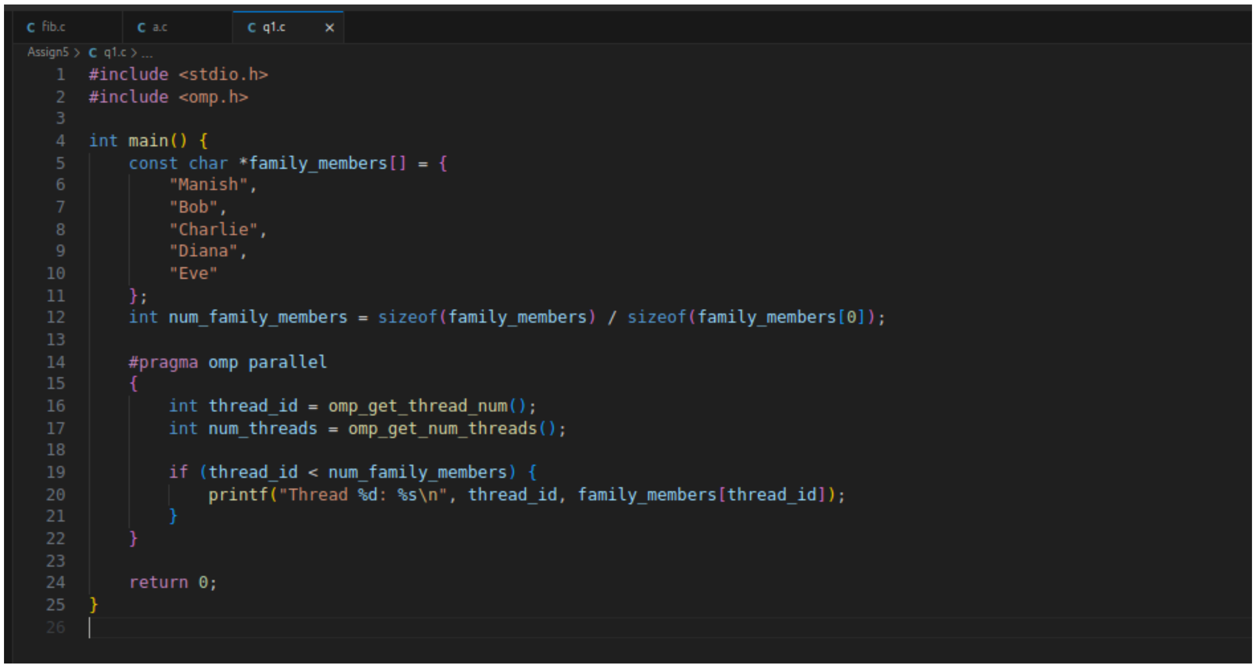
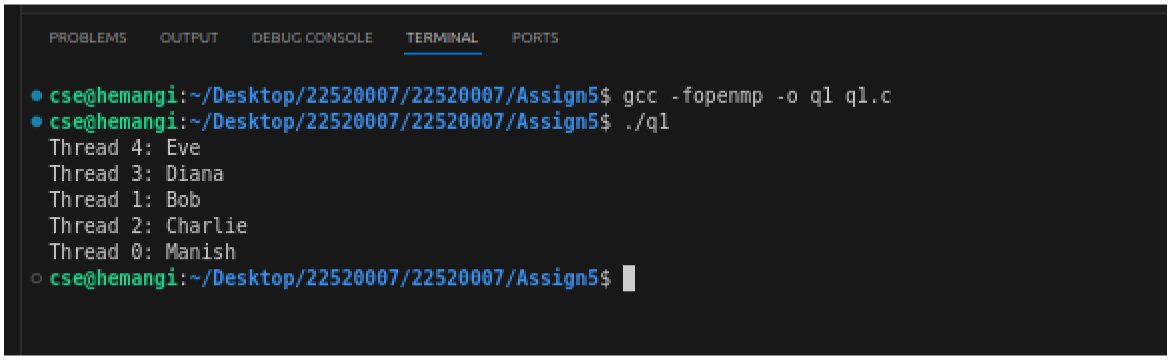
Practical No 5

Name : Manish Namdev Barage
PRN : 22520007(B6)

Q1. Write an OpenMP program such that, it should print the name of your family members, such that the names should come from different threads/cores. Also print the respective job id.



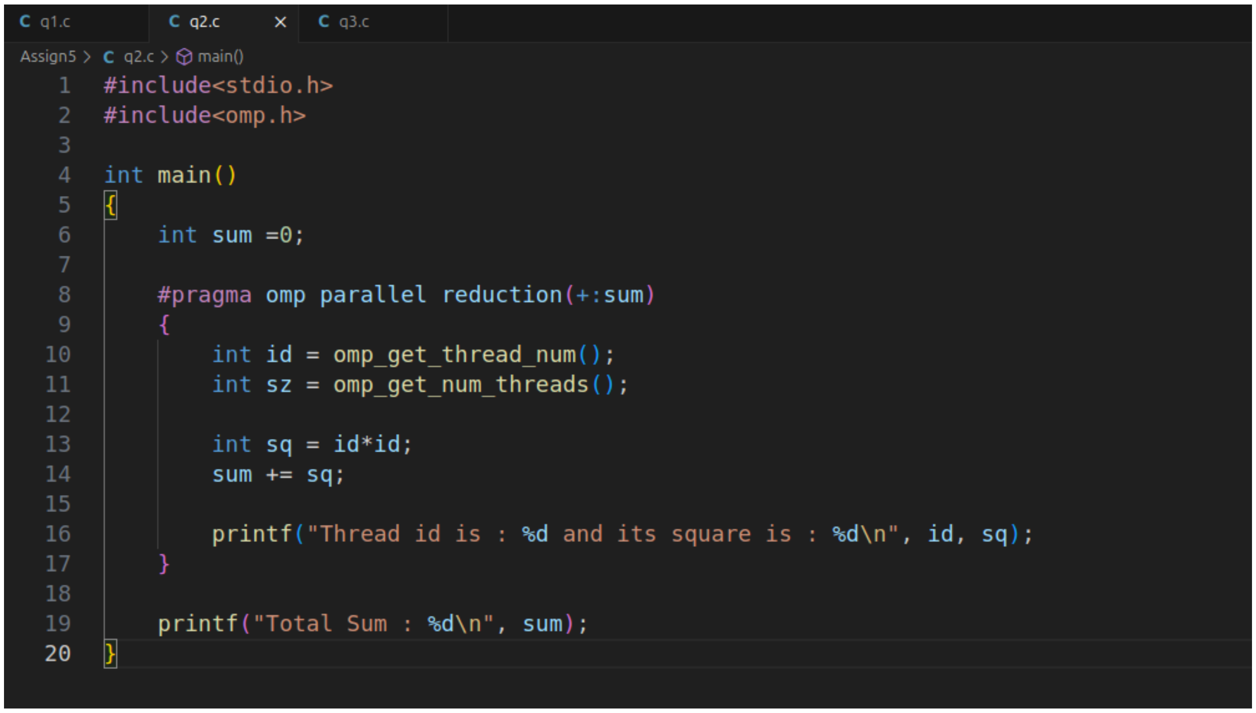
```c
#include <stdio.h>
#include <omp.h>

int main() {
    const char *family_members[] = {
        "Manish",
        "Bob",
        "Charlie",
        "Diana",
        "Eve"
    };
    int num_family_members = sizeof(family_members) / sizeof(family_members[0]);

    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        int num_threads = omp_get_num_threads();

        if (thread_id < num_family_members) {
            printf("Thread %d: %s\n", thread_id, family_members[thread_id]);
        }
    }

    return 0;
}
```



