

## Practical No 3

PRN: 22520005

Name: Aftab Imtiyaj Bhadgaonkar

Batch: B6

Course: High Performance Computing Lab

### Problem Statement 1:

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

*C Program to find the minimum scalar product of two vectors (dot product)*

### Code(Sequential):

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

#define N 100000

long long int minimum_scalar_product(int *vec1, int *vec2, int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = i + 1; j < size; j++) {
            if (vec1[i] > vec1[j]) {
                int temp = vec1[i];
                vec1[i] = vec1[j];
                vec1[j] = temp;
            }
        }
    }

    for (int i = 0; i < size - 1; i++) {
        for (int j = i + 1; j < size; j++) {
            if (vec2[i] < vec2[j]) {
                int temp = vec2[i];
```

```
vec2[i] = vec2[j];
vec2[j] = temp;
}
}
}

long long int result = 0;
for (int i = 0; i < size; i++) {
    result += (long long int)vec1[i] * vec2[i];
}

return result;
}

int main() {
    int vec1[N], vec2[N];
    srand(time(NULL));

    for (int i = 0; i < N; i++) {
        vec1[i] = rand() % 100;
        vec2[i] = rand() % 100;
    }

    clock_t start = clock();
    long long int min_scalar_product = minimum_scalar_product(vec1, vec2, N);

    clock_t end = clock();
    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Sequential minimum scalar product: %lld\n", min_scalar_product);
    printf("Sequential execution time: %f seconds\n", time_taken);

    return 0;
}
```

## Screenshots:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
● aftar@Aftar:~/Desktop/HPC/Assignment 3$ gcc -o p1 p1_seq.c
● aftar@Aftar:~/Desktop/HPC/Assignment 3$ ./p1
Sequential minimum scalar product: 161208876
Sequential execution time: 15.896246 seconds
○ aftar@Aftar:~/Desktop/HPC/Assignment 3$
```

## Code(Parallel):

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

#define N 100000

void sort_ascending(int *vec, int size) {
#pragma omp parallel for
for (int i = 0; i < size - 1; i++) {
for (int j = i + 1; j < size; j++) {
if (vec[i] > vec[j]) {
int temp = vec[i];
vec[i] = vec[j];
vec[j] = temp;
}
}
}
}

void sort_descending(int *vec, int size) {
#pragma omp parallel for
for (int i = 0; i < size - 1; i++) {
for (int j = i + 1; j < size; j++) {
if (vec[i] < vec[j]) {
int temp = vec[i];
vec[i] = vec[j];
```

```
vec[j] = temp;
}
}
}
}

long long int minimum_scalar_product(int *vec1, int *vec2, int size) {
    sort_ascending(vec1, size);
    sort_descending(vec2, size);
    long long int result = 0;
    #pragma omp parallel for reduction(+:result)
    for (int i = 0; i < size; i++) {
        result += (long long int)vec1[i] * vec2[i];
    }

    return result;
}

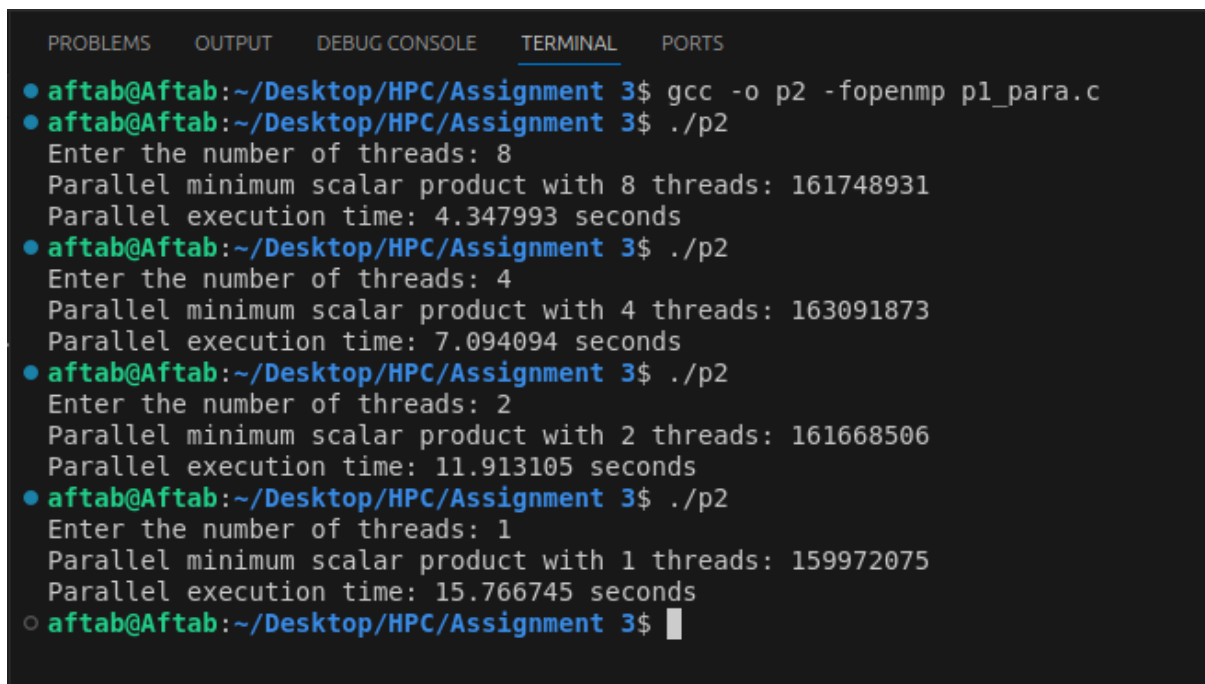
int main() {
    int vec1[N], vec2[N];
    int num_threads;
    printf("Enter the number of threads: ");
    scanf("%d", &num_threads);
    omp_set_num_threads(num_threads);

