Report for HPC LAB

Name: Bhaskar R
Roll No: CED18I009
Programming Environment: MPI
Problem: Matrix Addition
Date: 22nd October 2021

Hardware Configuration:

CPU NAME : Intel core i5 – 8250U @ 1.60 GHz
Number of Sockets : 1
Cores per Socket : 4
Threads per core : 2
L1 Cache size : 32KB (Per Core)
L2 Cache size : 256KB (Per Core)
L3 Cache size : 6MB (Shared)
RAM : 8 GB

Serial Code:

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define NR 5
#define NC 5
#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2int main()
{
	double start, end;
	int i, j;
	long double a[NR][NC], b[NC][NC], c[NR][NC];
	start = MPI_Wtime();
	for (i = 0; i < NR; i++)
		for (j = 0; j < NC; j++)
			a[i][j] = i + j * 1.785;
	for (i = 0; i < NR; i++)
		for (j = 0; j < NC; j++)
			b[i][j] = i + j * 0.987;
	for (i = 0; i < NR; i++)
		for (j = 0; j < NC; j++)
		{
			c[i][j] = a[i][j] + b[i][j];
		}
	printf("\nResultant Matrix:\n");
	for (i = 0; i < NR; i++)
	{
		printf("\n");
		for (j = 0; j < NC; j++)
			printf("%3.1Lf ", c[i][j]);
	}
	printf("\nFinished.\n");
```

