National University
of computer and emerging sciences

PDC – Assignment 2

CS-326 Parallel and Distributed Computing

BS(CS) – D

Batch 2018

Submitted By:

Hassan Shahzad 18i-0441

Submitted to:

Sir Ehtesham Zahoor

Date of Submission:

29-05-21

TABLE OF CONTENTS

IMPORTANT NOTE

While I was running the codes, I noticed that when I apply a greater number of threads, the execution time increases. It bothered me a lot as during my OS Project the values were fine. In order to test, I created a simple file (i.e., **test.c**) and it had a simple for loop. I ran it without any threads and then with different number of threads. The result is as follows:

test.c

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

int main()
{
    int x = 1;
    double secs = 0;
    clock_t begin = clock();
    //#pragma omp parallel num_threads (2)
    //{
      //  #pragma omp for schedule (static, 400)
        for (int i = 0; i<800; i++)
        {
            //printf("%d",x);
            x++;
        }
    //}
    clock_t end = clock();
    secs = (double)(end-begin) / CLOCKS_PER_SEC;
    printf("\nTime taken = %f\n", secs);
    return 0;
}
```

Fig 0.1 : Serial Version of Test Code

test.c

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

int main()
{
    int x = 1;
    double secs = 0;
    clock_t begin = clock();
    #pragma omp parallel num_threads (2)
    {
        #pragma omp for schedule (static, 400)
        for (int i = 0; i<800; i++)
        {
            //printf("%d",x);
            x++;
        }
    }
    clock_t end = clock();
    secs = (double)(end-begin) / CLOCKS_PER_SEC;
    printf("\nTime taken = %f\n", secs);
    return 0;
}
```

Fig 0.1 : Parallel Version of Test Code

```
hxn@hxn:~/Desktop/PDC A2$ gcc test.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out

Time taken = 0.000002
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp test.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out

Time taken = 0.000075
```

Fig 0.3: Serial and Parallel Execution time Side by Side

As can be seen, if we increase the number of threads, the execution time increases. There are many reasons for that. Some of them are listed as follows:

1. Whenever we create threads using ***"pragma omp parallel"***, the threads are created. The time taken to create the threads is also added to the execution time of the program. It is also called as **Thread Overhead.** The more the threads created, the more the overhead. If the program is solving big computations, then this overhead is negligible but, in our case, the program is solving almost minimal computations. Hence, the thread overhead is not neglected and is added in the execution time (in milliseconds).
2. The threads also depend on the number of threads that your processor can assign. It depends on your system specs. Hence, the values will vary depending upon your system.
3. The more the number of threads, the more the context switching between them which again takes time and that additional time is again non-negligible and is added to the execution time.[1]
4. Giving each thread a little amount of work makes the overhead of the starting and the terminating threads swamp the useful work and increase the execution time. (Which is valid in our case as we are not giving the threads any significant task).[2]

[1] https://stackoverflow.com/questions/44169993/what-happens-if-you-start-too-many-threads/67743712#67743712

[2] https://www.codeguru.com/cpp/sample_chapter/article.php/c13533/Why-Too-Many-Threads-Hurts-Performance-and-What-to-do-About-It.htm

TASK 1:

SOURCE CODE:

task2.c | task3.c | i180441_task1.c | task4.c

i180441_task1.c

```c
#include <stdio.h>
#include <sys/time.h>
#include <omp.h>

#define N 1024

int main()
{
    int i, k =10;
    int a[10] = {0,1,2,3,4,5,6,7,8,9};
    int c[1000];
    int b[N][N];
    int loc = -1;
    int tmp= -1;

    struct timeval start,stop;
    double secs = 0;
    gettimeofday(&start, NULL);

    for (i =0; i<k; i++)
    {
        b[i][k] = b[a[i]][k];
    }
    printf("%d %d", a[0], b[0][0]);

    for (i = 0; i<1000; i++)
    {
        tmp = tmp+1;
        c[i] = tmp;
    }

    for (i=0; i<1000; i++)
    {
        if (c[i]%4 == 0)
        {
            loc = i;
        }
    }

    gettimeofday(&stop, NULL);
    secs = (double)(stop.tv_usec - start.tv_usec) / 1000 + (double)(stop.tv_sec - start.tv_sec);


    printf("\nTime taken = %f\n", secs);
    return 0;
}
```

Fig 1.1: Source code provided

TIME TAKEN BY SOURCE CODE:

```
hxn@hxn:~/Desktop/PDC A2$ gcc i180441_task1.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
0 0
Time taken = 0.000070
```

Fig 1.2: Execution time of the source code

1(a) Dependency Analysis:

```
#define N 1024

int main()
{
        int i, k=10;
        int a[10]={0,1,2,3,4,5,6,7,8,9};
        int c[1000];
        int b[N][N];
        int loc=-1;
        int tmp=-1;

        for(i=0;i<k;i++)
                b[i][k]=b[a[i]][k];   Loop Independent

        printf("%d %d",a[0],b[0][0]);


        for(i=0;i<1000;i++)
        {
                tmp = tmp+1;          Flow (True) Dependency
                c[i]=tmp;             Cannot be Resolved
                                      Loop Independent
        }

        for(i=0;i<1000;i++)
                if (c[i]%4==0)
                        loc = i;      Loop Independent

        return 0;
}
```

Fig 1.3: Dependency Analysis

As can be seen from Fig 1.3, all 3 statements are loop-independent whereas there is a ***Flow Dependency*** in the second loop. This dependency can also be called ***Read after Write*** dependency and it's the only True dependency, hence cannot be resolved.

