# Report for HPC LAB

**Name:** Bhaskar R
**Roll No:** CED18I009
**Programming Environment:** MPI
**Problem:** Sum of N Numbers
**Date:** 29th 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:**

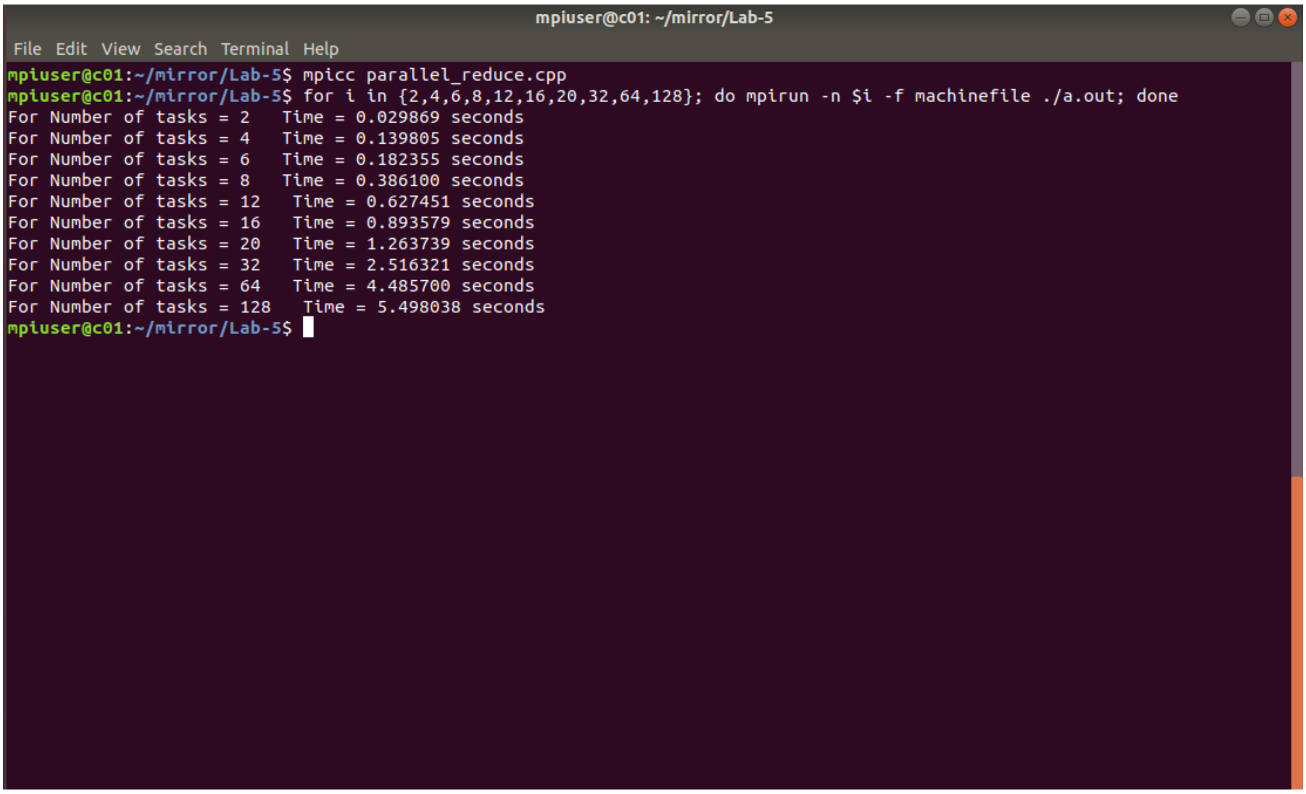
```
#include <stdio.h>
#include <time.h>
#include <omp.h>
#include <stdlib.h>
#define n 100
int main()
{
	double a[n], rtime;
	float startTime, endTime, execTime;
	int i, k, omp_rank;
	double sum;
	int thread[] = {1, 2, 4, 6, 8, 12, 16, 20, 32, 64};
	int thread_arr_size = 10;
	sum = 0.0;
	omp_set_num_threads(thread[k]);
	startTime = omp_get_wtime();
	for (i = 0; i < n; i++)
	{
		omp_rank = omp_get_thread_num();
		a[i] = (float)i * 1.67;
		for (int j = 0; j < 100010; j++)
			a[i] = a[i] + 2;
		sum = sum + a[i];
	}
	endTime = omp_get_wtime();
	execTime = endTime - startTime;
	rtime = execTime;
	printf("\n rtime=%f\n", rtime);
	return 0;
}
```