```
        end = MPI_Wtime();
        printf("\nTime= %f", end - start);
        return 0;
}
```

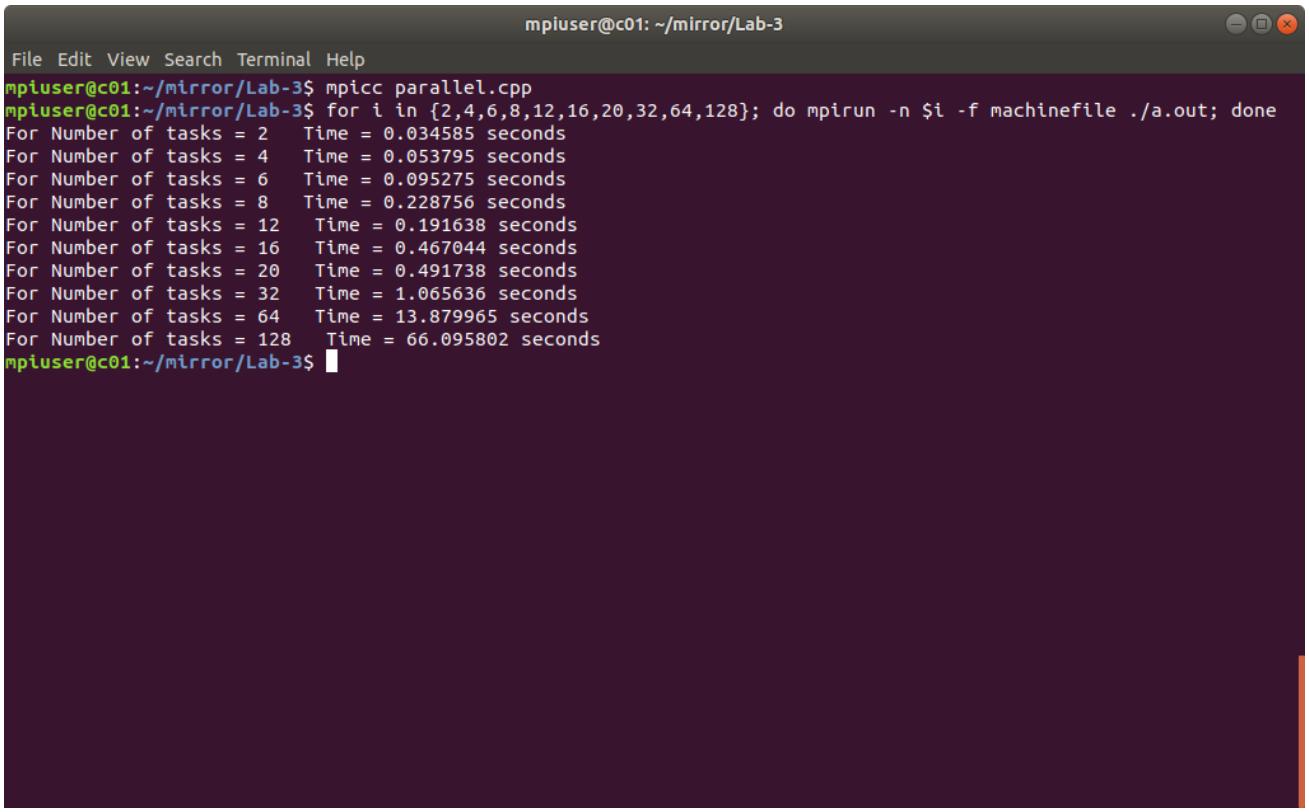
Parallel Code :

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#define NR 5
#define NC 5
#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2
// cluster parallel code
int main(int argc, char *argv[])
{
        int numtasks, taskid, numworkers, source, dest, mtype, rows, averow, extra, offset, i,
j,
                k, rc;
        long double a[NR][NC], b[NC][NC], c[NR][NC];
        MPI_Status status;
        double start, end;
        MPI_Init(&argc, &argv);
        start = MPI_Wtime();
        MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
        MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
        if (numtasks < 2)
        {
                printf("Need atleast two MPI tasks. Quiting...\n");
                MPI_Abort(MPI_COMM_WORLD, rc);
                exit(1);
        }
        char pro_name[MPI_MAX_PROCESSOR_NAME];
        int name_len;
        MPI_Get_processor_name(pro_name, &name_len);
        printf("-From from %s, rank %d, out of %d processors\n", pro_name, taskid,
                numtasks);
        numworkers = numtasks - 1;
        // master task:
        if (taskid == MASTER)
        {
                for (i = 0; i < NR; i++)
                        for (j = 0; j < NC; j++)
                                a[i][j] = i + j * 1.785;
                for (i = 0; i < NC; i++)
                        for (j = 0; j < NC; j++)
                                b[i][j] = i * j * 0.897;
                averow = NR / numworkers;
                extra = NR % numworkers;
                offset = 0;
```

```
                mtype = FROM_MASTER;
                for (dest = 1; dest <= numworkers; dest++)
                {
                        rows = (dest <= extra) ? averow + 1 : averow;
                        MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
                        MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
                        MPI_Send(&a[offset][0], rows * NC, MPI_LONG_DOUBLE, dest,
mtype,
                                        MPI_COMM_WORLD);
                        MPI_Send(&b[offset][0], rows * NC, MPI_LONG_DOUBLE, dest,
mtype,
                                        MPI_COMM_WORLD);
                        offset = offset + rows;
                }
                // receive from worker:
                mtype = FROM_WORKER;
                for (i = 1; i <= numworkers; i++)
                {
                        source = i;
                        MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
                        MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD,
&status);
                        MPI_Recv(&c[offset][0], rows * NC, MPI_LONG_DOUBLE, source,
mtype, MPI_COMM_WORLD, &status);
                }
                printf("\nResultant Matrix:\n");
                for (i = 0; i < NR; i++)
                {
                        printf("\n");
                        for (j = 0; j < NC; j++)
                                printf("%6.2Lf ", c[i][j]);
                }
                printf("\nDone.\n");
                end = MPI_Wtime();
                printf("\nTime= %f", end - start);
        }
        // Worker task:
        if (taskid > MASTER)
        {
                mtype = FROM_MASTER;
                MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD,
&status);
                MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD,
&status);
                MPI_Recv(&a, rows * NC, MPI_LONG_DOUBLE, MASTER, mtype,
                                MPI_COMM_WORLD, &status);
                MPI_Recv(&b, rows * NC, MPI_LONG_DOUBLE, MASTER, mtype,
                                MPI_COMM_WORLD, &status);
                // mat addition
                for (k = 0; k < NC; k++)
                        for (i = 0; i < rows; i++)
                        {
```

```
                        c[i][k] = a[i][k] + b[i][k];
                    }
                mtype = FROM_WORKER;
                MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
                MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
                MPI_Send(&c, rows * NC, MPI_LONG_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD);
        }
        MPI_Finalize();
}
```