    srand(time(NULL));

    for (int i = 0; i < N; i++) {
        vec1[i] = rand() % 100;
        vec2[i] = rand() % 100;
    }
}
```

```
double start = omp_get_wtime();  
long long int min_scalar_product = minimum_scalar_product(vec1, vec2, N);  
  
double end = omp_get_wtime();  
double time_taken = end - start;  
  
printf("Parallel minimum scalar product with %d threads: %lld\n",  
num_threads, min_scalar_product);  
printf("Parallel execution time: %f seconds\n", time_taken);  
return 0;  
}
```

## Screenshots:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS  
● aftab@Aftab:~/Desktop/HPC/Assignment 3$ gcc -o p2 -fopenmp p1_para.c  
● aftab@Aftab:~/Desktop/HPC/Assignment 3$ ./p2  
Enter the number of threads: 8  
Parallel minimum scalar product with 8 threads: 161748931  
Parallel execution time: 4.347993 seconds  
● aftab@Aftab:~/Desktop/HPC/Assignment 3$ ./p2  
Enter the number of threads: 4  
Parallel minimum scalar product with 4 threads: 163091873  
Parallel execution time: 7.094094 seconds  
● aftab@Aftab:~/Desktop/HPC/Assignment 3$ ./p2  
Enter the number of threads: 2  
Parallel minimum scalar product with 2 threads: 161668506  
Parallel execution time: 11.913105 seconds  
● aftab@Aftab:~/Desktop/HPC/Assignment 3$ ./p2  
Enter the number of threads: 1  
Parallel minimum scalar product with 1 threads: 159972075  
Parallel execution time: 15.766745 seconds  
○ aftab@Aftab:~/Desktop/HPC/Assignment 3$ █
```

## Analysis:

In terms of time:

Parallel is much faster than sequential code.

Threads	Parallel	Sequential
1	15.766745	15.986246
2	11.913105	-
4	7.094094	-
8	4.347993	-

## Problem Statement 2:

Write OpenMP 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 in calculate the execution time or use GPROF)

- i. **For each matrix size, change the number of threads from 2,4,8., and plot the speedup versus the number of threads.**
- ii. **Explain whether or not the scaling behaviour is as expected.**

## Code(Sequential):

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

void initialize_matrix(int **matrix, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            matrix[i][j] = rand() % 100;
        }
    }
}

void matrix_addition_sequential(int **matrixA, int **matrixB, int **matrixC,
int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            matrixC[i][j] = matrixA[i][j] + matrixB[i][j];
        }
    }
}

int main() {
    int sizes[] = {250, 500, 750, 1000, 2000};

    for (int s = 0; s < 5; s++) {
        int size = sizes[s];
        printf("\nMatrix size: %d x %d\n", size, size);

        int **matrixA = (int **)malloc(size * sizeof(int *));
```

```
int **matrixB = (int **)malloc(size * sizeof(int *));
int **matrixC = (int **)malloc(size * sizeof(int *));
for (int i = 0; i < size; i++) {
    matrixA[i] = (int *)malloc(size * sizeof(int));
    matrixB[i] = (int *)malloc(size * sizeof(int));
    matrixC[i] = (int *)malloc(size * sizeof(int));
}

initialize_matrix(matrixA, size);
initialize_matrix(matrixB, size);

clock_t start_time = clock();
matrix_addition_sequential(matrixA, matrixB, matrixC, size);
clock_t end_time = clock();