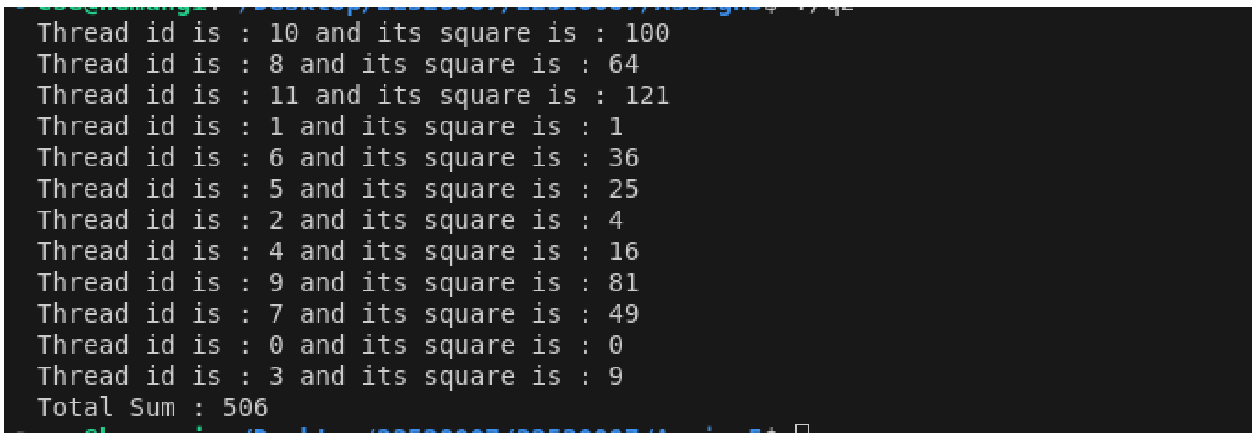
```
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ gcc -fopenmp -o q1 q1.c
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q1
Thread 4: Eve
Thread 3: Diana
Thread 1: Bob
Thread 2: Charlie
Thread 0: Manish
cse@hemangi:~/Desktop/22520007/22520007/Assign5$
```

Practical No 5

Q2. Write an OpenMP program such that, it should print the sum of square of the thread id's. Also make sure that, each thread should print the square value of their thread id.



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

int main()
{
    int sum =0;

    #pragma omp parallel reduction(+:sum)
    {
        int id = omp_get_thread_num();
        int sz = omp_get_num_threads();

        int sq = id*id;
        sum += sq;

        printf("Thread id is : %d and its square is : %d\n", id, sq);
    }

    printf("Total Sum : %d\n", sum);
}
```

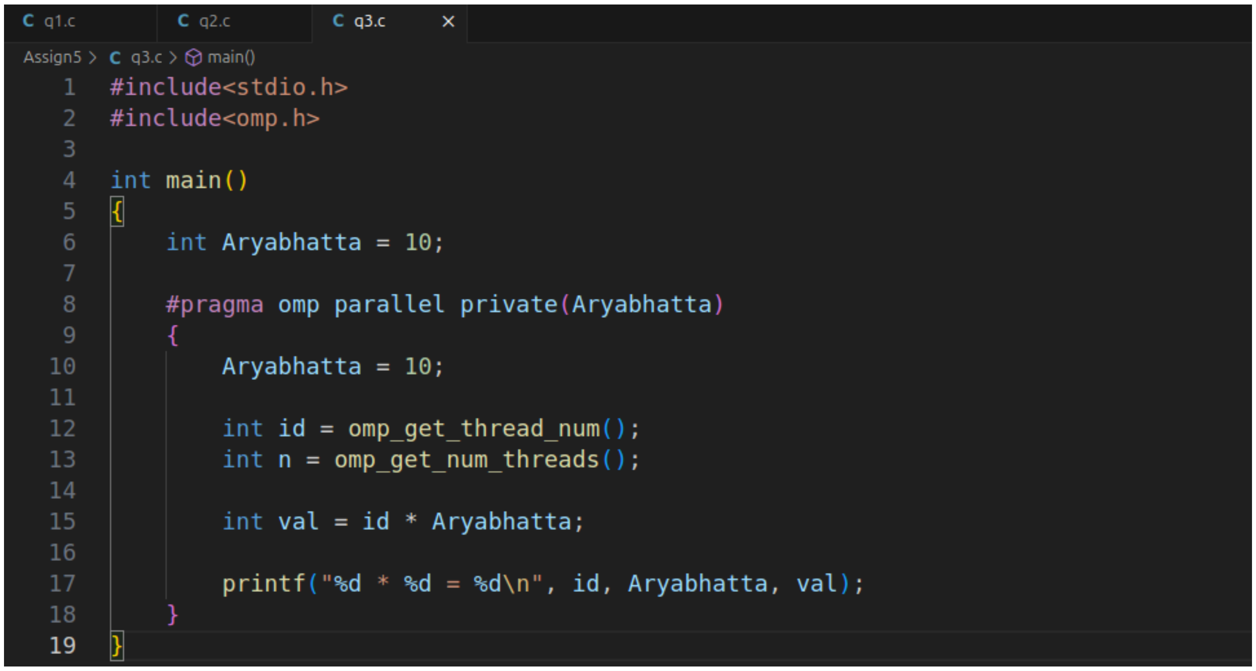


```
Thread id is : 10 and its square is : 100
Thread id is : 8 and its square is : 64
Thread id is : 11 and its square is : 121
Thread id is : 1 and its square is : 1
Thread id is : 6 and its square is : 36
Thread id is : 5 and its square is : 25
Thread id is : 2 and its square is : 4
Thread id is : 4 and its square is : 16
Thread id is : 9 and its square is : 81
Thread id is : 7 and its square is : 49
Thread id is : 0 and its square is : 0
Thread id is : 3 and its square is : 9
Total Sum : 506
```

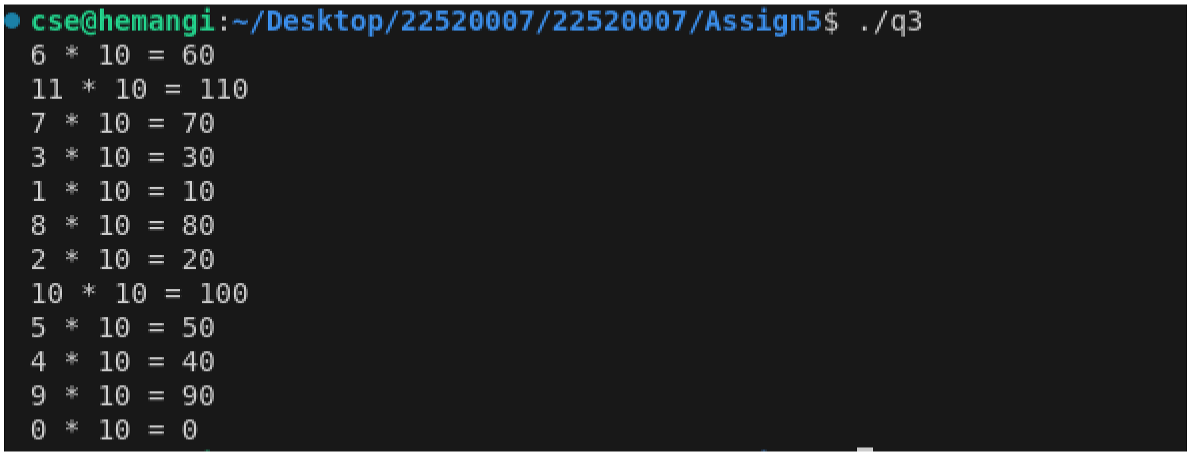
Practical No 5

Q3. Consider a variable called "Aryabhatta" declared as 10 (i.e int Arbhatta=10).Write an OpenMP program which should print the result of multiplication of thread id and value of the above variable.

Note*: The variable "Aryabhatta" should be declared as private



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

int main()
{
    int Aryabhatta = 10;

    #pragma omp parallel private(Aryabhatta)
    {
        Aryabhatta = 10;

        int id = omp_get_thread_num();
        int n = omp_get_num_threads();

        int val = id * Aryabhatta;

        printf("%d * %d = %d\n", id, Aryabhatta, val);
    }
}
```



```
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q3
6 * 10 = 60
11 * 10 = 110
7 * 10 = 70
3 * 10 = 30
1 * 10 = 10
8 * 10 = 80
2 * 10 = 20
10 * 10 = 100
5 * 10 = 50
4 * 10 = 40
9 * 10 = 90
0 * 10 = 0
```

