Report for HPC LAB

Name: Bhaskar R
Roll No: CED18I009
Programming Environment: OpenMP
Problem: Vector dot product using reduction and critical section
Date: 26th August 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 <omp.h>
#define n 50000
#define delay 50000
int main()
{
	double a[n], b[n], c[n], runtime;
	float startTime, endTime;
	int i, k, omp_rank;
	double dot;
	dot = 0.0;
	startTime = omp_get_wtime();
	for (i = 0; i < n; i++)
	{
		omp_rank = omp_get_thread_num();
		a[i] = (float)i * 4.92;
		b[i] = (float)i * 2.37;
		c[i] = 0.0;
		for (int j = 0; j < delay; j++)
			c[i] += a[i] * b[i];
		dot += c[i];
	}
	endTime = omp_get_wtime();
	runtime = endTime - startTime;
	printf("rtime = %f", runtime);
	return 0;
}
```

Parallel Code (Reduction):

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

#define n 50000
#define delay 50000

int main()
{
	double a[n], b[n], c[n], runtime[11];
	float startTime, endTime;
	int i, k, omp_rank;
	double dot;
	int threads[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64};
	for (k = 0; k < 11; k++)
	{
		dot = 0.0;
		omp_set_num_threads(threads[k]);
		startTime = omp_get_wtime();
#pragma omp parallel private(i)
		{
#pragma omp for reduction(+ : dot)
			for (i = 0; i < n; i++)
			{
				omp_rank = omp_get_thread_num();
				a[i] = (float)i * 4.92;
				b[i] = (float)i * 2.37;
				c[i] = 0.0;
				for (int j = 0; j < delay; j++)
					c[i] += a[i] * b[i];
				dot += c[i];
			}
		}
		endTime = omp_get_wtime();
		runtime[k] = endTime - startTime;
	}
	for (k = 0; k < 11; k++)
		printf("\n\nThread = %d     rtime =  %f", threads[k], runtime[k]);
	return 0;
}
```

Compilation and Execution:
For enabling OpenMP environment use -fopenmp flag while

compiling using g++. **g++ -fopenmp dotproduct.cpp**

For execution use

./a.out

Observations:

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	9.40	1	
2	4.58	2.05	102.4
4	2.90	3.24	92.1
6	3.01	3.12	81.5
8	2.62	3.58	82.3
10	2.49	3.77	81.6
12	2.60	3.61	78.8
16	3.47	2.70	67.1
20	3.45	2.72	66.5
32	3.43	2.74	65.5
64	3.50	2.68	63.6

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)**

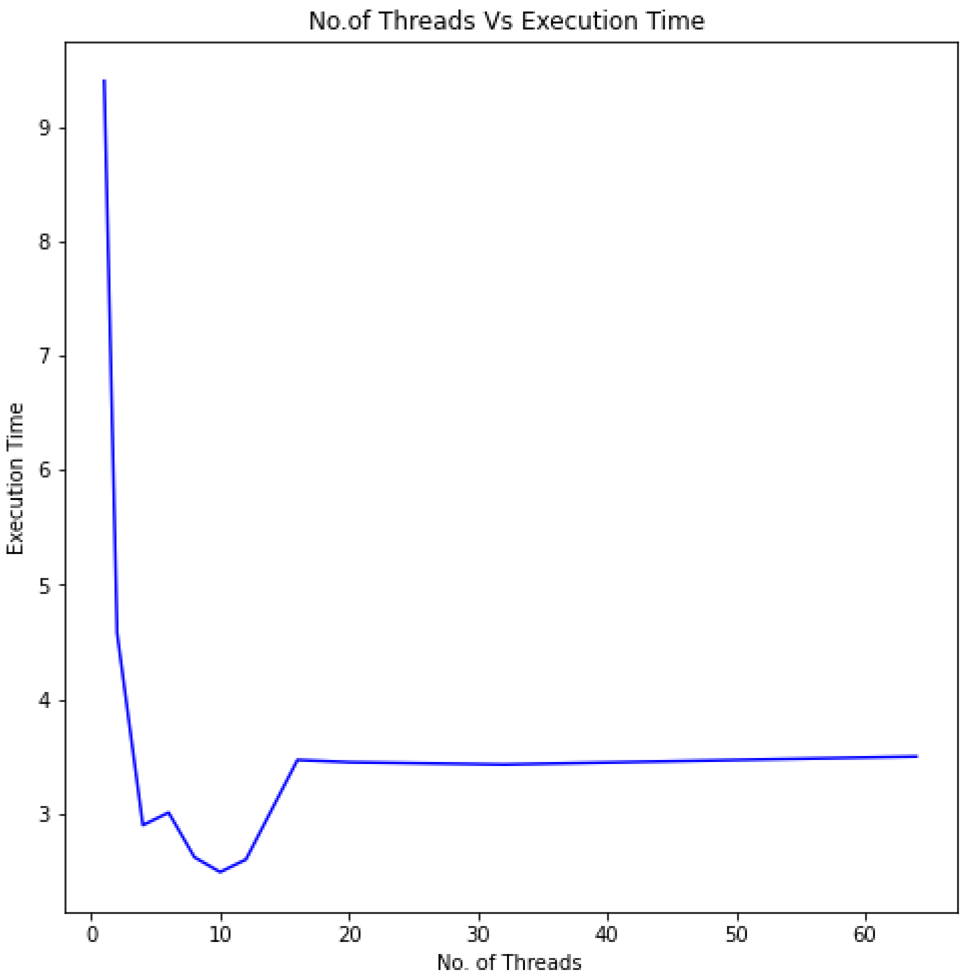
where, S(n) = Speedup for thread count ‘n’
n = Number of threads
p = Parallelization fraction

