# Distributed Computing Fundamentals

**CSInParallel Project**

July 26, 2012

# CONTENTS

CHAPTER

ONE

# PREREQUISITES

Before getting started with this module, please keep in mind that we assume you have **sufficient knowledge with C programming**.

CHAPTER

TWO

# INTRODUCTION TO CLUSTER

**Definition**: "A cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers cooperatively working together as a single, integrated computing resource." - by Rajkumar Buyya.

A cluster is usually a linux-based operating system. Basically, a cluster has four major components:

- Network is to provide communications between nodes and server.
- Each node has its own processor, memory, and storage.
- Server is to provide network services to the cluster.
- Gateway acts as a firewall between the outside world and the cluster.

In order to prepare for what you will be working on, you should have a good understanding of parallel computer architectures. We are going to look at two parallel computer architectures:

- Shared Memory Model
- Distributed Memory Model

General Characteristics of Shared Memory Model:

"

- 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." [1]

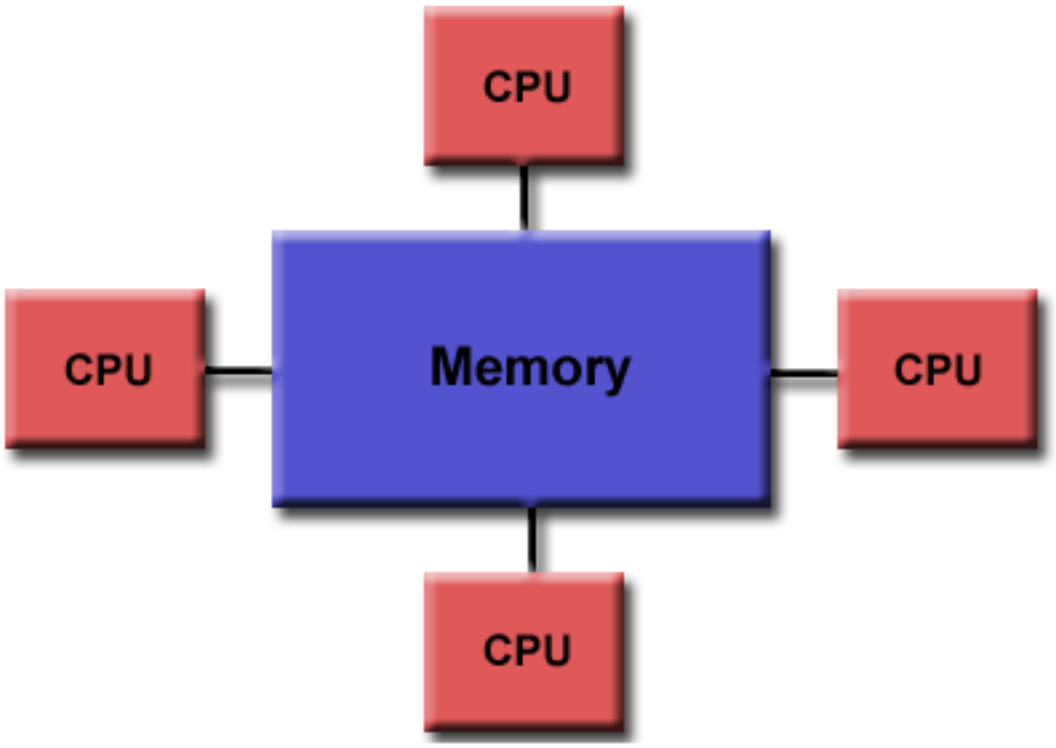


Figure 1: Shared Memory: Uniform Memory Access Obtained from www.computing.llnl.gov [2]

General Characteristics of Distributed Memory Model:

“

- 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.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer’s responsibility.” [3]

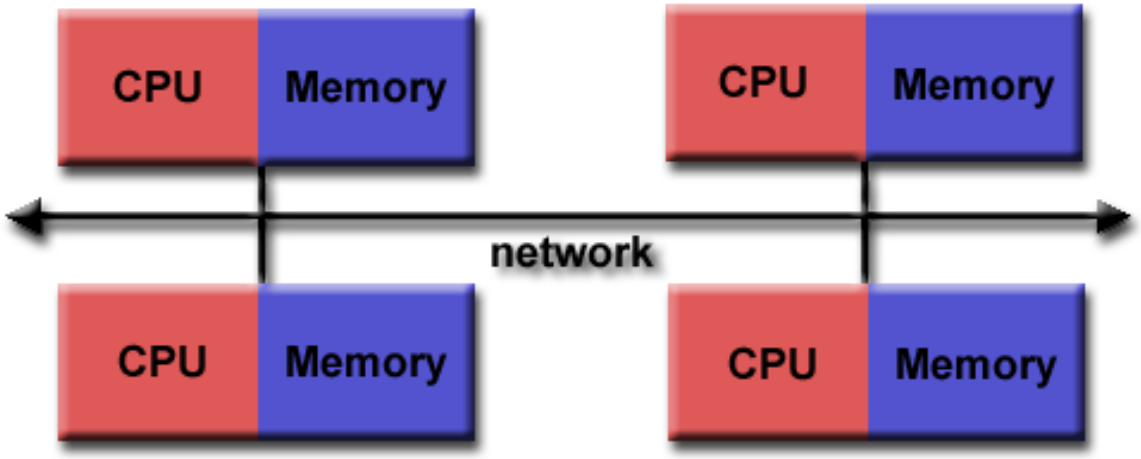


Figure 2: Distributed Memory System Obtained from www.computing.llnl.gov [4]

Some benefits of using clusters are:

- Inexpensive: Hardware and software of a cluster cost significantly much less than those of a supercomputer.
- Scalability: extra nodes can be added to a cluster when work exceeds the capacities of the current system in the cluster.
- Maintenance: A cluster is relatively easy to set up and maintain.
- High Performance: Operations should be optimized and efficient.
- Great capacity: Ability to solve a larger problem size.

