Summer Internship, (2022)

Message Parsing Interface and it's Application in High Performance Computings

Siddharth Sahu
Shiv Nadar University

Guide: (Mr. Abhishek Kumar)
IUAC (Inter University Accelerator Centre)

Certificate

This is to certify that the internship project report Message Parsing Interface and it's Application in High Performance Computing is the bona fide work of Siddharth Sahu, B. Tech. in Computer Science and Engineering of Shiv Nadar Institute of Eminence, Greater Noida carried out under my supervision.

Project Mentor:

Mr.Abhishek Kumar (Scientist - 'C')

Abstract

The project involves the usage of MPI library to write efficient code in C and Python for it's use in High Performance Computing. It also compares the timing and efficiency of the codes written in C and Python.

Acknowledgements

Firstly, I would like to thank my supervisor and mentor, **Mr.Abhishek Kumar (Scientist - 'C')** for his guidance, help and the many discussion sessions that gave me courage and enthusiasm to design and build this application.
This project would not have been possible without the help of many people. Here-after, I would like to thank them for their help, support, and/or advice they have given us while working on the project.

Contents

Chapter 1

Introduction to HPC (High Performance Computing)

It is the use of parallel processing for running advanced application programs efficiently and quickly. The term applies specially for a system that function above a Tera Flops 10^{12} (Floating Point Operations Per Second).
The term High-performance computing is occasionally used as a synonym for supercomputing. Although technically a supercomputer is a system that performs at or near currently highest operational rate for computers. Some supercomputers work at more than a Peta Flops (10^{15} floating point operations per second). The most common HPC system all scientific engineers & academic institutions. Some Government agencies particularly military are also relying on APC for complex applications.

1.1 Importance of High performance Computing

1. It is used for scientific discoveries, game-changing innovations, and to improve quality of life.

2. It is a foundation for scientific & industrial advancements.

3. It is used in technologies like IoT, AI, 3D imaging evolves & amount of data that is used by organization is increasing exponentially to increase ability of a computer, we use High-performance computer.

4. HPC is used to solve complex modeling problems in a spectrum of disciplines. It includes AI, Nuclear Physics, Climate Modelling, etc.

5. HPC is applied to business uses as well as data warehouses & transaction processing.

1.2 How does HPC work?

HPC solutions have three main components:

1. Compute

2. Network

3. Storage

To build a high performance computing architecture, compute servers are networked together into a cluster. Software programs and algorithms are run simultaneously on the servers in the cluster. The cluster is networked to the data storage to capture the output. Together, these components operate seamlessly to complete a diverse set of tasks.
To operate at maximum performance, each component must keep pace with the others. For example, the storage component must be able to feed and ingest data to and from the compute servers as quickly as it is processed. Likewise, the networking components must be able to support the high-speed transportation of data between compute servers and the data storage. If one component cannot keep up with the rest, the performance of the entire HPC infrastructure suffers.

