# Assignment No. 3

PRN: 2020BTECS00025
Name: Sahil Santosh Otari
Course: High Performance Computing Lab
Title of practical: Study and implementation of schedule, nowait, reduction, ordered and collapse clauses.

**Problem 1:**

Analyse and implement a Parallel code for below program using OpenMP.

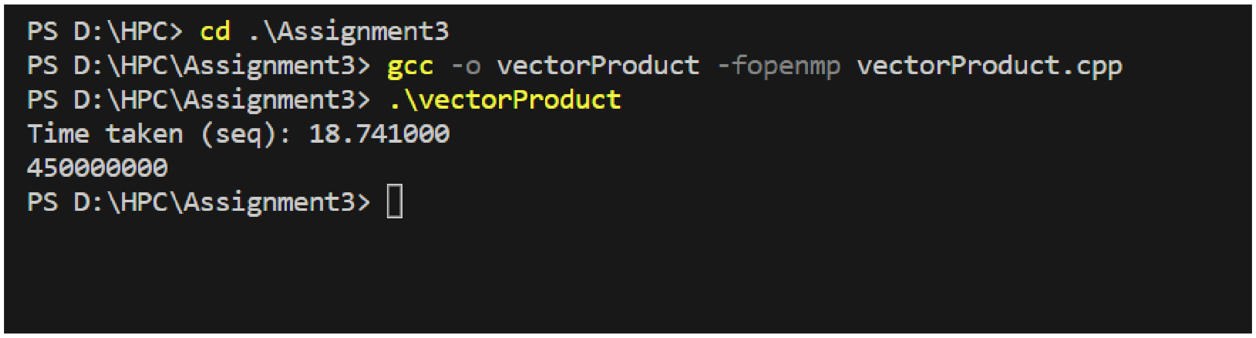
C program to find minimum product of two vectors (dot product).

Code (Sequential):

```
#include <stdio.h>

#include <time.h>

#define n 100000


void sort(int nums[]){

    int i, j;

    for (i = 0; i < n - 1; i++){

        for (j = 0; j < n - i - 1; j++){

            if (nums[j] > nums[j + 1])

            {

                int temp = nums[j];

                nums[j] = nums[j + 1];

                nums[j + 1] = temp;

            }

        }

    }

}
```

```
void sortDesc(int nums[]){
    int i, j;
    for (i = 0; i < n; i++){
        for (j = i + 1; j < n; j++) {
            if (nums[i] < nums[j]){
                int temp = nums[i];
                nums[i] = nums[j];
                nums[j] = temp;
            }
        }
    }
}

int main()
{
    int vec1[n], vec2[n];
    for (int i = 0; i < n; i++)
    {
        vec1[i] = 15;
    }
    for (int i = 0; i < n; i++)
    {
        vec2[i] = 300;
    }
    clock_t t = clock();
    sort(vec1);
    sortDesc(vec2);
    t = clock() - t;
    double time = ((double)t) / CLOCKS_PER_SEC;
    printf("Time taken (seq): %f\n", time);
    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        sum = sum + (vec1[i] * vec2[i]);
    }
    printf("%d\n", sum);
    return 0;
}
```

Output:



```
PS D:\HPC> cd .\Assignment3
PS D:\HPC\Assignment3> gcc -o vectorProduct -fopenmp vectorProduct.cpp
PS D:\HPC\Assignment3> .\vectorProduct
Time taken (seq): 18.741000
450000000
PS D:\HPC\Assignment3>
```

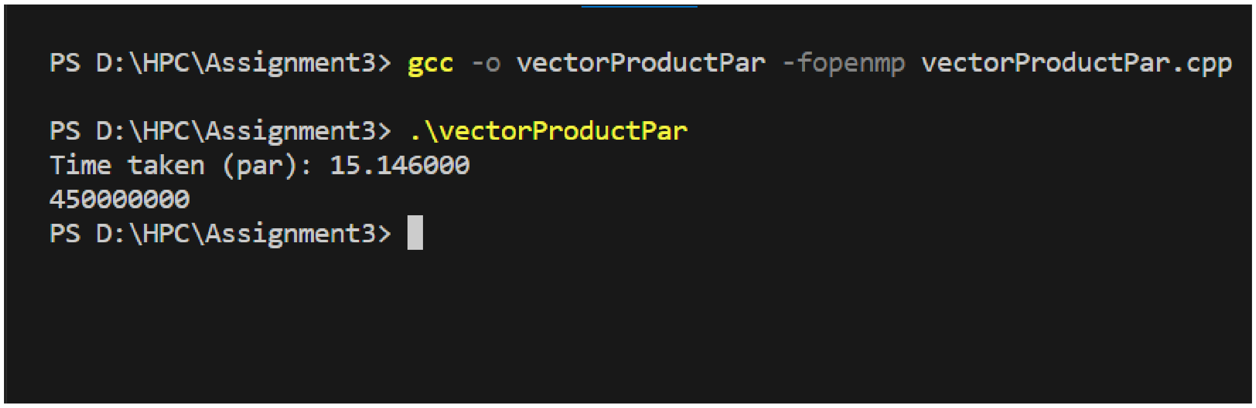
Code (Parallel):

```
#include <omp.h>
#include <stdio.h>
#include <time.h>
#define n 100000
void sort(int nums[]){
    int i, j;
```

```c
    for (i = 0; i < n; i++)
    {
        int turn = i % 2;
#pragma omp parallel for
        for (j = turn; j < n - 1; j += 2){
            if (nums[j] > nums[j + 1])
            {
                int temp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = temp;
            }
        }
    }
}
void sort_des(int nums[]){
    int i, j;
    for (i = 0; i < n; ++i){
        int turn = i % 2;
#pragma omp parallel for
        for (j = turn; j < n - 1; j += 2){
            if (nums[j] < nums[j + 1]){
                int temp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = temp;
            }
        }
    }
```

```c
}

int main()
{
    int vec1[n], vec2[n];
    for (int i = 0; i < n; i++)
    {
        vec1[i] = 15;
    }
    for (int i = 0; i < n; i++)
    {
        vec2[i] = 300;
    }
    clock_t t;
    t = clock();
    sort(vec1);
    sort_des(vec2);
    t = clock() - t;

    double time_taken = ((double)t) / CLOCKS_PER_SEC;
    printf("Time taken (par): %f\n", time_taken);
    int sum = 0;
    for (int i = 0; i < n; i++)
    {
        sum = sum + (vec1[i] * vec2[i]);
    }
    printf("%d\n", sum);
    return 0;

}
```

Output:



```
PS D:\HPC\Assignment3> gcc -o vectorProductPar -fopenmp vectorProductPar.cpp

PS D:\HPC\Assignment3> .\vectorProductPar
Time taken (par): 15.146000
450000000
PS D:\HPC\Assignment3> 
```

## Information:

### Key Features:

**1. Parallel Sorting:** The code employs parallel sorting techniques using OpenMP to sort two arrays (`nums1` and `nums2`) in both ascending and descending order. Parallel sorting can be beneficial for large arrays on multi-core processors.