There are many applications of clustering such as:

- Scientific computation
- Parametric Simulations
- Database Applications
- Internet Applications
- E-commerce Applications

**Recommended Reading:**

- Please read Cluster Computing: High-Performance, High-Availability, and High-Throughput Processing on a Network of Computers [5].
- Case Studies on Cluster Applications: read from page 16 - 22.

In order to use a cluster effectively, we need to have some programming environments such as Message Passing Interface (MPI), and OpenMP, etc. In this module, we will be learning about MPI on distributed memory cluster.

### References

CHAPTER

THREE

# LOCAL CLUSTER CONFIGURATIONS

There are two local clusters at Macalester College, one is Selkie, and the other is LittleFe. Here we want to give you a sense of how the two clusters work.

## 3.1 LittleFe

LittleFe is a 6-node distributed memory cluster. Each node is a motherboard consists of a CPU, I/O devices, and memory .etc. It uses Message Passing Interface (MPI) library to distribute the task to each node, and let each node work their own task simultaneously. Then we can combine the result from each node to get the overall output. Each node of LittleFe has nVidia GT 218 GPU, which enables LittleFe to do GPU programming, CUDA C. All the nodes connect via the network router.

LittleFe is Linux-based operating system, which is provided by Bootable Cluster CD (BCCD). BCCD also provides some cool applications that can be ran using MPI such as GalaxSee, Life, and Parameter-space, etc. Because of LittleFe's ability, we have also been working on Heterogeneous programming model, which combines both CUDA and MPI. You will see in the next module. In this module, we will be using LittleFe for our MPI programming activities. Below is a picture of LittleFe.

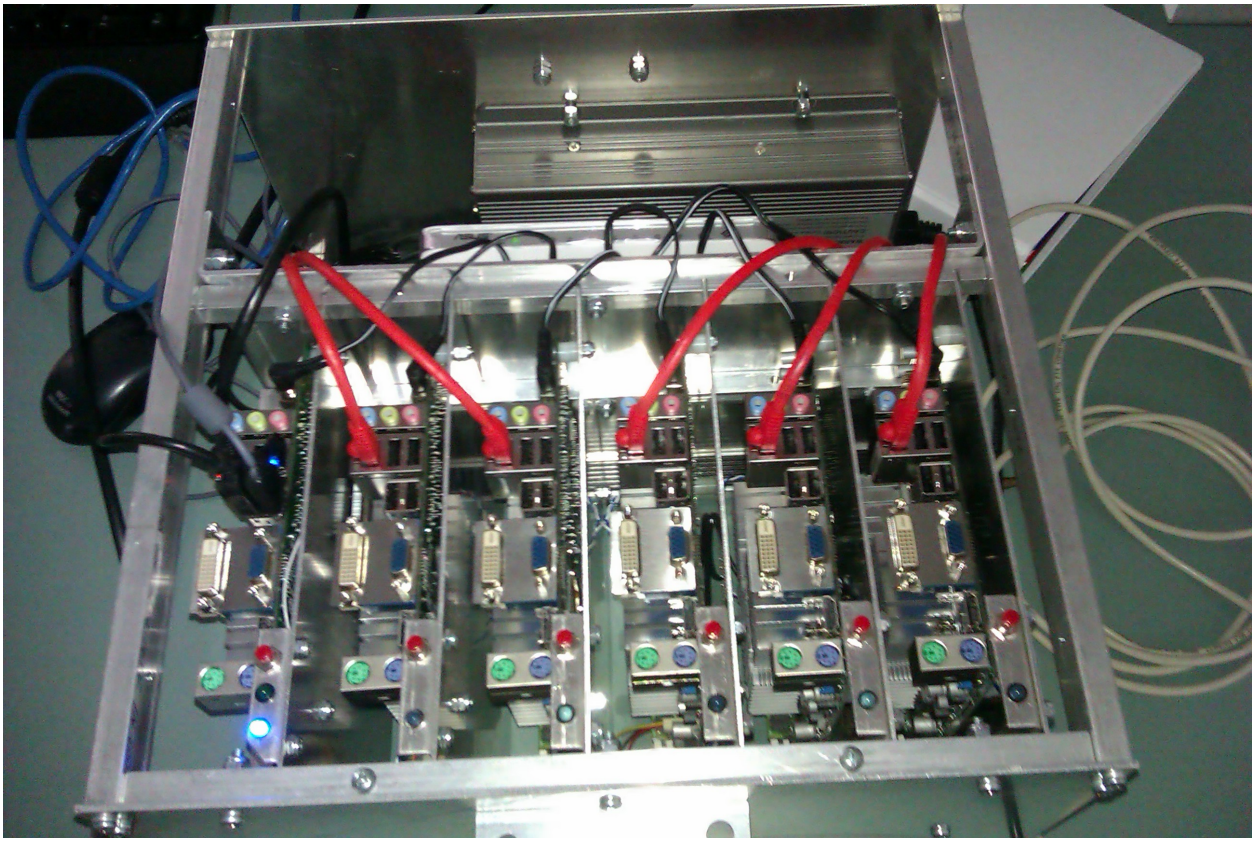

Figure 1: LittleFe

## 3.2 Selkie

Selkie is a virtual cluster, which was built in summer 2011, at Macalester College by a computer science summer research group. The hardware platforms of Selkie are Apple iMacs, 16 mid-2010 models and 28 new mid-2011 models. Those iMacs machines are quad-core models, and 8 GB of RAM. The mid-2010 models have a 1 TB hard drive, and each of 2011-models have a 500 GB hard drive. VirtualBox product was used for virtual machines because VirtualBox runs on Mac OS X, and it is open source. The operating system configuration on each virtual machine is Ubuntu linix Server version 10.04.