**(i) Parallel Code (Reduction) :**

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define N 100

int main(int argc, char **argv)
{
	int myid, numprocs;
	int i, x, start, end, rem;
	float val = 0, result, arr[N];
	double st, ed;
	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
	MPI_Comm_rank(MPI_COMM_WORLD, &myid);
	char pro_name[MPI_MAX_PROCESSOR_NAME];
	int name_len;
	MPI_Get_processor_name(pro_name, &name_len);
	if (0 == myid)
	{
		st = MPI_Wtime();
		for (i = 0; i < N; i++)
			arr[i] = i;
	}
	/* broadcast arr */
	MPI_Bcast(arr, N, MPI_FLOAT, 0, MPI_COMM_WORLD);
	/* add portion of arr */
	x = N / numprocs;
	start = myid * x;
	end = start + x;
	for (i = start; i < end; i++)
	{
		val += arr[i];
	}
	printf("Calculated %f in %d - %s\n", val, myid, pro_name);
	/* compute global sum */
	MPI_Reduce(&val, &result, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);
	if (myid==0)
	{
		rem = N % numprocs;
		for (i = N - rem; i < N; i++)
			result += arr[i];
		printf("Sum is %f.\n", result);
		ed = MPI_Wtime();
		printf("\nTime= %f", ed - st);
	}
	MPI_Finalize();
}
```

## Output :




## Compilation and Execution:

Compiling using mpic++ parallel_reduce.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 | 0.02 | 0.5 | -2.0 |
| 4 | 0.13 | 0.0769 | -16.0052 |
| 6 | 0.18 | 0.0556 | -20.3827 |
| 8 | 0.38 | 0.0263 | -42.3118 |
| 12 | 0.62 | 0.0161 | -66.6674 |
| 16 | 0.89 | 0.0112 | -94.1714 |