Assumption:

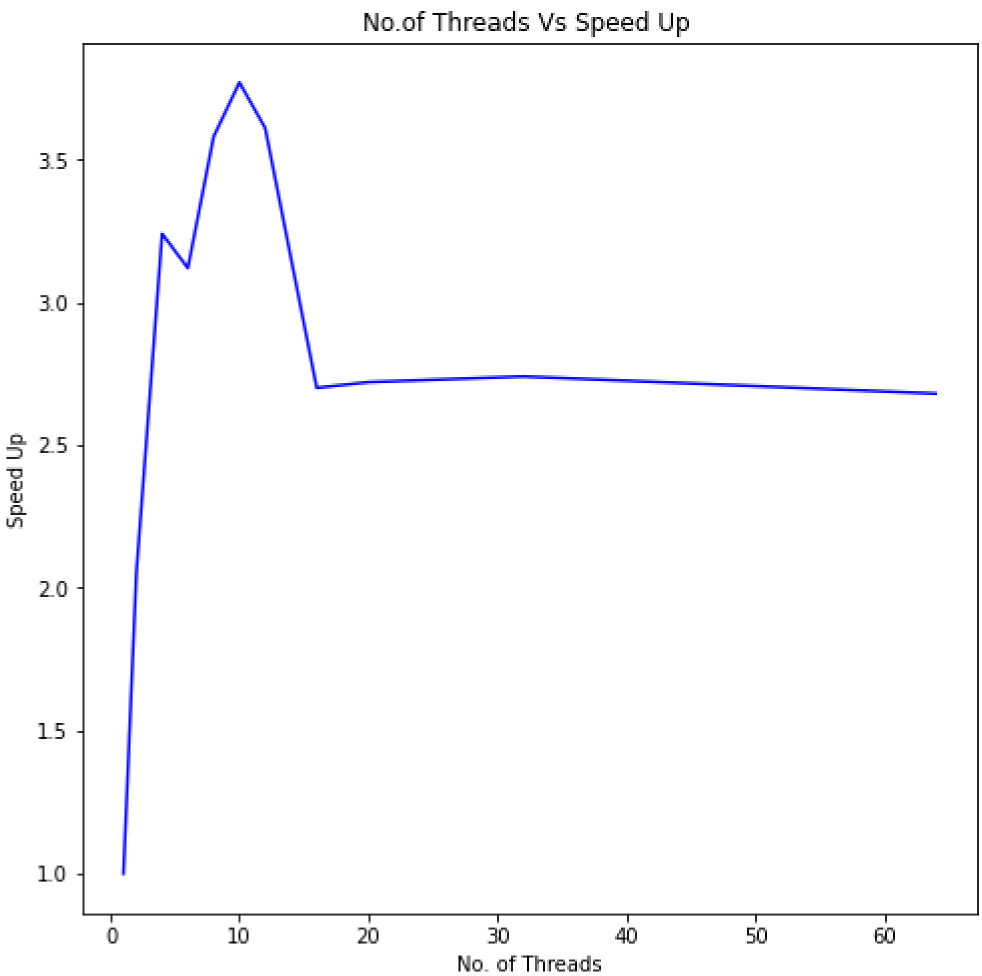
Following extra for loop is added to increase the number of operations in the parallel region to visualize the effect of multi-threading in the vector dot product.

```
for (int j = 0; j < delay; j++)
        c[i] += a[i] * b[i];
dot += c[i];
```