**2. Timing Measurement:** The code measures the time taken for both sorting operations and calculates the total time elapsed.

**3. Vector Dot Product:** It computes the dot product of the two sorted arrays after sorting and calculates the scalar product.

## Analysis:

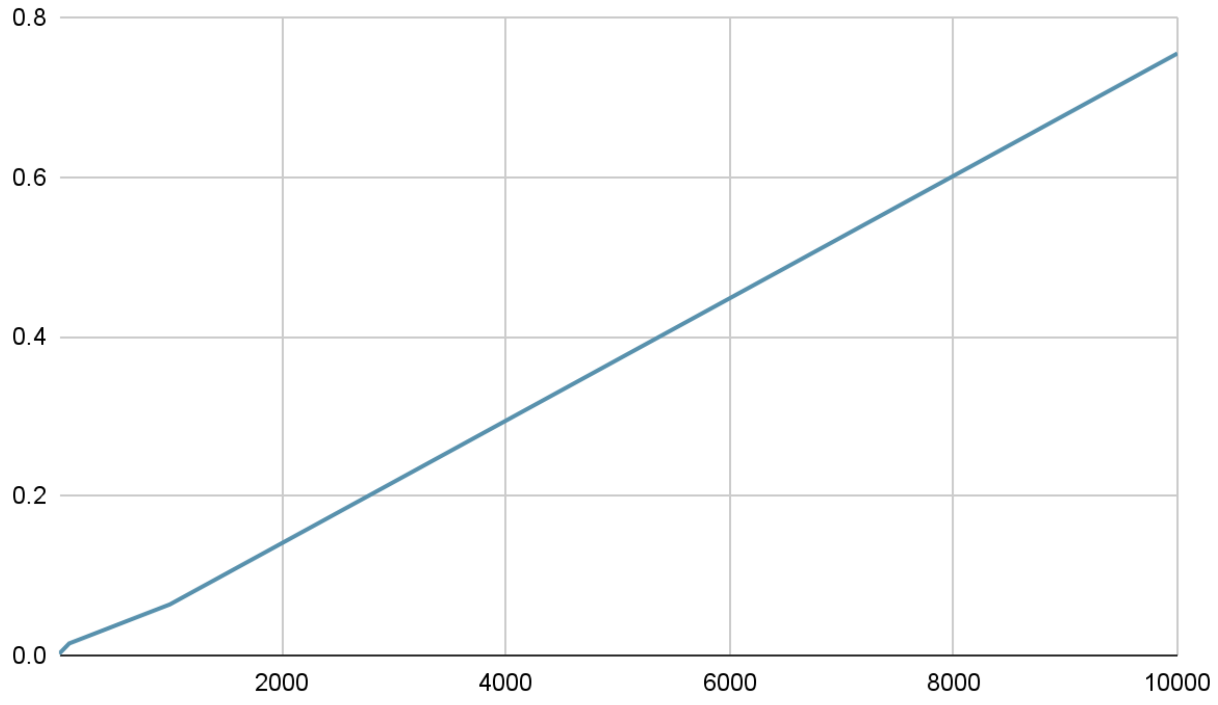
- **Parallel Sorting:** Parallel sorting is a suitable approach for optimizing performance when dealing with large arrays. In this code, sorting is parallelized using Open MP directives, which can lead to substantial speed-up on multicore processors. However, the effectiveness of parallel sorting depends on the size of the input data and the number of available CPU cores.
- **Timing Measurement:** The code includes timing measurements to assess the performance of the sorting operations. Timing measurements are essential for evaluating the impact of parallelism and can help identify performance bottlenecks.

- **Vector Dot Product:** After sorting, the code computes the dot product of the two sorted vectors. The dot product operation itself is not parallelized in this code, but it can potentially benefit from parallelization if needed.
- **Input Data:** The code initializes `nums1` and `nums2` arrays with constant values (10 and 20, respectively). In a real-world scenario, you would typically read data from external sources.
- **Efficiency Consideration:** The efficiency of parallel sorting depends on various factors, including the size of the arrays and the number of available CPU cores. For small arrays or single-core systems, parallel sorting might not provide a significant performance improvement.
- **Sequential vs. Parallel:** The code measures and prints the time taken for sorting and dot product calculations sequentially. It allows you to compare the performance of the parallel sorting approach against the sequential one.

Number of Threads = 4

| Input Size | 10 | 100 | 1000 | 10000 |
|---|---|---|---|---|
| Time | 0.002000 | 0.015000 | 0.064000 | 0.755000 |

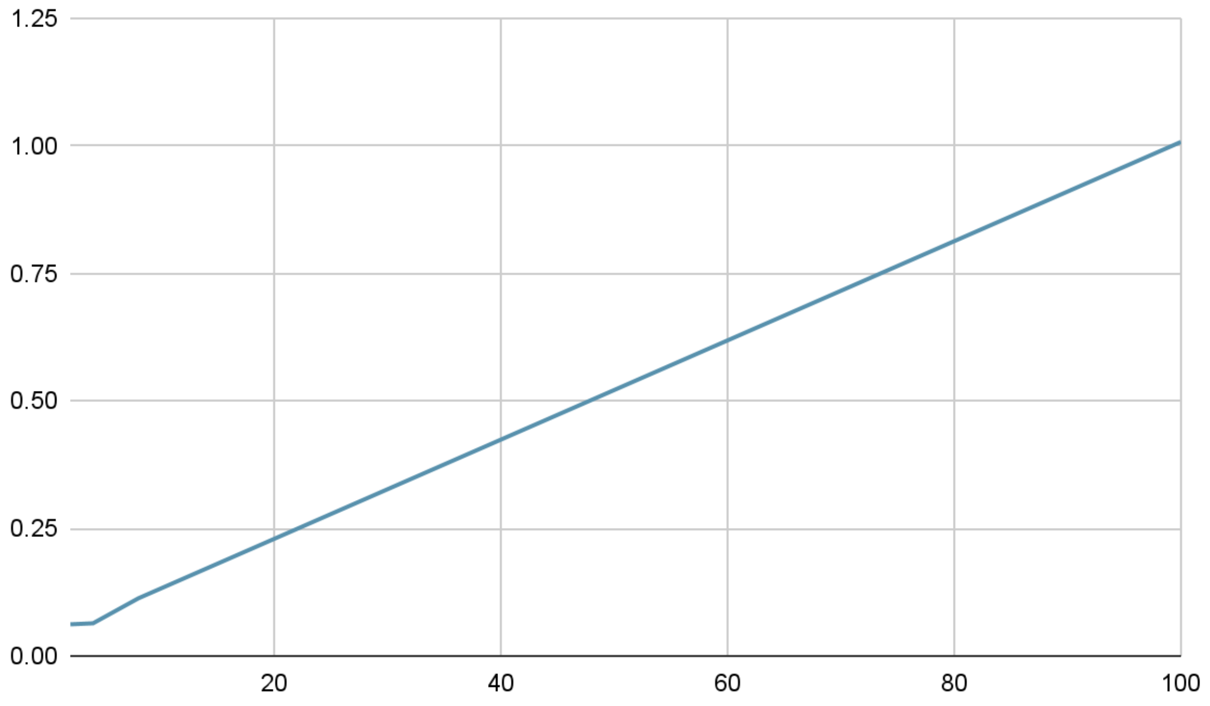
Graph:



For N = 1000

| Thread No. | 2 | 4 | 8 | 100 |
|---|---|---|---|---|
| Time | 0.062000 | 0.064000 | 0.113000 | 1.008000 |

Graph:



**Problem 2:**

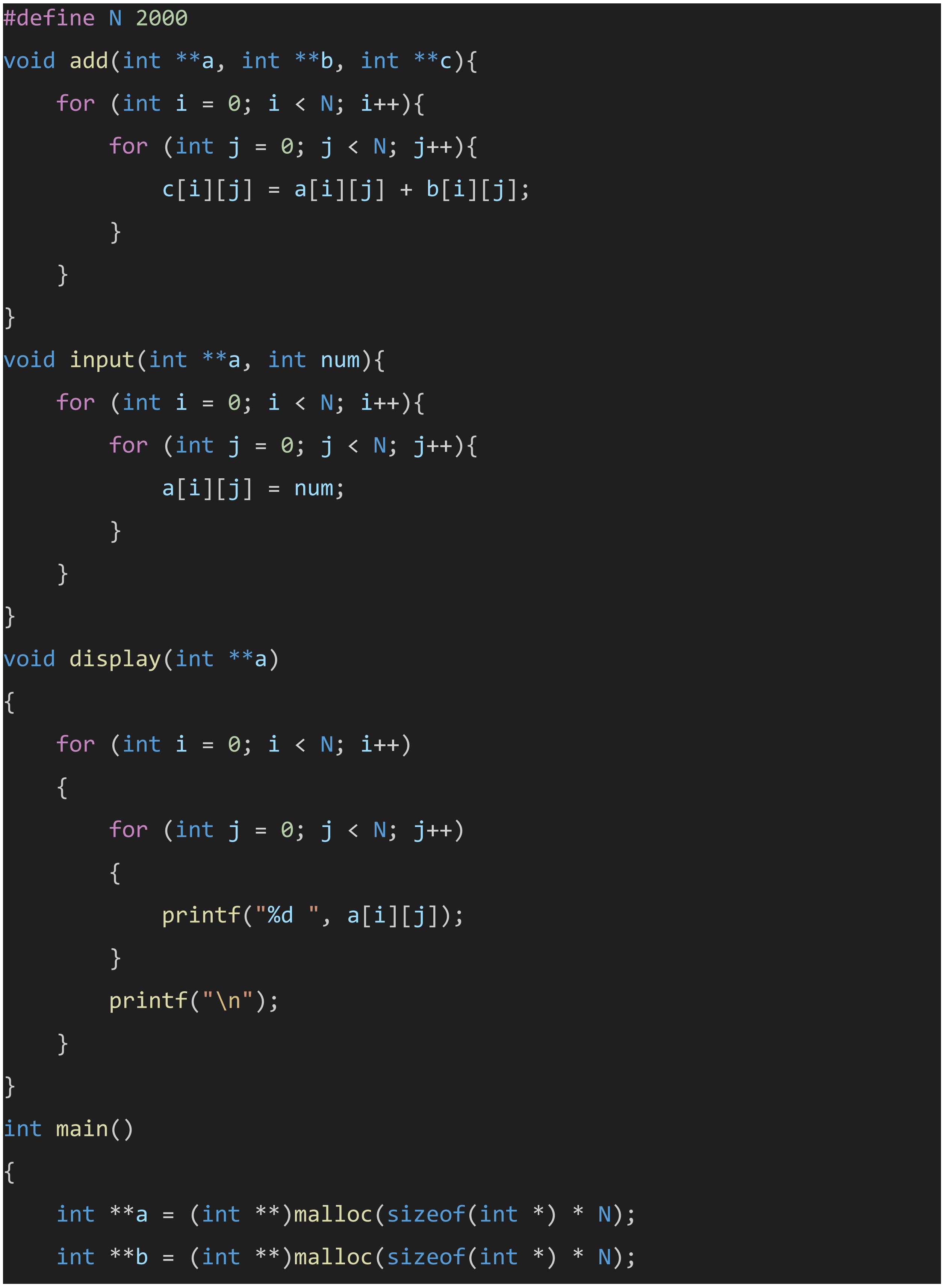
Write Open MP code for two 2D Matrix addition, vary the size of your matrices from 250, 500, 750, 1000, and 2000 and measure the runtime with one thread (Use functions in C to calculate the execution time or use GPROF)

i. For each matrix size, change the number of threads from 2,4,8., and plot the speed-up versus the number of threads.

ii. Explain whether the scaling behaviour is as expected.

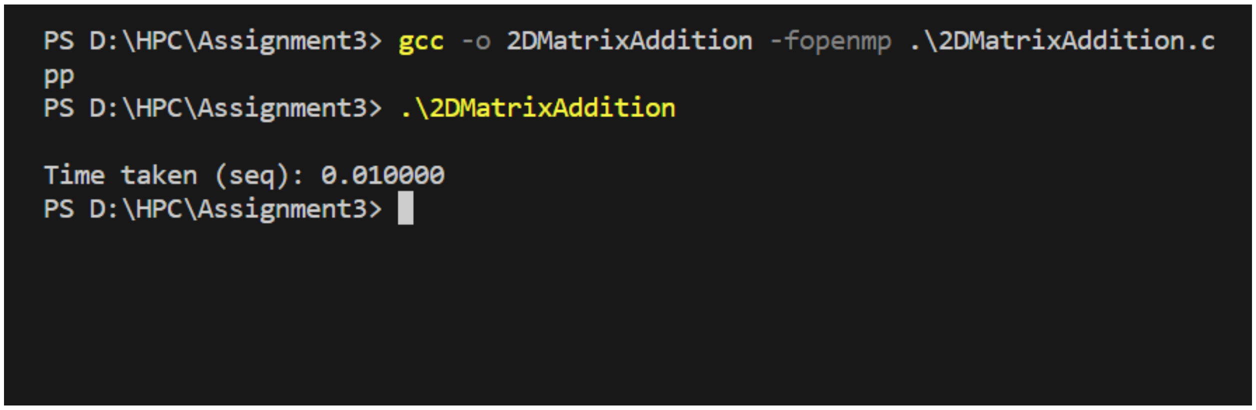
Code (Sequential):

```
#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

#include <time.h>
```



```c
#define N 2000
void add(int **a, int **b, int **c){
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            c[i][j] = a[i][j] + b[i][j];
        }
    }
}
void input(int **a, int num){
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            a[i][j] = num;
        }
    }
}
void display(int **a)
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
        {
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}
int main()
{
    int **a = (int **)malloc(sizeof(int *) * N);
    int **b = (int **)malloc(sizeof(int *) * N);
    int **c = (int **)malloc(sizeof(int *) * N);
    for (int i = 0; i < N; i++)
    {
        a[i] = (int *)malloc(sizeof(int) * N);
        b[i] = (int *)malloc(sizeof(int) * N);
        c[i] = (int *)malloc(sizeof(int) * N);
    }
    input(a, 1);
    input(b, 1);
    double startTime = omp_get_wtime();
    add(a, b, c);
    double endTime = omp_get_wtime();


    printf("\nTime taken (seq): %f\n", endTime - startTime);
}
```