Distance Vector:

The distance vector of this code is **zero** as the value of i will be the same on both sides of equality. Hence, sink – source = 0.

Direction Vector:

The direction vector of this code will be **(=)** as it is loop independent and the distance vector is 0.

1(b) Parallelization:

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define N 1024

int main()
{
    int i, k =10;
    int a[10] = {0,1,2,3,4,5,6,7,8,9};
    int c[1000];
    int b[N][N];
    int loc = -1;
    int tmp= -1;
    double secs = 0;
    clock_t begin = clock();

    for (i =0; i<k; i++){
        b[i][k] = b[a[i]][k];
    }
    printf("%d %d", a[0], b[0][0]);

    for (i = 0; i<1000; i++){
        tmp = tmp+1;
        c[i] = tmp;
    }

    #pragma omp parallel num_threads(2)
    {
        #pragma omp for schedule(static,500)
        for (i=0; i<1000; i++){
            if (c[i]%4 == 0){
                loc = i;
            }
        }
    }
    clock_t end = clock();
    secs = (double)(end-begin) / CLOCKS_PER_SEC;
    printf("\nTime taken = %f\n", secs);
    return 0;
}
```

Fig 1.4: Parallelized Code

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp i180441_task1.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
0 0
Time taken = 0.000051
```

Fig 1.5: Execution time of parallelized code with 2 processes

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp i180441_task1.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
0 0
Time taken = 0.000095
```

Fig 1.6: Execution time of parallelized code with 4 processes

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp i180441_task1.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
0 0
Time taken = 0.000382
```

Fig 1.7: Execution time of parallelized code with 8 processes

1(c) Performance Graph:

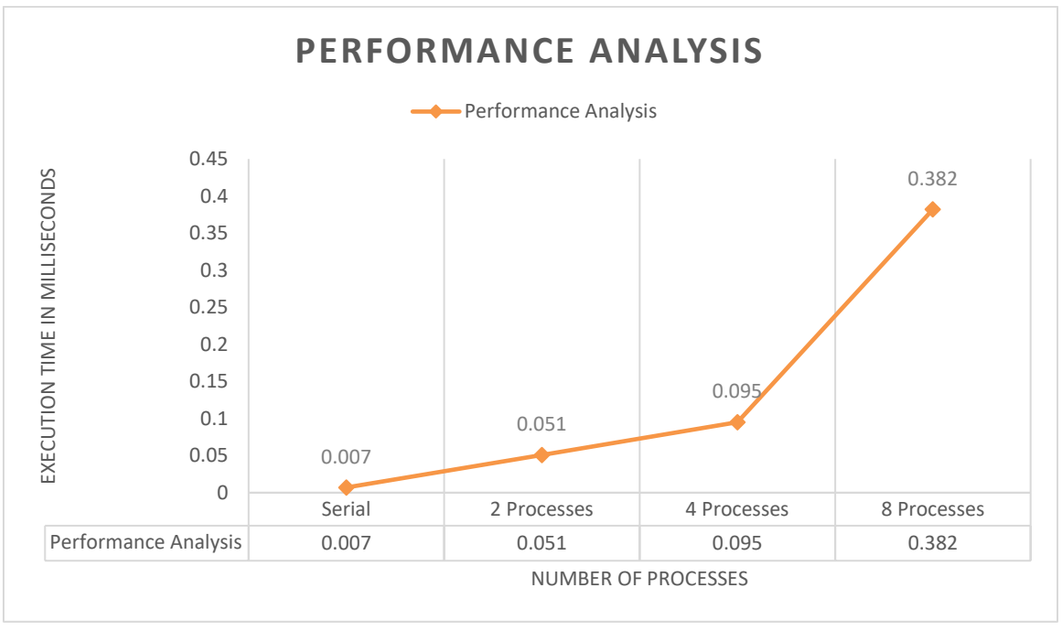

Fig 1.8: Performance Analysis Graph

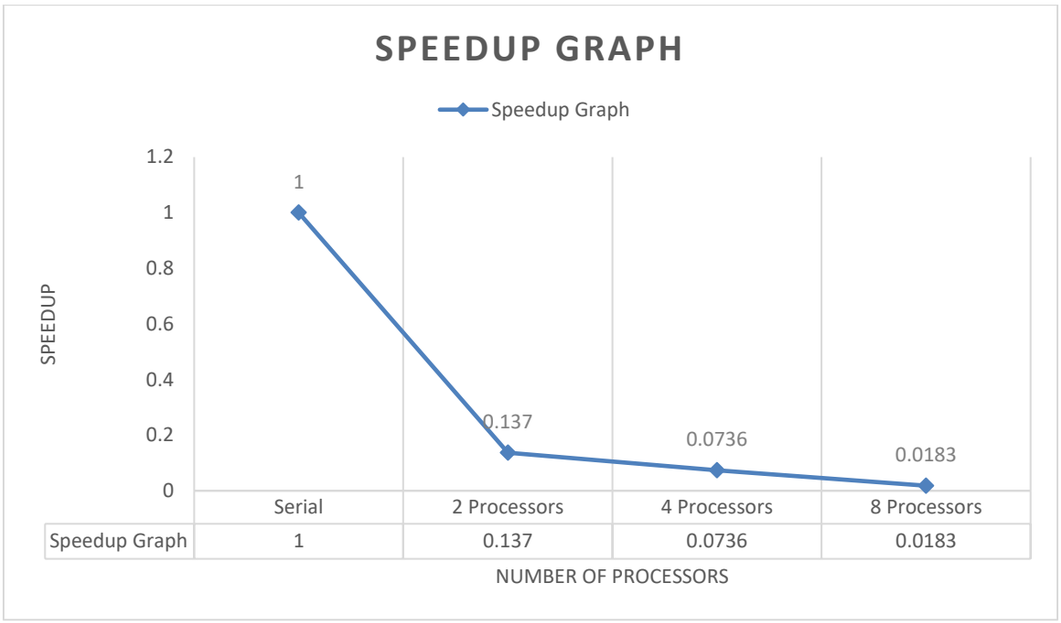

Fig 1.9: Speedup Graph

As can be seen from Fig 1.8 & Fig 1.9, as I increased the number of processes(threads) I saw an increase in the execution time of the program along with a significant decrease in its speedup. There are various reasons for that. Firstly, most of the code is serial whereas a small portion of the code is parallel. I did cross-check the outputs to see if the transformation is valid and the outputs were identical. We did not parallelize the first two loops of the code because they had no major task firstly, secondly as the computation being done there is having the time complexity of O(1). This wont change even after parallelizing; however, the time would definitely increase as the thread overhead will be added to the execution time. Various other reasons are listed above under the heading -> **Important Note**.

Please feel free to go through that.

The speedup was calculated by the following formula:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

TASK 2:

SOURCE CODE:

task2.c

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

#define N 800

int function_call(int j){
    int a;
    a = 2*2+j;
    return a;
}

int main()
{
    int i,j;
    int a[N][N];
    int b[N][N];
    int c[N][N];
    double secs = 0;
    clock_t begin = clock();

    for (i=1; i<N; i++){
        for (j=0; j<N; j++){
            b[i-1][j] = function_call(j);
        }
    }

    for (j=1; j<N-10; j++){
        for (i=0; i<N-10; i++){
            a[i][j+2] = b[i+2][j];
            c[i+1][j] = b[i][j+3];
        }
    }
    clock_t end = clock();
    secs = (double)(end-begin) / CLOCKS_PER_SEC;
    printf("\nTime taken = %f\n", secs)
    return 0;
}
```

Fig 2.1: Source code provided

TIME TAKEN BY SOURCE CODE:

```
hxn@hxn:~/Desktop/PDC A2$ gcc task2.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out

Time taken = 0.004940
```

Fig 2.2: Execution time of the source code

2(a) Dependency Analysis:

```
#include <stdio.h>
#define N 800

int function_call(int j) {
	int a;
	a=2*2+j;
	return a;
}

int main()
{
	int i,j;
	int a[N][N];
	int b[N][N];
	int c[N][N];

	for(i=1;i<=N;i++)
		for(j=0;j<N;j++)
			b[i-1][j]=function_call(j);

	for(j=0;j<N-10;j++)
		for(i=0;i<N-10;i++)
		{
			a[i][j+2] = b[i+2][j];
			c[i+1][j] = b[i][j+3];
		}