Practical No 5

Q4. Write an OpenMP program that calculates the partial sum of the first 20 natural numbers using parallelism. Each thread should compute a portion of the sum by iterating through a loop. Implement the program using the lastprivate clause to ensure that the final total sum is correctly computed and printed outside the parallel region.

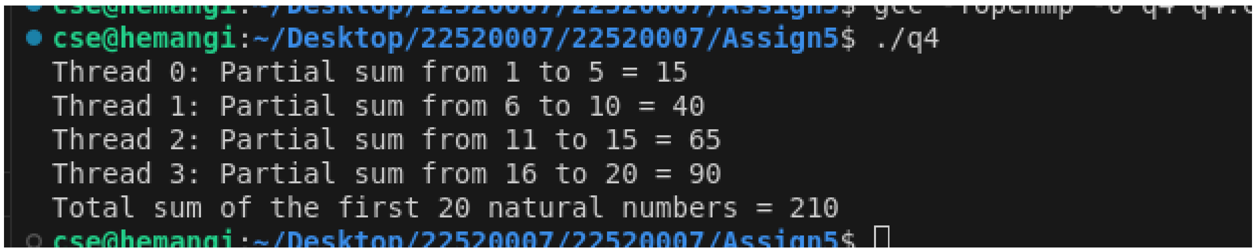
Hint:

1.Utilize OpenMP directives to parallelize the summation process.

2.Ensure that each thread has its private copy of partial sum.

3.Use the lastprivate clause to assign the value of the last thread's partial sum to the final total sum after the parallel region.

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

int main()
{
int n = 20;
int total_sum = 0;
int partial_sum = 0;
omp_set_num_threads(4);

#pragma omp parallel private(partial_sum) reduction(+:total_sum)
{
// Initialize partial_sum to 0 for each thread
partial_sum = 0;

// Get the thread ID and the total number of threads
int thread_id = omp_get_thread_num();

int num_threads = omp_get_num_threads();

// Calculate the range for each thread
int chunk_size = n / num_threads;
int start = thread_id * chunk_size + 1;
int end = (thread_id == num_threads - 1) ? n : start + chunk_size - 1;

// Compute the partial sum for the range of numbers assigned to this thread
for (int i = start; i <= end; ++i)
{
partial_sum += i;
}

total_sum += partial_sum;
// Print partial sum from each thread
printf("Thread %d: Partial sum from %d to %d = %d\n", thread_id, start, end, partial_sum);

// Use the lastprivate clause to ensure that the final value of partial_sum is captured
}

// #pragma omp single
// {
// total_sum = partial_sum;
// }

// Print the final total sum
printf("Total sum of the first %d natural numbers = %d\n", n, total_sum);

return 0;
}
```



```
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q4
Thread 0: Partial sum from 1 to 5 = 15
Thread 1: Partial sum from 6 to 10 = 40
Thread 2: Partial sum from 11 to 15 = 65
Thread 3: Partial sum from 16 to 20 = 90
Total sum of the first 20 natural numbers = 210
cse@hemangi:~/Desktop/22520007/22520007/Assign5$
```

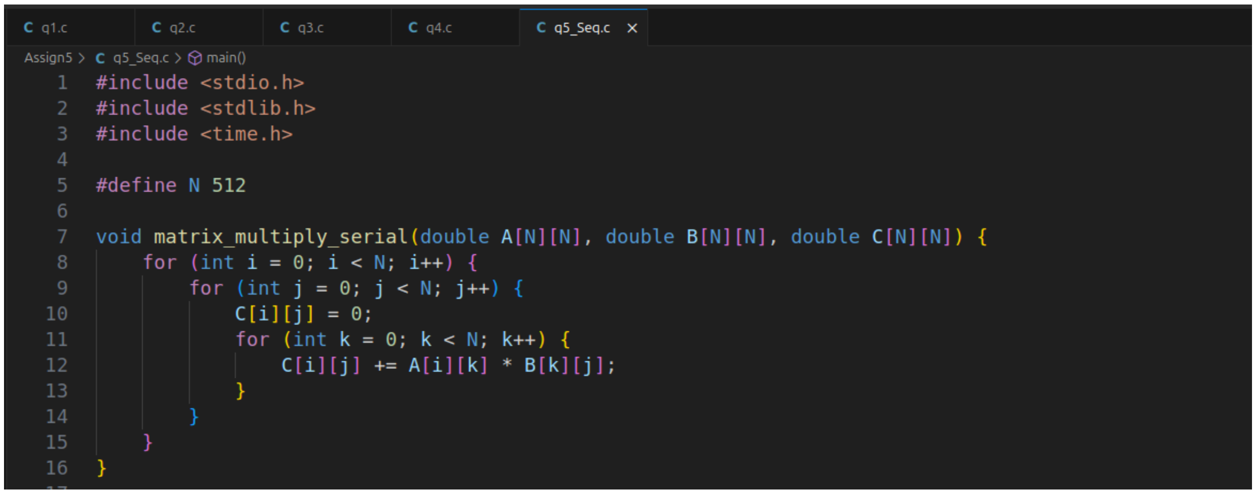
Practical No 5

Q5. Consider a scenario where you have to parallelize a program that performs matrix multiplication using OpenMP. Your task is to implement parallelization using both static and dynamic scheduling, and compare the execution time of each approach.

Note*:

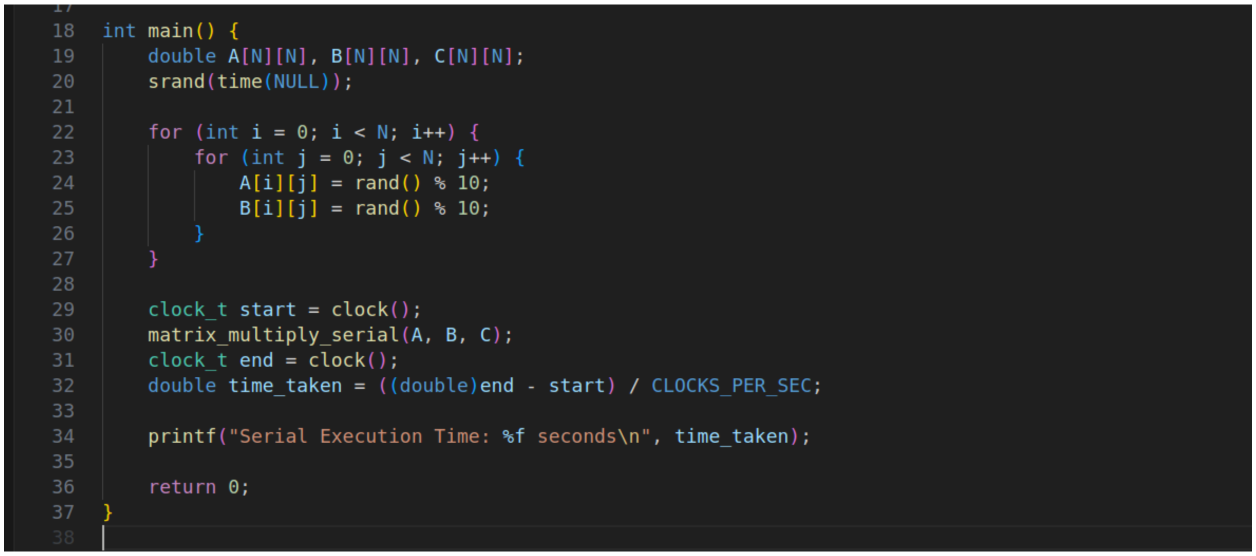
- Implement a serial version of matrix multiplication in C/C++.
- Parallelize the matrix multiplication using OpenMP with static scheduling.
- Parallelize the matrix multiplication using OpenMP with dynamic scheduling.
- Measure the execution time of each parallelized version for various matrix sizes.
- Compare the execution times and discuss the advantages and disadvantages of static and dynamic scheduling in this context.

Sequential :

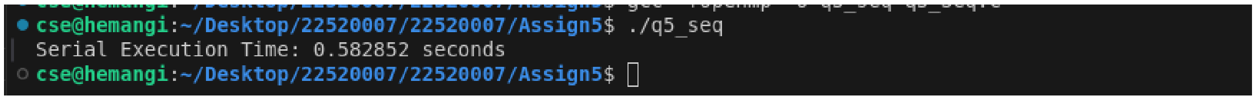