**Recommended Reading:**

- To read more on Selkie Cluster

### 3.2.1 How to log in onto your local clusters

Your instructor will provide the instructions of how to log in onto your local clusters.

CHAPTER

FOUR

# INTRODUCTION TO MPI

## 4.1 What is MPI ?

Message Passing Interface (MPI) is a subroutine or a library for passing messages between processes in a distributed memory model. MPI is not a programming language. MPI is a programming model that is widely used for parallel programming in a cluster. In the cluster, the head node is known as the master, and the other nodes are known as the workers. By using MPI, programmers are able to divide up the task and distribute each task to each worker or to some specific workers. Thus, each node can work on its own task simultaneously.

Since this is a small module, we will be focusing on only important and common MPI functions and techniques. For further study, there are a lot of free resources available on the internet.

## 4.2 Why MPI ?

There are many reasons of using MPI as our parallel programming model:

- MPI is a standard message passing library, and it is supported on all high-performance computer platforms.
- MPI program is able to run on different platforms that support the MPI standard without changing your source codes.
- Because of its parallel features, programmers are able to work on a much larger problem size with the faster computation.
- There are many functions defined in MPI Library for our usage.
- A variety of implementations are available.

## 4.3 How do I write MPI program ?

In order to get the MPI library working, you need to include the header file **#include <mpi.h>** or **#include "mpi.h"** in your C code.

### 4.3.1 MPI Program Structure

Like other programming languages you have seen, program that includes MPI library has its structure. The structure is shown in the figure below:

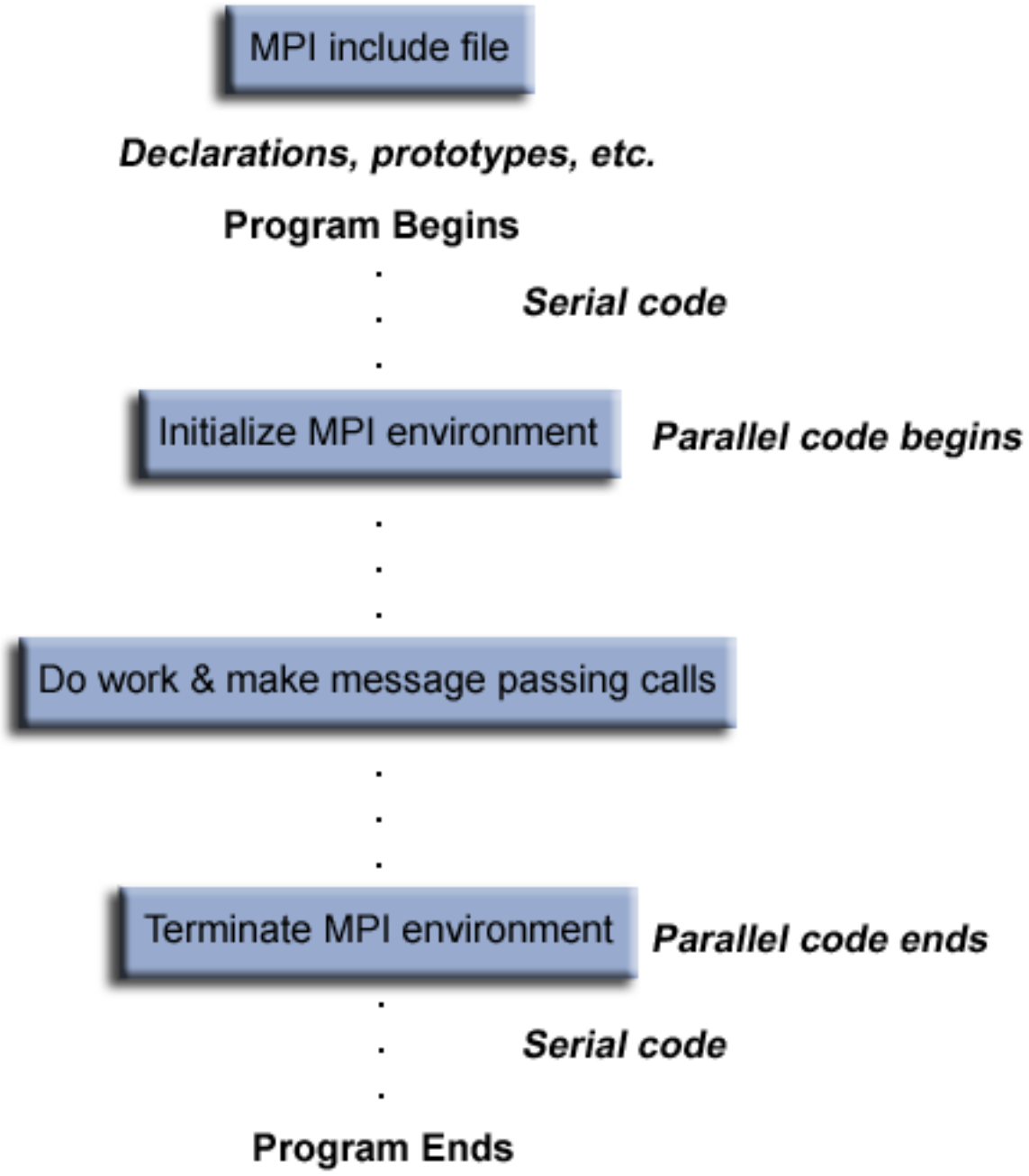


Figure 3: MPI program structure Obtained from computing.llnl.gov [1]

A MPI program is basically a C program with extending MPI library, SO DON'T BE SCARED. The program has two different parts, one is serial, and the other is parallel. Serial part contains variable declarations, etc., and the parallel part starts when MPI execution environment has been initialized, and ends when MPI_Finalize() has been called.

