PARALLEL PROGRAMMING - A PEDAGOGIC INTRODUCTION

Biji C. L.

ebook Series
University of Kerala

Internal Quality Assurance Cell
University of Kerala
2016

Parallel Programming - A Pedagogic Introduction

Biji C. L
Research Scholar,
Dept. of Computational Biology and Bioinformatics
University of Kerala

eBook Series No.5
Internal Quality Assurance Cell
2016

Contents

PREFACE

The eBook serves as a beginners guide on teach fundamentals of parallel programming with message passing interface and its application in Bioinformatics. The concepts of MPI are discussed in user friendly manner through simple C programs. The aim is to kindle interest in every user to develop MPI programs. The eBook is organized as 4 chapters. Chapter 1 gives an introduction to multitasking, parallel processing and evolution of programming language followed by an overview of the parallel computing paradigm, with a few metaphors establishing the relevance of parallel processing. Chapter 2 deals with the overview of Message Passing Interface (MPI). The basic terminologies and keywords related to MPI are briefly described. Chapter 3 provides parallel programming tutorials to help understand the fundamentals of MPI programming. Chapter 4 provides the application of parallel programming in Bioinformatics. A few toy examples and some of the parallel programming tools in Bioinformatics are also explained.

I wish to express my gratitude to my mentor Dr Achuthsankar S.Nair who kindled an interest in preparing this e-book by spotting my interest in technical writing. The opportunity to host a parallel computing seminar further fueled my resolved. I thank Dr. Satheesh Kumar, Dept. of Future Studies for introducing me to the concept of Parallel Programming. Technical support by the project students Shima V.M., Arun P.R, Manu K. Madhu and Jojo George during their tenure at the Dept. of Computational Biology Bioinformatics is acknowledged. I wish to thank all my fellow researchers, especially Mr. Sajil. C. K.

CHAPTER 1
INTRODUCTION

1.1 Multitasking

Multitasking is a basic human skill which everyone experiences in their day- to- day life. It's quite easy for human to walk and chat or to eat and watch a movie simultaneously. Similarly it comes automatically to a mother to watch television, talk on a phone, and watch over boiling milk, at the same taking care of her toddler. Now, taking the case of computer system, the concept of multitasking came up during 1950's. Multitasking is the ability to distribute different tasks at the same time on a single operating system.

Figure 1.1: A metaphor for multi-tasking in Indian tradition

Though everyone enjoys the pleasure of surfing the internet, playing music, texting and printing concurrently on a personal computer, many may not realize the power of multitasking in computers. It all started with the great vision of Alan Turing to create a machine that do human thinking and which of course had the inspiration of Charles Babbage during Victorian times. Pilot Ace Computer was the first computer that could do more than one task. It was designed and built in 1950's. The buzzword "Multitasking" for computer was coined during 1960's. Intuitively this describes the execution of various tasks by sharing the same resource (Central Processing Unit). In effect, computer is handling only one task at a time, but will switch the tasks very rapidly. The modern CPU is so powerful that, it can run thousands of instruction in a fraction of second. So, multitasking is time slicing of the task, so as to have a better utility of resources.

1.2 Parallel Processing

Parallel processing exploits the simultaneous use of various computer resources for solving a computational task. The various scientific domains from astrophysics to biology demand more computing power than traditional sequential computers for analyzing the huge data generated every day. But not all computational problems require parallel computing resources for executing the various operations. Therefore, a primary check is required for verifying the feasibility of parallel processes. An insightful metaphor is music. Consider the two genres of music- melody and harmony. In melody, notes follow one after the other while in harmony; certain selected compositions of notes are played together. Thus, a melody cannot be parallelized whereas a harmony can be. Many day-to-day errands are amenable to parallelization. Making fruit salad is an example. We can assign the task of cutting each of the required fruits to different person and can finally put together all the fruits and add sugar and ice cream. Thus, by dividing and paralleling the job, process is completed faster.

Figure 1.2: Parallelizable activities (a) harmony (b) Making of Fruit salad.
(Source: https://en.wikipedia.org/wiki/Orchestra, http://freefruitsaladrecipe.blogspot.in/2014/10/recipe-for-fresh-fruit-salad-fruit.html)

Let us consider task of summing 1 to 100. The problem execution can be paralleled for making it faster. For example, summing the numbers from 1 to 25 can be assigned to master processor(node), summing numbers from 26 to 50 can be assigned to processor 1, summing numbers from 51 to 75 can be assigned to processor 2 and the rest (from 76 to 100) to processor 3. Finally, every processor will give their sum values to the master node and it will again sum up all the intermediate values to get the final sum. Therefore each process requires one by fourth time to complete verification. Thus, parallel computing improves usage of time compared to that of sequential computing.

1.3 Evolution of Parallel Programming

Over years, programming languages have experienced tremendous evolution with respect to the different needs of mankind. During 1940's the programming languages were lengthy low level machine instructions. Later in 1950's Fortran (FORmula TRANslation) emerged as high level scientific language and its glory can be witnessed through its presence in aerospace, automotive and research institutes. The most popular C language came into existence around 1972. C may be viewed as parental language model through which many different languages have emerged. During early 1980's, C++ dominated the programming developers for low memory using applications. The idea of implementing scripting language was initiated by Guido van Rossum during mid 1980's. Python programming became popular since then as it is many times easier to use compared to the compiled language C and C++. With the development of massively parallel processors during late 1980's, parallel computing paradigms started to flourish. During late 1990's the cluster computers became more dominant and widely acceptable for many scientific parallel computing centers. Today, waves of parallel computing can be experienced in desktops and laptops. Though many different programming language support parallel processing, the scope of the book is limited to parallel programming in C with the support of Message Passing Interface (MPI). MPI emerged as a standard since mid- 1990's by simplifying the task of dividing multiple concurrent tasks through proper synchronization.

CHAPTER 2
CLUSTER COMPUTING AND MESSAGE PASSING INTERFACE[1]

Fueled by the rapid technological development and computing paradigm shift, many high through put and optimization problems which was hard to solve with a normal desktop computers can be solved effectively with cluster computers. A cluster is a type of parallel or distributed or independent computers that are interconnected by a high speed network system. It is also known as poor man's super computer or commodity super computer. When comparing to the super computer which owes a proprietary operating system, cluster use royalty free operating system. The key benefits of cluster computing are High performance, Scalability, System availability, High throughput, Optimization techniques, Parallel computing techniques, low cost and time etc. Cluster computing can be achieved in many different forms and Message passing interface (MPI) is one of them. MPI is the dominant model used in high performance computing (hpc) and generally used as the industry standard for writing Message Passing Programs on hpc platforms. It is a language independent communication protocol. It allows users to create programs that can run efficiently on most parallel architectures.

2.1 Some basic concepts in MPI

The basic terminologies using MPI is listed.

Communicator: Communicator is a group of processes that are able to communicate to each other.

Rank: To distinguish each process of the communicator, an ID assigned to each of them is termed as rank of the process. One processor communicates explicitly to another process using this ID.

Size: Size is the total number of processes belonging to a communicator.

2.1.1 MPI_Communication world

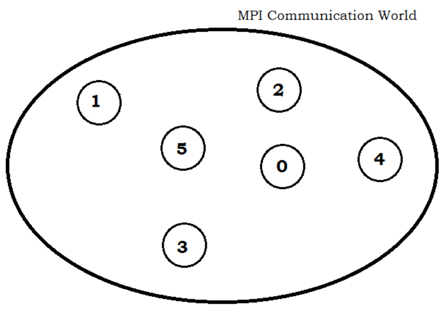


Figure 2.1: MPI Communication world

Figure 2.1 shows a schematic representation of an MPI Communication World. Communicator involves processors numbered from 0 to 5 i.e. total processors defined

[1] Earlier version of this chapter is published in Manu K. Madhu, Biji C.L., "Parallel Computing with Message Passing Interface ", CSI Communication volume No.38 (5), August 2014, pp 25-26.

is 6 hence the size of the communicator is 6. Each processor will be having a unique ID called rank. Generally, MPI supports two kinds of communications.

- Point to point communication: Communication involving one sender and one receiver is termed as Point to point communication. Figure 2.2 depicts a conventional representation of a Point to point communication.

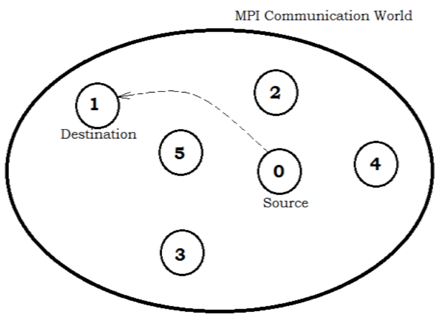


Figure 2.2: Schematic diagram of a Point to point communication

- Collective communication: While performing a task, it may be required for one processor to communicate with all other processors of the communicator. For instance, the Master node needs to communicate with all of its slave nodes for integrating the end result. MPI communication can be called using languages such as C, C++, FORTRAN77 and FORTRAN90. Figure 2.3 shows a schematic representation of a sample collective communication.