```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 512

void matrix_multiply_serial(double A[N][N], double B[N][N], double C[N][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Practical No 5



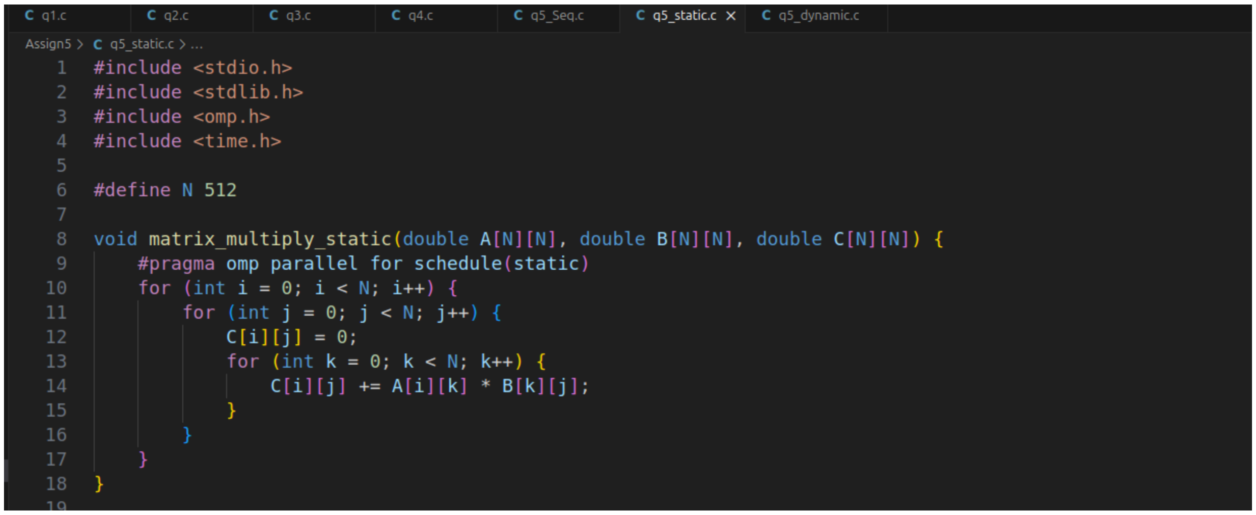
```c
int main() {
    double A[N][N], B[N][N], C[N][N];
    srand(time(NULL));

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
        }
    }

    clock_t start = clock();
    matrix_multiply_serial(A, B, C);
    clock_t end = clock();
    double time_taken = ((double)end - start) / CLOCKS_PER_SEC;

    printf("Serial Execution Time: %f seconds\n", time_taken);

    return 0;
}
```



```
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ gcc -fopenmp -o q5_seq q5_seq.c
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q5_seq
Serial Execution Time: 0.582852 seconds
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ 
```

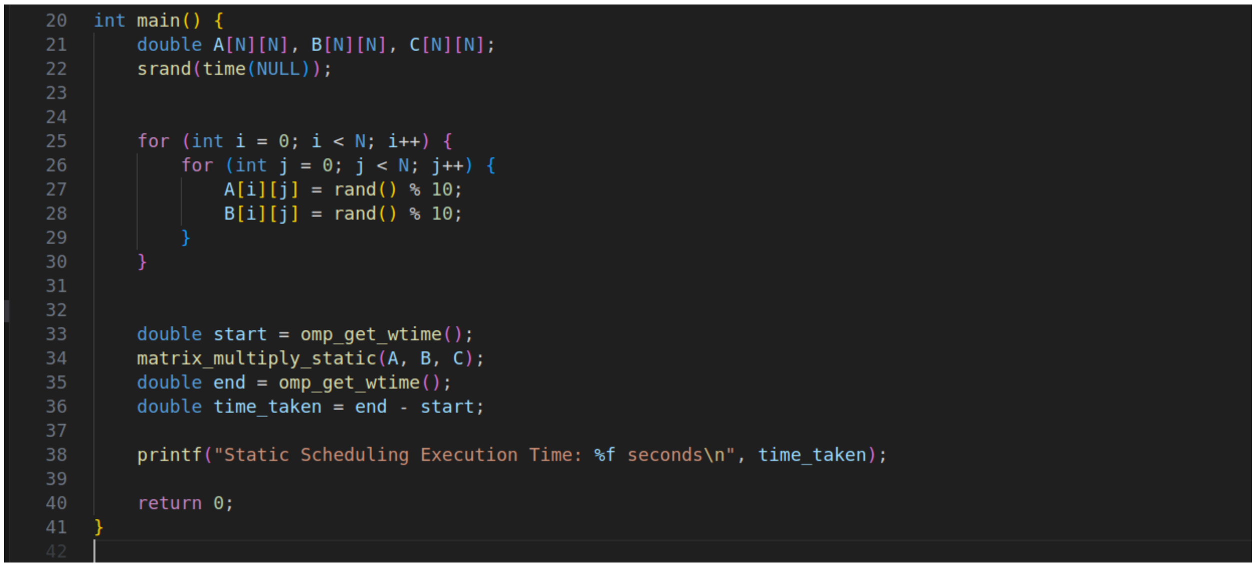
Parallel :