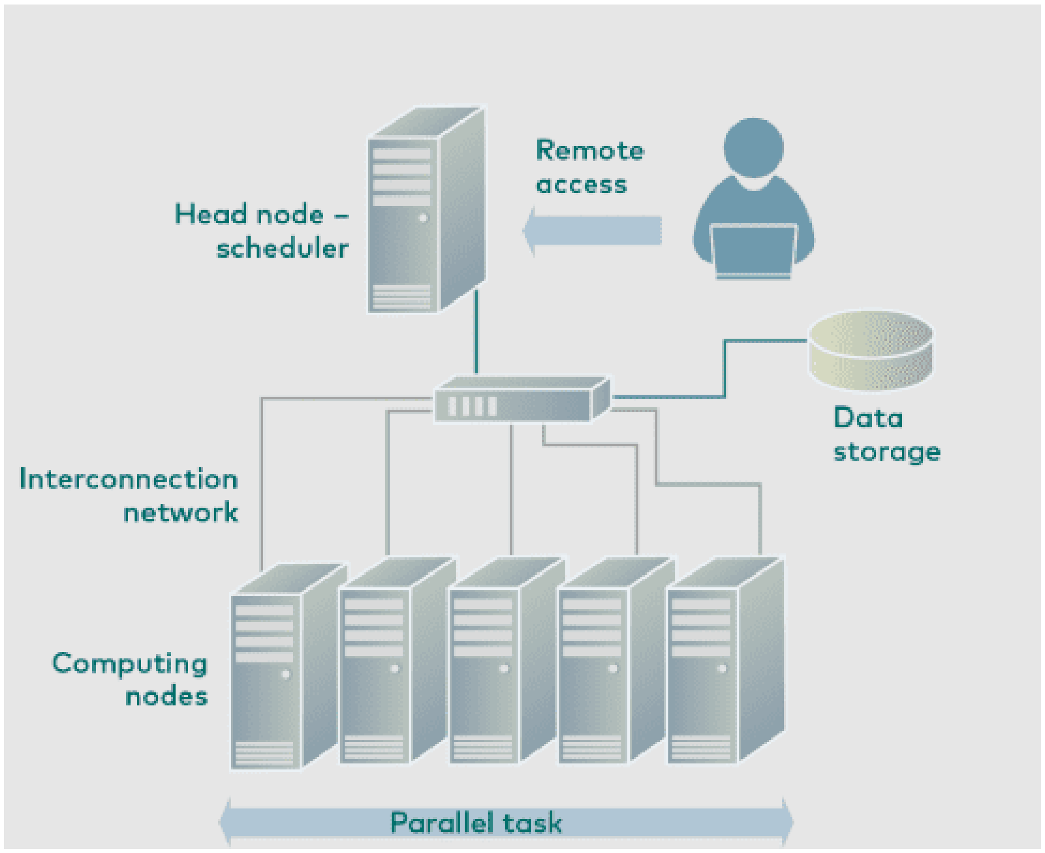


Figure 1.1: Working of HPC

Chapter 2

MPI (Message Parsing Interface)

2.1 Introduction

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed to function on parallel computing architectures.[1] The MPI standard defines the syntax and semantics of library routines that are useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran. There are several open-source MPI implementations, which fostered the development of a parallel software industry, and encouraged development of portable and scalable large-scale parallel applications.

2.2 Reasons of using MPI

- **Standardization** - MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has re-placed all previous message passing libraries.
- **Portability** - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.
- **Performance Opportunities** - Vendor implementations should be

able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.

- **Functionality** - There are over 430 routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.
- **Availability** - A variety of implementations are available, both vendor and public domain.

2.3 Installation of MPI on Linux Systems

The following instructions for installation of MPI was done on a system running Ubuntu(18.04)

For using MPI in C Program -
The following commands must be executed in the Linux Terminal

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

For Compiling the program, change the directory in the terminal where the C file is present using cd command and run the following command in terminal

```
mpicc -o hello_world hello_world.c
```

For running the program, again use the command below to run it

```
mpirun -np 4 ./hello_world
```

For using MPI in Python Program -
The following commands must be executed in the Linux Terminal

1.
```
sudo apt-get install python3
```
2.
```
sudo apt install python3-pip
```

3. `sudo pip3 install mpi4py`

For Python, we don't need to compile the program we can directly run it by the command below

```
mpiexec -n 4 python3 hello_world.py
```

2.4 Hello World program using MPI in C and Python

For C Program -

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv) {
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    // Print off a hello world message
    printf("Hi form rank %d \n",world_rank);
    // Finalize the MPI environment.
    MPI_Finalize();
return 0;
}
```

For Python program -

```python
from mpi4py import MPI
import sys

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

sys.stdout.write(
    "Hello, World! I am process %d of %d on %s.\n"
    % (rank, size, name))
```

Chapter 3

MPI Fucntions

1. **MPI_Send**
   MPI_Send is the standard send in MPI. Behind the scenes, it will issue a buffered send MPI_Bsend or a synchronous send MPI_Ssend. This decision will be based on whether the buffer attached for buffered sends contains enough free space for the message to send. If there is enough space, the buffered send MPI_Bsend will be issued, otherwise it will revert to a synchronous send MPI_Ssend. MPI implementations may provide a buffer by default, therefore not having explicitly assigned a buffer for buffered send does not guarantee that an MPI_Send will issue an MPI_Ssend.

```
MPI_Send(&token, 1, MPI_INT, (world_rank + 1) %
    world_size,0, MPI_COMM_WORLD);
```

Listing 3.1: Code snippet from ring.c

```
comm.send(shared,dest=(rank+1)%size)
```

Listing 3.2: Code snippet from ring.py

2. **MPI_Recv**
   MPI_Recv receives a message in a blocking fashion: it will block until completion, which is reached when the incoming message is copied to the buffer given.