Output:



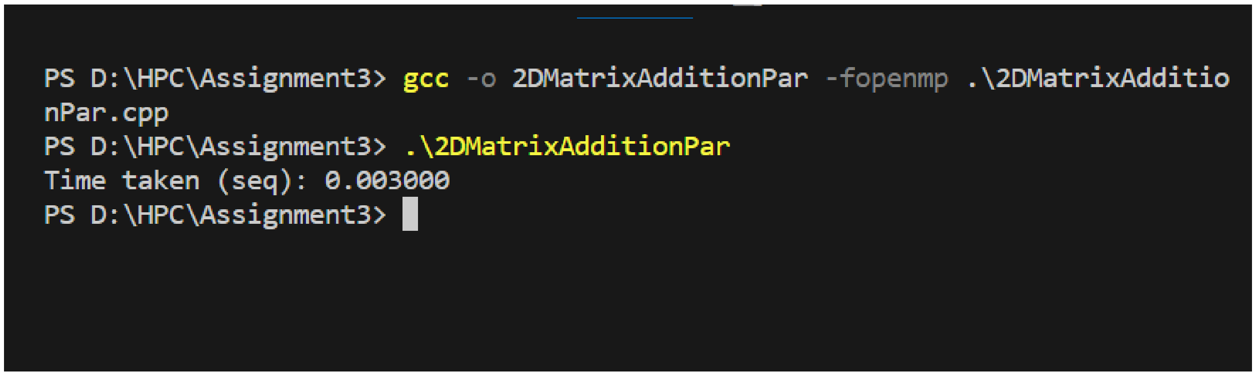
```
PS D:\HPC\Assignment3> gcc -o 2DMatrixAddition -fopenmp .\2DMatrixAddition.c
pp
PS D:\HPC\Assignment3> .\2DMatrixAddition

Time taken (seq): 0.010000
PS D:\HPC\Assignment3> 
```

Code (Parallel)

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

```c
#define N 1000
void add(int **a, int **b, int **c){
#pragma omp parallel for
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            c[i][j] = a[i][j] + b[i][j];
        }
    }
}
void input(int **a, int num){
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            a[i][j] = num;
        }
    }
}
void displayMatrix(int **a){
    for (int i = 0; i < N; i++){
        for (int j = 0; j < N; j++){
            printf("%d ", a[i][j]);
        }
        printf("\n");
    }
}

int main(){
    int **a = (int **)malloc(sizeof(int *) * N);
    int **b = (int **)malloc(sizeof(int *) * N);
    int **c = (int **)malloc(sizeof(int *) * N);
    for (int i = 0; i < N; i++){
        a[i] = (int *)malloc(sizeof(int) * N);
        b[i] = (int *)malloc(sizeof(int) * N);
        c[i] = (int *)malloc(sizeof(int) * N);
    }
    input(a, 1);
    input(b, 1);
    double startTime = omp_get_wtime();
    add(a, b, c);
    double endTime = omp_get_wtime();
    // display(c);
    printf("Time taken (Par): %f\n", endTime - startTime);
}
```

Output:



```
PS D:\HPC\Assignment3> gcc -o 2DMatrixAdditionPar -fopenmp .\2DMatrixAdditio
nPar.cpp
PS D:\HPC\Assignment3> .\2DMatrixAdditionPar
Time taken (seq): 0.003000
PS D:\HPC\Assignment3>
```

**Information and Analysis:**

This C code performs parallel matrix addition on two square matrices of size 1000x1000 using Open MP. Here's a brief analysis:

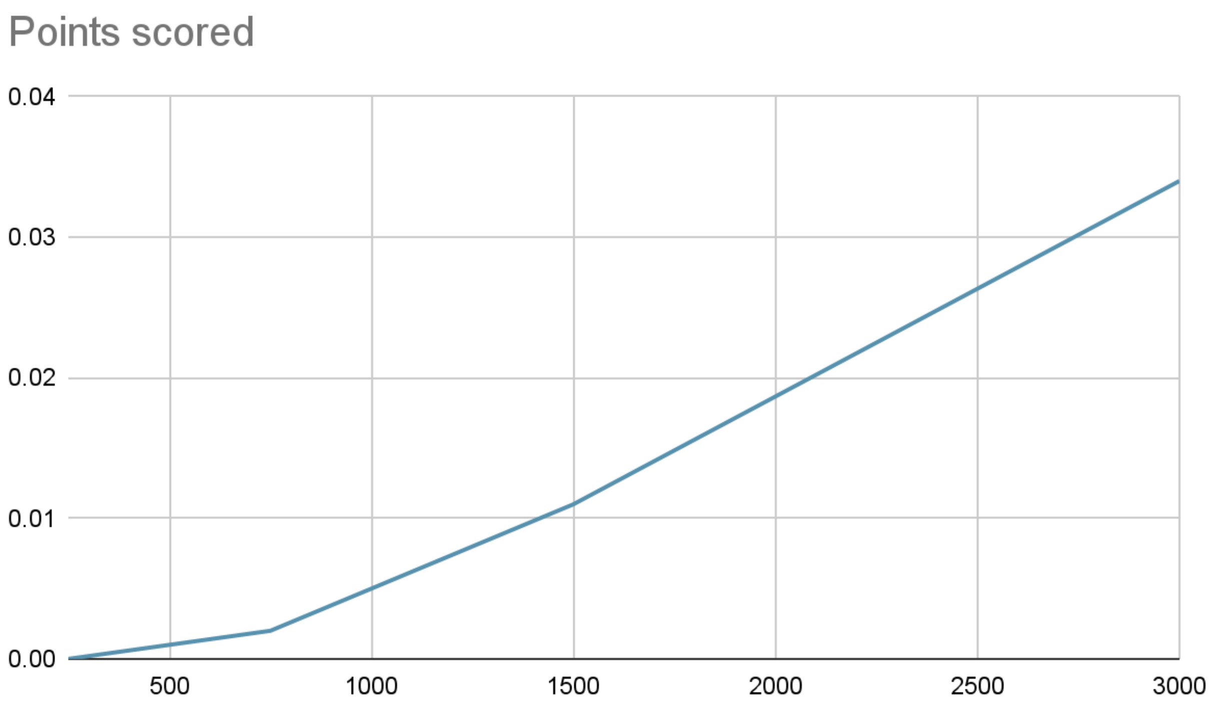
- **Parallelization (Open MP):** The code utilizes Open MP's parallelization with the `#pragma omp parallel for` directive, which distributes the work of matrix addition among multiple threads, potentially improving performance on multicore processors.
- **Matrix Operations:** Functions are provided for initializing matrices (`input`), performing matrix addition (`add`), and displaying matrices (`displayMatrix`).
- **Memory Allocation:** The code dynamically allocates memory for three matrices `a`, `b`, and `c`. However, there's no memory deallocation (`free`).