	return 0;
}
```

Loop Dependent

source sink

Loop Dependent

sink source

Fig 2.3: Dependency Analysis

As can be seen from Fig 2.3, all 2 statements are dependent hence there is loop-carried dependency in both of the nested loops as the values within depend on another iteration of the loop.

Distance Vector:

In the first nested loop, the Distance Vector will be **zero** as both the values of source and sink are same in case of j. Hence, sink – source = 0.

In the second nested loop, in the first line, the source will be on the left-hand side whereas on the second line, the source will be on the right-hand side. We will start by taking values (i=1, j=1):

First Line = (3,1) – (1,3) => **(2,-2)**

Second Line = (2,1) – (1,4) => **(1,-3)**

Direction Vector:

The direction vector of this code will be **"="** as it is loop independent and the distance vector is 0.

In second nested loop:

The first line has distance vector (2,-1). Hence, the direction vector will be **(<,>).**

The second line has distance vector (1,-3). Hence, the direction vector will be **(<,>).**

2(b) Parallelization:

task2.c

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

#define N 800

int function_call(int j){
    int a;
    a = 2*2+j;
    return a;
}

int main()
{
    int i,j;
    int a[N][N];
    int b[N][N];
    int c[N][N];
    double secs = 0;
    clock_t begin = clock();

    #pragma omp parallel num_threads(2)
    {
        for (i=1; i<N; i++){
            for (j=0; j<N; j++){
                b[i-1][j] = function_call(j);
            }
        }
    }


    #pragma omp parallel num_threads(2)
    {
        #pragma omp for schedule(static,395)
        for (j=1; j<N-10; j++){
            for (i=0; i<N-10; i++){
                a[i][j+2] = b[i+2][j];
                c[i+1][j] = b[i][j+3];
            }
        }
    }

    clock_t end = clock();
    secs = (double)(end-begin) / CLOCKS_PER_SEC;
    printf("\nTime taken = %f\n", secs);
    return 0;
}
```

Fig 2.4: Parallelized Code

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task2.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out

Time taken = 0.018006
```

Fig 2.5: Execution time of parallelized code with 2 processes

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task2.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out

Time taken = 0.077145
```

Fig 2.6: Execution time of parallelized code with 4 processes

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task2.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out

Time taken = 0.120706
```

Fig 2.7: Execution time of parallelized code with 8 processes

2(c) Performance Graph:

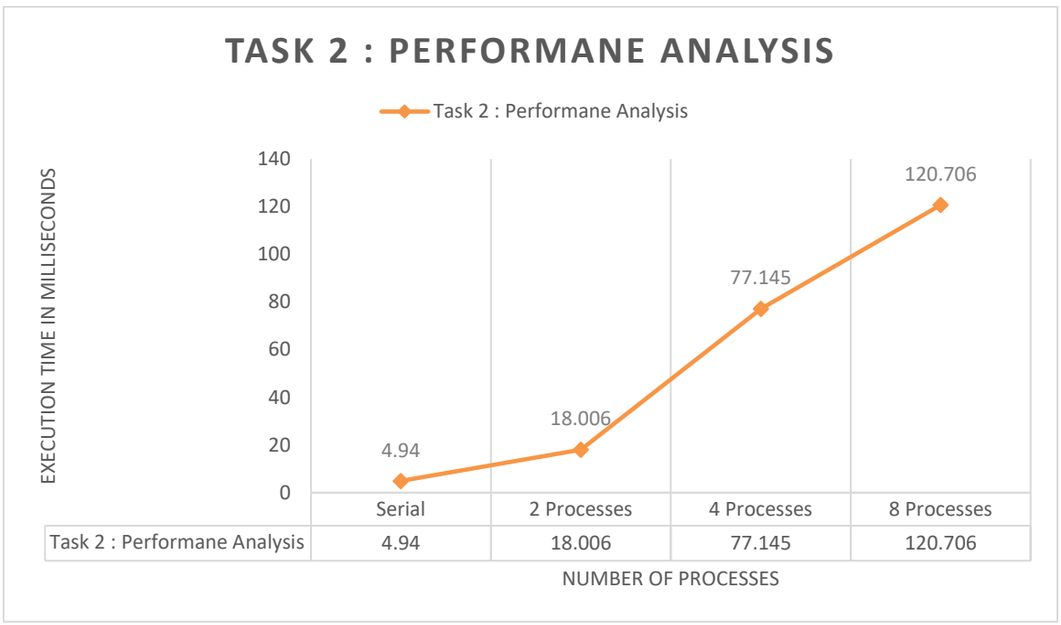

Fig 2.8: Performance Analysis Graph

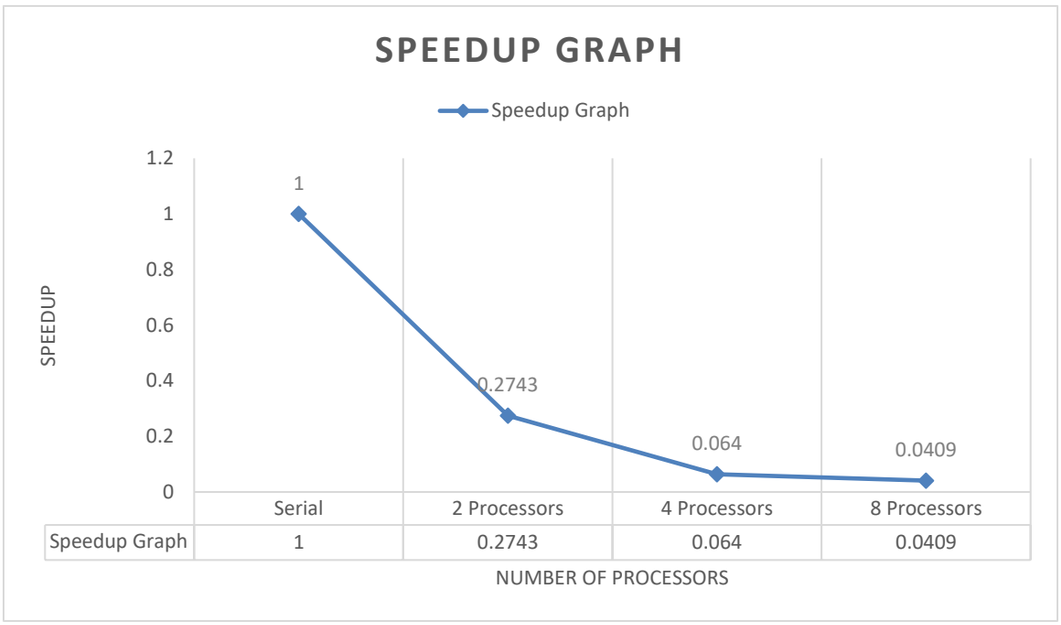

Fig 2.9: Speedup Graph

As can be seen from Fig 2.8 & Fig 2.9, as I increased the number of processes(threads) I saw an increase in the execution time of the program along with a significant decrease in its speedup . There are various reasons for that. Firstly, half of the code is serial whereas the other half is parallel. I did cross-check the outputs to see if the transformation is valid and the outputs were identical. We did not parallelize the first loop of the code because they had no major task and parallelizing wouldn't have made any useful effect to the execution time other than simply adding the thread overhead. Furthermore, the computation being done there is having the time complexity of O(1). Various other reasons are listed above under the heading => **Important Note**

Please feel free to go through that.

The speedup was calculated by the following formula:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$

TASK 3:

SOURCE CODE:

task3.c

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define N 1024

int main()
{
    int i,j;
    int X[N][N];
    int Y[N];
    int Z[N];
    int k=1;

    double secs = 0;
    clock_t begin = clock();

    for (i=0; i<N; i++)
    {
        Y[i] = k;
        k=k*2;
        Z[i] = -1;
        for (j=0; j<N; j++)
        {
            X[i][j] =2;
        }
    }

    for (i=0; i<N; i++)
    {
        for (j=0; j<N; j++)
        {
            Z[i] += Y[j] + X[i][j];
        }
    }

    clock_t end = clock();
    secs = (double)(end-begin) / CLOCKS_PER_SEC;
    printf("\nTime taken = %f\n", secs);
    return 0;
}
```