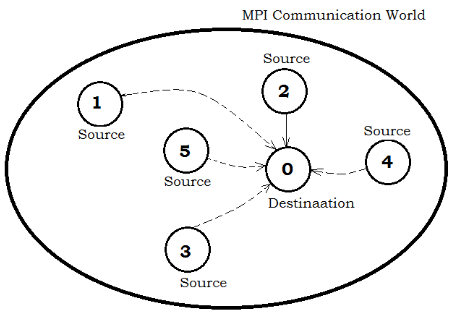


Figure 2.3: Schematic representation of a sample Collective communication

MPI Using C

For writing an MPI program in C language, the following facts need to remember.

- Include the header file mpi.h using the command include<mpi.h>. In C programming, all library functions are included in different header files, in different categories with .h extension. For an MPI program, 'include<mpi.h>' command will include all MPI subroutines. During the MPI subroutine call from the main program, compiler will go to MPI subroutine definition, which is available at the MPI library and executes the function definition. After execution, result is returned to the program from where it was called.
- C language is case-sensitive. All the MPI subroutines have the form MPI_Subroutine. For instance in MPI_Init, MPI_Comm_size where MPI, I, and

C are in upper-case. Constants defined in mpi.h are all in upper-case such as MPI_INT, MPI_SUM, and MPI_COMM_WORLD.

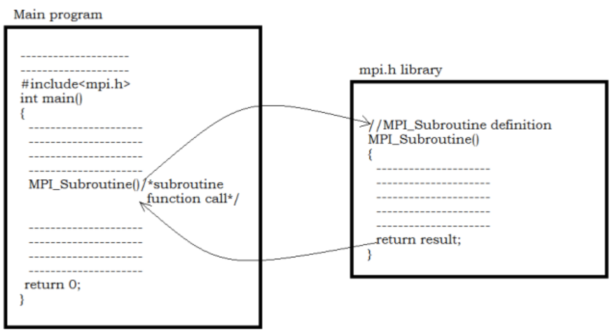
- In an MPI function call, the arguments that specify the address of a buffer should be specified as pointers.
- Return code of an MPI function call should be an integer value.
- Data types defined in the C semantics is in way that is more individual. A comparison of pre-defined MPI data types with C data types is listed in Table1.

Table 2.1: Comparison of MPI_Datatypes & C Datatypes

MPI Datatypes	C Datatypes
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_SHOR	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI Subroutines

MPI subroutine communicates among different processors in the MPI communication world for performing jobs in parallel. MPI subroutine can be called from languages such as C, C++, FORTRAN77 and FORTRAN90. As shown in figure 2.4, During the MPI subroutine call in the main program, compiler will execute the function and returns result back to the main program.



```
--------------------
--------------------
#include<mpi.h>
int main()
{
  --------------------
  --------------------
  --------------------
  --------------------
  MPI_Subroutine()/*subroutine
                   function call*/

  --------------------
  --------------------
  --------------------
  --------------------
 return 0;
}
```

```
//MPI_Subroutine definition
MPI_Subroutine()
{
  --------------------
  --------------------
  --------------------
  --------------------
  --------------------
  return result;
}
```

Figure 2.4: Flow of program with MPI_Subroutine

- Subroutines defined inside MPI can be classified as
- Environmental Subroutines
- Collective Communication Subroutines

- Point to point Communication Subroutines

2.1.2 Environmental Subroutines

These include a group of subroutines, which helps to initialize finalize the MPI execution environment, querying processor's rank and querying the total number of processors in the MPI communication world. The various environmental subroutines are listed below.

MPI_Init

The MPI execution environment is initialized using the MPI_Init subroutine. MPI should be initialized only once and should be called before any other subroutine. Subsequent calls to this routine is erroneous. Users can select the number of processors for executing the task through command line arguments.The syntax is int MPI_Init(int *argc, char **argv). The input parameters are 'argc'- pointer to the number of arguments and 'argv'- pointer to the argument vector.

MPI_COMM_size

The MPI_COMM_size subroutine determines the size of the group associated with a communicator. The syntax is int MPI_Comm_size(MPI_Comm comm, int *size), where MPI_Comm is a MPI object, comm represents communicator and size represents the number of processors in the group of comm.

MPI_Comm_rank

MPI_Comm_rank subroutine is used to distinguish each processor of the communicator; an ID is assigned to each processor and is called rank of the processor. One processor communicate explicitly to another processor using rank as its ID and the syntax is MPI_Comm_rank(MPI_Comm comm, int *rank)

MPI_Finalize

MPI_Finalize subroutine will terminate the MPI execution environment. No other MPI call can be made after calling MPI_Finalize()

2.1.3 Collective Communication Subroutines

For executing jobs in parallel, it is required to communicate with different processors. The subroutines used to perform collective communication are listed below.

MPI_Reduce

This subroutine performs a global reduction operation across all the members of a group, and brings the result to the master node.Figure2 shows schematic representation of process behind MPI_Reduce subroutine. In this example values 10, 40, 20 and 30 are fetched from each processor for performing a specific operation and results returned to the output buffer of the processor 0 or root node. That is, when MPI_Reduce subroutine is called it will combine the inputs provided in the input buffer of each processor in the communicator, using the operation '*', then returns the result after the operation to the output buffer of the root node.

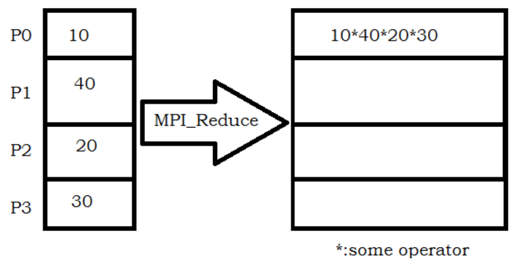


Figure 2.5: Schematic representation of process behind MPI_Reduce subroutine

Syntax-int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

Input Parameters

sendbuf: Address of send buffer (choice).
count :Number of elements in send buffer (integer).
Datatype: Data type of elements of send buffer (handle). op :Reduce operation (handle).
Root: Rank of root process (integer). comm :Communicator (handle).

Output Parameters

recvbuf: Address of receive buffer (choice, significant only at root). Reduce operations can be any of the following. MPI_Reduce support a set of predefined operations, which are listed below

Table 2.2: Possible MPI_Reduce Operations

MPI Name	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

2.14 Point to point Communication Subroutine

It includes the subroutines used to perform point to point communication.

MPI_Send

MPI_Send performs a standard mode block send operation, i.e. these functions do not return value until the communication is finished. The syntax is Int

MPI_Send(void
*sendbuf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm), where MPI_Datatype and MPI_Comm are object.

Input Parameters

sendbuf :Initial address of send buffer (choice)
count: Number of elements in send buffer (nonnegative integer) datatype
:Datatype of each send buffer element (handle)
dest :Rank of destination (integer) tag:
Message tag (integer)
comm : communicator (handle)

MPI_Recv

Performs a blocking receive operation. Receive buffer is a storage for count number of consecutive elements of type specified by data type. Message received must be less than or equal to the length of the receive buffer. The syntax is int MPI_Recv(void *recvbuf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

Input Parameters

Count: Maximum number of elements to receive (integer).
Datatype: Datatype of each receive buffer entry (handle).
source: Rank of source (integer).
tag :Message tag (integer). Comm:
Communicator (handle).

Output Parameters

recvbuf: Initial address of receive buffer (choice). Status:
Status object (status).

The data flow during MPI_Send and MPI_Recv function call is depicted below. When the processor 0 calls an MPI_Send, the data stored in sendbuf(send buffer) is copied into the sysbuf (system buffer). And when processor 1 makes a MPI_Recv function call, data from sysbuf of processor0 will be copied into processor 1's sysbuf. Then from the sysbuf data is copied into recvbuf (receive buffer) of processor 0.

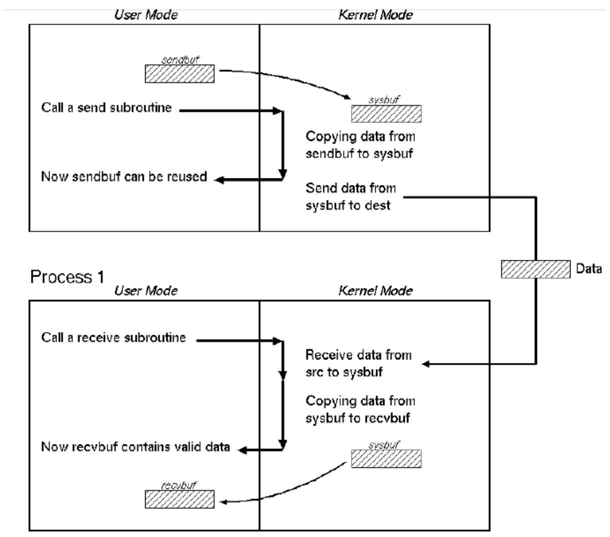

User Mode
Kernel Mode
sendbuf
sysbuf
Call a send subroutine
Copying data from sendbuf to sysbuf
Now sendbuf can be reused
Send data from sysbuf to dest
Data
Process 1
User Mode
Kernel Mode
Call a receive subroutine
Receive data from src to sysbuf
Copying data from sysbuf to recvbuf
Now recvbuf contains valid data
sysbuf
recvbuf

Figure 2.6: Data movement during MPI_Send and MPI_Recv function calls

Now, let us start MPI programming with some toy examples. For better understanding even, an ordinary C program is included before the MPI program. We have included three different examples to brief the concept of MPI programming.

