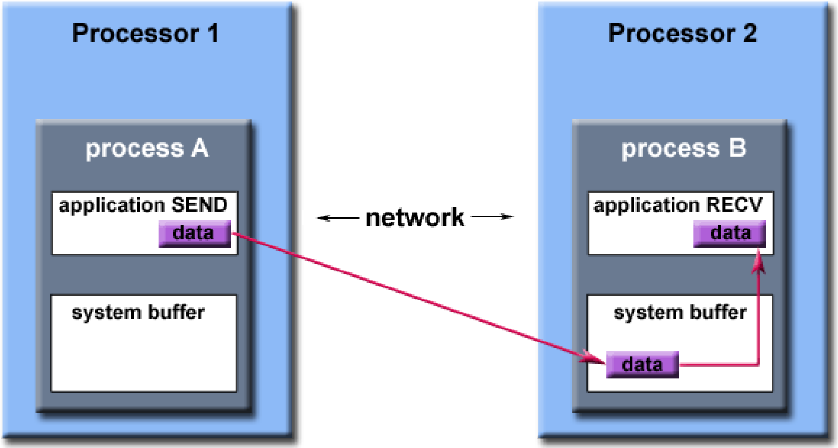
- We have considered two processes:
  - one sender
  - one receiver
- This is called point-to-point communication
  - simplest form of message passing
  - relies on matching send and receive
- Close analogy to sending personal emails

### Blocking point-to-point communication

- **Sending** and **receiving** are the two foundational concepts of MPI.
- Almost every single function in MPI can be implemented with basic send and receive calls.
- MPI's send and receive calls operate:
    - First, process A decides a message needs to be sent to process B.
    - Process A then packs up all of its necessary data into a buffer(envelopes) for process B.
    - The location of the message is defined by the process's rank.
    - MPI allows senders and receivers to also specify message IDs with the message (known as tags)

# Message Passing Interface (MPI). . .

## Blocking point-to-point communication . . .



# Message Passing Interface (MPI). . .

## Blocking point-to-point communication . . .

- Let's look at the prototypes for the MPI sending functions.

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

# Message Passing Interface (MPI)...

## Blocking point-to-point communication ...

- Let's look at the prototypes for the MPI receiving functions.

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

## Elementary MPI datatypes

| MPI Datatype | C datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short char |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | char |

# Message Passing Interface (MPI). . .

## MPI send / recv program

```
// Find out rank, size
int world_rank; int world_size; int number;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",number);
}
```

## Six Function MPI

- MPI is very simple. These six functions allow you to write many programs:
    1. MPI_Init
    2. MPI_Finalize
    3. MPI_Comm_size
    4. MPI_Comm_rank
    5. MPI_Send
    6. MPI_Recv

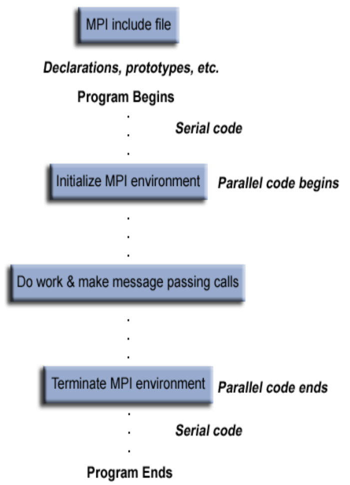
# Message Passing Interface (MPI). . .

### Communication modes

- Sending a message can either be synchronous or asynchronous
- A synchronous send is not completed until the message has started to be received
- An asynchronous send completes as soon as the message has gone
- Receives are usually synchronous - the receiving process must wait until the message arrives

# Message Passing Interface (MPI)...

## General MPI Program Structure:



# Message Passing Interface (MPI). . .

## Example in C Language

```
// required MPI include file
   #include "mpi.h"
   #include <stdio.h>
   int main(int argc, char *argv[]) {
   int  numtasks, rank, len, rc;
   char hostname[MPI_MAX_PROCESSOR_NAME];
   MPI_Init(&argc,&argv);    // initialize MPI
   MPI_Comm_size(MPI_COMM_WORLD,&numtasks); // get number of tasks
   MPI_Comm_rank(MPI_COMM_WORLD,&rank); // get my rank
   MPI_Get_processor_name(hostname, &len);
   printf ("Number of tasks= %d My rank= %d Running on %s\n",
           numtasks,rank,hostname);
   MPI_Finalize();    // done with MPI
   }
```

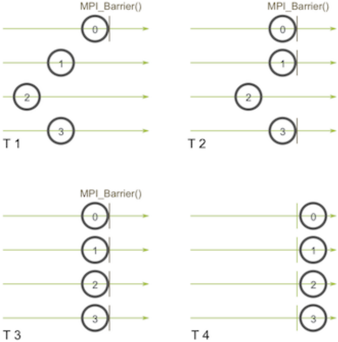
# Message Passing Interface (MPI)...