**Communicator**: a set of processes that have a valid rank of source or destination fields. The predefined communicator is MPI_COMM_WORLD, and we will be using this communicator all the time in this module. MPI_COMM_WORLD is a default communicator consisting all processes. Furthermore, a programmer can also define a new communicator, which has a smaller number of processes than MPI_COMM_WORLD does.

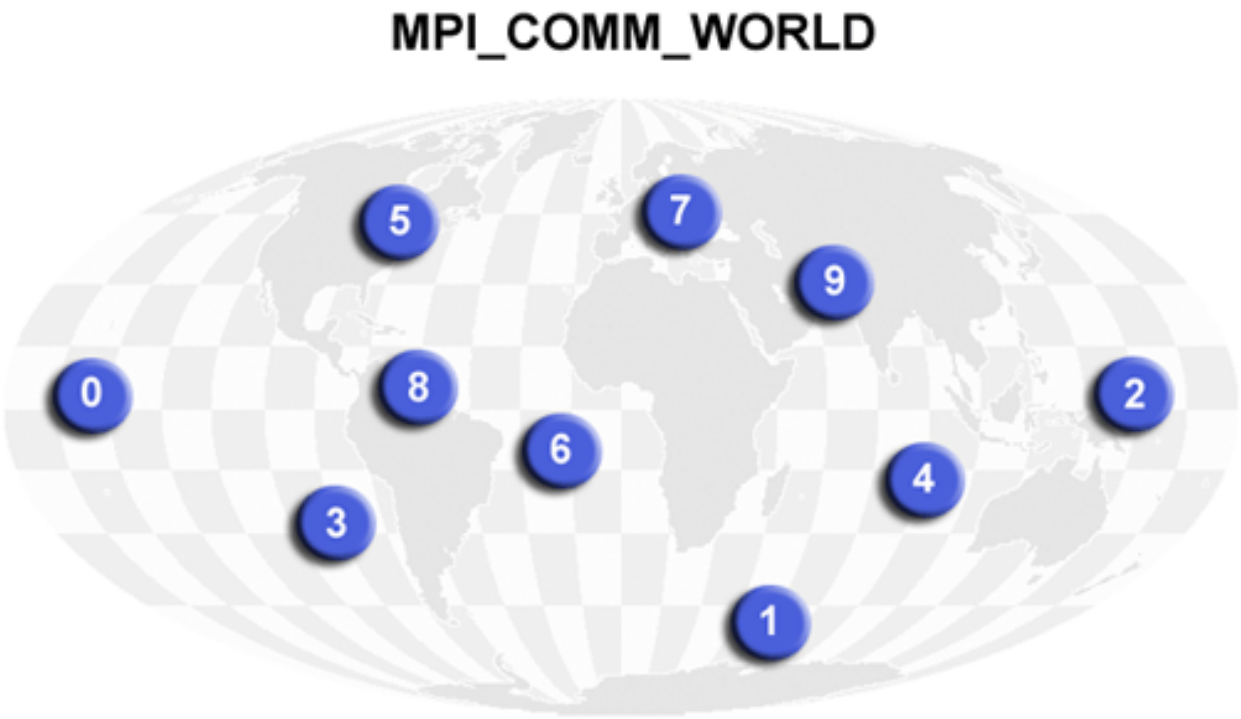


Figure 4: MPI_COMM_WORLD Obtained from computing.llnl.gov [2]

**Processes**: For this scope of knowledge, we just need to know that processes belong to the MPI_COMM_WORLD. If there are *p* processes, then each process is defined by its rank, which starts from *0* to *p - 1*. The master has the rank *0*.

### 4.3.2 Some Common Functions:

The following functions are the functions that are commonly used in MPI programs:

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

This function has to be called in every MPI program. It is used to initialize the MPI execution environment.

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

This function determines the number of processes in the communicator. The number of processes get store in the variable *size*. All processes in a communicator have the same value of *size*.

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

This function determines the rank of the calling process within the communicator. Each process is assigned uniquely by integer rank from *0* to *number of processes - 1*, and its rank gets stored in the variable *rank*.

```
MPI_Get_processor_name(name, &len)
```

This function returns the unique processor name. Variable *name* is the array of char for storing the name, and *len* is the length of the name.

```
MPI_Wtime()
```

This function returns an elapsed wall clock time in seconds on the calling processor. This function is often used to measure the running time of an MPI program. There is no defined starting point; therefore, in order to measure the running time, a programmer needs to call two different MPI_Wtime(), and find the difference.

```
MPI_Finalize()
```

This function terminates the MPI execution environment. MPI_Finalize() has to be called by all processes before exiting.

### Example 1: Hello World MPI

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

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

        MPI_Init (&argc, &argv);        /* creates MPI execution environment */
        MPI_Comm_rank (MPI_COMM_WORLD, &rank);        /* get current process rank */
        MPI_Comm_size (MPI_COMM_WORLD, &nprocs);/* get number of processes */
        printf("Hello world from process %d of %d\n", rank, nprocs);
        MPI_Finalize();                        /* terminates the MPI execution environment */
        return 0;
}
```

**Comments** In this hello world program, it illustrates how to use some basic functions of MPI. First, it initializes the MPI execution environment. Then it prints "Hello world from process rank of number of processes". Finally, it terminates the MPI execution environment.

### References

CHAPTER

FIVE

# MPI COMMUNICATIONS

## 5.1 Point-to-point Communication

Point-to-point communication is a way that pair of processors transmits the data between one another, one processor sending, and the other receiving. MPI provides SEND(MPI_Send) and RECEIVE(MPI_Recv) functions that allow point-to-point communication taking place in a communicator.