Fig 3.1: Source code provided

TIME TAKEN BY SOURCE CODE:

```
hxn@hxn:~/Desktop/PDC A2$ gcc task3.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out

Time taken = 0.004756
```

Fig 3.2: Execution time of the source code

3(a) Dependency Analysis:

```
#define N 1024

int main()
{
        int i,j;
        int X[N][N];
        int Y[N];
        int Z[N];

        int k=1;
        for(i=0;i<N;i++)
        {
                Y[i] = k;
                k = k*2;
                Z[i]=-1;

                for(j=0;j<N;j++)
                        X[i][j]=2;
        }

        for(i=0;i<N;i++)
                for(j=0;j<N;j++)
                Z[i] += Y[j] + X[i][j];

        return 0;
}
```

Anti (False) Dependency
Write after Read
Loop Independent

Loop Independent

Fig 3.3: Dependency Analysis

As can be seen from Fig 3.3, all highlighted statements are loop-independent whereas there is a ***Anti Dependency*** in the second nested-loop. This dependency can also be called ***Write after Read*** dependency and it's a False dependency, hence it can be resolved by changing variable name. But an important **NOTE** here: If we try to change the value of k in either of the lines, the value of k will stop updating. Hence, leading to a different output making the transformation invalid. Therefore, it was decided to not remove this dependency.

Distance Vector:

The distance vector of this code is **(0)** as the value of i will be the same on both sides of equality. Hence, sink – source = 0.

Direction Vector:

The direction vector of this code will be **(=)** as it is loop independent and the distance vector is 0.