## Collective Communications

- MPI also provides primitives for collective communication.
- There are many instances where communication between groups of processes is required
- Collective communication involves many tasks of the application.
- Two simple collective operations:
  - MPI_Bcast spreads data from the root task to all tasks in the communicator comm.
  - MPI_Reduce combines data from all processes in the communicator (using operation), and returns the result to the task root.

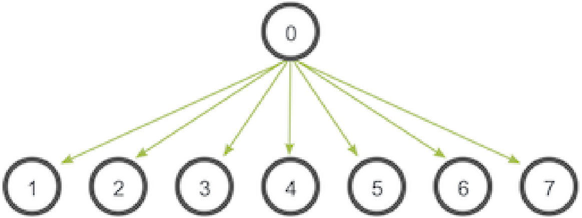
## MPI Broadcast

- One of the things to remember about collective communication is that it implies a synchronization point among processes
- MPI has a special function that is dedicated to synchronizing processes:
  - `MPI_Barrier(MPI_Comm communicator)`



## Broadcasting with MPI_Bcast

- A broadcast is one of the standard collective communication techniques.
- During a broadcast, one process sends the same data to all processes in a communicator.
- One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.
- The communication pattern of a broadcast looks like this:



# Message Passing Interface (MPI)...

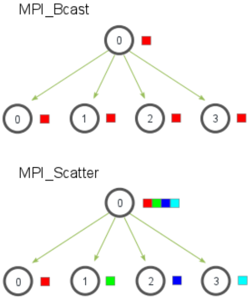
## Broadcasting with MPI_Bcast ...

- process zero is the root process, and it has the initial copy of data. All of the other processes receive the copy of data.
- In MPI, broadcasting can be accomplished by using MPI_Bcast
- The function prototype looks like this:

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)
```

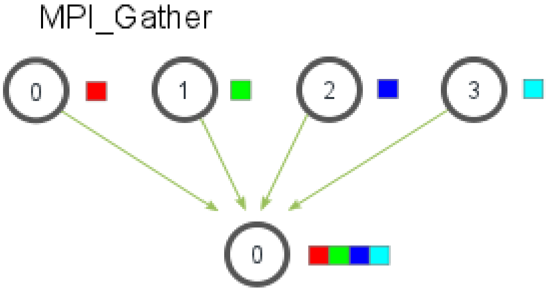
## MPI_Scatter

- MPI_Scatter is a collective routine that is very similar to MPI_Bcast
- MPI_Scatter involves a designated root process sending data to all processes in a communicator.
- MPI_Bcast sends the same piece of data to all processes while MPI_Scatter sends chunks of an array to different processes.



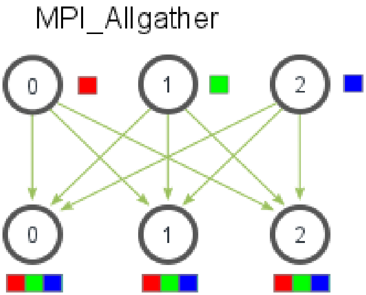
## MPI_Gather

- MPI_Gather is the inverse of MPI_Scatter
- MPI_Gather takes elements from many processes and gathers them to one single process.



## MPI_Allgather

- MPI_Allgather will gather all of the elements to all the processes.
- In the most basic sense, MPI_Allgather is an MPI_Gather followed by an MPI_Bcast



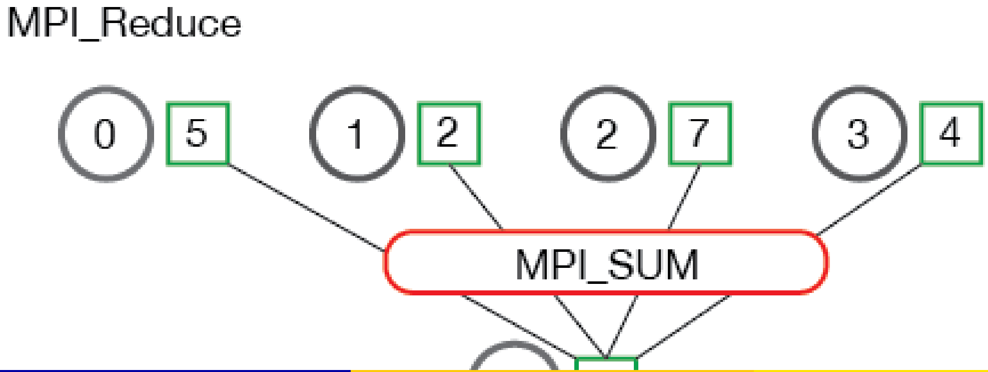
# Message Passing Interface (MPI)...

## MPI Reduce and Allreduce

- Data reduction involves reducing a set of numbers into a smaller set of numbers via a function.
- Reducing this list of numbers with the sum function would produce `sum([1, 2, 3, 4, 5]) = 15`.
- The reduction operations defined by MPI include:
  - MPI_MAX - Returns the maximum element.
  - MPI_MIN - Returns the minimum element.
  - MPI_SUM - Sums the elements.
  - MPI_PROD - Multiplies all elements.
  - MPI_LAND - Performs a logical and across the elements.
  - MPI_LOR - Performs a logical or across the elements.
  - MPI_BAND - Performs a bitwise and across the bits of the elements.
  - MPI_BOR - Performs a bitwise or across the bits of the elements.
  - MPI_MAXLOC - Returns the maximum value and the rank of the process that owns it.
  - MPI_MINLOC - Returns the minimum value and the rank of the process that owns it.

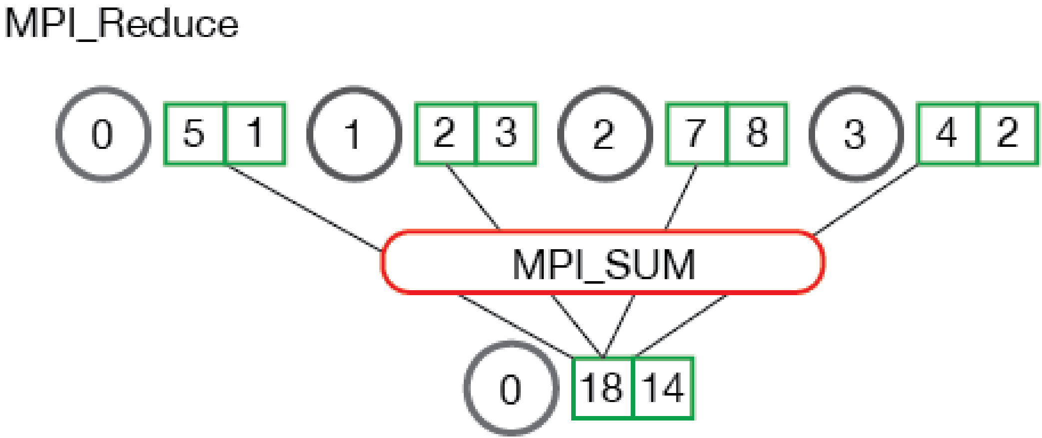
## MPI Reduce and Allreduce

- MPI_Reduce is called with a root process of 0 and using MPI_SUM as the reduction operation.
- The four numbers are summed to the result and stored on the root process.



## MPI Reduce

- The illustration below shows reduction of multiple numbers per process.



# Message Passing Interface (MPI)...

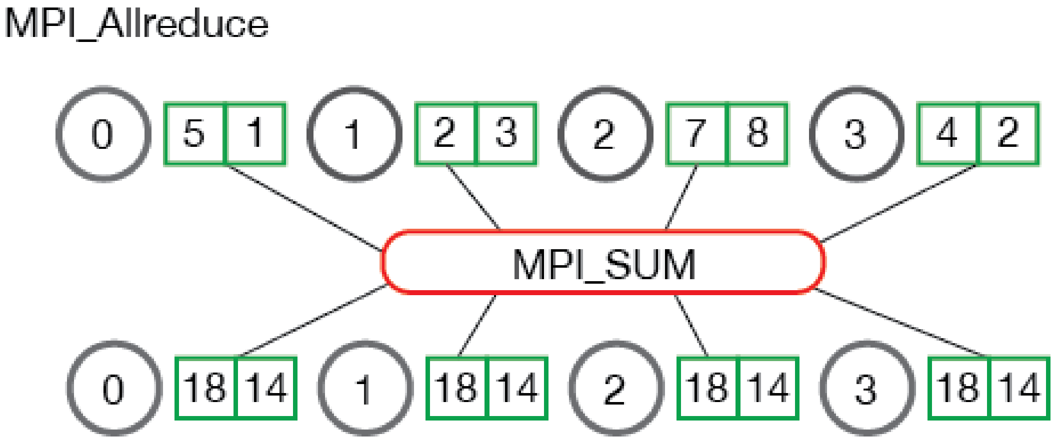
## MPI Allreduce

- Many parallel applications will require accessing the reduced results across all processes rather than the root process
- In a similar complementary style of MPI_Allgather to MPI_Gather, MPI_Allreduce will reduce the values and distribute the results to all processes.
- The function prototype is the following:

```
MPI_Allreduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm communicator)
```

## MPI Allreduce . . .

- The following illustrates the communication pattern of MPI_Allreduce:



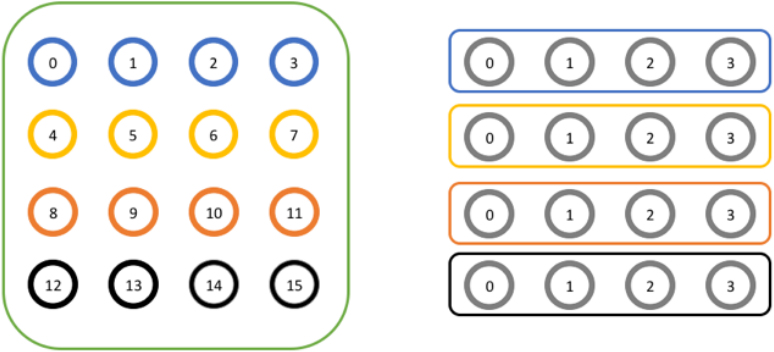
### Groups and Communicators

- For simple applications, it's not unusual to do everything using MPI_COMM_WORLD
- For more complex use cases, it might be helpful to have more communicators.
- the first and most common function used to create new communicators:

```
MPI_Comm_split(
    MPI_Comm comm,
    int color,
    int key,
    MPI_Comm* newcomm)
```

## Groups and Communicators

Split a Large Communicator Into Smaller Communicators



# Message Passing Interface (MPI)...

## Groups and Communicators

- Let's look at the code for this:

```
/ Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
```

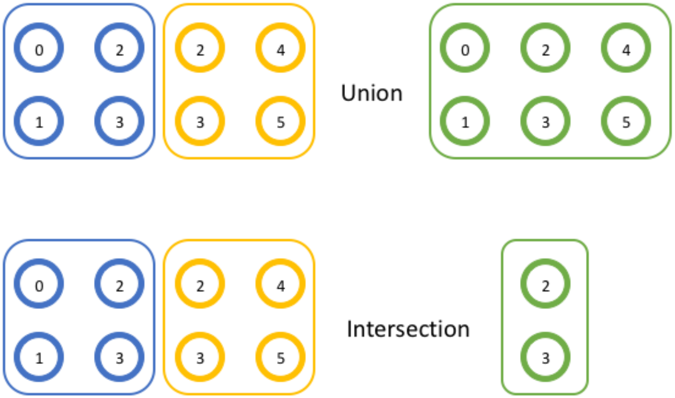
### Groups and Communicators

- Another function is MPI_Comm_create.
- Its signature is almost identical:

```
MPI_Comm_create(
    MPI_Comm comm,
    MPI_Group group,
    MPI_Comm* newcomm)
```

# Message Passing Interface (MPI)...

## Groups and Communicators



### Groups and Communicators

- In MPI, it's easy to get the group of processes in a communicator with the API call, MPI_Comm_group.

```
MPI_Comm_group(
    MPI_Comm comm,
    MPI_Group* group)
```

# Message Passing Interface (MPI)...

## Groups and Communicators

- Once you have a group or two, performing operations on them is straightforward. Getting the union looks like this:

```
MPI_Group_union(
    MPI_Group group1,
    MPI_Group group2,
    MPI_Group* newgroup)
```

### Groups and Communicators

- And you can probably guess that the intersection looks like this:

```
MPI_Group_intersection(
    MPI_Group group1,
    MPI_Group group2,
    MPI_Group* newgroup)
```

## Groups and Communicators

- this function takes an MPI_Group object and creates a new communicator that has all of the same processes as the group.

```
MPI_Comm_create_group(
    MPI_Comm comm,
    MPI_Group group,
    int tag,
    MPI_Comm* newcomm)
)
```

# Download example and slides

1. clone the source code with git or

```
git clone  https://github.com/mesfind/openmpi.git
```

2. Use the this link to download manually https://github.com/mesfind/openmpi

# Run the examples on HPC

1. comiple the source with parallel version
2. modified the `submit.sh` file
3. submit the job with `qsub submit.sh`

# Exercises: Message-Passing Programming

### Installing MPI locally on your ubuntu

1. `sudo apt-get install gcc`
2. `sudo apt-get install openmpi-bin`
3. `sudo apt-get install libopenmpi-dev`

# Exercises: Message-Passing Programming

## Hello World

1. Write an MPI program which prints the message “Hello World”.
2. Compile and run on several processes in parallel
3. Modify your program so that each process prints out both its rank and the total number of processes P that the code is running on, i.e. the size of MPI_COMM_WORLD.
4. Modify your program so that only the master process (i.e. rank 0) prints out a message (very useful when you run with hundreds of processes)
5. What happens if you omit the final MPI procedure call in your program?

# Exercises: Message-Passing Programming

### Sum of Array in c

1. Write a program that sums all rows in an array using MPI parallelism.