2.1.4 MPI Subroutines for Manipulating Files

MPI provides special functions for handling files such as opening, reading, writing, closing etc.

MPI_File_open
The first thing you always want to do with a file is to open it. The subroutine used in C is MPI_File_open:
The syntax is int MPI_File_open (MPI_Comm comm, char filename, int amode, MPI_Info info, MPI_File *fh);
Input Parameters
Comm: Communicator (handle). filename:
Name of file to open amode: File access mode
info: info object (handle)
Output Parameters
fh: New file handle

The MPI_File_open will open the file using the specified file name in the comm communicator group. The file can be accessed according to the amode specified. There is different access mode which is shown in the table given below.

Table 2.3: Different MPI Access Modes

MODE	DESCRIPTION
MPI_MODE_RDONLY	Read Only
MPI_MODE_RDWR	Read and Write
MPI_MODE_WRONLY	Write Only
MPI_MODE_CREATE	Create the file if it does not exist
MPI_MODE_EXCL	Raise an error if the file already exists
MPI_MODE_DELETE_ON_CLOSE	Delete file on close
MPI_MODE_UNIQUE_OPEN	File will not open concurrently else where
MPI_MODE_SEQUENTIAL	File will only be accessed concurrently
MPI_MODE_APPEND	Set initial position of all file pointers to the end of the file

MPI_File_close This subroutine closes the mentioned file. We must ensure that all requests associated with the file have completed before we close it. The syntax is
int MPI_File_close (MPI_File *fh);
Input Parameters
fh: File handle(handle)
MPI_File_get_size
The subroutine will return the size of the mentioned file.
The syntax is int MPI_File_get_size (MPI_File fh, MPI_Offset size);
Input Parameters
fh: file handle (handle)
Output Parameters
size : size of the file in bytes (nonnegative integer)
size is of type MPI_Offset. MPI_Offset is an integer type of size sufficient to represent the size of the largest file supported by MPI.
MPI_File_read
MPI_File_read reads from the file specified by a file pointer. The count and datatype of the file content should be specified. The data is stored in the buffer specified by the user. The syntax is
int MPI_File_read(MPI_File fh, void *buf, int count, MPI_Datatype datatype, MPI_Status *status)
Input Parameters
fh: File handle (handle)
count : Number of elements in buffer (nonnegative integer)
datatype : Datatype of each buffer element (handle)
Output Parameters
buf: Initial address of buffer
status: Status object (Status)

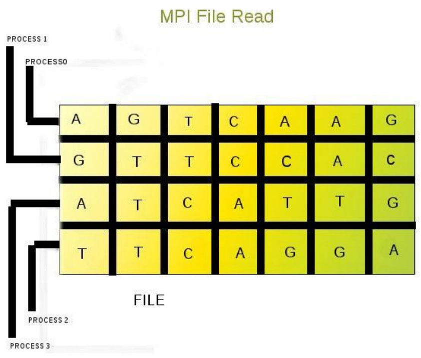


Figure 2.7: Schematic representation of multiple process reading a single file

MPI_File_write

Writes to a file specified by the file pointer . The content to be written should be stored in the buffer. The count and datatype must be specified. The syntax is

int MPI_File_write (MPI_File fh, void *buffer, int count, MPI_Datatype datatype, MPI_Status *status);

Input Parameters

fh :file handle (handle)

buf :initial address of buffer (choice)

count :number of elements in buffer (nonnegative integer) datatype

:datatype of each buffer element (handle)

Output Parameters

status : status object (Status)

If no status required then use MPI_STATUS_IGNORE These two functions, MPI_File_write and MPI_File_read are non-collective, i.e., each process does the reads on its own. Each process can read the file differently and in its own way and time.

MPI_File_read_at_all

Unlike MPI_File_read this will read collectively from file using explicit offset, ie, Read chunk of data from a file from specified start. The syntax is

int MPI_File_read_at_all(MPI_File fh, MPI_Offset offset, void *buf,int count, MPI_Datatype datatype, MPI_Status *status)

Input Parameters

fh: File handle

offset: file offset (nonnegative integer)

count: number of elements in buffer (nonnegative integer)

datatype: datatype of each buffer element (handle)

Output Parameters

status: status object (Status) buf: Initial

address of the buffer

2.1.5 MPI Subroutines for Memory Allocation

For remote memory access and message passing MPI subroutines are used for allocating and freeing memory. The subroutines are

MPI_Alloc_mem

This subroutines is for allocating memory. The syntax is

int MPI_Alloc_mem(MPI_Aint size, MPI_Info info, void *baseptr)

Input Parameters

size: size of memory segment in bytes (non-negative integer) info:
info argument (handle)

Output Parameters

baseptr: pointer to beginning of memory segment allocated

MPI_Free_mem

This subroutine for freeing the memory allocated by using MPI_Alloc_meThe syntax is

int MPI_Free_mem(void *base)

Input Parameters

base: Initial address of memory segment allocated by MPI_Alloc_mem (choice).

Program 2.1 Example for MPI_Alloc_mem and MPI_Free_mem

```
#include <stdio.h>
int main(int argc,char **argv)
{
        int     (* arr)[50][50];
        MPI_Alloc_mem(sizeof(int)*50*50, MPI_INFO_NULL, &arr);
        (*arr)[5][3] = 2;
        printf("%d",arr[5][3]);
        MPI_Free_mem(arr);

}
```

CHAPTER 3

PARALLEL PROGRAMMING TUTORIAL USING MPI[2]

Installation

MPI enabled parallel computing environment can be installed on our personal computers or laptops with Linux operating system using the following command on the terminal.

```
>sudo apt-get install libcr-dev mpich2
mpich2-doc
```

If you are using your personal computers or laptops, then the number of processors are limited to either 2 or 4 based on dual or quad core processors.

The experiments can even be run over a Rocks cluster (Rocks version 6.1 (Mamba) with Cent OS 6.3-64 bit version) which is an implementation of "Beowulf" cluster, running Sun Grid scheduler for job submissions.

For better understanding the concepts of parallel programming, ordinary C program is included before the MPI Programming.

Exercise 1: A trivial example for printing

Case 1: Using C

Program 3.1 A trivial example for printing using C

```
#include <stdio.h>
int main(int argc,char **argv)
{
        printf("Namasthe Keralam!\n");
}
```

The command line syntax for the compilation of C program

```
$gcc namste.c
$./a.out
```

The output is listed below.

```
Namaste Keralam!
```

Case 2: Using MPI

Program 3.2 A trivial example for printing using MPI

```
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv)
{
        int myid,numprocs,i;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD   ,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD   ,&myid);
        printf("Namasthe Keralam...         %d of %d\n",myid,numprocs);
```

[2] Programs are developed in association with the project students Arun P.R, Manu K. Madhu and Jojo George during their tenure at the Department of Computational Biology & Bioinformatics

```
        MPI_Finalize();
}
```

In the above program 'myid' denotes the identity of processor and 'numprocs' denotes the number of processors required for distributing the given task. The command line syntax for the compilation of MPI program and the output is listed. In the above example, number of process defined is 4.Hence, four different copies of the program is generated and will be distributed to the 4 processors for executing the jobs in parallel.

```
$mpicc namaste.c
$mpirun -np 4 ./a.out
Namaste  Keralam!  from  processor  0  of  4
Namaste  Keralam!  from  processor  1  of  4
Namaste  Keralam!  from  processor  2  of  4
Namaste Keralam! from processor 3 of 4
```

Now, consider another program of printing odd numbers from 1 to 100.

Exercise 2: Print the odd numbers from 1 to 100

Case 1: Using C

Program 3.3 Print the odd numbers from 1 to 100 using C

```
#include <stdio.h>
main()
{
        inti;
        i=1;
        while(i<100)
        {
                if(i%2==1){
                printf("%d\n",i);
        }
      i++;
      }
}
```

On Compilation using gcc odd.c will print all the odd numbers between 1 to 100. So let us try to parallelize the operations for better utilization of the processor time by distributing the task among 10 different processors.

Case 2: Using MPI

Program 3.4 Print the odd numbers from 1 to 100 using MPI

```
#include<stdio.h>
#include<mpi.h>
int main(int argc, char **argv)
{
        int myid,numprocs,i;
        MPI_INIT(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD  ,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD  ,&myid);
        for(i=myid*10+1;i<=myid*10+10;i++)
        {
            if(i%2==1)
                printf("\n Odd numbers from %d to %d ",myid+1,i);

        }
        MPI_Finalize();

}
```

On compilation using
$mpicc odd_mpi.c
$mpirun –np 10 ./a.out will distribute the task of computing odd numbers from 1 to 10 on processor 1 followed by 11 to 20 on processor 2 till 91 to 100 on processor 3. The processors will execute the task concurrently and will display the result.
As a next case consider the program of printing square of 10 numbers.