3(b) Parallelization:

```c
#include <stdio.h>
#include <time.h>
#include <omp.h>
#define N 1024

int main()
{
    int i,j;
    int X[N][N];
    int Y[N];
    int Z[N];
    int k=1;

    double secs = 0;
    clock_t begin = clock();

    #pragma omp parallel num_threads(2)
    {
        #pragma omp for schedule(static,512)
        for (i=0; i<N; i++)
        {
            Y[i] = k;
            k=k*2;
            Z[i] = -1;
            for (j=0; j<N; j++)
            {
                X[i][j] =2;
            }
        }
    }

    #pragma omp parallel num_threads(2)
    {
        #pragma omp for schedule(static,512)
        for (i=0; i<N; i++)
        {
            for (j=0; j<N; j++)
            {
                Z[i] += Y[j] + X[i][j];
            }
        }
    }
    clock_t end = clock();
    secs = (double)(end-begin) / CLOCKS_PER_SEC;
    printf("\nTime taken = %f\n", secs);
    return 0;
}
```

Fig 3.4: Parallelized Code

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task3.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out

Time taken = 0.013789
```

Fig 3.5: Execution time of parallelized code with 2 processes

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task3.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out

Time taken = 0.036461
```

Fig 3.6: Execution time of parallelized code with 4 processes

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task3.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out

Time taken = 0.049252
```

Fig 3.7: Execution time of parallelized code with 8 processes

3(c) Performance Graph:

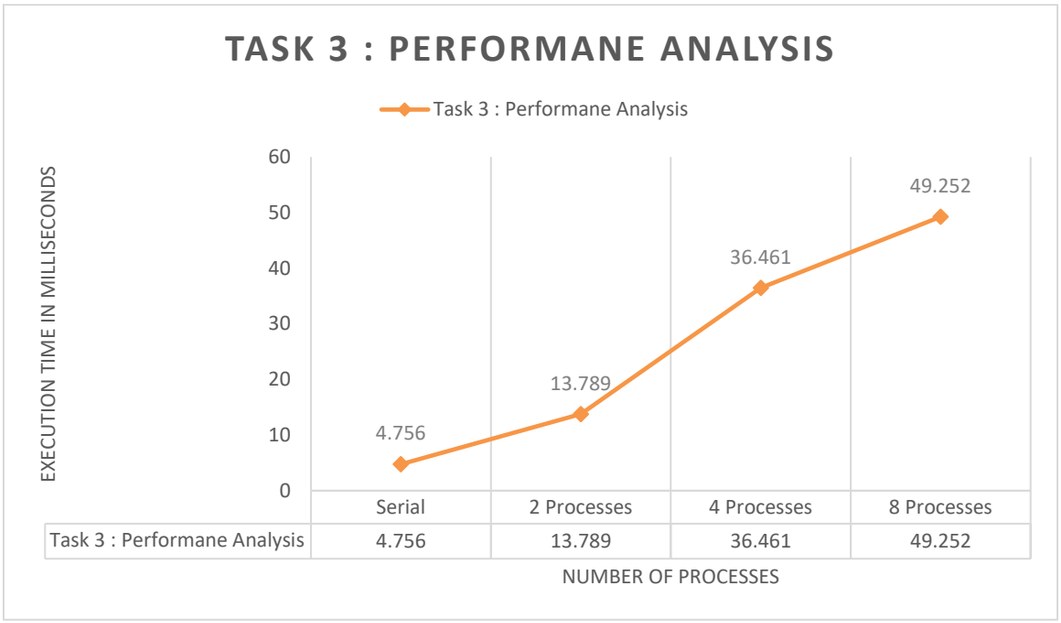

Fig 3.8: Performance Analysis Graph

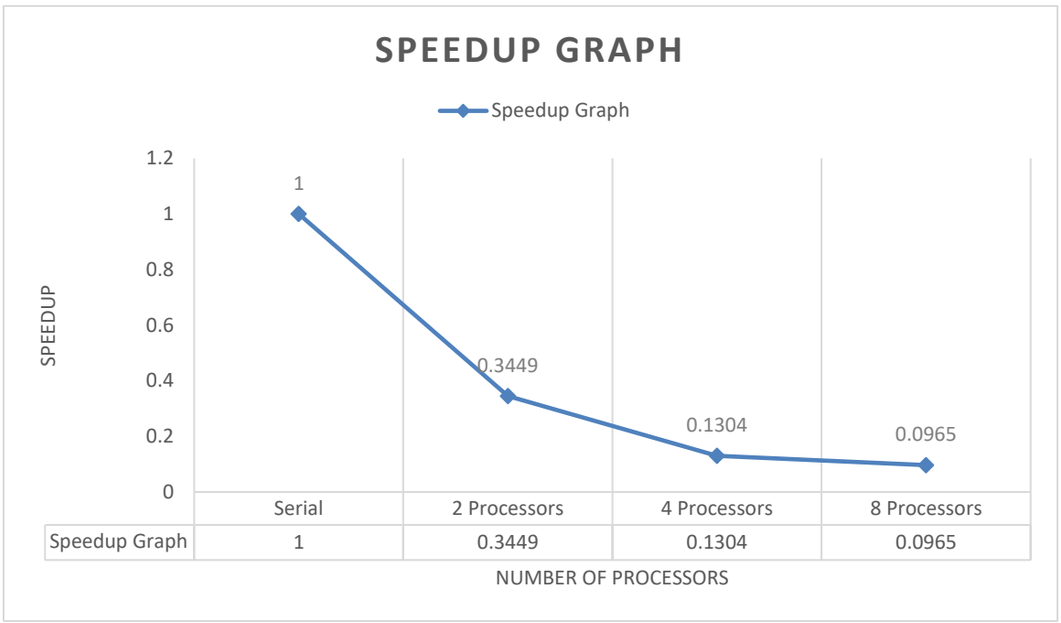

Fig 3.9: Speedup Graph

As can be seen from Fig 3.8 & Fig 3.9, as I increased the number of processes(threads) and even after most of the code being parallelized, I saw an increase in the execution time of the program and decrease in the speedup. I did cross-check the outputs to see if the transformation is valid and the outputs were identical. There are various reasons for this increase in time. Some of them are listed above and can be accessed by clicking => **Important Note**

Please feel free to go through that.

The speedup was calculated by the following formula:

$$\text{Speedup} = \frac{\textbf{Serial Time}}{\textbf{Parallel Time}}$$

TASK 4:

SOURCE CODE:

task4.c

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

int main()
{
    int m=4, n =16384;
    int i,j;

    double a[m][n];
    double s[m];

    double secs = 0;
    clock_t begin = clock();

    #pragma omp parallel for private(i,j), shared(s,a)
        for (i=0; i<m; i++)
        {
            s[i] = 1.0;
            for (j=0; j<n; j++)
            {
                s[i] = s[i] + a[i][j];
            }
        }
        printf("%f", s[1]);
    clock_t end = clock();
    secs = (double)(end-begin) / CLOCKS_PER_SEC;
    printf("\nTime taken = %f\n", secs);
    return 0;
}
```