- **Matrix Size**: Matrices of size N*N are processed, which is a moderately sized workload and should be manageable in memory.
- **Execution Time Measurement:** The code measures the execution time of parallel matrix addition using `omp_get_wtime()` and prints the result to the console.
- **Potential Improvements:** Ensure proper memory deallocation with `free` to prevent memory leaks. Additionally, you can experiment with different thread numbers using Open MP directives for performance tuning.

Number of Threads = 1

| Input Size | 250 | 750 | 1500 | 3000 |
|---|---|---|---|---|
| Time | 0.000000 | 0.002000 | 0.011000 | 0.034000 |

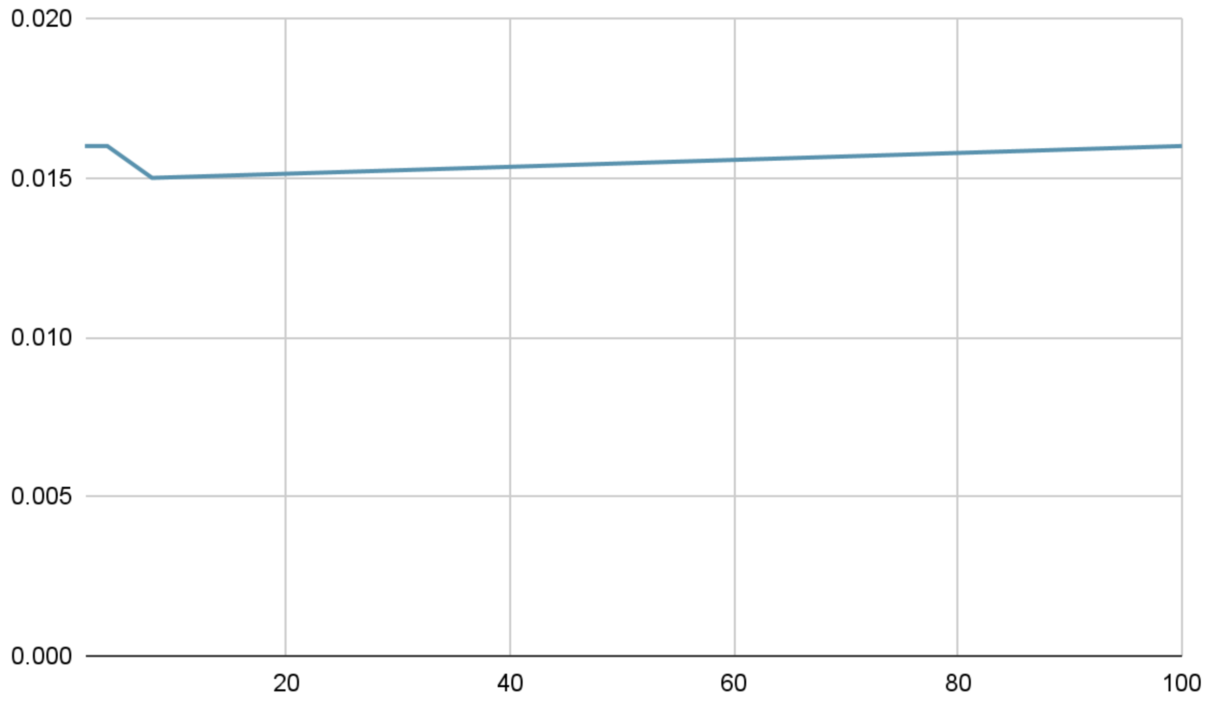
Graph:



For N = 3000

| Thread No. | 2 | 4 | 8 | 100 |
|---|---|---|---|---|
| Time | 0.016000 | 0.016000 | 0.015000 | 0.016000 |

Graph:



**Problem Statement 3:**

Q3. For 1D Vector (size=200) and scalar addition, Write an Open MP code with the
following:

i. Use the STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyse the speed-up.

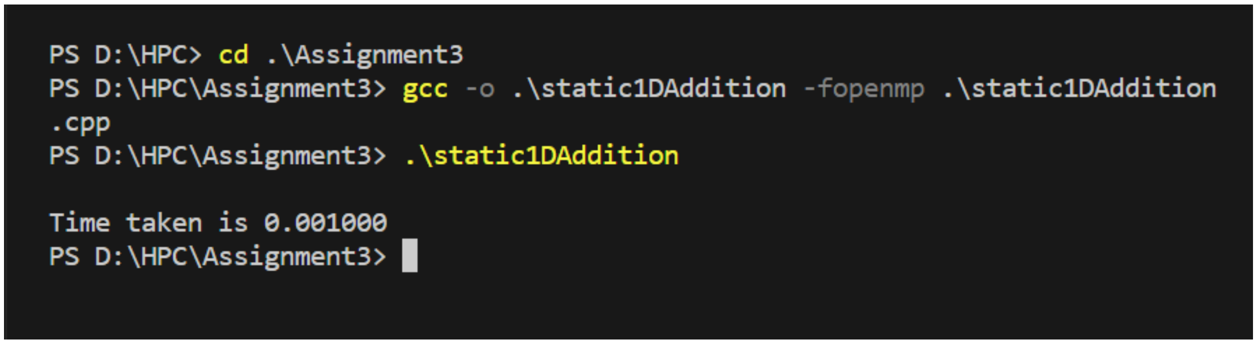
ii. Use the DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyse the speed-up.

iii. Demonstrate the use of nowait clause.