| 20 | 1.26 | 0.0079 | -132.1919 |
| 32 | 2.51 | 0.004 | -257.0323 |
| 64 | 4.48 | 0.0022 | -460.7446 |
| 128 | 5.49 | 0.0018 | -558.9221 |

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)

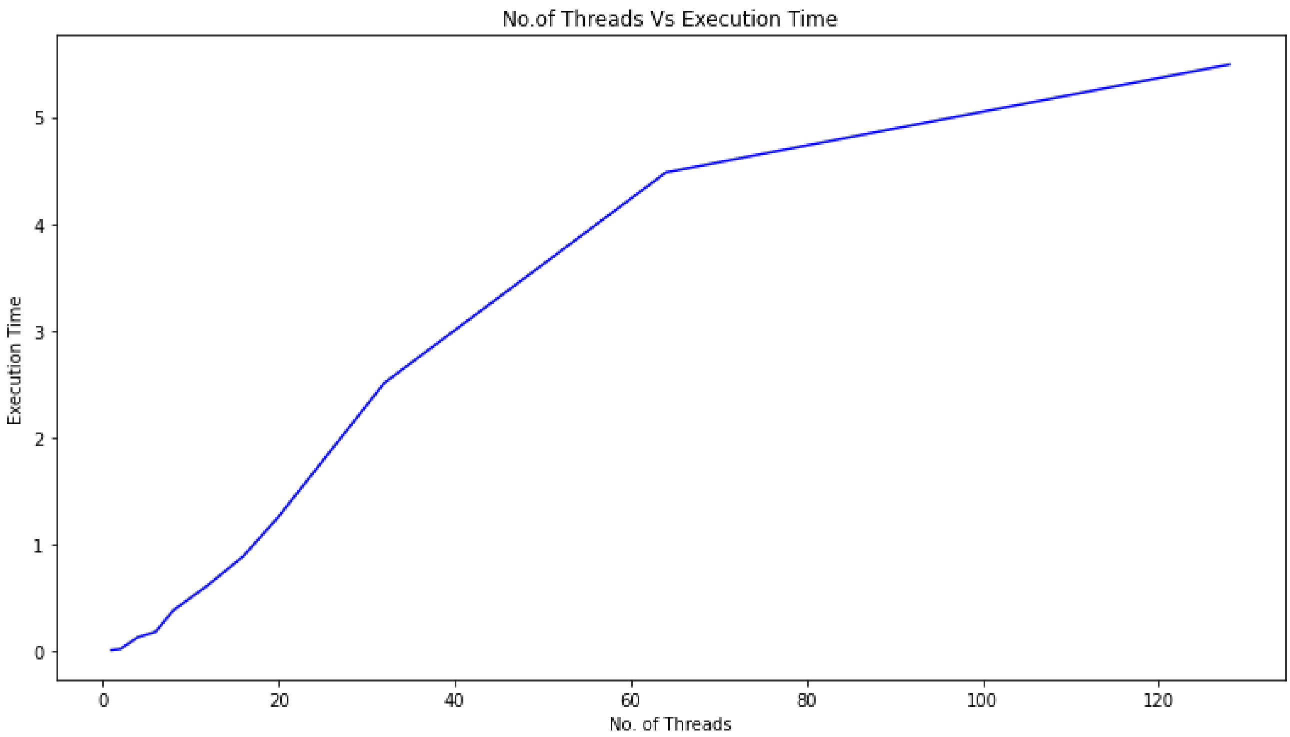
Parallelization Fraction can be found using the following formula, **S(n)=1/((1 - p) + p/n)**

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

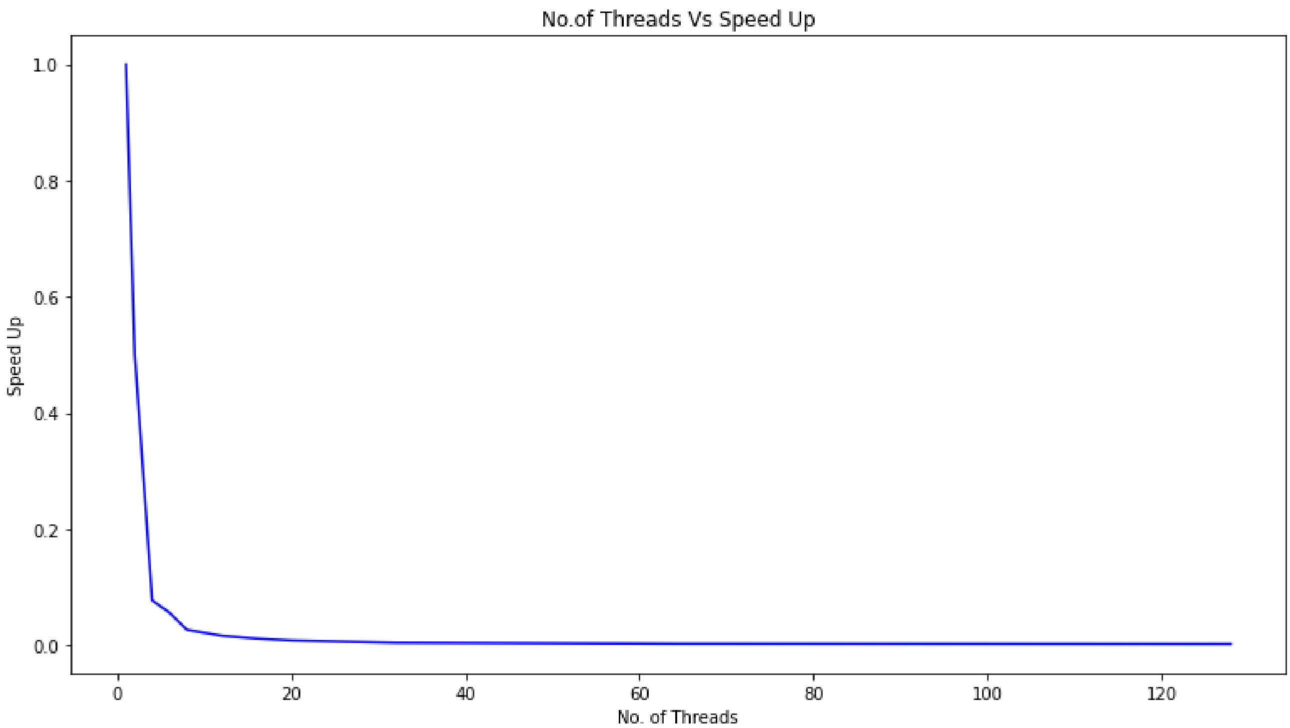
n = Number of threads

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.
Since the problem is of smaller complexity the overheads of parallelization seem to have more effects here.

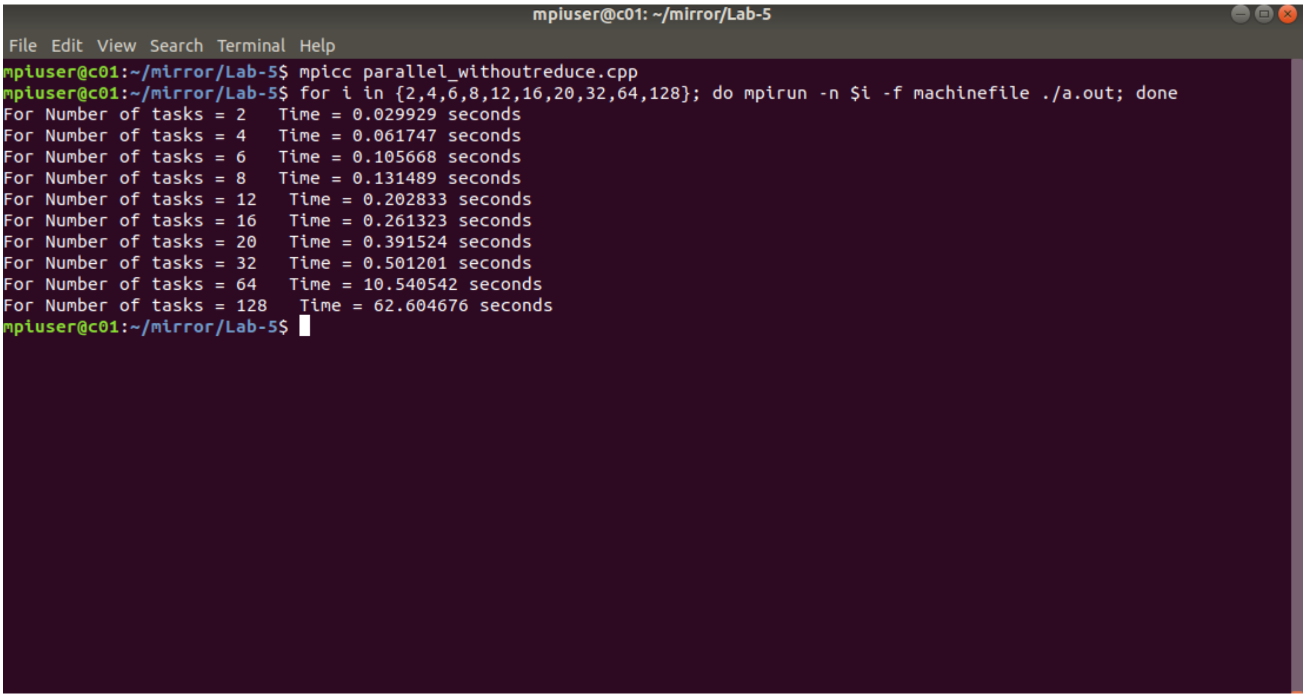
**(ii) Parallel Code (Without Reduction) :**

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define SIZE 100
#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2

int main(int argc, char *argv[])
{
        int numtasks, taskid, numworkers, source, dest, mtype,
                segment, aveseg, extra, offset, i, j, k, rc;
        long double a[SIZE], b[SIZE], sum = 0.0, temp = 0.0; //b is temporary
        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 %s, rank %d, out of %d processors\n", pro_name, taskid, numtasks);
        numworkers = numtasks - 1;
        //master task:
        if (taskid == MASTER)
        {
                for (i = 0; i < SIZE; i++)
                {
                        a[i] = i;
                }
                aveseg = SIZE / numworkers;