Static Scheduling :

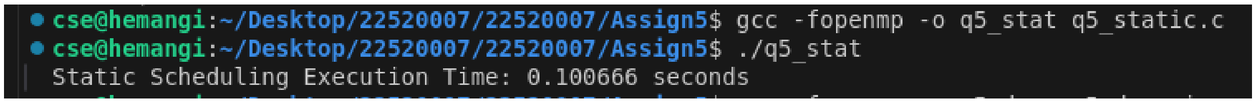


```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

#define N 512

void matrix_multiply_static(double A[N][N], double B[N][N], double C[N][N]) {
    #pragma omp parallel for schedule(static)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Practical No 5

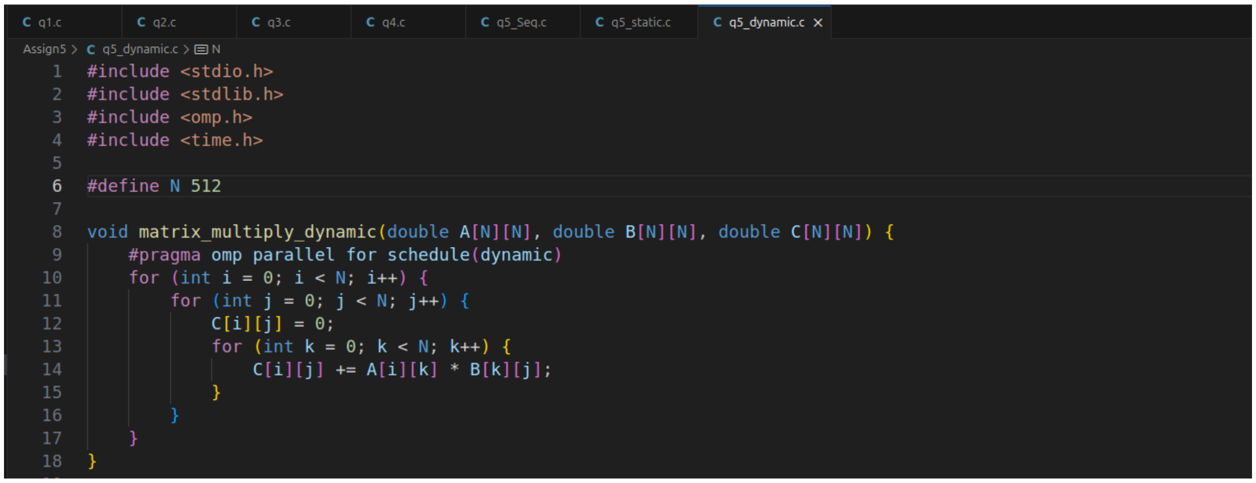


```c
int main() {
    double A[N][N], B[N][N], C[N][N];
    srand(time(NULL));


    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
        }
    }


    double start = omp_get_wtime();
    matrix_multiply_static(A, B, C);
    double end = omp_get_wtime();
    double time_taken = end - start;

    printf("Static Scheduling Execution Time: %f seconds\n", time_taken);

    return 0;
}
```



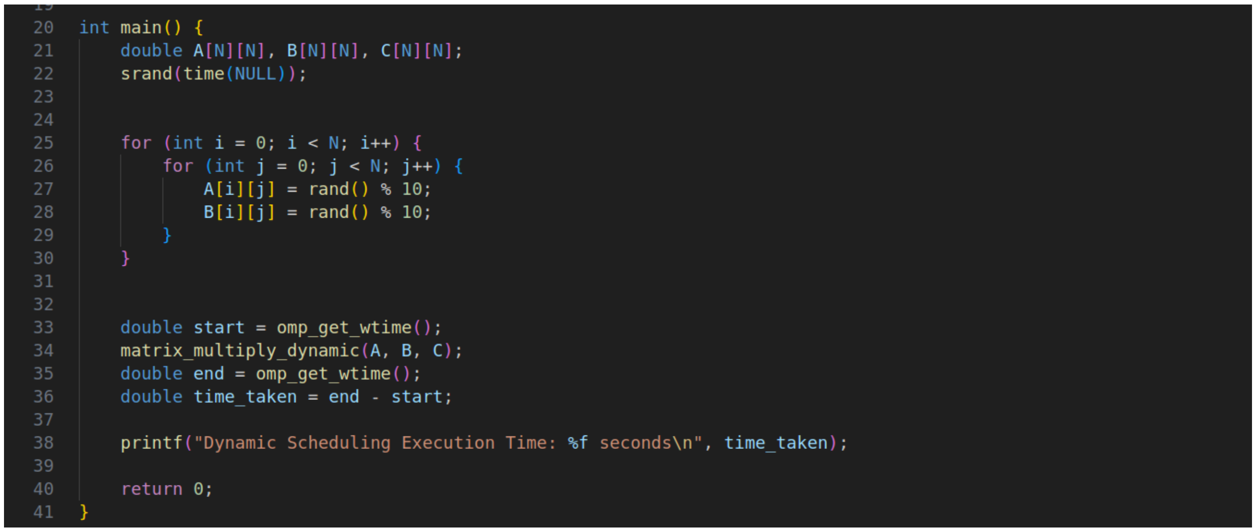
```
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ gcc -fopenmp -o q5_stat q5_static.c
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q5_stat
Static Scheduling Execution Time: 0.100666 seconds
```

Dynamic Scheduling :

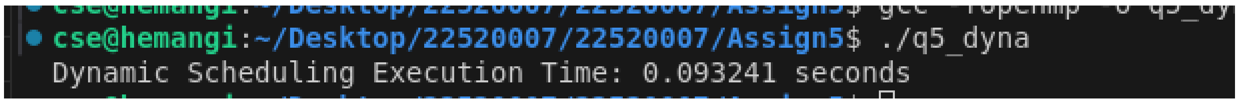


```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

#define N 512

void matrix_multiply_dynamic(double A[N][N], double B[N][N], double C[N][N]) {
    #pragma omp parallel for schedule(dynamic)
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