Static Schedule:

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 60
int main(){
    int *a = (int *)malloc(sizeof(int) * N);
    int *c = (int *)malloc(sizeof(int) * N);
    int b = 10;
    omp_set_num_threads(8);
    for (int i = 0; i < N; i++)
    {
        a[i] = 0;
    }
    double startTime, endTime, time;
    startTime = omp_get_wtime();
#pragma omp parallel for schedule(static, 2)
    for (int i = 0; i < N; i++)
    {
        c[i] = a[i] + b;
    }
    endTime = omp_get_wtime();
    time = endTime - startTime;
    printf("\nTime taken is %f\n", time);
}
```

Output:

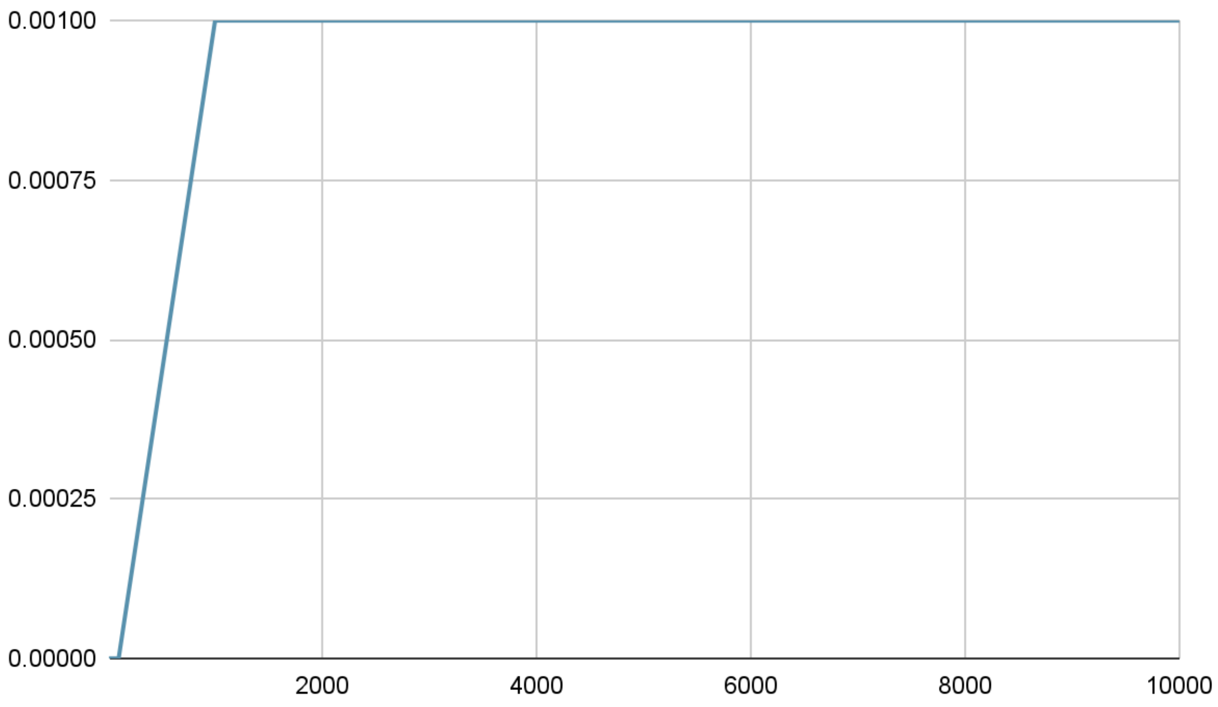


```
PS D:\HPC> cd .\Assignment3
PS D:\HPC\Assignment3> gcc -o .\static1DAddition -fopenmp .\static1DAddition
.cpp
PS D:\HPC\Assignment3> .\static1DAddition

Time taken is 0.001000
PS D:\HPC\Assignment3>
```

Number of Threads = 4

| Input Size | 10 | 100 | 1000 | 10000 |
|---|---|---|---|---|
| Time | 0.000000 | 0.000000 | 0.001000 | 0.001000 |

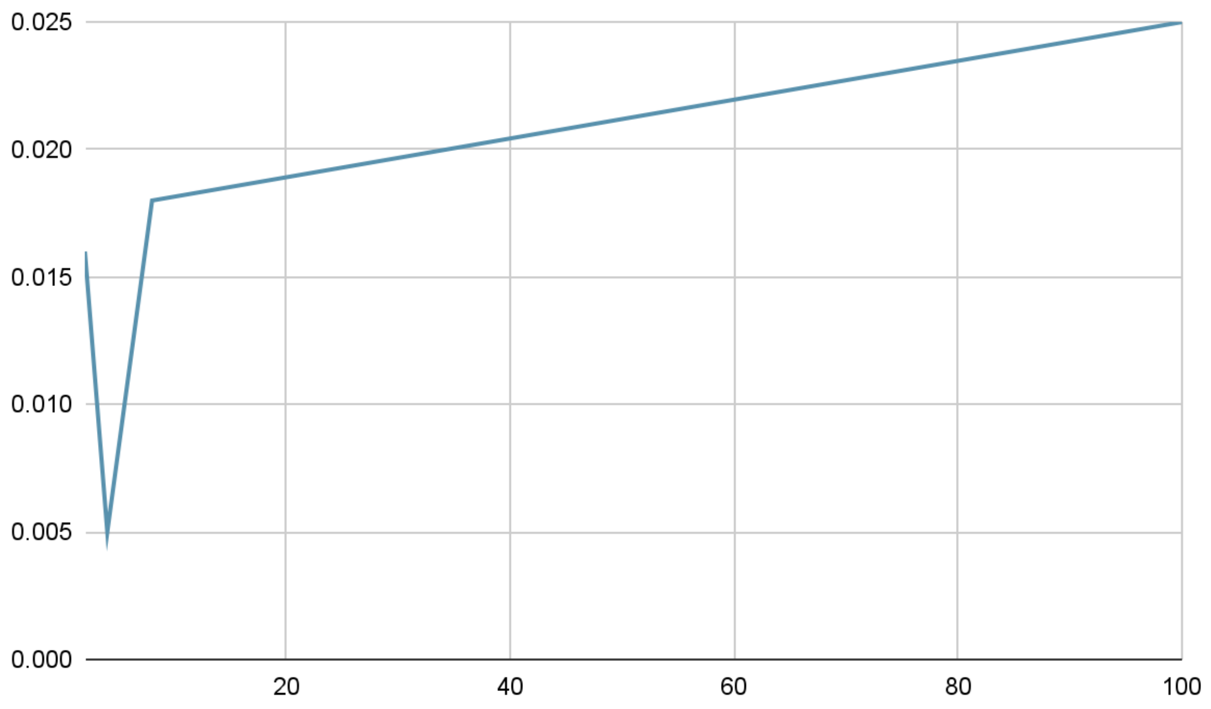
Graph:



For N = 1000000

| Thread No. | 2 | 4 | 8 | 100 |
|---|---|---|---|---|
| Time | 0.016000 | 0.005000 | 0.018000 | 0.025000 |

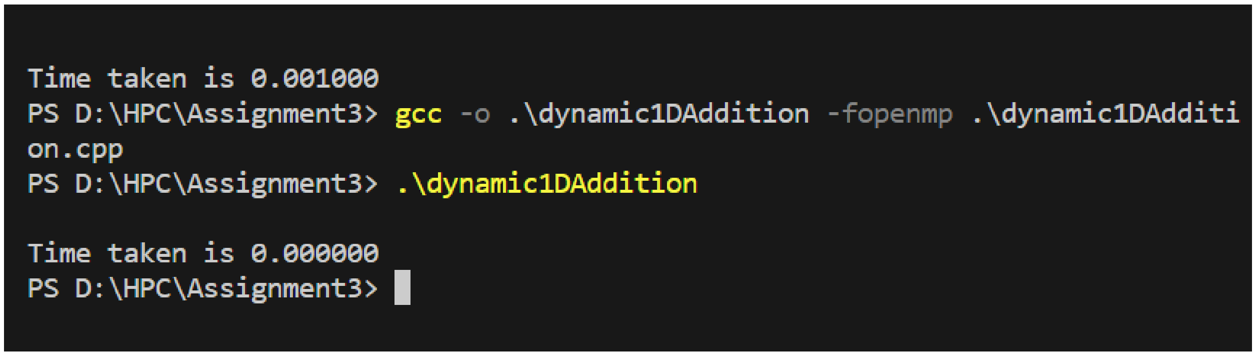
Graph:



Dynamic Schedule:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 60
int main()
{
    int *a = (int *)malloc(sizeof(int) * N);
    int *c = (int *)malloc(sizeof(int) * N);
    int b = 10;
    omp_set_num_threads(8);
    for (int i = 0; i < N; i++)
    {
        a[i] = 0;
    }
    double startTime, endTime, time;
    startTime = omp_get_wtime();
#pragma omp parallel for schedule(dynamic, 2)
    for (int i = 0; i < N; i++)
    {
        c[i] = a[i] + b;
    }
    endTime = omp_get_wtime();
    time = endTime - startTime;
    printf("\nTime taken is %f\n", time);
}
```