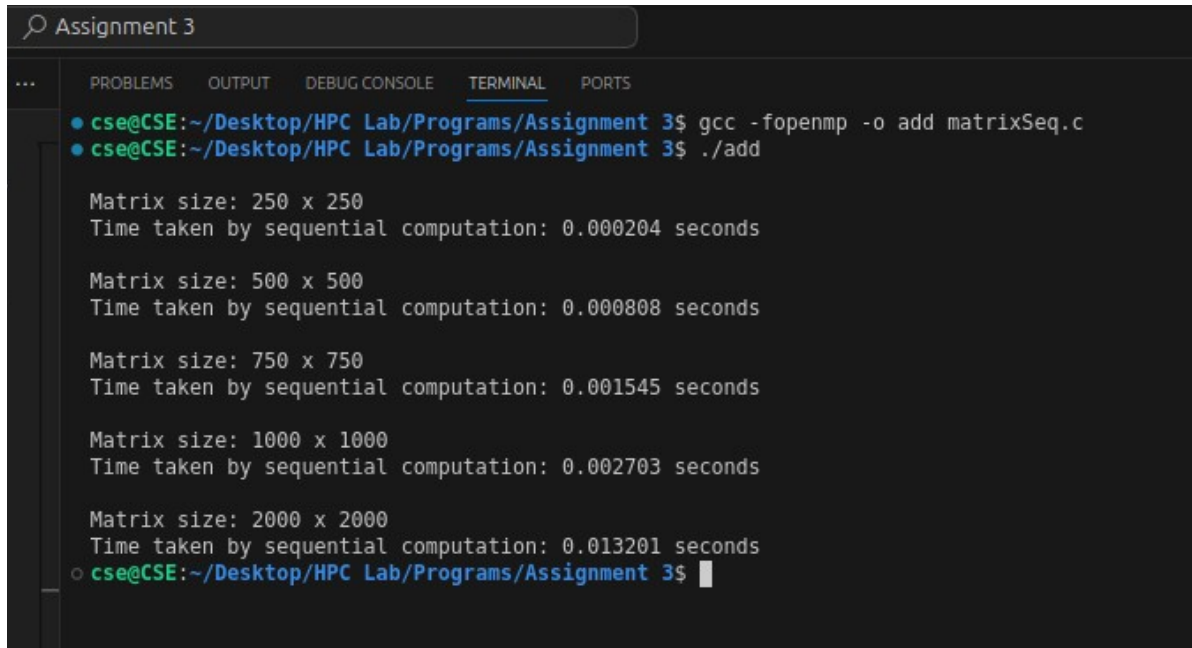
double time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;
printf("Time taken by sequential computation: %.6f seconds\n", time_taken);

for (int i = 0; i < size; i++) {
    free(matrixA[i]);
    free(matrixB[i]);
    free(matrixC[i]);
}
free(matrixA);
free(matrixB);
free(matrixC);
}

return 0;
}
```



## Screenshot:



```
Assignment 3
... PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
• cse@CSE:~/Desktop/HPC Lab/Programs/Assignment 3$ gcc -fopenmp -o add matrixSeq.c
• cse@CSE:~/Desktop/HPC Lab/Programs/Assignment 3$ ./add