Exercise 3: Print the square of 10 numbers

Case 1: Using C

Program 3.5 Print the square of 10 numbers using C

```
#include<stdio.h>
main()
{
    int a[10], i;
    printf("Enter 10 integers\n");
    for (i=0; i<10; i++)
    {
        scanf("%d", &a[i]);
    }
    for (i=0; i<10; i++)
    {
        printf("square of %d =%d\n", a[i], a[i]*a[i]);
    }
}
```

On compilation, the program asks for inputting 10 different numbers. Later the squares are displayed on the screen.
While performing parallelization, for better utilization of memory collective communication model can be included. In this case, we can collect the input array on the master node and the required input data may be shared among different processor based on the task. MPI_Bcast subroutine can broadcast the data from master node to the different processors. Let us try to parallelize the serial code by distributing the task among 5 different processors.

Program 3.6 Print the square of 10 numbers using MPI

```
#include<stdio.h>
#include<mpi.h>
main (int argc,char **argv)
{
    int a[10],j;
    int unused   attribute   ((unused));
    printf(" Enter 10 integers \n");
    for(j =0;j<10;j++)
    {
        unused = scanf("%d",&a[j]);
    }
    int myid,numprocs ,k;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Bcast(&a,10,MPI_INT,0,MPI_COMM_WORLD);
    for (k= myid*2;k <= myid*2+2;k++){
    printf (" square from %d\n of %d\n is %d\n",myid+1,a[k],a[k]*a[k]) ;
    }
    MPI_Finalize() ;
    return 0;
}
```

On execution with $mpicc squarempi.c
$mpirun –np 5 ./a.out, the first two data will be transmitted to first processor, then next two data to processor 2 till the final two data to processor 5. Finally the squares will be listed out by different processors.

Let us try to further explore the world of MPI with the calculation of factorial of a number.

Exercise 4: Print the factorial of a number n

Case 1: Using C

Program 3.7 Print the factorial of a number n using C

```
#include<stdio.h>
int factorial (int n)
{
    inti;
    int fact=1;
    if (n>1)
    for (i=2;i<=n;i++)
    fact*=i;
    return(fact);
}
```

```
main()
{
        int n;
        printf("enter a number");
        scanf("%d", &n);
        printf("/n Factorial of %d = %d \n",n, factorial(n));
}
```

As factorial involves a continuous multiplication, a separate function may be created tor computing the factorial of number. On execution the input will be passed to the factorial function and in a serial way the multiplication is performed.
Now let us try to automatically distribute the work among different processor using MPI! MPI_Bcast and MP_Reduce subroutines will help for collective communication and synchronization of concurrent task.

Case 2: Using MPI
Program 3.8 Print the factorial of a number n using MPI

```
#include<stdio.h>
#include<mpi.h>
int main(int argc,char **argv)
{
        int myid,numprocs,i,n,lm,j,mod;
        int fact,rslt=1;
        MPI_Init (&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD   ,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD   ,&myid);
        int unused    attribute   ((unused));
        if(myid==0)
        {
                printf(" Enter the number to find the factorial \n ");
                unused = scanf("%d",&n);
        }
        MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
        lm=n/numprocs;
        mod=n%numprocs;
        for(i=myid*lm+1;i<=myid*lm+lm;i++)
        {
         rslt=rslt*i;
        if(mod!=0)
        {
        if(myid==numprocs-1)
        {
        for(j=i;j<=(myid+1)*lm+mod;j++)
        {
        rslt=rslt*j;
        printf(" The multiplication from %d is %d\n",myid+1,rslt);
        }
        }
        }
        MPI_Reduce(&rslt,&fact,1,MPI_INT,MPI_PROD,0,MPI_COMM_WORLD);
        if(myid==0)
        {
        printf("\n The factorial of the Given number %d is %d ",n,fact);
        }
        MPI_Finalize() ;
        return 0;
        }
```

On execution based on the available processor, the number will be divided among different processors and will carry a local multiplication at different processors and the products are accumulated at master node using MPI_Reduce.

Example 5: MPI Program to find sum of marks obtained in 1st 2nd and 3rd years and total marks obtained in degree exam.

Table 3.1: The marks scored by Rinky in her degree exams.

	sub1	sub2	sub3	sub4	sub5
1st year	40	30	50	30	40
2nd year	50	30	50	40	40
3rd year	30	30	40	50	50

Case1 : Using C

Program 3.9 Program to find sum of marks

```
#include <stdio.h>
int main(intargc,char **argv)
{
        int myid, numprocs,i,j,sum[3],tsum=0;
        int degreeMarks[3][5]=
        {
                {40, 30, 50, 30,40} ,           //Degree first year marks
                {50, 30, 50, 40,40} , //Degree second year marks
                {30, 30, 40, 50,50}         //Degree third year marks
        };
        for(i=0;i<3;i++)
        {
                sum[i]=0;
        }
        for(i=0;i<3;i++)
        {
        for(j=0;j<5;j++)
        {
                sum[i]=sum[i]+degreeMarks[i][j];
        }
        printf("Year %d total marks=%d\n",i+1,sum[i]);
        }
        for(i=0;i<3;i++)
        {
                tsum=tsum+sum[i];
        }
        printf("Total marks obtained in Degree Examinations=%d\n",tsum);
}
```

OUTPUT

```
$ gcc degreeMarksN.c
$ ./a.out
Year 1 total marks=190 Year 2
total marks=210 Year 3 total
marks=200
Total marks obtained in Degree Examinations=600
```

Case 2: Using MPI

Program 3.10 Program to find sum of marks using MPI

```
#include<stdio.h>
#include<mpi.h>
int main(int argc,char **argv)
{
    int myid, numprocs,i,sum=0,tsum=0;
    int degreeMarks[3][5]=
    {
    {40,30,50,30,40},
    {50,30,50,40,40},
    {30,30,40,50,50}
    };
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD  ,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD  ,&myid);
    for(i=0;i<5;i++)
    {
    sum=sum+degreeMarks[myid][i];
    }
    printf("\nMARK OF %d YEAR IS %d\n",myid+1,sum);
    MPI_Reduce(&sum,&tsum,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
    if(myid==0){
    printf("\nTOTAL MARKS OBTAINED IN DEGREE EXAM IS %d\n",tsum);
    }
    MPI_Finalize();
}
```

OUTPUT

```
$ mpicc degreeMarks.c
$ mpirun -np 3 ./a.out Year 1
total marks=190 Year 2 total
marks=210 Year 3 total
marks=200
Total marks obtained in Degree Examinations=600
```

Let us focus on MPI_Send and MPI_Recv subroutine used for point to point communication. These subroutines are used to pass a message from one processor to another.

Example 6: Write a MPI program to send the message "Message from Neymar is: Hi Messi, welcome to MPI WORLD " from Processor 0 to processor 1.

Program 3.11 Program for Point to Point Communication

```
#include<stdio.h>
#include<mpi.h>
#define MESSY 0
#define NEYMAR 1
int main(int argc, char *argv[])
{
        int myid,numprocs;
        char *msg;
        MPI_INIT(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD   ,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD   ,&myid);
        MPI_Status status;
        if(myid==MESSY)
        {
                MPI_Recv(&msg,30,MPI_CHAR,1,123,MPI_COMM_WORLD   ,&status);
                printf("\nMessage from NEYMAR IS %s \n",msg);
        }
        if(myid==NEYMAR)
        {
                msg="Hi Messy, Welcome to MPI World";
                MPI_Send(&msg,30,MPI_CHAR,0,123,MPI_COMM_WORLD);
                printf(" \nMessage sent...\n");
        }
  MPI_Finalize();
}
```

OUTPUT

```
$ mpicc Message.c
$ mpirun -np 2 ./a.out
Message from Processor 0 to Processor 1 is: Welcome to the University of Kerala.
```

Program 3.12 Example for MPI_File_read

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

int main (int argc ,char *argv[] )
{
        int numprocs,rank,size,mysize;
        char buffer[100];
        MPI_File fh;
        MPI_Status status;
        MPI_Init(&argc,&argv);
        MPI_Offset filesize;
        MPI_Comm_size(MPI_COMM_WORLD   ,&size);
        MPI_Comm      MPI_File_open(MPI_COMM_WORLD,      "test.txt",      MPI_MODE_RDONLY,
        MPI_INFO_NULL, &fh);
        MPI_File_get_size(fh, &filesize);
        MPI_File_read(fh, buffer, 50, MPI_CHAR,MPI_STATUS_IGNORE);
        printf(" Rank %d \n %s",rank,buffer);
        MPI_File_close(&fh);
        MPI_Finalize() ;
        return 0;