Fig 4.1: Source code provided

TIME TAKEN BY SOURCE CODE:

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task4.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
1.000000
Time taken = 0.000371
```

Fig 4.2: Execution time of the source code (1 Process)

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task4.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
1.000000
Time taken = 0.000294
```

Fig 4.3: Execution time of the source code (2 Processes)

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task4.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
1.000000
Time taken = 0.000347
```

Fig 4.4: Execution time of the source code (4 Processes)

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task4.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
1.000000
Time taken = 0.000979
```

Fig 4.5: Execution time of the source code (8 Process)

4(a) Dependency Analysis:

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

int main()
{
	int m=4,n=16384;
	int i,j;
	double a[m][n];
	double s[m];

	#pragma omp parallel for private(i,j), shared(s,a)
	for (i=0;i<m;i++)
	{
		s[i]=1.0;
			for(j=0;j<n;j++)
				s[i]=s[i]+a[i][j];  Loop Independent
	 }
	printf("%f",s[1]);
}
```

Fig 4.3: Dependency Analysis

As can be seen from Fig 4.3, second statement is loop-independent. I had many ideas in my mind to improve this code and after a lot of testing one was selected which is discussed in => **4(c) Performance Graph:**

Distance Vector:

The distance vector of this code is **(0)** as the value of i will be the same on both sides of equality. Hence, sink – source = 0.

Direction Vector:

The direction vector of this code will be **(=)** as it is loop independent and the distance vector is 0.

4(b) Parallelization:

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

int main()
{
    int m=4, n =16384;
    int i,j;

    double a[m][n];
    double s[m];

    double secs = 0;
    clock_t begin = clock();

    #pragma omp parallel num_threads (2)
    {
        #pragma omp for schedule(static, 12)
        for (i=0; i<m; i++) // 24 iterations
        {
            s[i] = 1.0;
            for (j=0; j<n; j++) // 16384 iterations
            {
                s[i] = s[i] + a[i][j];
            }
        }

    }
    printf("%f", s[1]);
    clock_t end = clock();
    secs = (double)(end-begin) / CLOCKS_PER_SEC;
    printf("\nTime taken = %f\n", secs);
    return 0;
}
```

Fig 4.4: Parallelized Code

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task4.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
1.000000
Time taken = 0.000254
```

Fig 4.5: Execution time of parallelized code with 1 process

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task4.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
1.000000
Time taken = 0.000237
```

Fig 4.6: Execution time of parallelized code with 2 processes

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task4.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
1.000000
Time taken = 0.000272
```

Fig 4.7: Execution time of parallelized code with 4 processes