Matrix size: 250 x 250
Time taken by sequential computation: 0.000204 seconds

Matrix size: 500 x 500
Time taken by sequential computation: 0.000808 seconds

Matrix size: 750 x 750
Time taken by sequential computation: 0.001545 seconds

Matrix size: 1000 x 1000
Time taken by sequential computation: 0.002703 seconds

Matrix size: 2000 x 2000
Time taken by sequential computation: 0.013201 seconds
○ cse@CSE:~/Desktop/HPC Lab/Programs/Assignment 3$
```

### **Code(Parallel):**

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

void initialize_matrix(int **matrix, int size) {
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            matrix[i][j] = rand() % 100;
        }
    }
}

void matrix_addition_parallel(int **matrixA, int **matrixB, int **matrixC, int
size, int num_threads) {
    omp_set_num_threads(num_threads);
    #pragma omp parallel for collapse(2)
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            matrixC[i][j] = matrixA[i][j] + matrixB[i][j];
        }
    }
}

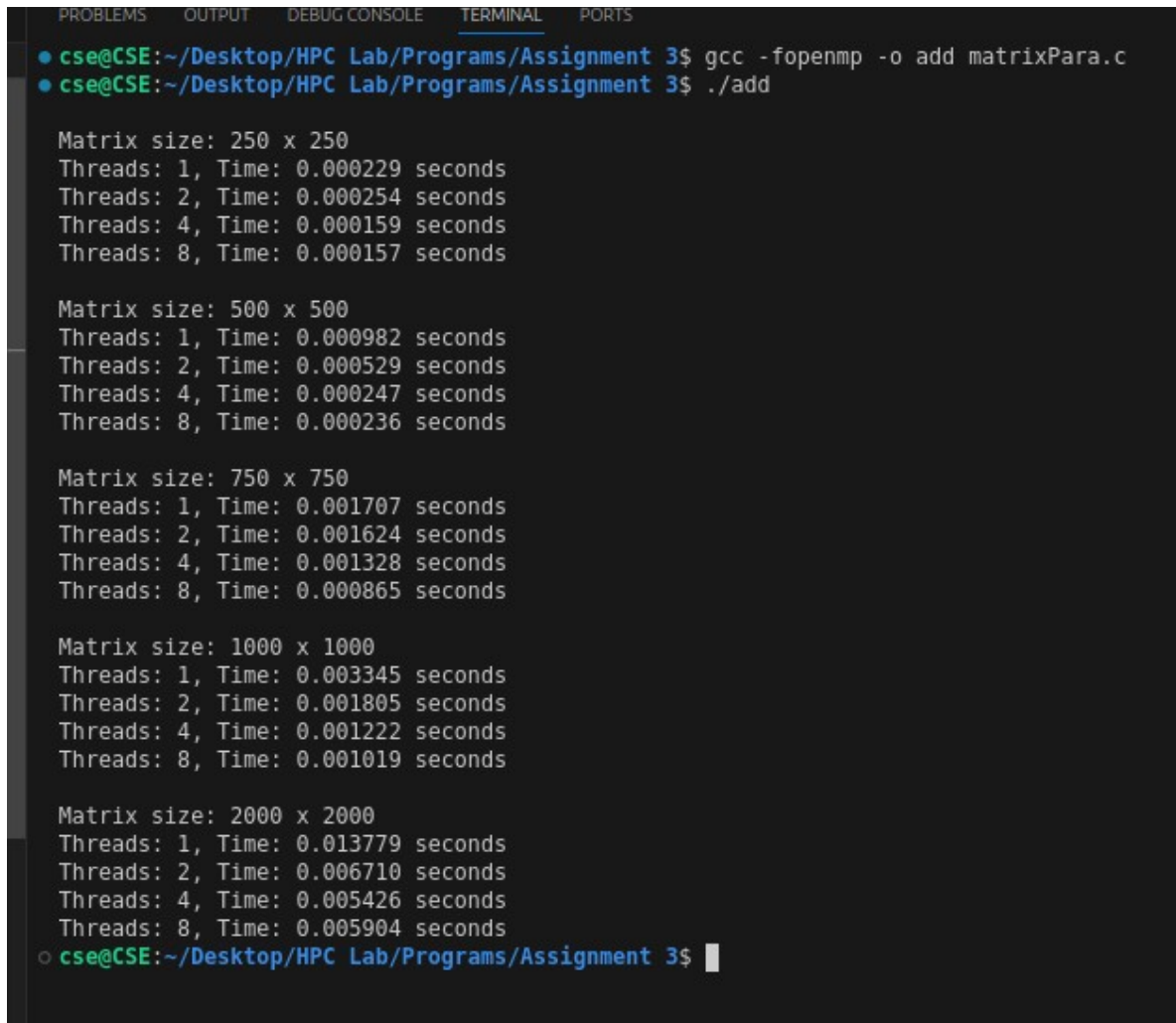
int main() {
    int sizes[] = {250, 500, 750, 1000, 2000};
    int num_threads_list[] = {1, 2, 4, 8};

    for (int s = 0; s < 5; s++) {
        int size = sizes[s];
        printf("\nMatrix size: %d x %d\n", size, size);

        int **matrixA = (int **)malloc(size * sizeof(int *));
        int **matrixB = (int **)malloc(size * sizeof(int *));
        int **matrixC = (int **)malloc(size * sizeof(int *));
        for (int i = 0; i < size; i++) {
            matrixA[i] = (int *)malloc(size * sizeof(int));
            matrixB[i] = (int *)malloc(size * sizeof(int));
```

```
matrixC[i] = (int *)malloc(size * sizeof(int));  
}  
  
initialize_matrix(matrixA, size);  
initialize_matrix(matrixB, size);  
  
for (int t = 0; t < 4; t++) {  
    int num_threads = num_threads_list[t];  
  
    double start_time = omp_get_wtime();  
    matrix_addition_parallel(matrixA, matrixB, matrixC, size, num_threads);  
    double end_time = omp_get_wtime();  
  
    printf("Threads: %d, Time: %.6f seconds\n", num_threads, end_time -  
        start_time);  
}  
  
for (int i = 0; i < size; i++) {  
    free(matrixA[i]);  
    free(matrixB[i]);  
    free(matrixC[i]);  
}  
free(matrixA);  
free(matrixB);  
free(matrixC);  
}  
  
return 0;  
}
```