```
if (world_rank != 0) {
     MPI_Recv(&token, 1, MPI_INT, world_rank-1, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\
  n", world_rank, token, world_rank - 1);
```

Listing 3.3: Code snippet from ring.c

```
data=comm.recv(source=(rank-1)%size)
```

Listing 3.4: Code snippet from ring.py

3. **MPI_Bcast**
   MPI_Bcast broadcasts a message from a process to all other processes in the same communicator. This is a collective operation; it must be called by all processes in the communicator.

```
    MPI_Bcast(b, N*N, MPI_INT, 0, MPI_COMM_WORLD);
```

Listing 3.5: Code snippet from matrixmultMPI.c

```
b = comm.bcast(b, root=0)
```

Listing 3.6: Code snippet from matrixmultMPI.py

4. **MPI_Gather**
   MPI_Gather collects data from all processes in a given communicator and concatenates them in the given buffer on the specified process. The concatenation order follows that of the ranks. This is a collective operation; all processes in the communicator must invoke this routine.

```
    MPI_Gather(cc, N*N/size, MPI_INT, c, N*N/size,
  MPI_INT, 0, MPI_COMM_WORLD);
```

Listing 3.7: Code snippet from matrixmultMPI.c

```
    data = comm.gather(split, root=0)
```

Listing 3.8: Code snippet from matrixmultMPI.py

5. **MPI_Scatter**
   MPI_Scatter dispatches data from a process across all processes in the same communicator. As a blocking operation, the buffer passed can be safely reused as soon as the routine returns. Also, MPI_Scatter is a collective operation; all processes in the communicator must invoke this routine.

```
    MPI_Scatter(a, N*N/size, MPI_INT, aa, N*N/size,
   MPI_INT,0,MPI_COMM_WORLD);
```

Listing 3.9: Code snippet from matrixmultMPI.c

```
    split = comm.scatter(split, root=0)
```

Listing 3.10: Code snippet from matrixmultMPI.py

6. **MPI_Reduce**
   MPI_Reduce is the means by which MPI process can apply a reduction calculation. The values sent by the MPI processes will be combined using the reduction operation given and the result will be stored on the MPI process specified as root. MPI_Reduce is a collective operation; it must be called by every MPI process in the communicator given.

```
totaltime = comm.reduce(duration,op = MPI.SUM, root = 0)
```

Listing 3.11: Code snippet from matrixmultMPI.py

```
    MPI_Reduce(&duration,&global,1,MPI_DOUBLE,MPI_SUM,0,
   MPI_COMM_WORLD);
```

Listing 3.12: Code snippet from matrixmultMPI.c

Chapter 4

Performance Analysis between C and Python in MPI

I compared and analysed codes using the MPI library to find out the performance difference between C and Python programs.

4.1 Integration

Finding out the total integration under the curve for a fixed function with limits 0 to 2.

$$f(x) = \int_{0}^{1} x^2$$

I have used the Trapezoidal Rule for calculating the same. It is a Numerical technique to find the definite integral of a function.The function is divided into many sub-intervals and each interval is approximated by a Trapezium. Then the area of trapeziums is calculated to find the integral which is basically the area under the curve. The more is the number of trapeziums used, the better is the approximation.
Algorithm for calculating the integration under the curve is as follows

```
    h = (b-a)/n;

    local_n = n/size;

    local_a = a + my_rank * local_n * h;

```

```c
    local_b = (local_a + local_n) * h;

    integral = Trap(local_a, local_b, local_n, h);