```
MPI_Send(void* message, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm comm)

        - message:      initial address of the message
        - count:        number of entries to send
        - datatype:     type of each entry
        - destination:  rank of the receiving process
        - tag:          message tag is a way to identify the type of a message
        - comm:         communicator (MPI_COMM_WORLD)

MPI_Recv(void* message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Sta

        - source:       rank of the sending process
        - status:       return status
```

**Note:** To read more on MPI_Status, please read MPI_Status.

### 5.1.1 MPI Datatype

In most MPI functions, which you will be using, require you to specify the datatype of your message. Below is the table showing the corresponding datatype between MPI Datatype and C Datatype.

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

**Table 1: Corresponding datatype between MPI and C**

### 5.1.2 Example 2: Send and Receive Hello World

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

#define FROM_MASTER 1

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

    int rank, nprocs;
    char message[12] = "Hello, world";

    /* status for MPI_Recv */
    MPI_Status status;

    /* Initialize MPI execution environment */
    MPI_Init(&argc, &argv);
    /* Determines the size of MPI_COMM_WORLD */
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs)
    /* Give each process a unique rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* If the process is the master */
    if ( rank == 0 )
        /* Send message to process whose rank is 1 in the MPI_COMM_WORLD */
        MPI_Send(&message, 12, MPI_CHAR, 1, FROM_MASTER, MPI_COMM_WORLD);

    /* If the process has rank of 1 */
    else if ( rank == 1 ) {
        /* Receive message sent from master */
        MPI_Recv(&message, 12, MPI_CHAR, 0, FROM_MASTER, MPI_COMM_WORLD, &status);
        /* Print the rank and message */
        printf("Process %d says : %s\n", rank, message);
    }

    /* Terminate MPI execution environment */
    MPI_Finalize();
}
```

**Comments** This MPI program illustrates the use of MPI_Send and MPI_Recv functions. Basically, the master sends a message, “Hello, world”, to the process whose rank is 1, and then after having received the message, the process prints the message along with its rank.

## 5.2 Collective Communication

Collective communication is a communication that must have all processes involved in the scope of a communicator. We will be using MPI_COMM_WORLD as our communicator; therefore, the collective communication will include all processes.

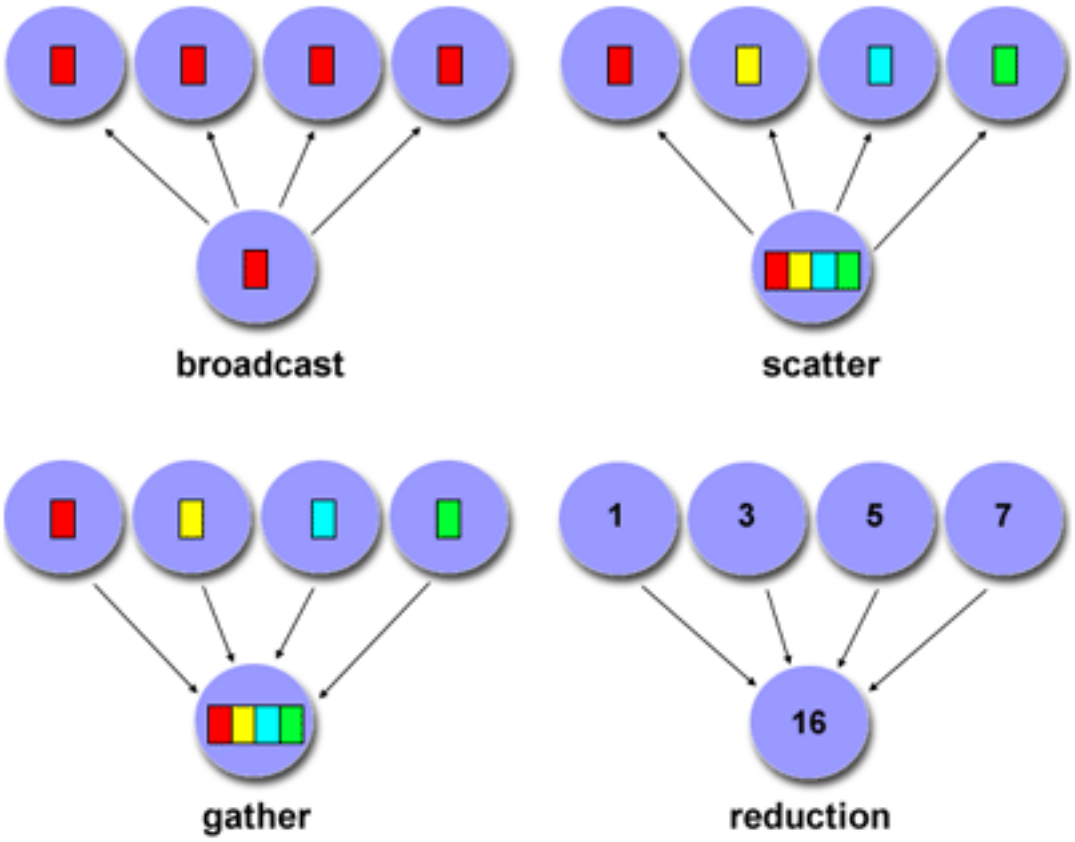


Figure 5: Collective Communications Obtained from computing.llnl.gov [1]

```
MPI_Barrier(comm)
```

This function creates a barrier synchronization in a commmunicator(MPI_COMM_WORLD). Each task waits at MPI_Barrier call until all other tasks in the communicator reach the same MPI_Barrier call.

```
MPI_Bcast(&message, int count, MPI_Datatype datatype, int root, comm)
```

This function displays the message to all other processes in MPI_COMM_WORLD from the process whose rank is root.

```
MPI_Reduce(&message, &receivemessage, int count, MPI_Datatype datatype, MPI_Op op, int root, comm)
```

This function applies a reduction operation on all tasks in MPI_COMM_WORLD and reduces results from each process into one value. MPI_Op includes for example, MPI_MAX, MPI_MIN, MPI_PROD, and MPI_SUM, etc.