                extra = SIZE % numworkers;
                offset = 0;
                mtype = FROM_MASTER;
```

```
		for (dest = 1; dest <= numworkers; dest++)
		{
			segment = (dest <= extra) ? aveseg + 1 : aveseg;
			MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
			MPI_Send(&segment, 1, MPI_INT, dest, mtype,
MPI_COMM_WORLD);
			MPI_Send(&a[offset], segment, MPI_LONG_DOUBLE, dest, mtype,
MPI_COMM_WORLD);
			offset = offset + segment;
		}
		//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(&segment, 1, MPI_INT, source, mtype,
MPI_COMM_WORLD, &status);
			MPI_Recv(&temp, 1, MPI_LONG_DOUBLE, source, mtype,
MPI_COMM_WORLD, &status);
			sum += temp;
		}
		printf("Sum is %Lf.\n", sum);
		end = MPI_Wtime();
		printf("\nTime= %f\n", end - start);
	}
	//Worker task:
	if (taskid > MASTER)
	{
		mtype = FROM_MASTER;
		MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD,
&status);
		MPI_Recv(&segment, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD,
&status);
		MPI_Recv(&a, segment, MPI_LONG_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD, &status);
		temp = 0.0;
		for (i = 0; i < segment; i++)
			temp += a[i];
		mtype = FROM_WORKER;
		MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
		MPI_Send(&segment, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
		MPI_Send(&temp, 1, MPI_LONG_DOUBLE, MASTER, mtype,
MPI_COMM_WORLD);
	}
	MPI_Finalize();
	return 0;
}
```