## Screenshots:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
cse@CSE:~/Desktop/HPC Lab/Programs/Assignment 3$ gcc -fopenmp -o add matrixPara.c
cse@CSE:~/Desktop/HPC Lab/Programs/Assignment 3$ ./add

Matrix size: 250 x 250
Threads: 1, Time: 0.000229 seconds
Threads: 2, Time: 0.000254 seconds
Threads: 4, Time: 0.000159 seconds
Threads: 8, Time: 0.000157 seconds

Matrix size: 500 x 500
Threads: 1, Time: 0.000982 seconds
Threads: 2, Time: 0.000529 seconds
Threads: 4, Time: 0.000247 seconds
Threads: 8, Time: 0.000236 seconds

Matrix size: 750 x 750
Threads: 1, Time: 0.001707 seconds
Threads: 2, Time: 0.001624 seconds
Threads: 4, Time: 0.001328 seconds
Threads: 8, Time: 0.000865 seconds

Matrix size: 1000 x 1000
Threads: 1, Time: 0.003345 seconds
Threads: 2, Time: 0.001805 seconds
Threads: 4, Time: 0.001222 seconds
Threads: 8, Time: 0.001019 seconds

Matrix size: 2000 x 2000
Threads: 1, Time: 0.013779 seconds
Threads: 2, Time: 0.006710 seconds
Threads: 4, Time: 0.005426 seconds
Threads: 8, Time: 0.005904 seconds
cse@CSE:~/Desktop/HPC Lab/Programs/Assignment 3$
```

## Analysis:

In terms of time:

From the output, keeping the matrix size same and increasing the number of threads reduces the matrix multiplication time.

### Problem Statement 3:

For 1D Vector (size=200) and scalar addition, Write a OpenMP code with the following: i. Use STATIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. ii. Use DYNAMIC schedule and set the loop iteration chunk size to various sizes when changing the size of your matrix. Analyze the speedup. iii. Demonstrate the use of nowait clause.

### Code(Sequential):

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

#define VECTOR_SIZE 200

void init_vec(int *vec, int size) {
    for (int i = 0; i < size; i++) {
        vec[i] = rand() % 100;
    }
}

void add_static(int *vec, int scalar, int size, int chunk) {
    #pragma omp parallel for schedule(static, chunk)
    for (int i = 0; i < size; i++) {
        vec[i] += scalar;
    }
}

void add_dynamic(int *vec, int scalar, int size, int chunk) {
    #pragma omp parallel for schedule(dynamic, chunk)