---

**Note:** To read more on MPI_Op, please read MPI_Op.

---

```
MPI_Scatter(&message, int count, MPI_Datatype, &receivemessage, int count, MPI_Datatype, int root, co
```

This function divides a message into equal contiguous parts and sends each part to each node. The master gets the first part, and the process whose rank is 1 gets the second part, and so on. The number of elements get sent to each worker is specified by count.

```
MPI_Gather(&message, int count, MPI_Datatype, &receivemessage, int count, MPI_Datatype, int root, co
```

This function collects distinct messages from each task in the communicator to a single task. This function is the reverse operation of MPI_Scatter.

### References

CHAPTER

SIX

# COMPILING AND ACTIVITES

## 6.1 Compiling an MPI program

To compile an MPI program on your local cluster, you can enter the following commands in the terminal:

First, we need to make an executable file from the MPI program by using **mpicc**:

```
mpicc -o filename filename.c
```

Then you are able to execute it using **mpirun** :

```
mpirun -machinefile machines -np #processes ./filename
```

---

**Note:** **machines**: is the instruction for running the executable file on a cluster. It tells the executable file to run on which nodes and how many times on those nodes. For example, machines on LittleFe has structure as follow:

- node000.bccd.net slots = 1
- node011.bccd.net slots = 1
- node012.bccd.net slots = 1
- node013.bccd.net slots = 1
- node014.bccd.net slots = 1
- node015.bccd.net slots = 1

---

Moreover, you can also compile an MPI program without using **machines**, you can use the following command to run only on the master node:

```
mpirun -np #processes ./filename
```

---

**Note:** Please ask your instructor for instructions on how to log in onto your local machine.

---

## 6.2 Activity 1: What is my $\pi$ ?

In this activity, we are going to compute $\pi$ using integration. We have formula:

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

Therefore, we can compute the area under the curve to get the value of the integral.

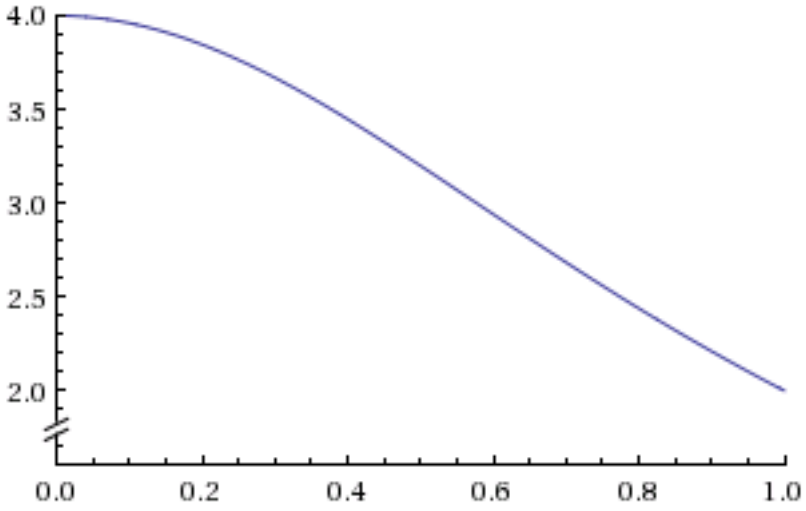


Figure 4: Graph for function

**Comments**

- We can split the area under the curve into bins. The idea is to group the bins into smaller chunks, and so we can use each process to calculate each chunk, and then combine the result into one value. Remember, that we can get a more accurate result if you split the area under the curve into more number of bins.
- In this activity, we also want you to time your computation by using MPI_Wtime() function. We provide you some parts of the code, and would like you to complete **TO DO**, and then you can experiment with the different number of bins you are using. Moreover, we want you to execute your program with different number of processes, and compare your timings. We will walk you through the code step by step.
- First, you need to initialize the MPI execution environment, define the size of communicator, and define the rank of each process. This should be straight forward for you. You are asked to complete this task.
- Then you want to let each process know the **number of bins** you are using. Therefore, you need to broadcast the **number of bins** to all processes in our MPI_COMM_WORLD. You should use **MPI_Bcast** to broadcast the **number of bins**. You are asked to complete this part of the code.
- Now we are ready to ask each process compute their task. We want to evaluate the integral of $\frac{4}{1+x^2}$ from *0* to *1*, and we can do so by finding the sum of all bins from *0* to *1*. Each bin is approximately $\frac{1}{n} * \frac{4}{1+x^2}$ (**n** is the number of bins). We are iterating over the number of bins, and we start from *0*; therefore, to find the center of each bin, we need to add + *0.5* to variable *i*. Moreover, in the **for loop**, we ask the rank *0* to compute the first bin, the (nprocs) bin, and so on, rank *1* to compute the second bin, the (nprocs + 1) bin, and so on, ..., as long as value of *i* is less than **n**. This can be done by using the following piece of code:

```
/* Calculating for each process */
step = 1.0 / (double) n;
sum = 0.0;
for (i = rank; i < n; i += nprocs) {
    x = step * ((double)i + 0.5);
    sum += (4.0/(1.0 + x*x));
```

```
}

mypi = step * sum;
```

- When all processes have finished their computations, their results are stored in **mypi**. Therefore, we can reduce all their results into one result, which is the value of $\pi$. Your task is to complete this part by using MPI_Reduce.

To download the source code to do your activity: `download mpi_pi_todo.c`

To download the entire source code from www.mcs.anl.gov [1]: `download mpi_pi_done.c`

## 6.3 Activity 2: Vector Matrix Multiplication

In this activity, we are going to compute vector matrix multiplication. This activity illustrates the use of MPI_Bcast, MPI_Scatter, and MPI_Gather to do this multiplication. First, we want you to complete this MPI program by filling codes at **TO DO**. After having completed this task, try to run this MPI program by using different number of processes. Try to explain yourself what is happening !