**Output :**




**Compilation and Execution:**

Compiling using mpic++ parallel_withoutreduce.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 | 0.02 | 0.5 | -2.0 |
| 4 | 0.06 | 0.1667 | -6.6651 |
| 6 | 0.10 | 0.1 | -10.8 |
| 8 | 0.13 | 0.0769 | -13.7187 |
| 12 | 0.20 | 0.05 | -20.7273 |
| 16 | 0.26 | 0.0385 | -26.639 |
| 20 | 0.39 | 0.0256 | -40.0658 |
| 32 | 0.50 | 0.02 | -50.5806 |
| 64 | 10.54 | 0.0009 | -1127.7319 |
| 128 | 62.60 | 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)

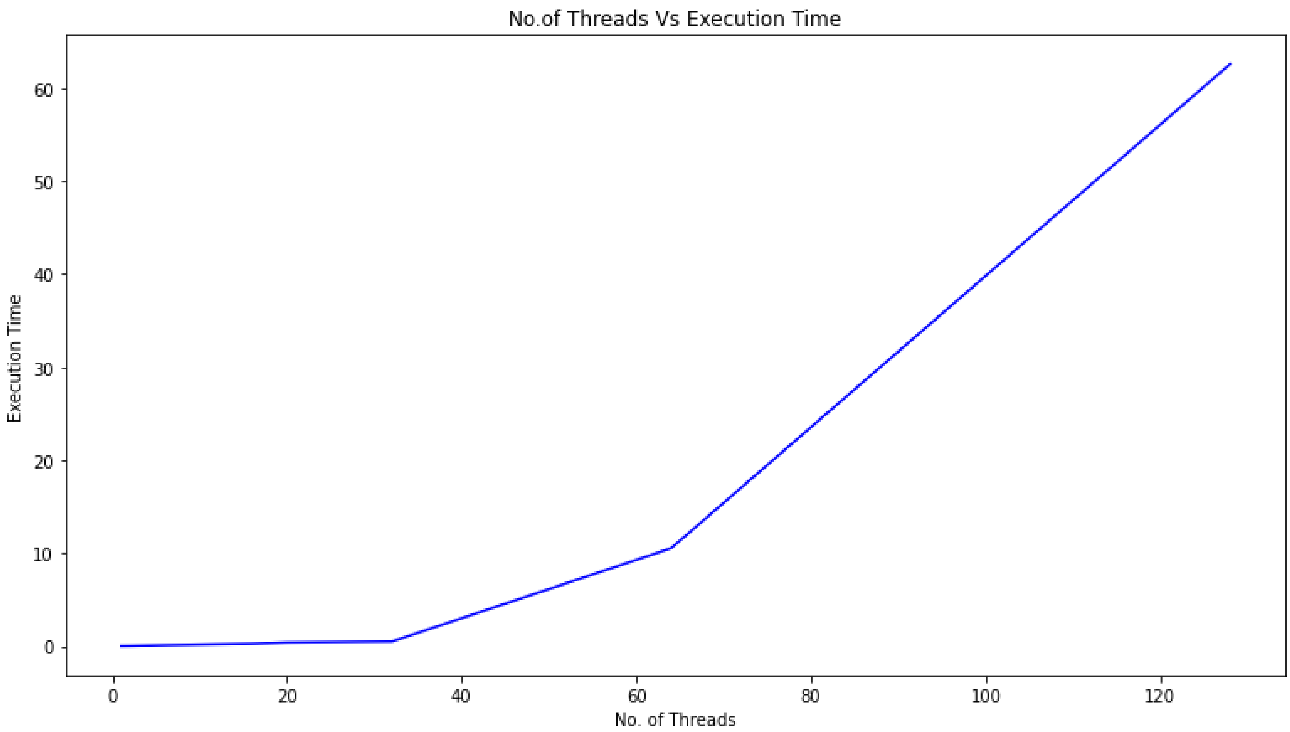
Parallelization Fraction can be found using the following formula, **S(n)=1/((1 - p) + p/n)**