Number of Threads vs Execution Time:



Number of Threads vs Speed Up:



Inference:

(Note: Execution time, graph and inference will be based on hardware configuration**)**

- At thread count 10 maximum speedup is observed as the maximum number of parallel threads supported by the hardware is 8.
- If the thread count is more than 10 then the execution time increases slightly and remains the all most same after 16 threads.

ii) Parallel Code:(Critical Section)

```
#include <stdio.h>
#include <omp.h>
#define n 50000
#define delay 50000
int main()
{
	double a[n], b[n], c[n], runtime[11];
	float startTime, endTime;
	int i, k, omp_rank;
	double dot, fdot;
	int threads[] = {1, 2, 4, 6, 8, 10, 12, 16, 20, 32, 64};
	for (k = 0; k < 11; k++)
	{
		dot = 0.0;
		omp_set_num_threads(threads[k]);
		startTime = omp_get_wtime();
#pragma omp parallel private(i)
		{
#pragma omp for
			for (i = 0; i < n; i++)
			{
				omp_rank = omp_get_thread_num();
				a[i] = (float)i * 4.92;
				b[i] = (float)i * 2.37;
				c[i] = 0.0;
				for (int j = 0; j < delay; j++)
					c[i] += a[i] * b[i];
				dot += c[i];
			}
#pragma omp critical(finaldot)
			fdot += dot;
		}
		endTime = omp_get_wtime();
		runtime[k] = endTime - startTime;
	}
	for (k = 0; k < 11; k++)
		printf("\n\nThread Count: %d     Run Time: %f", threads[k], runtime[k]);
	return 0;
}
```

Compilation and Execution:
For enabling OpenMP environment use -fopenmp flag while

compiling using **g++ -fopenmp dotproduct_cs.cpp**

For execution use

./a.out

Observations:

Number of Threads	Execution Time	Speed-up	Parallelization Fraction
1	10.53	1	
2	5.04	2.08	103
4	4.21	2.50	80
6	3.32	3.17	82.1
8	2.99	3.52	81.8
10	2.97	3.54	79.7
12	3.48	3.02	72.9
16	3.74	2.81	68.7
20	3.68	2.86	68.4
32	3.76	2.80	66.3
64	3.86	2.72	64.2

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)

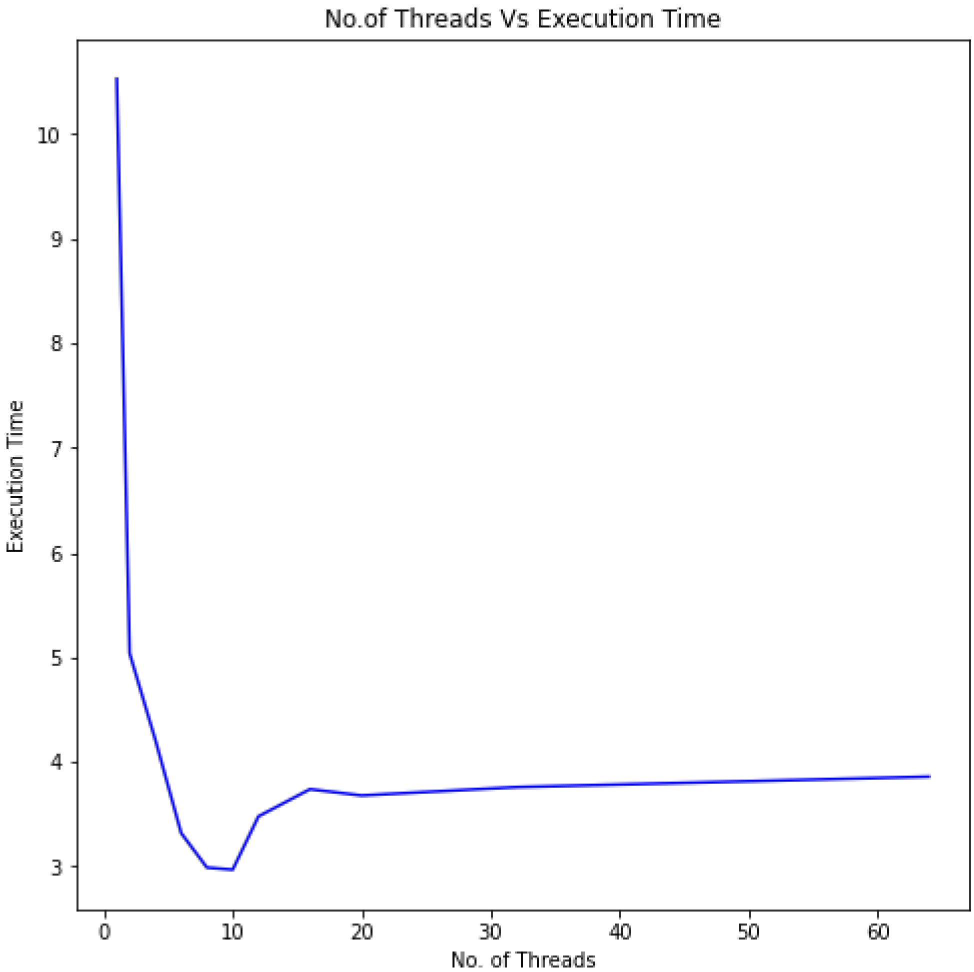
Parallelization Fraction can be found using the following formula, **S(n)=1/((1 - p) + p/n)**