```
hxn@hxn:~/Desktop/PDC A2$ gcc -fopenmp task4.c
hxn@hxn:~/Desktop/PDC A2$ ./a.out
1.000000
Time taken = 0.000642
```

Fig 4.8: Execution time of parallelized code with 8 processes

4(c) Performance Graph:

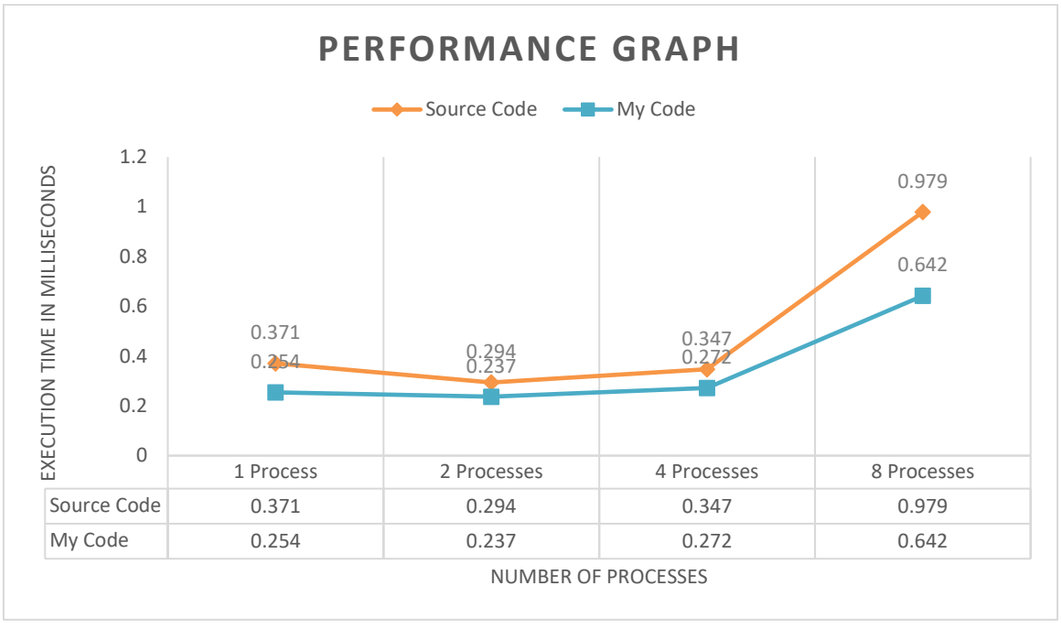

Fig 4.9: Performance Analysis Graph

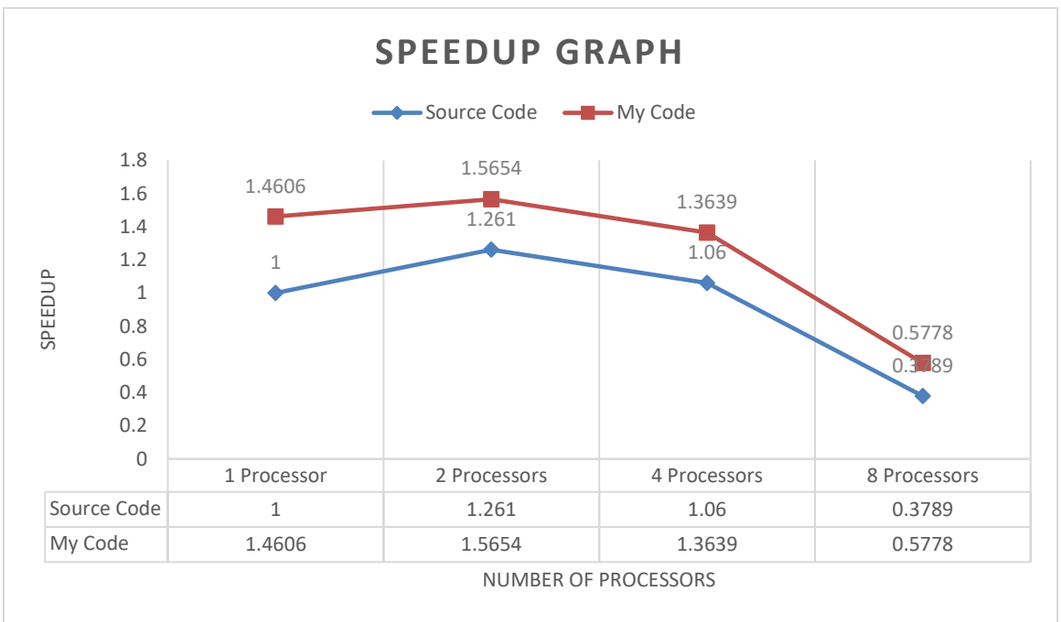

Fig 4.10: Speedup Graph

As can be seen from Fig 4.9 & Fig 4.10, as can be seen, the execution time decreased a lot after changing the code along with significant speedup. The basic idea behind changing the given code was to firstly parallelize the code along with assigning different iterations to different threads. This way those iterations can run simultaneously and hence improve the performance. But as we can see, as soon as we increase the number of threads, the execution time increases and the speedup decreases. This is due to various reasons that have been discussed above and can be seen by clicking => **Important Note**

Please feel free to go through that.

The speedup was calculated by the following formula:

$$\text{Speedup} = \frac{\text{Serial Time}}{\text{Parallel Time}}$$