    if (my_rank == 0){
        total = integral;
        for (source = 1; source < size; source++){
            MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
  MPI_COMM_WORLD, &status);
            total += integral;
        }
    }
    else {
        MPI_Send(&integral, 1, MPI_FLOAT, dest, tag,
  MPI_COMM_WORLD);
    }
```

Listing 4.1: Code snippet from IntegrationMPI.c

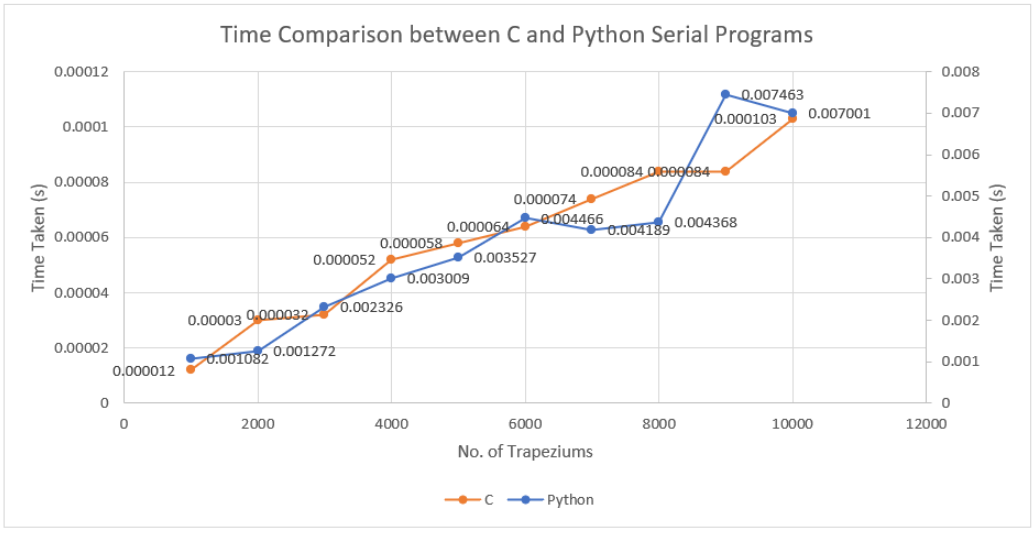
4.1.1 For Serial Program in C and Python

No. of Trapeziums	Time Taken (s)	Accuracy
1000	0.001082	2.66666867
2000	0.001272	2.66666743
3000	0.002326	2.66666681
4000	0.003009	2.66666675
5000	0.003527	2.66666672
6000	0.004466	2.66666670
7000	0.004189	2.66666669
8000	0.004368	2.66666669
9000	0.007463	2.66666669
10000	0.007001	2.66666669

Data for Python Code

No. of Trapeziums	Time Taken (s)	Accuracy
1000	0.000012	2.66662788
2000	0.000030	2.66671276
3000	0.000032	2.66658711
4000	0.000052	2.66660619
5000	0.000058	2.66650033
6000	0.000064	2.66644287
7000	0.000074	2.66667461
8000	0.000084	2.66645622
9000	0.000084	2.66647649
10000	0.000103	2.66674614

Data for C Code



Comparison between C and Python serial programs

4.1.2 For MPI Program in C and Python

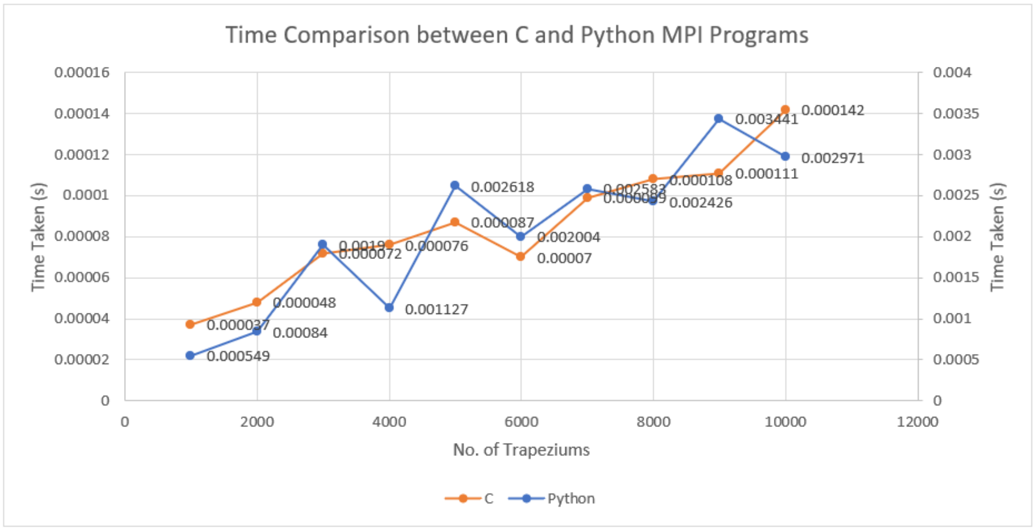