}
```

OUTPUT

```
$ mpicc example.c
$ mpirun -np 2
./a.out Rank 0
Hai how are you
University of
Kerala Rank1
Hai how are you
Universiy of Kerala
```

Program 2.2 Example for MPI_File_write

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

int main (int argc ,char *argv[] )
{
        int numprocs,rank,size,mysize,i;
        int buffer[100];
        MPI_File fh;
        MPI_Status status;
        MPI_Init(&argc,&argv);
        MPI_Offset filesize;
        MPI_Comm_size(MPI_COMM_WORLD   ,&size);
        MPI_Comm_rank(MPI_COMM_WORLD   ,&rank);
        for (i=0; i < 100; i++)
        {
                printf("        Rank %d \n",rank);//Display Rank
                buffer[i] = rank*100+i;//Calculate value to write into file
                printf("%d ",buffer[i]);//Display buffer value
        }
        //Open file in write mode
        MPI_File_open(MPI_COMM_WORLD,"test1.txt",MPI_MODE_CREATE |
MPI_MODE_WRONLY, MP
        //File write
        MPI_File_write(fh, buffer, 100, MPI_INT, MPI_STATUS_IGNORE);
        printf(" Rank %d has wrote the data\n",rank);
        MPI_File_close(&fh);
        MPI_Finalize() ;
        return 0;


}

/*The file is written as binary file. So in order to view the
file content use command
>hexdump -v -e '7/4 "%10d "' -e '"\n"' <file path>
Example
>hexdump -v -e '7/4 "%10d "' -e '"\n"' //home/user/Desktop/TEST/test1.txt
*/
```

Bonus Exercise

1. Write a program to print all the prime numbers between 1 and 100 using message passing interface library.
2. Write a program to check the palindromes of a given sentence using MPI

CHAPTER 4
PARALLEL PROGRAMMING IN BIOINFORMATICS

With the advent of new biology, a big data is emerging in Tera byte range from next sequencing machine. The ever growing troves of information in turn urges for parallel programming in many computational task in bioinformatics. Ordinary desktop computers are lacking the power of handling big biological dataset. The scientist has to mine the large data for patterns that may advance the understanding of modern medicine. Many computationally intensive problems in computational biology like BLAST and phylogenetic analysis have already been well adapted to high performance computing (Albert Y. Zomaya, 2006). The execution on parallel architecture drastically improves the speed. Above all, the need for distributing tasks over many computer processors is important for performing analysis on large datasets. One of the greatest challenges faced by the bioinformatics community is to adopt to the new parallel programming languages for concurrent processing. The main objective of the section is to bridge the gap and make the user feel confident of adapting to the parallel programming environment in Bioinformatics using message passing interface. Initially parallel programming concepts in Bioinformatics are explained using a few toy examples which is further followed by overview of a few existing parallel programming tools in Bioinformatics.

4.1 Toy examples

Bioinformatics is the science of telling story of life using a set of sequences. Sequence analysis is one of the major research areas in the field of computational biology and Bioinformatics. Let us start Parallel programming in Bioinformatics by reading and printing a simple fasta file.

Exercise 1: Reading and printing a simple fasta file using C

Program 4.1 Program for reading and printing simple fasta file

```
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int main()
{
    char ch, file_name[50];
    FILE *fp;
    printf("Enter the name of file you wish to see\n");
    gets(file_name);
    fp = fopen(file_name,"r"); //file open                read mode
    if( fp == NULL )
    {
        perror("Error while opening the file.\n");
        exit(EXIT_FAILURE);


    }
    printf("The contents of %s file are :\n", file_name);
    ch=getc(fp);
    /* Printing The contents of the file */
    while(ch != EOF)
```

```
        {
        printf("%c",ch);
        ch=getc(fp);
        }
}
```

Assume that we have a small fasta file. On compiling gcc seqprint.c will provide an output in a sequential order as shown below.
The contents of tst.fa file are: AAGTCTC
AAGGTTTTCC AACCCCTTTT

Exercise 1: Reading and printing a simple fasta file using MPI

Program 4.2 Program for Reading and printing a simple fasta file using MPI

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

int main (int argc ,char *argv[] )
{
        intnumprocs,rank,i=0,j=0,k,size,mysize,start,end;
        char ch,contents[100][300];
        int ierr;
        char *chunk;
        /*file handle*/
        MPI_File in;
        MPI_Offset filesize,globalstart,globalend;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD ,&size);
        MPI_Comm_rank(MPI_COMM_WORLD ,&rank);
        /*Open file in read mode*/
        MPI_File_open(MPI_COMM_WORLD,"sss.fa",
        MPI_MODE_RDONLY, MPI_INFO_NULL, &in);
        /*No: of characters in file including 'EOF' and '\n'*/
        MPI_File_get_size(in,&filesize);
        printf("\n File Size is %lld", filesize);
        filesize--;                /* get rid of EOF */

         /*Divide the filesize into equal sizes so that
         each process can read this much
         mysize = filesize/size;
         printf("\n My size.. %d\n",mysize);
         /*Read should start at this position*/
         globalstart = rank * mysize;
         /*Read should end at this position*/
         globalend            = globalstart + mysize -
         /*If last rank then read should end at the   file character*/
         if(rank==size-1) globalend = filesize-1;
         /* Buffer to store the data*/
         chunk = malloc((mysize + 1)*sizeof(char));
         /*Read mysize characters
         from the file except last process*/
         MPI_File_read_at_all(in, globalstart,
         mysize, MPI_CHAR,
         chunk[mysize] = '\0';//Get rid of '\n'
         printf("\nProcess %d \n",rank);
         printf("%s",chunk);
         MPI  File  close(&in);
         MPI  Finalize() ;
         return 0;
}
```

On compilation mpicc seqprintpara.c and running mpirun -np 3 ./a.out

Exercise 2: Printing dimer frequency pattern and count using C

Program 4.3. Program for printing dimer frequency pattern and count using C.

```
/* This program get input from a file the file contain the characters of DNA
sequence A,C,G,T and find diffrent combinations of that
  * characters like AA, CC,AT,AG .... etc... and count the no.of patterns in
  that sequence. This program done in serialy */

#include<stdio.h>
#include<string.h>
#include<stdlib.h>

int main(int argc,char **argv)
{
intmyid,numprocs,i,j,l,count,fl=0,p,x,y,a,b=0,cm,lm,cl=0,m,r,s;
char     names[25];
charsr[6]="",temp[25]="",test[25],ch,file_name[25],chr,saq[25][25];
FILE *fp;
/* the structure created for sequence it contain the sequence, patterns
in that sequence no.of patterns in the sequence*/
typedef struct pat
{
char seq[25];
char pt[2];

int c;
};
    printf("Enter the name of file you wish to see\n");
      gets(file_name);
      fp = fopen(file_name,"r"); // read mode
      if( fp == NULL )
      {
           perror("Error while opening the file.\n");
           exit(EXIT_FAILURE);
      }
      printf("The contents of %s file are :\n", file_name);
    ch=getc(fp);
i=0;
j=0;
/* Printing The contents of the file */

while(ch != EOF)
{
printf("%c",ch);
saq[i][j]=ch;
j++;
if (ch == '\n')
{
```

```
cl=cl+1;
i=i++;
j=0;
}
ch=getc(fp);
}


/* Creating structure variable for each line for example in the file contain 2 line t

struct pat pa[cl][10];
for(i=0;i<cl;i++)
{
strcpy(pa[i][0].seq,saq[i]);
}
fclose(fp);
l=strlen(names);
p=l;
printf("%d\n",cl);
/* finding the pattern and counting the no.of patter in that sequence */
for(i=0;i<cl;i++)
{
/* Codes for finding the patterns in the sequence */
strcpy(test,pa[i][0].seq);
l=strlen(test);
m=0;
 j=0;
 sr[0]=test[0];
 sr[1]=test[1];
 strcpy(pa[i][j].pt,sr);
 y=0;
 while(m<l)
 {
 sr[0]=test[m];
 sr[1]=test[m+1];
 fl=strcmp(pa[i][j].pt,sr);
 y=0;
 while(y<=j)
 {
 a=strcmp(pa[i][y].pt,sr);
 if(a==0)
 {
 b++;
 }
 y++;
 }
 if ((fl != 0) && (b == 0))
```

```
{
j++;
strcpy(pa[i][j].pt,sr);
}
m=m+2;
b=0;
}

/* Codes for counting the no.of patterns in that sequence */
 for(fl=0;fl<=j;fl++)
{
strcpy(names,test);
strcpy(sr,pa[i][fl].pt);
for(r=0;r<l/2;r++)
{
cm=strncmp(sr,names,2);
if(cm==0)
{
count=count+1;
}
strcpy(temp,"");
for(x=0,y=2;y<l;x++,y++)
{
temp[x]=names[y];
}
strcpy(names,temp);
}
pa[i][fl].c=count;
count=0;
}
/* Printing the sequence,patterns and no.of patterns in that sequence */
printf(" The no of sequences in the given line %s is \n",pa[i][0].seq);
for(s=0;s<j;s++)
{
printf(" %s                ",pa[i][s].pt);
printf("%d\n",pa[i][s].c);
}}}
```

On Compilation and running the program will serially computer all the dimers will display the result as shown below

```
Enter the name of file you wish to see
tst.fa The contents of tst.fa file are :
AGTCTC
AAGGTTTTCC
AACCCCTTTT
The no of sequences in the given line
```

```
AGTCTC is AG   1
TC             2
The no of sequences in the given line AAGGTTTTCC is
        AA          1
        GG          1
        TT      2
        CC          1
The no of sequences in the given line
AACCCCTTTT is AA        1
```

Exercise 2: Printing Dimer frequency pattern and count using MPI

Program 4.4 Program for printing dimer frequency pattern and count using MPI.