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

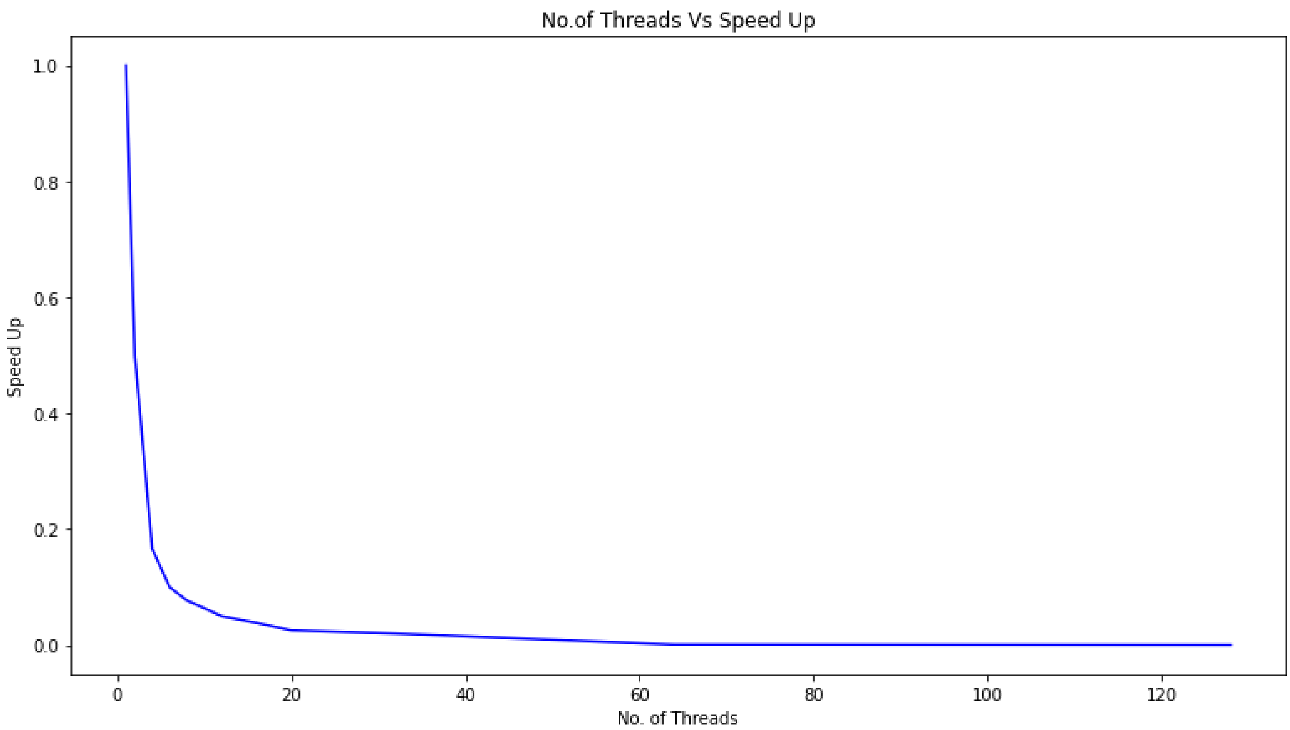
n = Number of threads

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.

Since the problem is of smaller complexity the overheads of parallelization seem to have more effects here.