For MPI, all the computations was divided and allocated to 4 processors.

No. of Trapeziums	Time Taken (s)	Accuracy
1000	0.000549	2.66017401
2000	0.000840	2.66341850
3000	0.001900	2.66450082
4000	0.001127	2.66504213
5000	0.002618	2.66536696
6000	0.002004	2.66558354
7000	0.002583	2.66573824
8000	0.002426	2.66585428
9000	0.003441	2.66594453
10000	0.002971	2.66601674

Data for Python Code

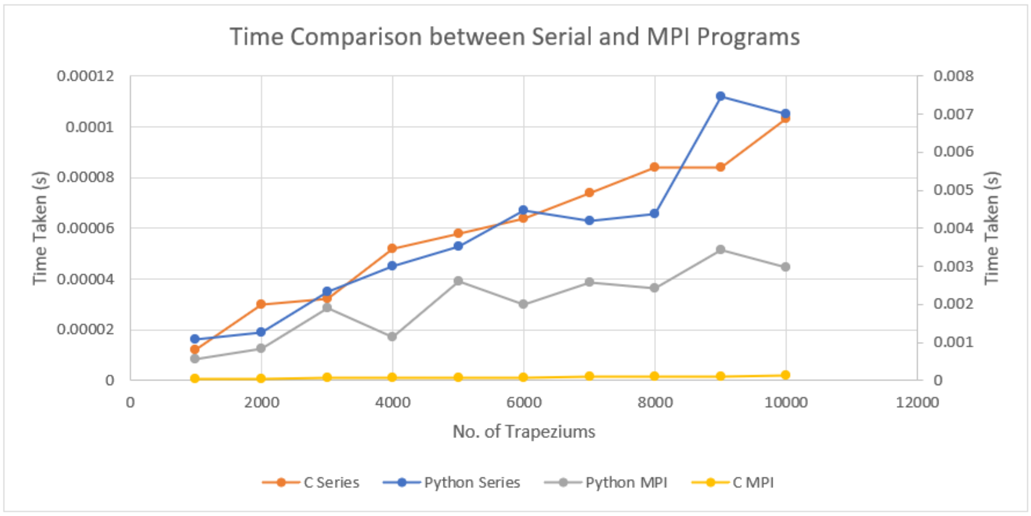
No. of Trapeziums	Time Taken (s)	Accuracy
1000	0.000037	2.66016173
2000	0.000048	2.66345334
3000	0.000072	2.66444683
4000	0.000076	2.66499472
5000	0.000087	2.66526604
6000	0.000070	2.66551304
7000	0.000099	2.66580105
8000	0.000108	2.66578150
9000	0.000111	2.66587353
10000	0.000142	2.66612148

Data for C Code



Comparison between C and Python MPI programs

4.1.3 Comparison between MPI and Serial Program



Comparison between Serial and MPI programs

4.2 Matrix Multiplication

Matrix Multiplication can be done more efficiently by using MPI.
I have used Square matrix as the data set to compute the multiplication. All the computation was divided and allocated to 4 processors.
For Python, MPI for matrix multiplication only works with the Numpy Library, the code snippet for the same is below