Output :



Compilation and Execution:

Compiling using mpic++ parallel.cpp

For execution use

```
for i in {2,4,6,8,12,16,20,32,64,128}; do mpirun -n $i -f machinefile ./a.out
```

Observations:

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	0.01	1	
2	0.03	0.3333	-4.0006
4	0.05	0.2	-5.3333
6	0.09	0.1111	-9.6011
8	0.22	0.0455	-23.9749
12	0.19	0.0526	-19.6488
16	0.46	0.0217	-48.0885
20	0.49	0.0204	-50.547
32	1.06	0.0094	-108.7824
64	13.87	0.0007	-1450.2313
128	66.09	0.0002	-5038.3622

Speed up can be found using the following formula,

S(n)=T(1)/T(n)

where, S(n) = Speedup for thread count ‘n’

T(1) = Execution Time for Thread count ‘1’ (serial code)

T(n) = Execution Time for Thread count ‘n’ (serial code)

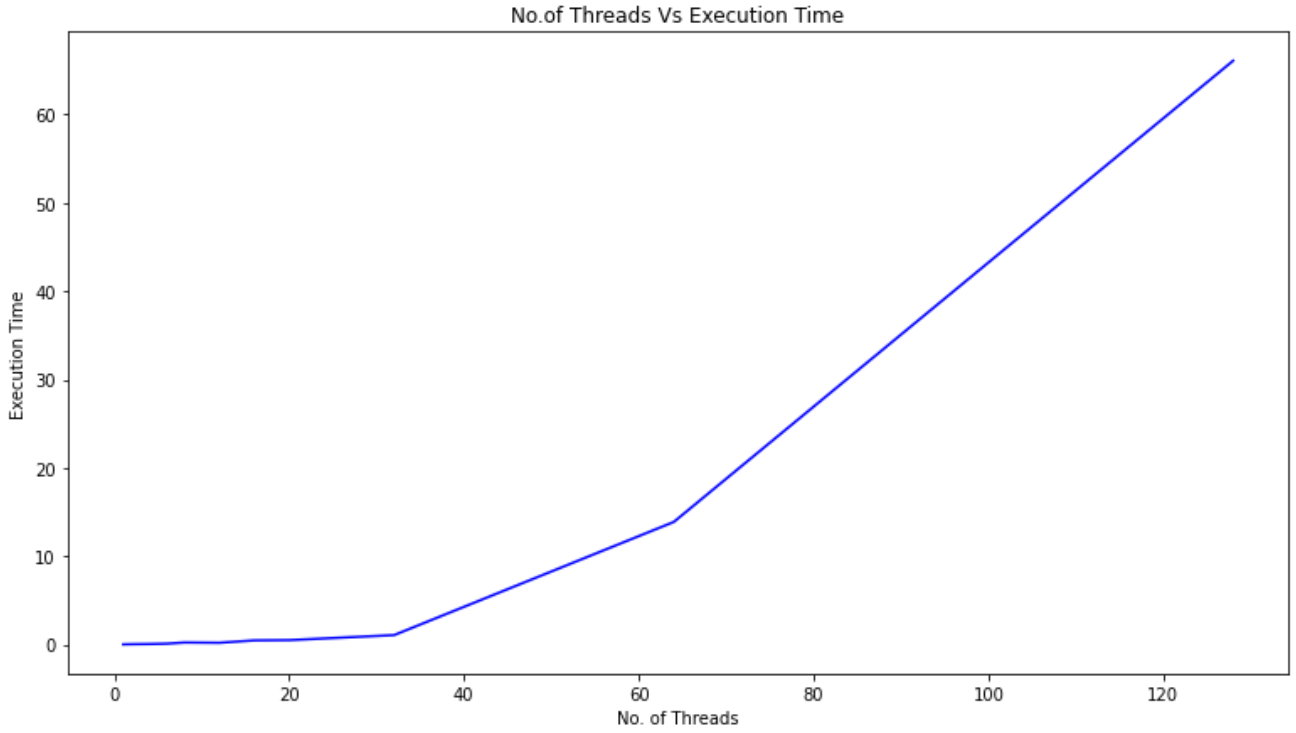
Parallelization Fraction can be found using the following formula, **S(n)=1/((1 - p) + p/n)**