Practical No 5



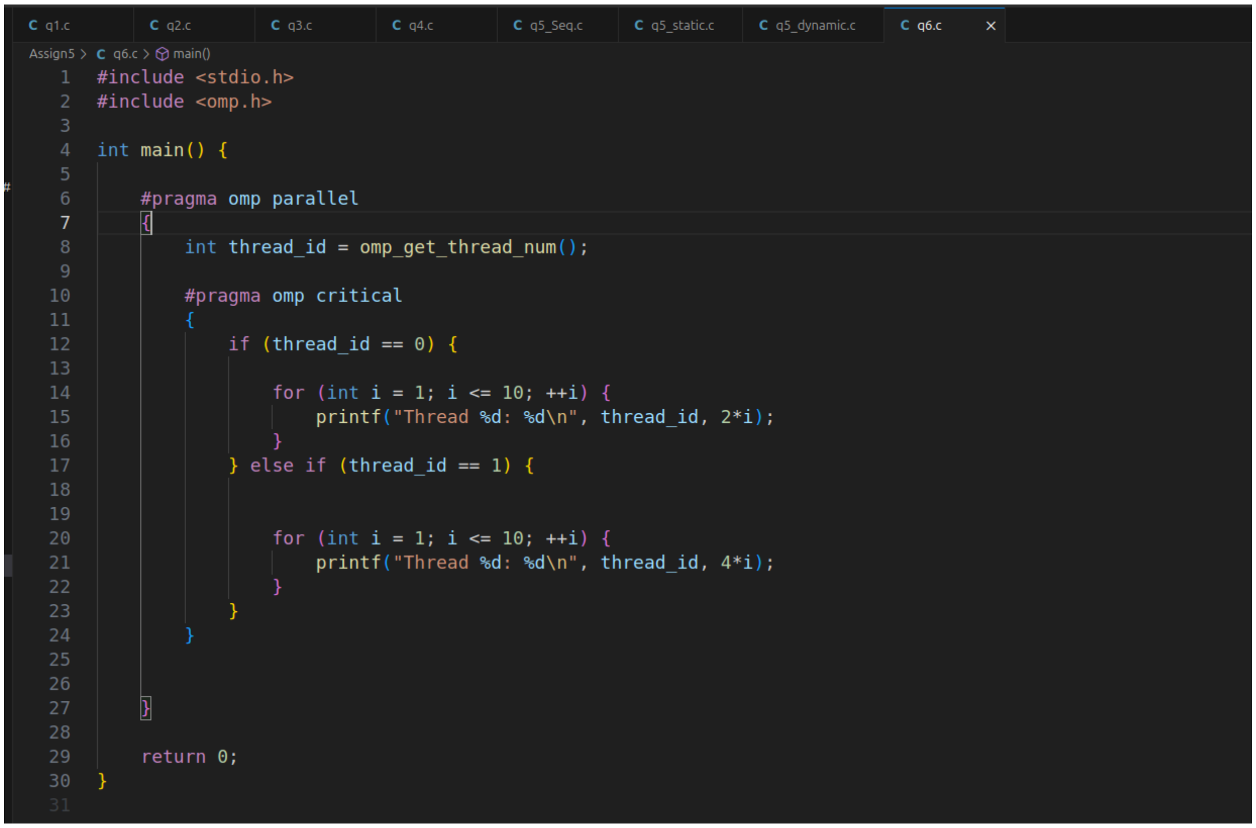
```c
int main() {
    double A[N][N], B[N][N], C[N][N];
    srand(time(NULL));


    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            A[i][j] = rand() % 10;
            B[i][j] = rand() % 10;
        }
    }


    double start = omp_get_wtime();
    matrix_multiply_dynamic(A, B, C);
    double end = omp_get_wtime();
    double time_taken = end - start;

    printf("Dynamic Scheduling Execution Time: %f seconds\n", time_taken);

    return 0;
}
```



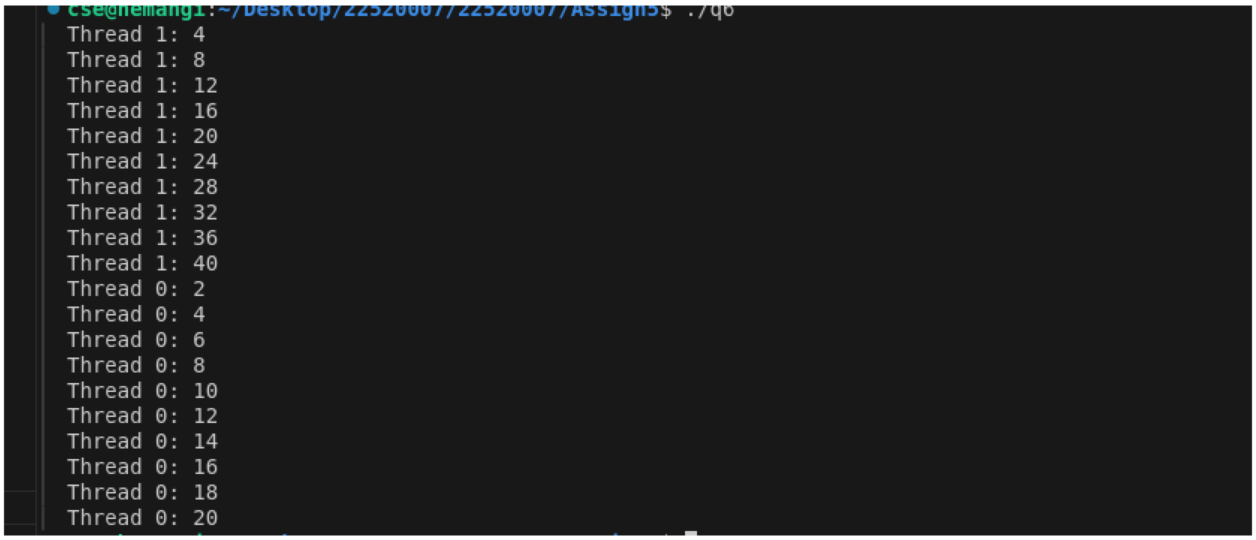
```
● cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q5_dyna
  Dynamic Scheduling Execution Time: 0.093241 seconds
```

Practical No 5

Q6. Write a Parallel C program which should print the series of 2 and 4. Make sure both should be executed by different threads !



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

int main() {

    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();

        #pragma omp critical
        {
            if (thread_id == 0) {

                for (int i = 1; i <= 10; ++i) {
                    printf("Thread %d: %d\n", thread_id, 2*i);
                }
            } else if (thread_id == 1) {


                for (int i = 1; i <= 10; ++i) {
                    printf("Thread %d: %d\n", thread_id, 4*i);
                }
            }
        }