```python
b = comm.bcast(b, root=0)

c = np.dot(a, b)

if size == 1:

    result = np.dot(a, b)

else:

    if rank == 0:

        a_row = a.shape[0]

        if a_row >= size:

            split = np.array_split(a, size, axis=0)

    else:

        split = None

    split = comm.scatter(split, root=0)

    split = np.dot(split, b)

    data = comm.gather(split, root=0)
```

Listing 4.2: Code snippet from amtrixmultMPI.py

For the C program the code snippet is as follows

```c
    MPI_Scatter(a, N*N/size, MPI_INT, aa, N*N/size, MPI_INT
   ,0,MPI_COMM_WORLD);

    MPI_Bcast(b, N*N, MPI_INT, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);

            for (i = 0; i < N; i++)
              {
                      for (j = 0; j < N; j++)
                      {
                              sum = sum + aa[j] * b[j][i];
                      }
                      cc[i] = sum;
                      sum = 0;
              }

    MPI_Allgather(cc, N*N/size, MPI_INT, c, N*N/size, MPI_INT
   , MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
```
Listing 4.3: Code snippet from amtrixmultMPI.c

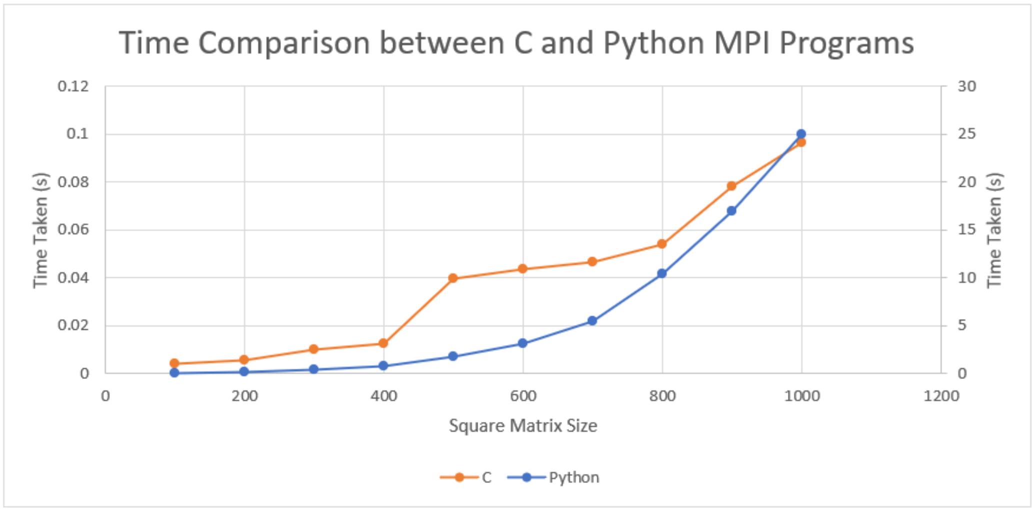
4.2.1 For MPI Program in C and Python

Matrix Size	Time Taken (s)
100x100	0.09769
200x200	0.16343
300x300	0.38058
400x400	0.83123
500x500	1.72779
600x600	3.16170
700x700	5.53350
800x800	10.47173
900x900	16.94393
1000x1000	24.93814

Data for Python Code

Matrix Size	Time Taken (s)
100x100	0.00429
200x200	0.00557
300x300	0.00984
400x400	0.01241
500x500	0.03970
600x600	0.04381
700x700	0.04673
800x800	0.05384
900x900	0.07845
1000x1000	0.09638

Data for C Code



Comparison between C and Python MPI programs

Conclusion

Through this project I was able to find out the efficiency of program in C and Python using the MPI Library and it was seen that Python Programs took more time to compute than the C counterpart.

Appendix A

Codes

A.1 helloworld.c

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv) {
    // Initialize the MPI environment
    MPI_Init(&argc, &argv);
    // Get the number of processes
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // Get the rank of the process
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    // Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);
    // Print off a hello world message
    printf("Hi form rank %d \n",world_rank);
    // Finalize the MPI environment.
    MPI_Finalize();
return 0;
}
```

A.2 helloworld.py

```python

from mpi4py import MPI
import sys

size = MPI.COMM_WORLD.Get_size()
rank = MPI.COMM_WORLD.Get_rank()
name = MPI.Get_processor_name()

sys.stdout.write(
    "Hello, World! I am process %d of %d on %s.\n"
    % (rank, size, name))
```

A.3 ring.c

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

int main(int argc, char* argv[])
{
MPI_Init(&argc, &argv);
int token;
int world_rank;
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
if (world_rank != 0) {
    MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n",
           world_rank, token, world_rank - 1);
} else {
    token = -1;
}
MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size,
         0, MPI_COMM_WORLD);

if (world_rank == 0) {
    MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n",
            world_rank, token, world_size - 1);
}
MPI_Finalize();
}
```

A.4 ring.py

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank=comm.rank
size=comm.size

shared=(rank)

comm.send(shared,dest=(rank+1)%size)
data=comm.recv(source=(rank-1)%size)
print ('Rank:',rank)
print ('Recieved:',data,'which came from rank:',(rank-1)%size
   )
```

A.5 IntegrationSerial.c

```c
#include <stdio.h>
#include <math.h>
#include <time.h>

int main(){
    float integral;
    float a,b;
    int n;
    float h;
    float x;
    clock_t start,end;
    double cpu_time_used;


    float f(float x);
    printf("Enter a, b and n \n");
    scanf("%f %f %d", &a, &b, &n);
    start = clock();

    h = (b-a)/n;
    integral = (f(a) + f(b))/2.0;
    x = a;
    for( int i = 1; i <= n-1; i++){
        x += h;
        integral += f(x);
    }
    integral *= h;
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC
   ;
    printf(" With n = %d trapezoids, our estimate \n", n);
    printf("of the integral form %f to %f = %f\n", a, b,
   integral);
    printf("Program took %f seconds to execute \n",
   cpu_time_used);

}

float f(float x){
    return x*x;
}
```