Output:

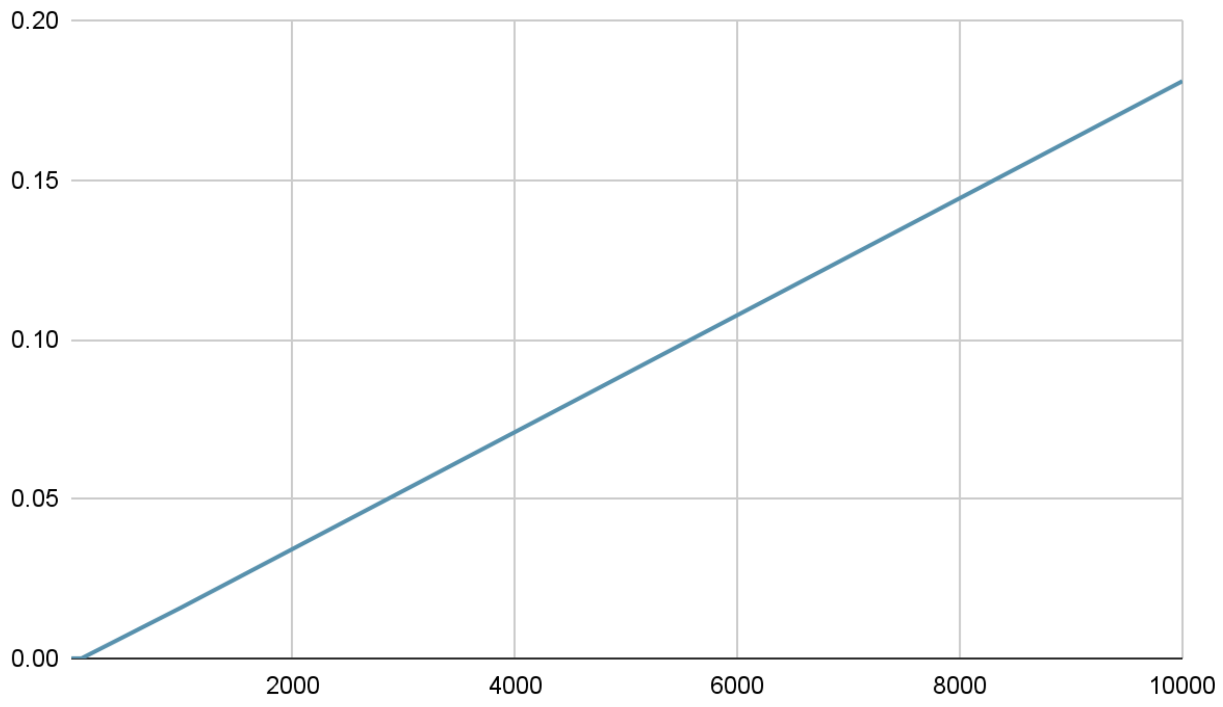


```
Time taken is 0.001000
PS D:\HPC\Assignment3> gcc -o .\dynamic1DAddition -fopenmp .\dynamic1DAdditi
on.cpp
PS D:\HPC\Assignment3> .\dynamic1DAddition

Time taken is 0.000000
PS D:\HPC\Assignment3>
```

Number of Threads = 4

| Input Size | 10 | 100 | 1000 | 10000000 |
|---|---|---|---|---|
| Time | 0.000000 | 0.000000 | 0.016000 | 0.181000 |