    }

    return 0;
}
```



```
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q6
Thread 1: 4
Thread 1: 8
Thread 1: 12
Thread 1: 16
Thread 1: 20
Thread 1: 24
Thread 1: 28
Thread 1: 32
Thread 1: 36
Thread 1: 40
Thread 0: 2
Thread 0: 4
Thread 0: 6
Thread 0: 8
Thread 0: 10
Thread 0: 12
Thread 0: 14
Thread 0: 16
Thread 0: 18
Thread 0: 20
```

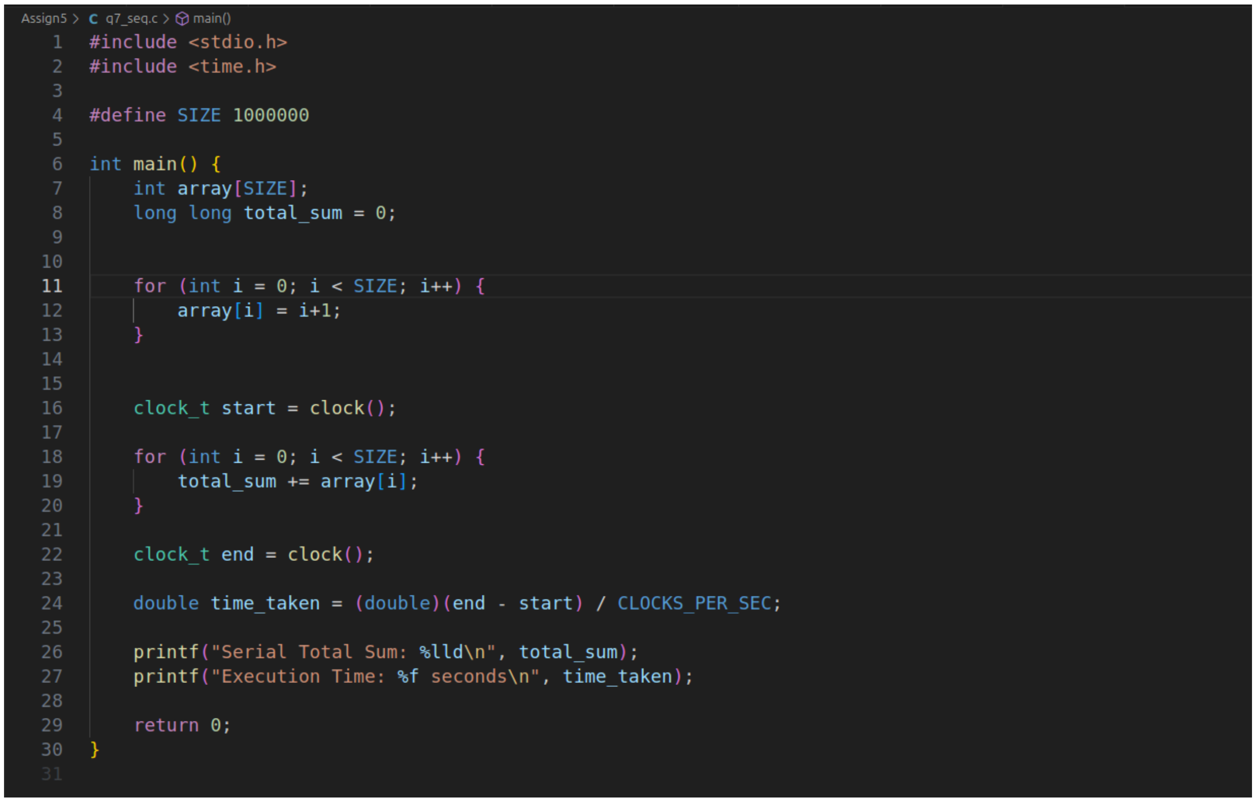
Practical No 5

Q7. Consider a scenario where you have a shared variable total_sum that needs to be updated concurrently by multiple threads in a parallel program. However, concurrent updates to this variable can result in data races and incorrect results. Your task is to modify the program to ensure correct synchronization using OpenMP's critical and atomic constructs.

Note*:

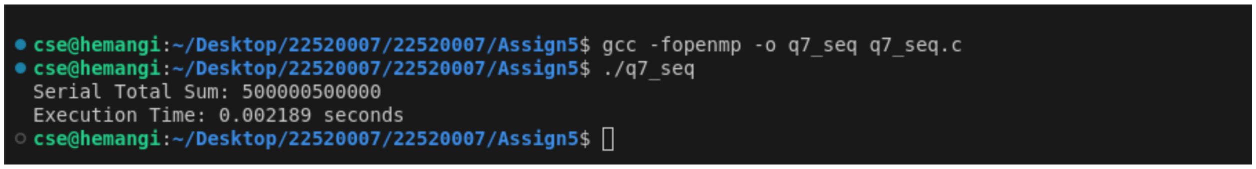
- Implement a simple parallel program in C that initializes an array of integers and calculates the sum of its elements concurrently using OpenMP.
- Identify potential issues with concurrent updates to the total_sum variable in the parallelized version of the program.
- Modify the program to use OpenMP's critical/atomic directive to ensure synchronized access to the total_sum variable.
- Measure and compare the performance of synchronized versions against the unsynchronized implementation.

Sequential :



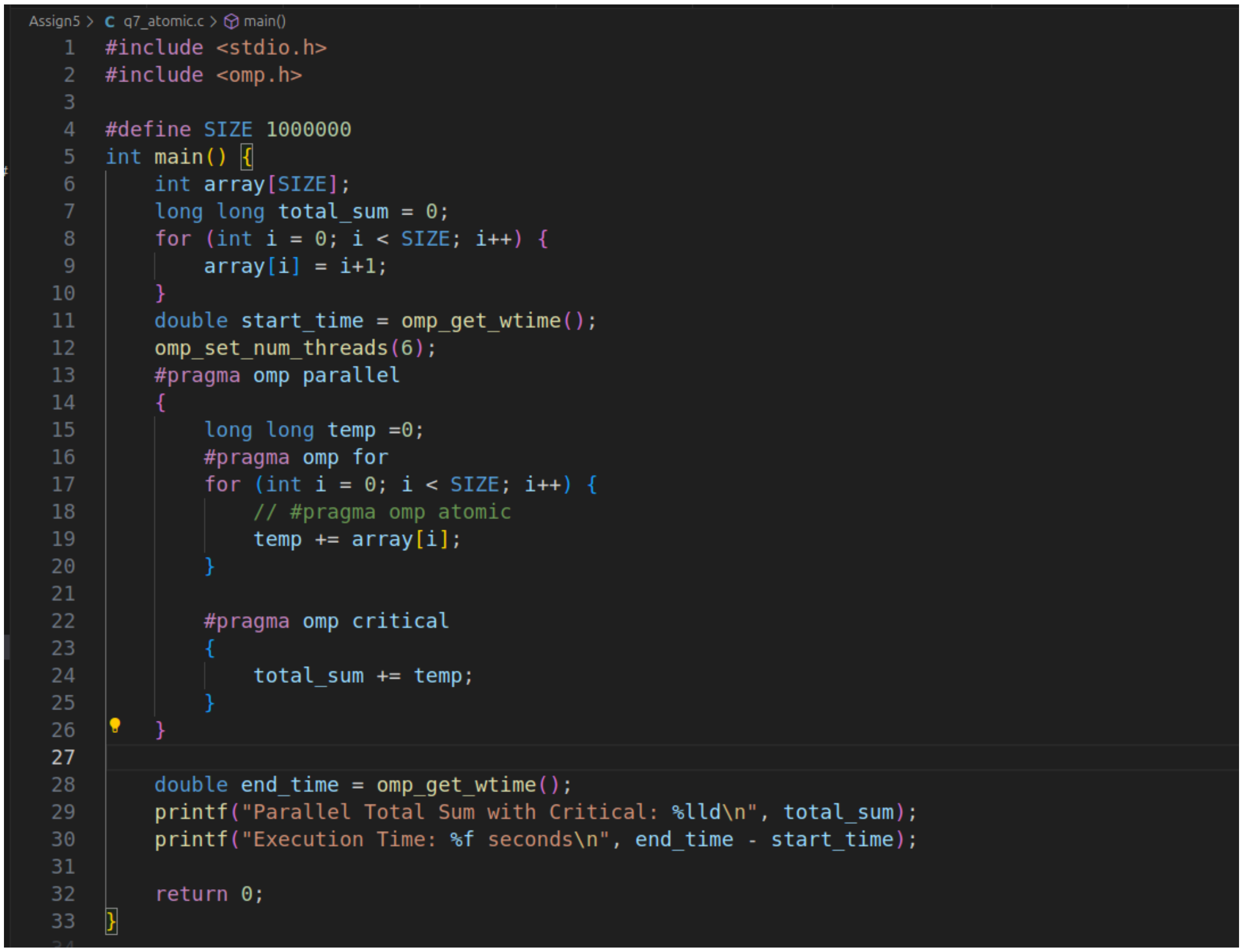
Assign5 > q7_seq.c > main()

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

#define SIZE 1000000

int main() {
    int array[SIZE];
    long long total_sum = 0;


    for (int i = 0; i < SIZE; i++) {
        array[i] = i+1;
    }


    clock_t start = clock();

    for (int i = 0; i < SIZE; i++) {
        total_sum += array[i];
    }

    clock_t end = clock();

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

    printf("Serial Total Sum: %lld\n", total_sum);
    printf("Execution Time: %f seconds\n", time_taken);

    return 0;
}
```