A.6 IntegrationSerial.py

```
import time

a = float(input("Enter Value for upper limit: \n"))
b = float(input("Enter vale for lower limit: \n"))
n = int(input("Enter number of trapezoids to form: \n"))

num1 = int(a)
num2 = int(b)
begin = time.time()
def fx(x):
    return x*x

height = (b-a)/n
integral = (fx(a) + fx(b))/2.0
x = a
for i in range(1,n):
    x += height
    integral += fx(x)


integral *= height
end = time.time()
totaltime = end - begin
print(" With n = %d trapezoids, our estimate \n" %(n))
print("of the integral from %f to %f = %0.8f\n" %(a,b,
   integral))
print("Total runtime of the program is %f" %(totaltime))
```

A.7 IntegrationMPI.c

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"

int main(int argc, char** argv){
    int my_rank;
    double time1, time2, duration, global;
    int size;
    float a ;
    float b ;
    int n ;
    float h;
    float local_a;
    float local_b;
    int local_n;
    float integral;
    float total;
    int source =0;
    int dest = 0;
    int tag = 0;
    MPI_Status status;

    float Trap(float local_a, float local_b, int local_n,
   float h);

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

```

```
 MPI_Comm_size(MPI_COMM_WORLD, &size);
 if (my_rank == 0){
     printf("Enter a, b and n \n");
     scanf("%f %f %d", &a, &b, &n);
     for ( dest = 1 ; dest < size; dest++){
         MPI_Send(&a, 1 , MPI_FLOAT, dest , tag=0,
MPI_COMM_WORLD);
         MPI_Send(&b, 1 , MPI_FLOAT, dest , tag=1,
MPI_COMM_WORLD);
         MPI_Send(&n, 1 , MPI_INT, dest , tag=2,
MPI_COMM_WORLD);
     }


 }
 else{
     MPI_Recv(&a, 1, MPI_FLOAT, source, tag=0,
MPI_COMM_WORLD, &status);
     MPI_Recv(&b, 1, MPI_FLOAT, source, tag=1,
MPI_COMM_WORLD, &status);
     MPI_Recv(&n, 1, MPI_INT, source, tag=2,
MPI_COMM_WORLD, &status);
 }
 MPI_Barrier(MPI_COMM_WORLD);
 time1 = MPI_Wtime();
 h = (b-a)/n;

 local_n = n/size;

 local_a = a + my_rank * local_n * h;

 local_b = (local_a + local_n) * h;

 integral = Trap(local_a, local_b, local_n, h);

 if (my_rank == 0){
     total = integral;
     for (source = 1; source < size; source++){
         MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
MPI_COMM_WORLD, &status);
         total += integral;
     }
 }
 else {
     MPI_Send(&integral, 1, MPI_FLOAT, dest, tag,
   MPI_COMM_WORLD);
    }
    time2 = MPI_Wtime();
    duration = time2 - time1;
    MPI_Reduce(&duration, &global,1,MPI_DOUBLE,MPI_SUM,0,
   MPI_COMM_WORLD);

    if (my_rank == 0){
        printf("With n = %d trapezoids, our estimate \n", n);
        printf("of the integral from %f to %f = %0.8f\n",a,b,
   total);
        printf("Global runtime is %f\n",global);
    }
    printf("Runtime at %d is %f \n", my_rank,duration);
    MPI_Finalize();

}

float Trap(float local_a, float local_b, int local_n, float h
   ){
    float integral;
    float x;
    int i;

    float f(float x);
    integral = (f(local_a) + f(local_b))/2.0;
    x = local_a;
    for (int i = 1; i <= local_n-1; i++){
        x += h;
        integral += f(x);
    }
    integral *= h;

}

float f(float x){
    return x*x;
}
```

A.8 IntegrationMPI.py

```python
from mpi4py import MPI
import mpi4py

def fx(x):
    return x*x

def Trap(local_a , local_b, local_n, h):
    integral = (fx(local_a) + fx(local_b))/2.0
    x = local_a
    for i in range(1,int(local_n)):
        x += h
        integral += fx(x)
    integral *= h
    return integral


comm = MPI.COMM_WORLD
rank=comm.rank
size=comm.size


source = 1
dest = 0
a = 0.0
b = 0.0
n = 0

if rank == 0:

    a = float(input("Enter Upper Limit \n"))
    b = float(input("Enter Lower Limit \n"))
    n = int(input("Enter number of trapezoids \n"))
    for i in range(1,size):
        dest += 1
        comm.send(a, dest, tag = 0)
        comm.send(b, dest, tag = 1)
        comm.send(n, dest, tag = 2)

else:
    a=comm.recv(source = 0, tag = 0)
    b=comm.recv(source = 0, tag = 1)
    n=comm.recv(source = 0, tag = 2)
time1 = mpi4py.MPI.Wtime()
```