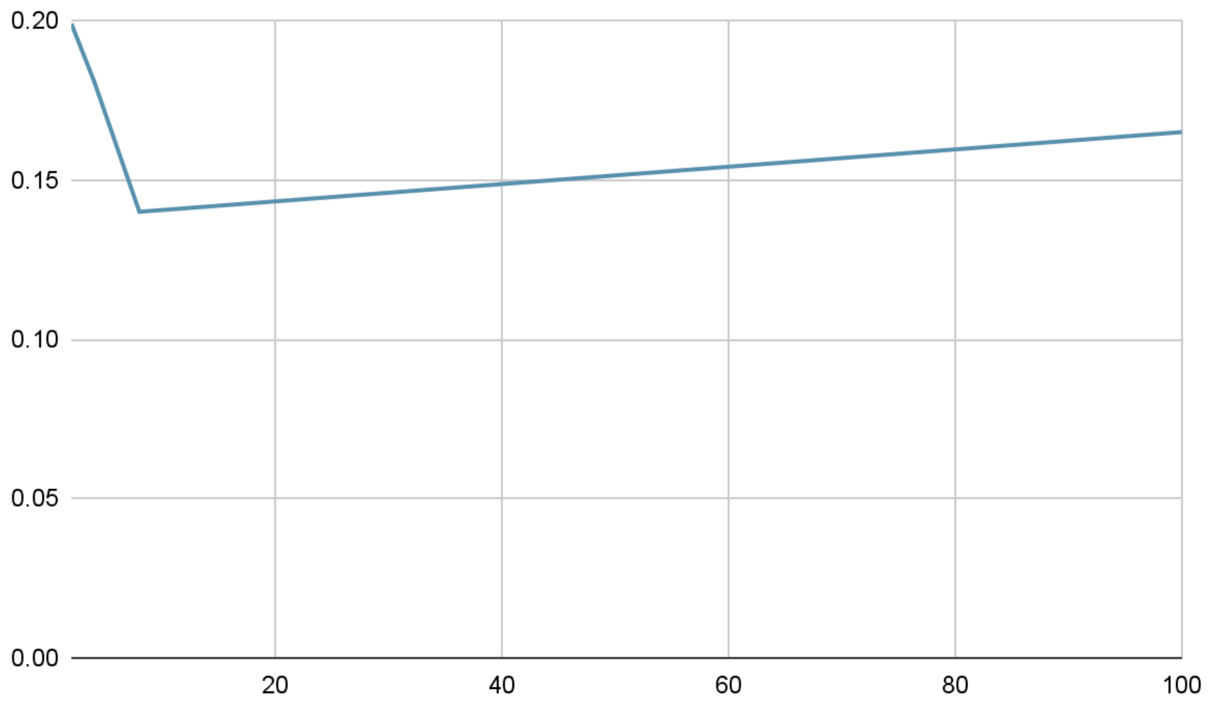
Graph:



For N = 10000000

| Thread No. | 2 | 4 | 8 | 100 |
|---|---|---|---|---|
| Time | 0.199000 | 0.181000 | 0.140000 | 0.165000 |

Graph:

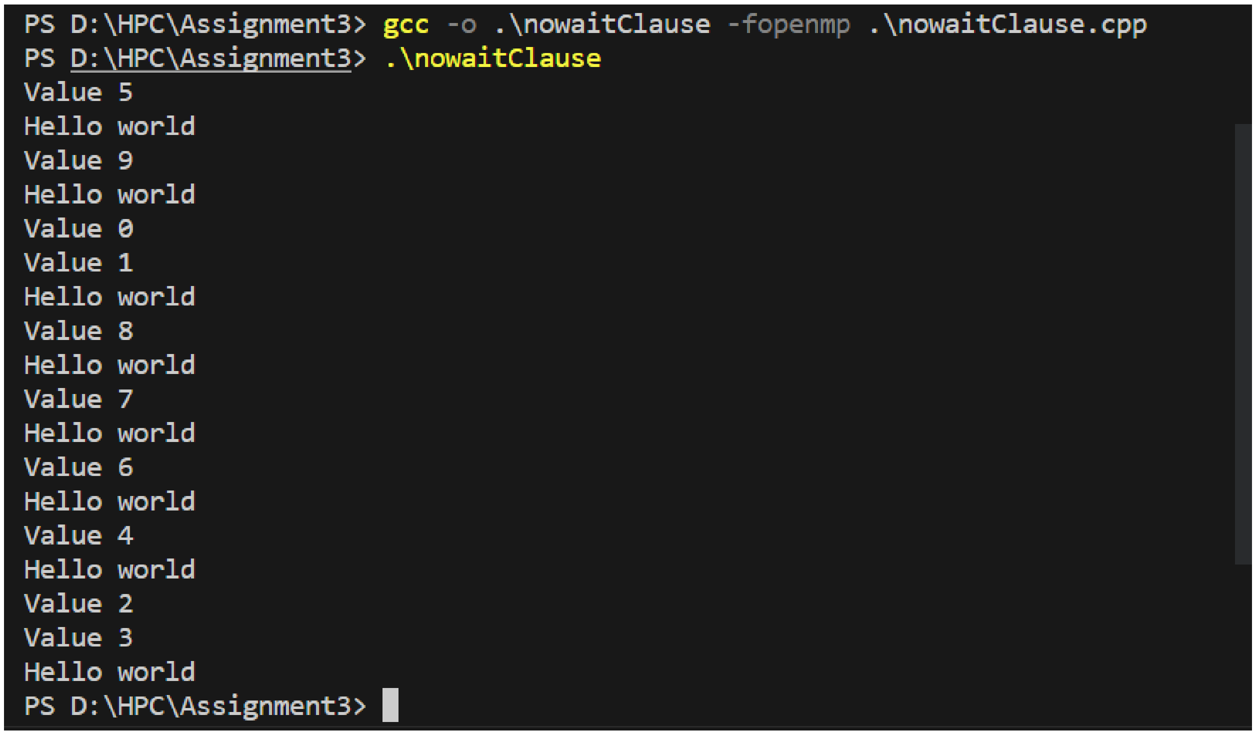


Nowait Clause:

```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 10

void hello_world(){
    printf("Hello world\n");
}
void print(int i){
    printf("Value %d\n", i);
}
int main(){
#pragma omp parallel
    {
```

```
#pragma omp for nowait

        for (int i = 0; i < N; i++){
            print(i);
        }
        hello_world();
    }
}
```

Output:



```
PS D:\HPC\Assignment3> gcc -o .\nowaitClause -fopenmp .\nowaitClause.cpp
PS D:\HPC\Assignment3> .\nowaitClause
Value 5
Hello world
Value 9
Hello world
Value 0
Value 1
Hello world
Value 8
Hello world
Value 7
Hello world
Value 6
Hello world
Value 4
Hello world
Value 2
Value 3
Hello world
PS D:\HPC\Assignment3>
```

**Information and Analysis:**

The code utilizes Open MP directives to parallelize the loop inside the `main` function. It splits the loop into multiple threads, each printing a different value of `i`.

The parallel loop runs from 0 to N (10) and each thread calls the `print` function to display its value. The `nowait` clause indicates that threads can proceed without waiting for all iterations to complete.

After the loop, all threads execute the `hello_world` function concurrently, potentially leading to interleaved "Hello world" messages.

The code produces output with values of `i` printed by multiple threads and "Hello world" messages. The order of output may vary due to parallel execution.

This code serves as a simple example of parallelization with Open MP. For more complex tasks, you can utilize thread synchronization mechanisms to control the order of execution and prevent race conditions,

Overall, this code demonstrates how to use Open MP to create parallel regions and parallelize a loop with multiple threads, allowing for concurrent execution of tasks.