I will explain how the vector matrix multiplication works. First, let's say we have a matrix $A$, and a vector $x$ as below:

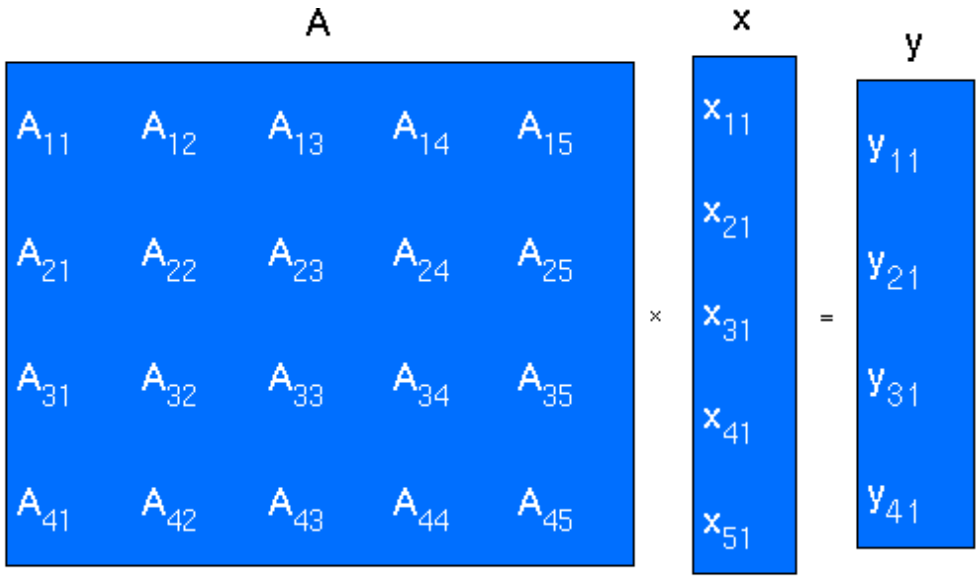


Figure 5: vector matrix multiplication Obtained from cms.uni-konstanz.de [2]

This multiplication produces a new vector whose length is the number of rows of matrix $A$. The multiplication is very simple. We take a row of matrix $A$ dot product with vector $x$, and this produces an element of the result vector. For instance, the first row of matrix $A$ dot products with vector $x$ will produce the first element in vector $y$.

**Comments**

- We will step you through this activity step by step. Since this is an MPI program, we need to create the MPI execution environment, define the size of the communicator, and give each process a unique rank. You are asked to completed this part of the code.
- After having initialized the MPI environment, we want to ask the master to initialize the vector and matrix we are going to multiply. In order to do that, we check if the process is master. If so, we initialize the matrix and vector. We initialize every entry to 1 because of its simplicity.

```
if (rank == 0) {
    /* Initialize the matrix and vector */
    for(i=0; i < WIDTH; i++) {
        vector[i] = 1;
        for(j = 0; j < WIDTH; j++) {
            matrix[i][j] = 1;
        }
    }
}
```

- Since the vector is not very large and all processes must have this vector to do the multiplication, we will broadcast the entire vector to all processes. We do this by using MPI_Bcast. In addition, we want to distribute the matrix to each process in the MPI_COMM_WORLD. We would do this using MPI_Scatter. You are asked to complete this task.
- When all processes can see the vector and some rows of matrix, they are now able to do the multiplication. We need to store their result in the result vector.

```
for(i = 0; i < chunk_size; i++) {
    result[i] = 0;
    for(j = 0; j < WIDTH; j++) {
        result[i] += local_matrix[i][j] * vector[j];
    }
}
```

- The last part you need to complete is to gather all result vectors in all processes, and store them in the output vector, called global_result vector. This will be our result vector. Moreover, we can print out the value of each element in the global_result vector, and then terminate the MPI execution environment.

To download the source code to do your activity: `download vector_matrix_todo.c`

To download the entire source code from www.public.asu.edu/~dstanzoi [3]: `download vector_matrix_done.c`

### References

CHAPTER

SEVEN

# DECOMPOSITION AND ACTIVITIES

Decomposition is a very important aspect to optimize the performance of parallel programming models. Decomposition is a way to divide up the task fairly; thus, each task can be distributed to each process. There are many ways to break up a task, and you should choose the way that the best suits your code. For instance, you should split your matrix by row, rather than by column if you want to compute matrix multiplication. Below is an example of decomposition method. You will see this method very often in your activities.

## 7.1 Example 3: Decompose the matrix by row

```
averow = ROW/numworkers;
extra = ROW%numworkers;
offset = 0;
mtype = FROM_MASTER;

for (dest = 1; dest <= numworkers; dest++) {
    rows = (dest <= extra) ? averow + 1 : averow;
    // Then send to each worker the number of rows
```

**Comments**

- **rows = (dest <= extra) ? averow + 1 : averow** means if **dest <= extra**, we have **rows = averow + 1**. Otherwise, we have **rows = averow**. This is a shorter version of if and else statement.
- In this example, **ROW** is the number of rows of the matrix, so each process will get at least **averow** rows. The **extra** is the extra rows when **ROW** is not divisible by number of workers. In order to send each task to each worker, we need to iterate over the number of workers. Then if we have extra rows, we know that number of extra rows must be less than the number of workers, so we can give one more row to workers whose ranks are less than or equal to extra.

### 7.1.1 Activity 3: Vector Matrix Multiplication Improved Version

In this activity, we will be using decomposition technique, MPI_Send, and MPI_Recv to improve the efficiency and accuracy of vector matrix multiplication. We already seen that by using MPI_Scatter, we do not get the right result if the length of vector is not divisible by the number of workers. Thus, we want to use the decomposition technique to help us divide the task fairly among each worker. Then, we can send each task to each worker by using MPI_Send. After the workers having received their tasks, they will compute their own task, and send their results back to the master. Finally the master will receive results from workers, and combine them into a result vector.