```c
        /* This program get input from a file the file contain the
        characters of DNA sequence A,C,G,T and find different combinations of that
        *     * characters like AA, CC,AT,AG ... count the no.of patterns in that sequence.
        This program done in parallel */

      #include<stdio.h>
      #include<mpi.h>
      #include<string.h>
      #include<stdlib.h>
      int main(int argc,char **argv)
      {
      intmyid,numprocs,i,j,l,count,fl=0,p,x,y,a,b=0,cm,lm,cl=0,m,r,s;

     char     names[25];
     charsr[6]="",temp[25]="",test[25],ch,file_name[25],chr,saq[25][25];
     FILE *fp;
     /* the structure created for sequence it contain the sequence ,
     patterns in that sequence no.of patterns in the sequence*/
     typedef struct pat
     {
     char seq[25];
     char pt[2];
     int c;
     };
     MPI_Init(&argc,&argv);
     MPI_Comm_size(MPI_COMM_WORLD   ,&numprocs);
     MPI_Comm_rank(MPI_COMM_WORLD   ,&myid);
     MPI_Status status;
     /* The file name is copied to file_name the input data is in tst.fa*/
       strcpy(file_name,"tst.fa");
          fp = fopen(file_name,"r"); //file open                    read mode
          if( fp == NULL )
          {
                 perror("Error while opening the file.\n");
                 exit(EXIT_FAILURE);
          }
          printf("The contents of %s file are :\n", file_name);
        ch=getc(fp);
     i=0;
     j=0;
     /* Printing The contents of the file */
     while(ch != EOF)
     {
     printf("%c",ch);
     saq[i][j]=ch;
     j++;
     if (ch == '\n')
     {
     cl=cl+1;
     i=i++;
     j=0;
     }
     ch=getc(fp);
     }
     /* Creating structure variable for each line for example in the file contain
     2 line then it will create 2 structure variable*/
     struct pat pa[cl][10];
     for(i=0;i<cl;i++)
     {
     strcpy(pa[i][0].seq,saq[i]);
     }
     fclose(fp);

        printf("%d \n",cl);
        /* Each sequence is processed separately in Each processor */
        printf("The Out put from processor %d is", myid+1);
        i=myid;
        /* Codes for finding the patterns in the sequence */
        strcpy(test,pa[i][0].seq);
        l=strlen(test);
        m=0;
        j=0;
        sr[0]=test[0];
        sr[1]=test[1];
        strcpy(pa[i][j].pt,sr);
        y=0;
        while(m<l)
        {
        sr[0]=test[m];
        sr[1]=test[m+1];
        fl=strcmp(pa[i][j].pt,sr);
        y=0;
        while(y<=j)
        {
        a=strcmp(pa[i][y].pt,sr);
        if(a==0)
        {
        b++;
        }
        y++;
        }
        if ((fl != 0) && (b == 0))
        {
        j++;
        strcpy(pa[i][j].pt,sr);
        }
        m=m+2;
        b=0;
        }
        /* Codes for counting the no.of patterns in that sequence */
        for(fl=0;fl<=j;fl++)
        {
        strcpy(names,test);
        strcpy(sr,pa[i][fl].pt);
        for(r=0;r<l/2;r++)
       {
       cm=strncmp(sr,names,2);
       if(cm==0)
       {
       count=count+1;
       }
       strcpy(temp,"");
       for(x=0,y=2;y<l;x++,y++)
       {
       temp[x]=names[y];
       }
       strcpy(names,temp);
       }
       pa[i][fl].c=count;
       count=0;
       }
       /* Printing the sequence,patterns and no.of patterns in that sequence */
       printf(" The no of sequences in the given line %s is \n",pa[i][0].seq);
       for(s=0;s<j;s++)
       {
       printf(" %s                    ",pa[i][s].pt);
       printf("%d\n",pa[i][s].c);
```

```
}
MPI_Finalize();
}
```

On compilation using mpicc dimermpi.c and running mpirin –np 3 ./a.out will produce the following result.

The contents of tst.fa file are :

```
AGTCTC
AAGGTTTTCC AACCCCTTTT
```

```
The Output from processor 1 is
The no of sequences in the given line AGTCTC is
AG           1
TC           2
AGTCTC AAGGTTTTCC
AACCCCTTTT
The Output from processor 2 is
The no of sequences in the given line AAGGTTTTCC is
    AA      1
    GG      1
    TT      2
    CC      1
    AGTCTC
AAGGTTTTCC AACCCCTTTT
The Output from processor 3 is
The no of sequences in the given line AACCCCTTTT is
AA           1
CC           2
TT           2
```

4.2 CLUSTAL- MPI

The huge data emerges out from the Bioinformatics research needs sophisticated algorithmic methodologies for its storage, analysis, and processing. High performance computing become inevitable in biological problems due to large run time and memory requirements. For instance, ClustalW is a tool for aligning multiple protein or nucleotide sequences. The technique of sequence alignment has an integral part in many research avenues of Computational Biology Bioinformatics. Now consider the following English strings BADDATA, BAGDATA, BIGDATA and BIODATA. It's very trivial example to understand the concept of alignment. The sequences are written one below the other to highlight their maximum similarity. All the string has sequence length as 7. As shown in figure, the * indicates similar sequences among the distributed data and a gap indicates the dissimilarities in the sequences.