```
h = (b-a)/n
local_n = n/size
local_a = a + rank * local_n * h
local_b = (local_a + local_n) * h

integral = Trap(local_a, local_b, local_n, h)

if rank == 0:
    total = integral
    while (source < size):
        integral = comm.recv(source = source,tag = 0)
        source += 1
        total = total + integral

else:
    comm.send(integral, dest = 0, tag = 0)
time2 = mpi4py.MPI.Wtime()
duration = time2 - time1
totaltime = comm.reduce(duration,op = MPI.SUM, root = 0)
print("Runtime at %d is %f" %(rank,duration))

if rank == 0:
    print(" With n = %d trapezoids, our estimate \n" %(n))
    print("of the integral from %f to %f = %0.8f\n" %(a,b,
    total))
    print("Totaltime = ", totaltime
```

A.9 matrixmultMPI.c

```
#define N 16
\\Change value of N above to a multiple of 4 to run the code
    properly
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include "mpi.h"


void print_results(char *prompt, int a[N][N]);

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

```c
    int i, j, k, rank, size, tag = 99, sum = 0;
    int a[N][N];
    int b[N][N];
    int c[N][N];
    int aa[N],cc[N];
    int row,col;
    int dest = 0;
    int source;
    double time1, time2, duration, global;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if(rank == 0){


        printf("enter the number of row =");
        scanf("%d",&row);
        printf("enter the number of column =");
        scanf("%d",&col);

        srand(time(NULL));
        for(i=0;i<row;i++) {
            for(j=0;j<col;j++){
                a[i][j] = rand() % 10;
            }
        }

        for(i=0;i<row;i++){
            for(j=0;j<col;j++){
                b[i][j] = rand() % 10;
            }
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    time1 = MPI_Wtime();
    MPI_Scatter(a, N*N/size, MPI_INT, aa, N*N/size, MPI_INT
   ,0,MPI_COMM_WORLD);

    MPI_Bcast(b, N*N, MPI_INT, 0, MPI_COMM_WORLD);

    MPI_Barrier(MPI_COMM_WORLD);

```

```
          for (i = 0; i < N; i++)
            {
                    for (j = 0; j < N; j++)
                    {
                            sum = sum + aa[j] * b[j][i];
                    }
                    cc[i] = sum;
                    sum = 0;
            }

    MPI_Allgather(cc, N*N/size, MPI_INT, c, N*N/size, MPI_INT
   , MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    time2 = MPI_Wtime();
    duration = time2 - time1;
    MPI_Reduce(&duration,&global,1,MPI_DOUBLE,MPI_SUM,0,
   MPI_COMM_WORLD);
    if(rank == 0) {
        printf("Global runtime is %f\n",global);
    }
    printf("Runtime at %d is %f \n", rank,duration);
    MPI_Finalize();
    if (rank == 0)
      print_results("C = ", c);
}

void print_results(char *prompt, int a[N][N])
{
    int i, j;

    printf ("\n\n%s\n", prompt);
    for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                    printf(" %d", a[i][j]);
            }
            printf ("\n");
    }
    printf ("\n\n");
}
```

A.10 matrixmultMPI.py

```python
from mpi4py import MPI
import mpi4py
import numpy as np

comm = MPI.COMM_WORLD
size = comm.size
rank = comm.Get_rank()
time1 = mpi4py.MPI.Wtime()


a = np.random.randint(10, size=(100, 100))
if rank == 0:
    b = np.random.randint(10, size=(100, 100))
    print(b)
else:
    b = None

b = comm.bcast(b, root=0)

c = np.dot(a, b)

if size == 1:

    result = np.dot(a, b)

else:

    if rank == 0:

        a_row = a.shape[0]

        if a_row >= size:

            split = np.array_split(a, size, axis=0)

    else:

        split = None

    split = comm.scatter(split, root=0)

    split = np.dot(split, b)

    data = comm.gather(split, root=0)

time2 = mpi4py.MPI.Wtime()
duration = time2 - time1
totaltime = comm.reduce(duration,op = MPI.SUM, root = 0)
print("Runtime at %d is %f" %(rank,duration))

if rank == 0:

    result = np.vstack(data)
    print(result)
    print(totaltime)
```

Appendix B

References

1. **https://rookiehpc.github.io/index.html**
2. **https://en.wikipedia.org/wiki/Message_Passing_Interface**
3. **https://www.cuemath.com/trapezoidal-rule-formula/**