**Comments**

- I will walk you through the code step by step. First, we will need to initialize the MPI environment, define the size of the communicator, and give a rank to each process. This should be straight forward to you because you have seen this many times already.

```
/* Initialize MPI execution environment */
MPI_Init( &argc,&argv);
MPI_Comm_rank( MPI_COMM_WORLD, &rank);
MPI_Comm_size( MPI_COMM_WORLD, &nprocs);
```

- Then we want to initialize the vector and matrix on master node. This can be done by:

```
if (rank == 0) {
    /* Initialize Matrix and Vector */
    for(i = 0; i < WIDTH; i++) {
    vector[i] = 1;
        for(j = 0; j < WIDTH; j++) {
            matrix[i][j] = 1;
        }
    }
}
```

- We have seen that using the collective communications without decomposition is not the best way of doing this problem. Here is a better way that will work for any number of processes. We will be using the decomposition technique above to split the task for each process. Then, the master will be sending each process the number of rows (**rows**) of matrix, and the vector. You are asked to complete this part of the code.
- ---

  **Note:** You should use MPI_Send to send the starting rows, and number of rows, and some rows of matrix, and the vector to the workers. Moreover, when you send some rows of matrix, you should use MPI_Send(&matrix[starting row][0], number of elements, ...). This will send the rows of matrix, which contain the number of elements, and it starts from the first element in that row.

  ---

- ```
  averow = ROW/numworkers;
  extra = ROW%numworkers;
  offset = 0;
  mtype = FROM_MASTER;

  for (dest = 1; dest <= numworkers; dest++) {
      rows = (dest <= extra) ? averow + 1 : averow;
      // TO DO
      ..............
      // end TO DO
  }
  ```

- After having sent the messages to all workers, we need to ask workers to receive the messages from the master. We check if the process is not the master, we will use MPI_Recv to receive the messages sent from the master. You are asked complete this task.

```
if (rank > 0) {
    mtype = FROM_MASTER;
    /* Receive the task from master */
    // TO DO:
    ..............
```
```

```
    // end TO DO
}
```

- Each worker now can compute their task, and then we have to send the results back to the master. Sending results back to master should not be difficult. Since result is just a vector, we can send the starting rows, number of rows, and entire result vector to the master. You are asked to complete this part.

```
/* Each worker works on their computation */
for(i = 0; i < rows; i++) {
    result[i] = 0;
    for(j = 0; j < WIDTH; j++) {
        result[i] += matrix[i][j] * vector[j];
    }
}

/* send the result back to the master */
mtype = FROM_WORKER;
// TO DO:
//.............
//end TO DO
```

- Finally, the master has to receive the results from all workers. You are asked to complete this task.

```
/* Receiving the work from each worker */
mtype = FROM_WORKER;
for (i = 1; i <= numworkers; i++) {
    source = i;
    // TO DO
    //..............
    // end TO DO
    printf("Received results from task %d\n", source);
}
```

To download the source code to do your activity: `download vector_matrix_todo.c`

To download the entire source code: `download vector_matrix_done.c`

### 7.1.2 Activity 4: Matrix Multiplication

In this activity, we want you to use decomposition technique, MPI_Send, and MPI_Recv in previous activities to complete the matrix multiplication program. If you have not seen matrix multiplication before, please click on matrix multiplication to read how matrix multiplication works.

**Comments**

- This activity is not much different from the previous activity. First, we use the decomposition method in the previous activity. Then we want to send some rows from the first matrix, and entire second matrix to each worker. Note that this is not the most efficient method of doing matrix multiplication because when the second matrix gets really large, we might not be able to send entire matrix to each worker. We use this method because of its simplicity.

```
/* Computing the row and extra row */
averow = ROWA/numworkers;
extra = ROWA%numworkers;
```

```
offset = 0;
mtype = FROM_MASTER;

/* Distributing the task to each worker */
for (dest = 1; dest <= numworkers; dest++) {
    /*If the rank of a process <= extra row, then add one more row to process*/
    rows = (dest <= extra) ? averow+1 : averow;
    printf("Sending %d rows to task %d offset=%d\n", rows, dest, offset);
    // TO DO:
    // ............
    // end TO DO
}
```

- Next we want each worker to receive messages sent from the master, and then we can use these matrices to do matrix multiplication on each worker. The result is then stored in matrix **c**. Your task is to receive the messages sent from the master.

```
if (taskid > MASTER) {
    mtype = FROM_MASTER;

    /* Each worker receive task from master */
    // TO DO
    // ..............
    // end TO DO

    /* Each worker works on their matrix multiplication */
    for (k = 0; k < COLB; k++){
        for (i = 0; i < rows; i++) {
            c[i][k] = 0.0;
            for (j = 0; j < COLA; j++)
                c[i][k] = c[i][k] + a[i][j] * b[j][k];
        }
    }
}
```

- After each worker has computed the matrix multiplication, all workers have to send the results back to the master. Each worker needs to send their matrix **c** to master. You are asked to complete this task.

```
/* Each worker sends the output back to master */
mtype = FROM_WORKER;
// TO DO
// ...........
// end TO DO
```

- Then master can receive the results from all workers, and combine them into a single result matrix. You are asked to complete this task.

```
/* Receive results from worker tasks */
mtype = FROM_WORKER; /* message comes from workers */
for (i = 1; i <= numworkers; i++) {
    source = i; /* Specifying where it is coming from */
    // TO DO
    // ..............
    // end TO DO
    printf("Received results from task %d\n",source);
}
```

To download the source code to do your activity: `download matrix_multiplication_todo.c`

To download the entire source code from computing.llnl.gov [1]: `download matrix_multiplication.c`

### References