```
seq1    BADDATA 7
seq2    BAGDATA 7
seq3    BIGDATA 7
seq4    BIODATA 7
        *  ****
```

Figure 4.1: Example of Sequence Similarity using English string

Now considering the case with DNA sequences, CLUSTALW is the software used analyzing the similarities among different sequences. In ClustalW, the sequences are processed serially and the time will progressively increase as length or number of sequences increased and the order of complexity is O(N2). The alignment is achieved via three steps: pairwise alignment, guide-tree generation and progressive alignment. ClustalW-MPI is a distributed and parallel implementation of ClustalW.The source code is available http://www.bii.astar.edu.sg/achievements/applications/clustalw/index.php. The above specified example of English string can even be computer using ClustalW-MPI. As sequence similarities are calculated by distributing the task, in a much faster manner, we can get the aligned output similar to Figure 4.1. ClustalW-MPI reported a scale up of 15.8 using 16 processor on the 500-sequence test data.

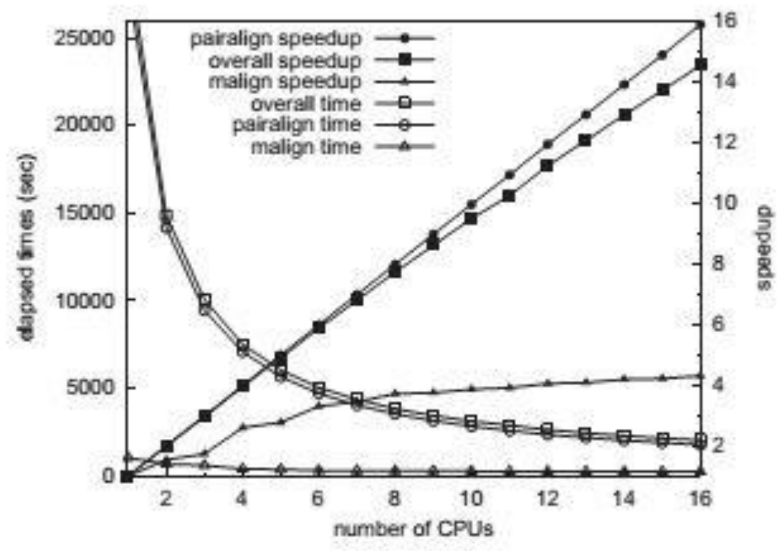
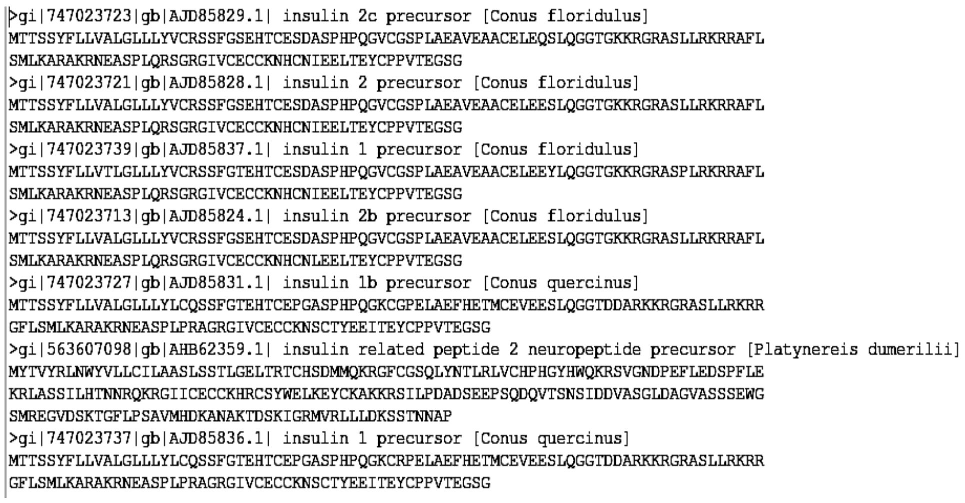


Figure 4.2: Performance improvement of CLUSTAL_MPI

To run the ClustalW-MPI requires a rock cluster with MPICH installed on it. Multiple options are available for running CLUSTALW-MPI To make a full multiple sequence alignment: (using one master node and 4 computing nodes), run the following command.

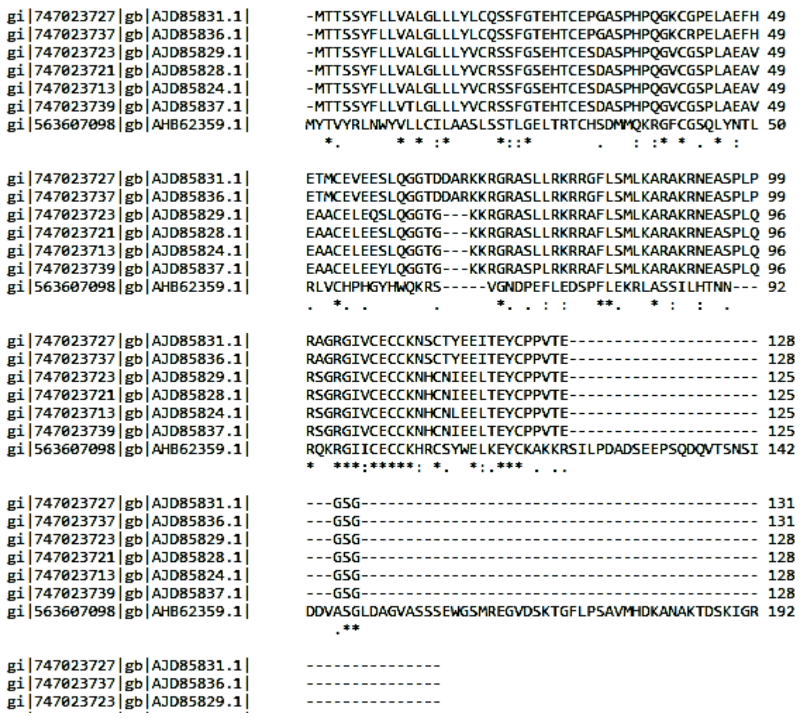
%mpirun -np 5 ./clustalw-mpi -infile=dele.input

%mpirun -np 5 ./clustalw-mpi -infile=CFTR.input Let us try to align a few insulin protein sequences. Consider that we have 7 input sequences in the test input file as shown in Figure 4.3. All the sequences need to be pair wise aligned. If the file is processed serially, pair align time = 0.046 sec. On the other hand distributing the task on 6 different processor will reduce the pair align time as 0.014 sec. The final aligned output is shown in Figure 4.4.



```
>gi|747023723|gb|AJD85829.1| insulin 2c precursor [Conus floridulus]
MTTSSYFLLVALGLLLYVCRSSFGSEHTCESDASPHPQGVCGSPLAEAVEAACELEQSLQGGTGKKRGRASLLRKRRAFL
SMLKARAKRNEASPLQRSGRGIVCECCKNHCNIEELTEYCPPVTEGSG
>gi|747023721|gb|AJD85828.1| insulin 2 precursor [Conus floridulus]
MTTSSYFLLVALGLLLYVCRSSFGSEHTCESDASPHPQGVCGSPLAEAVEAACELEESLQGGTGKKRGRASLLRKRRAFL
SMLKARAKRNEASPLQRSGRGIVCECCKNHCNIEELTEYCPPVTEGSG
>gi|747023739|gb|AJD85837.1| insulin 1 precursor [Conus floridulus]
MTTSSYFLLVTLGLLLYVCRSSFGTEHTCESDASPHPQGVCGSPLAEAVEAACELEEYLQGGTGKKRGRASPLRKRRAFL
SMLKARAKRNEASPLQRSGRGIVCECCKNHCNIEELTEYCPPVTEGSG
>gi|747023713|gb|AJD85824.1| insulin 2b precursor [Conus floridulus]
MTTSSYFLLVALGLLLYVCRSSFGSEHTCESDASPHPQGVCGSPLAEAVEAACELEESLQGGTGKKRGRASLLRKRRAFL
SMLKARAKRNEASPLQRSGRGIVCECCKNHCNLEELTEYCPPVTEGSG
>gi|747023727|gb|AJD85831.1| insulin 1b precursor [Conus quercinus]
MTTSSYFLLVALGLLLYLCQSSFGTEHTCEPGASPHPQGKCGPELAEFHETMCEVEESLQGGTDDARKKRGRASLLRKRR
GFLSMLKARAKRNEASPLPRAGRGIVCECCKNSCTYEEITEYCPPVTEGSG
>gi|563607098|gb|AHB62359.1| insulin related peptide 2 neuropeptide precursor [Platynereis dumerilii]
MYTVYRLNWYVLLCILAASLSSTLGELTRTCHSDMMQKRGFCGSQLYNTLRLVCHPHGYHWQKRSVGNDPEFLEDSPFLE
KRLASSILHTNNRQKRGIICECCKHRCSYWELKEYCKAKKRSILPDADSEEPSQDQVTSNSIDDVASGLDAGVASSSEWG
SMREGVDSKTGFLPSAVMHDKANAKTDSKIGRMVRLLLDKSSTNNAP
>gi|747023737|gb|AJD85836.1| insulin 1 precursor [Conus quercinus]
MTTSSYFLLVALGLLLYLCQSSFGTEHTCEPGASPHPQGKCRPELAEFHETMCEVEESLQGGTDDARKKRGRASLLRKRR
GFLSMLKARAKRNEASPLPRAGRGIVCECCKNSCTYEEITEYCPPVTEGSG
```

Figure 4.3: Sample Input text file for multiple alignment



```
gi|747023727|gb|AJD85831.1|      -MTTSSYFLLVALGLLLYLCQSSFGTEHTCEPGASPHPQGKCGPELAEFH 49
gi|747023737|gb|AJD85836.1|      -MTTSSYFLLVALGLLLYLCQSSFGTEHTCEPGASPHPQGKCRPELAEFH 49
gi|747023723|gb|AJD85829.1|      -MTTSSYFLLVALGLLLYVCRSSFGSEHTCESDASPHPQGVCGSPLAEAV 49
gi|747023721|gb|AJD85828.1|      -MTTSSYFLLVALGLLLYVCRSSFGSEHTCESDASPHPQGVCGSPLAEAV 49
gi|747023713|gb|AJD85824.1|      -MTTSSYFLLVALGLLLYVCRSSFGSEHTCESDASPHPQGVCGSPLAEAV 49
gi|747023739|gb|AJD85837.1|      -MTTSSYFLLVTLGLLLYVCRSSFGTEHTCESDASPHPQGVCGSPLAEAV 49
gi|563607098|gb|AHB62359.1|      MYTVYRLNWYVLLCILAASLSSTLGELTRTCHSDMMQKRGFCGSQLYNTL 50
                                   *.      * * :*     *::*       .   : :* * . * :

gi|747023727|gb|AJD85831.1|      ETMCEVEESLQGGTDDARKKRGRASLLRKRRGFLSMLKARAKRNEASPLP 99
gi|747023737|gb|AJD85836.1|      ETMCEVEESLQGGTDDARKKRGRASLLRKRRGFLSMLKARAKRNEASPLP 99
gi|747023723|gb|AJD85829.1|      EAACELEQSLQGGTG---KKRGRASLLRKRRAFLSMLKARAKRNEASPLQ 96
gi|747023721|gb|AJD85828.1|      EAACELEESLQGGTG---KKRGRASLLRKRRAFLSMLKARAKRNEASPLQ 96
gi|747023713|gb|AJD85824.1|      EAACELEESLQGGTG---KKRGRASLLRKRRAFLSMLKARAKRNEASPLQ 96
gi|747023739|gb|AJD85837.1|      EAACELEEYLQGGTG---KKRGRASPLRKRRAFLSMLKARAKRNEASPLQ 96
gi|563607098|gb|AHB62359.1|      RLVCHPHGYHWQKRS-----VGNDPEFLEDSPFLEKRLASSILHTNN--- 92
                                 .  *. .       .      *. . : :   **.   * :  :  .