    for (int i = 0; i < size; i++) {
        vec[i] += scalar;
    }
}

void add_nowait(int *vec, int scalar, int size) {
    #pragma omp parallel
    {
```

```
#pragma omp for nowait
for (int i = 0; i < size; i++) {
    vec[i] += scalar;
}
}
}

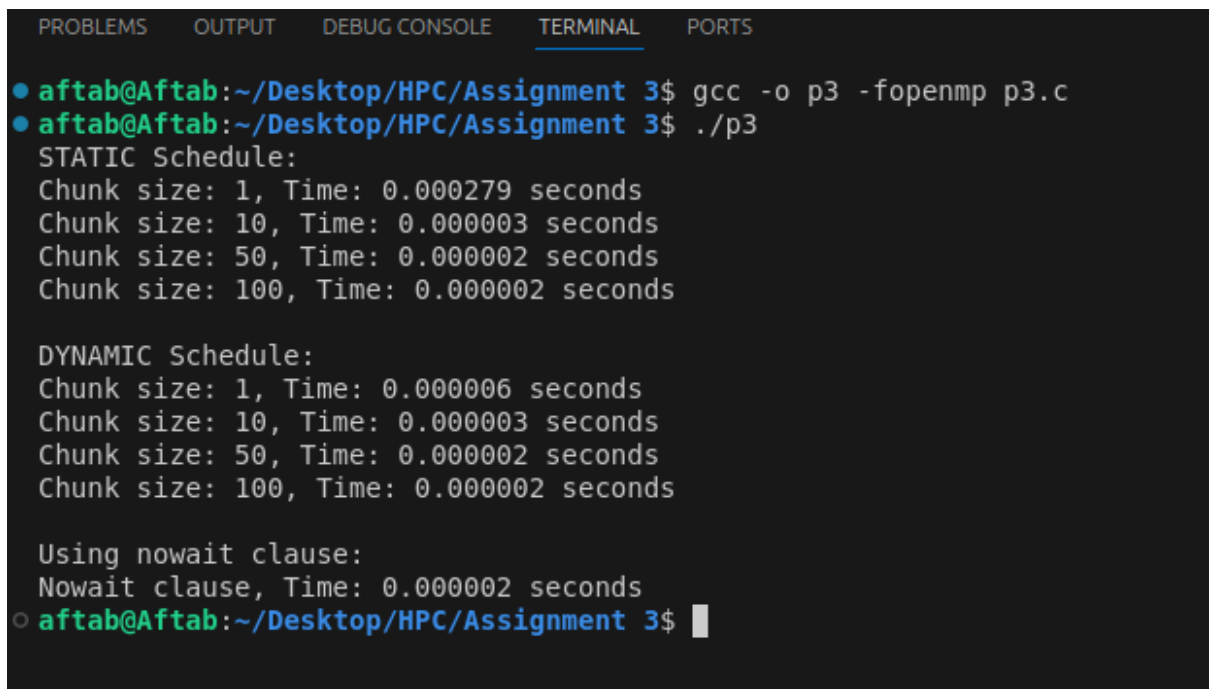
int main() {
    int vec[VECTOR_SIZE];
    int scalar = 10;
    int chunks[] = {1, 10, 50, 100};

    init_vec(vec, VECTOR_SIZE);

    printf("STATIC Schedule:\n");
    for (int i = 0; i < 4; i++) {
        double start_time = omp_get_wtime();
        add_static(vec, scalar, VECTOR_SIZE, chunks[i]);
        double end_time = omp_get_wtime();
        printf("Chunk size: %d, Time: %.6f seconds\n", chunks[i], end_time -
            start_time);
    }
    printf("\nDYNAMIC Schedule:\n");
    for (int i = 0; i < 4; i++) {
        double start_time = omp_get_wtime();
        add_dynamic(vec, scalar, VECTOR_SIZE, chunks[i]);
        double end_time = omp_get_wtime();
        printf("Chunk size: %d, Time: %.6f seconds\n", chunks[i], end_time -
            start_time);
    }
    printf("\nUsing nowait clause:\n");
    double start_time = omp_get_wtime();
    add_nowait(vec, scalar, VECTOR_SIZE);
    double end_time = omp_get_wtime();
    printf("Nowait clause, Time: %.6f seconds\n", end_time - start_time);

    return 0;
}
```

## Screenshot:



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
● aftab@Aftab:~/Desktop/HPC/Assignment 3$ gcc -o p3 -fopenmp p3.c
● aftab@Aftab:~/Desktop/HPC/Assignment 3$ ./p3
STATIC Schedule:
Chunk size: 1, Time: 0.000279 seconds
Chunk size: 10, Time: 0.000003 seconds
Chunk size: 50, Time: 0.000002 seconds
Chunk size: 100, Time: 0.000002 seconds

DYNAMIC Schedule:
Chunk size: 1, Time: 0.000006 seconds
Chunk size: 10, Time: 0.000003 seconds
Chunk size: 50, Time: 0.000002 seconds
Chunk size: 100, Time: 0.000002 seconds

Using nowait clause:
Nowait clause, Time: 0.000002 seconds
○ aftab@Aftab:~/Desktop/HPC/Assignment 3$
```

## Analysis:

With increase in a chunk size, the time required for vector scalar addition reduces significantly in case of both static and dynamic scheduling.