where, S(n) = Speedup for thread count ‘n’

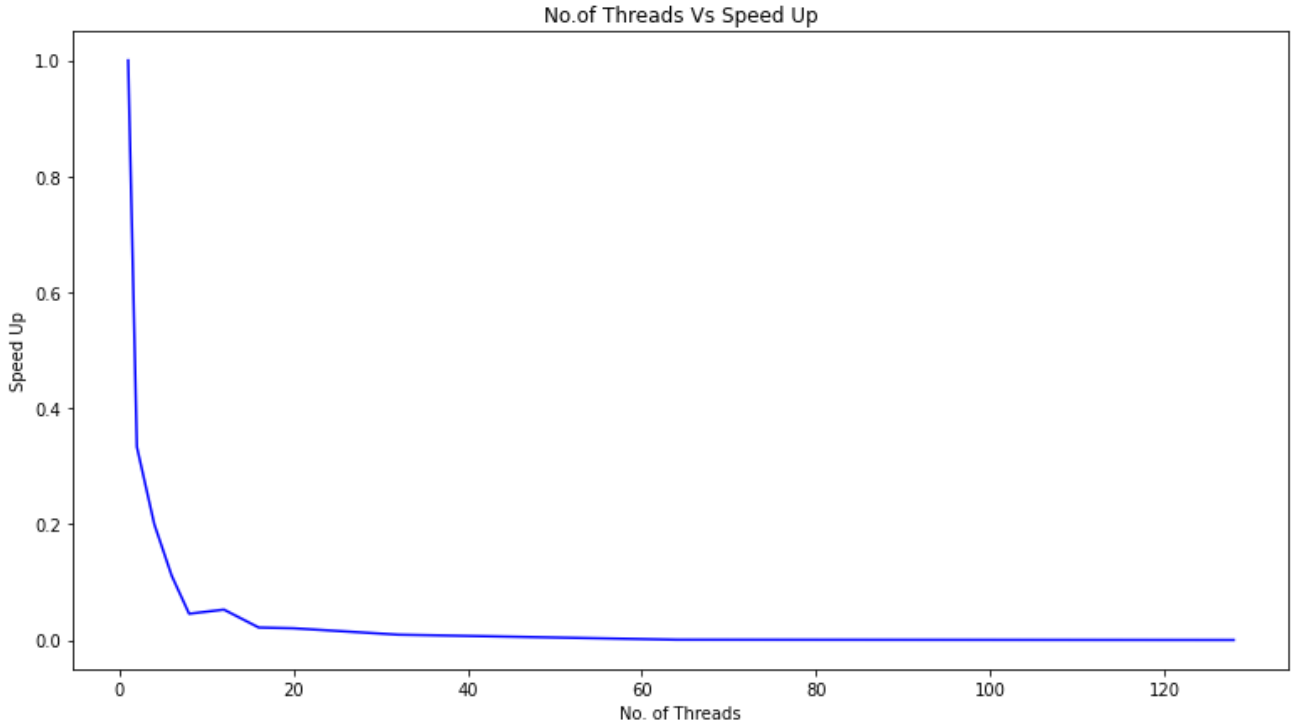
n = Number of threads

p = Parallelization fraction

Number of Threads vs Execution Time:



Number of Threads vs Speed Up:



Inference:

• Execution time is increasing with an increase in the number of threads, But at 6 threads less execution time occurred. Since the problem is of smaller complexity the overheads of parallelization seem to have more effects here.