gi|747023727|gb|AJD85831.1|      RAGRGIVCECCKNSCTYEEITEYCPPVTE--------------------- 128
gi|747023737|gb|AJD85836.1|      RAGRGIVCECCKNSCTYEEITEYCPPVTE--------------------- 128
gi|747023723|gb|AJD85829.1|      RSGRGIVCECCKNHCNIEELTEYCPPVTE--------------------- 125
gi|747023721|gb|AJD85828.1|      RSGRGIVCECCKNHCNIEELTEYCPPVTE--------------------- 125
gi|747023713|gb|AJD85824.1|      RSGRGIVCECCKNHCNLEELTEYCPPVTE--------------------- 125
gi|747023739|gb|AJD85837.1|      RSGRGIVCECCKNHCNIEELTEYCPPVTE--------------------- 125
gi|563607098|gb|AHB62359.1|      RQKRGIICECCKHRCSYWELKEYCKAKKRSILPDADSEEPSQDQVTSNSI 142
                                 *  ***:*****: *.  *:.*** . ..

gi|747023727|gb|AJD85831.1|      ---GSG-------------------------------------------- 131
gi|747023737|gb|AJD85836.1|      ---GSG-------------------------------------------- 131
gi|747023723|gb|AJD85829.1|      ---GSG-------------------------------------------- 128
gi|747023721|gb|AJD85828.1|      ---GSG-------------------------------------------- 128
gi|747023713|gb|AJD85824.1|      ---GSG-------------------------------------------- 128
gi|747023739|gb|AJD85837.1|      ---GSG-------------------------------------------- 128
gi|563607098|gb|AHB62359.1|      DDVASGLDAGVASSSEWGSMREGVDSKTGFLPSAVMHDKANAKTDSKIGR 192
                                    .**

gi|747023727|gb|AJD85831.1|      ---------------
gi|747023737|gb|AJD85836.1|      ---------------
gi|747023723|gb|AJD85829.1|      ---------------
```

Figure 4.4: Output alignment file from CLUSTAL-MPI

4.3 COMRAD- MPI [3]

The big data storage challenges in Bioinformatics emphasize the need for High performance computing solutions for managing large genomic data. The genomic compression helps to reduce the on disk “foot print” of large data volumes of sequences. Therefore, it is of interest to describe a parallel-computing approach using message-passing library for distributing the different compression stages in clusters. Though the distribution of various tasks or genome data over many different computers is difficult, genomic revolution trends demand for high performance computing solutions for data storage and management. Compression algorithm for large dataset requires a vast processing power and memory, which is rather difficult to process on desktop computers. COMRAD is a sequential iterative algorithm designed for compressing DNA sequence.

The experimental analysis of COMRAD with DNA data set in gigabytes range demands for a long run time. For processing the malus domestica genome, 7% of time is spending for codebook creation, substitution, clean up and encoding stages. In the current study, our objective is to reduce the computational time by parallelizing the COMRAD algorithm. As a first step in this direction COMRAD MPI introduce data parallelism by dividing equally the whole genome into n batches and each batch is processed simultaneously by a processor in the cluster computer. Further the parallelization of substitution, clean up and encoding stages were also incorporated. Figure 4.5 shows the flow chart of COMRAD MPI.

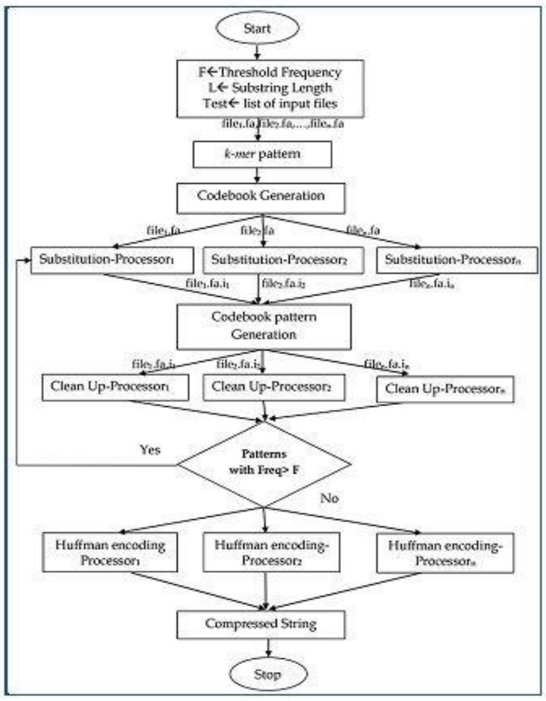


Figure 4.5: Flow Chart of Compression of Large Genome Dataset using COMRAD-MPI

[3] Earlier version of this chapter is published in Christopher Leela Biji,Manu K Madhu, Vineetha Vishnu, Satheesh Kumar K, Vijayakumar and Achuthsankar S Nair, "Compression of Large genomic datasets using COMRAD on Parallel Computing Platform", Bioinformation. 2015; 11(5): 267–271.

The source codes are available. The experiments can be run over Rocks cluster or on Ubuntu system with MPI environment installed.
To compress using COMRAD-MPI:

1. Split the input genomes into chunks of equal size using split -n 6 filename.fa
2. Copy the names of all the files that need to be compressed into a file
3. Run the command

```
./comrad.sh <file of file names>
```

```
Usage:
comrad.sh [OPTIONS] FILE comrad.sh
[OPTIONS] FILE
-n: No:of processors in the cluster
-f: Frequency threshold (default 4)
-l: Initial substring length (default 8)
-o: Output directory (default /tmp/comrad)
FILE: File name containing files to be compressed (include full path names for each file)
```

eg :Compression of multiples files using two processors

```
./comrad.sh -n 2 test
```

To decompress using COMRAD: 1.Run the command

```
./decomrad.sh <file of file names>
```

The performance of Genome Compression using parallel computing tool can be analyzed based on the Compression run time (Sec), Compression in bits per base (bpb) and Speedup ratio (S). Compression in bits per base and Speedup ratio is defined as

$$CB=\frac{\text{Compressed size in bits}}{\text{Total No:of bases in the sequence}}$$

$$\text{Speedup ratio}=\frac{\text{Serial Compression time}}{\text{Parallel Compression time}}$$

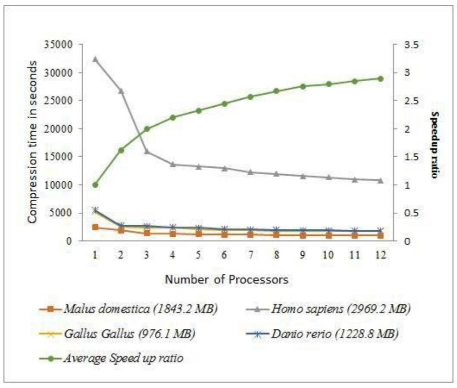


Figure 4.6: Speedup ratio and Compression time improvement with number of processors

Figure 4.6 shows the elapsed wall time improvement and speedup ratio for Homospaiens (Mammals), Malus-domestica (Plants), Gallus gallus (Bird) and Danio rerio (Fishes) in COMRAD-MPI. Experiment is repeated after splitting the whole genome equally between different processors on the cluster from n=2 to 12. As the number of processors is increased, the elapsed wall time is reduced. The sequential COMRAD require 8 hours, 91 minutes, 86 minutes and 41 minutes to effectively compress the homospaiens, danio rerio, gallus gallus and malus domestica genome but the implemented COMRAD-MPI could effectively compress it in just 3 hours, 30 minutes, 29 minutes and 15 minutes. While adding more dataset, redundancy with in the dataset is increased and COMRAD -MPI was able to compress multiple files relatively faster than COMRAD while maintaining the same compression.

References

1. Michael J. Quinn, Parallel Programming in C with Mpi and Openmp, 2004

2. Achuthsankar S.Nair, T. Mahalekshmi, "Data Structures in C", PHI Learning Limited, 2009

3. Manu K. Madhu, Biji C.L., "Parallel Computing with Message Passing Interface", Computer Society of India, volume No.38 (5), August 2014, pp 25-26.

4. Manu K. Madhu, Biji C. L.," Parallel Computing with Message Passing Interface-Part –II", Computer Society of India, Volume No:38(6),September 2014, pp 32-36

5. Albert Y. Zomaya. (2006) Parallel Computing for Bioinformatics and Computational Biology: Models, Enabling Technologies, and Case Studies. 1st ed. John wiley sons.

6. Li, Kuo-Bin. "ClustalW-MPI: ClustalW analysis using distributed and parallel computing." Bioinformatics, 2003, 19(12), 1585-1586.

7. Christopher Leela Biji, Manu K Madhu, Vineetha Vishnu, Satheesh Kumar K, Vijayakumar and Achuthsankar S Nair, "Compression of Large genomic datasets using COMRAD on Parallel Computing Platform", Bio information. 2015; 11(5): 267–271.