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

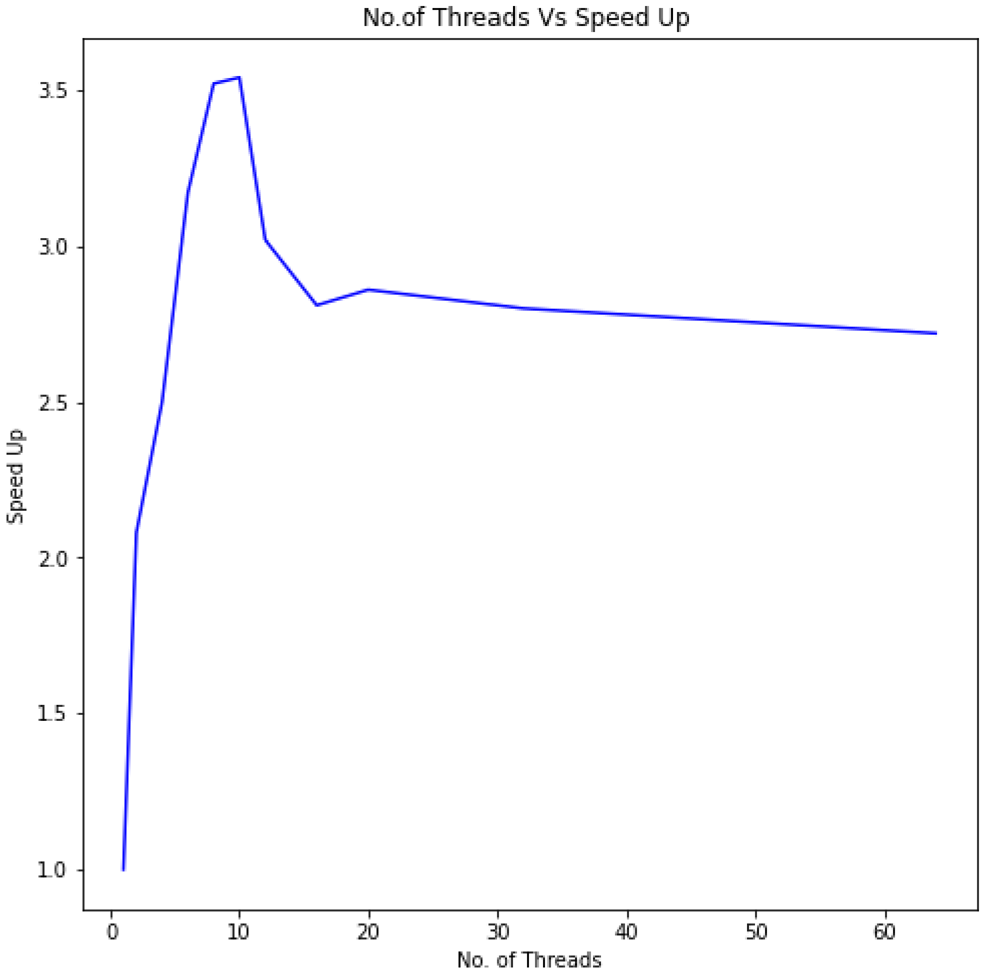
n = Number of threads

p = Parallelization fraction

Number of Threads vs Execution Time:



Number of Threads vs Speed Up:



Inference:

(Note: Execution time, graph and inference will be based on hardware configuration**)**

• At thread count 10 maximum speedup is observed as the maximum number of parallel threads supported by the hardware is 8.

• If the thread count is more than 10 then the execution time increases slightly and remains almost the same after 16 threads.