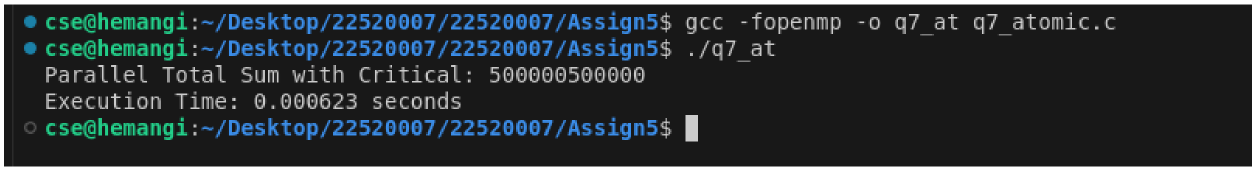
```
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ gcc -fopenmp -o q7_seq q7_seq.c
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q7_seq
Serial Total Sum: 500000500000
Execution Time: 0.002189 seconds
cse@hemangi:~/Desktop/22520007/22520007/Assign5$
```

Parallel Using Critical :



Assign5 > C q7_atomic.c > main()

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

#define SIZE 1000000
int main() {
    int array[SIZE];
    long long total_sum = 0;
    for (int i = 0; i < SIZE; i++) {
        array[i] = i+1;
    }
    double start_time = omp_get_wtime();
    omp_set_num_threads(6);
    #pragma omp parallel
    {
        long long temp =0;
        #pragma omp for
        for (int i = 0; i < SIZE; i++) {
            // #pragma omp atomic
            temp += array[i];
        }

        #pragma omp critical
        {
            total_sum += temp;
        }
    }

    double end_time = omp_get_wtime();
    printf("Parallel Total Sum with Critical: %lld\n", total_sum);
    printf("Execution Time: %f seconds\n", end_time - start_time);

    return 0;
}
```



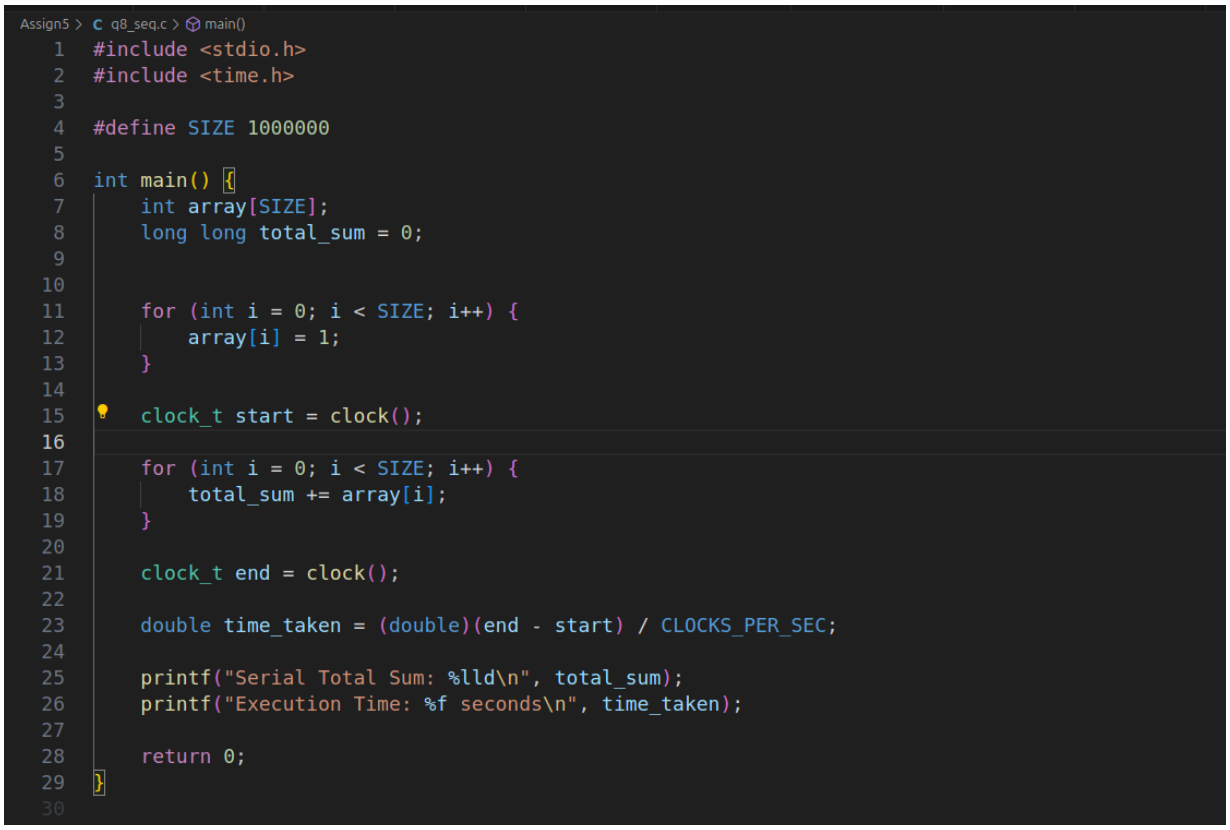
```
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ gcc -fopenmp -o q7_at q7_atomic.c
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q7_at
Parallel Total Sum with Critical: 500000500000
Execution Time: 0.000623 seconds
cse@hemangi:~/Desktop/22520007/22520007/Assign5$
```

Q8. Consider a scenario where you have a large array of integers, and you need to find the sum of all its elements in parallel using OpenMP. The array is shared among multiple threads, and parallelism is needed to expedite the computation process. Your task is to write a parallel program that calculates the sum of all elements in the array using OpenMP's reduction clause.

Note*:

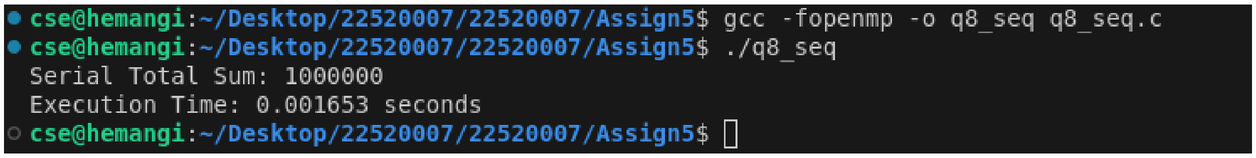
- Implement a sequential version of the program that calculates the sum of all elements in the array without using any parallelism.
- Identify potential bottlenecks and limitations of the sequential implementation in handling large arrays efficiently.
- Modify the program to utilize OpenMP's reduction clause to parallelize the summation process across multiple threads.
- Test the program with different array sizes and thread counts to evaluate its scalability and performance.
- Discuss the advantages of using the reduction clause for parallel summation and its impact on program efficiency.

Sequential :



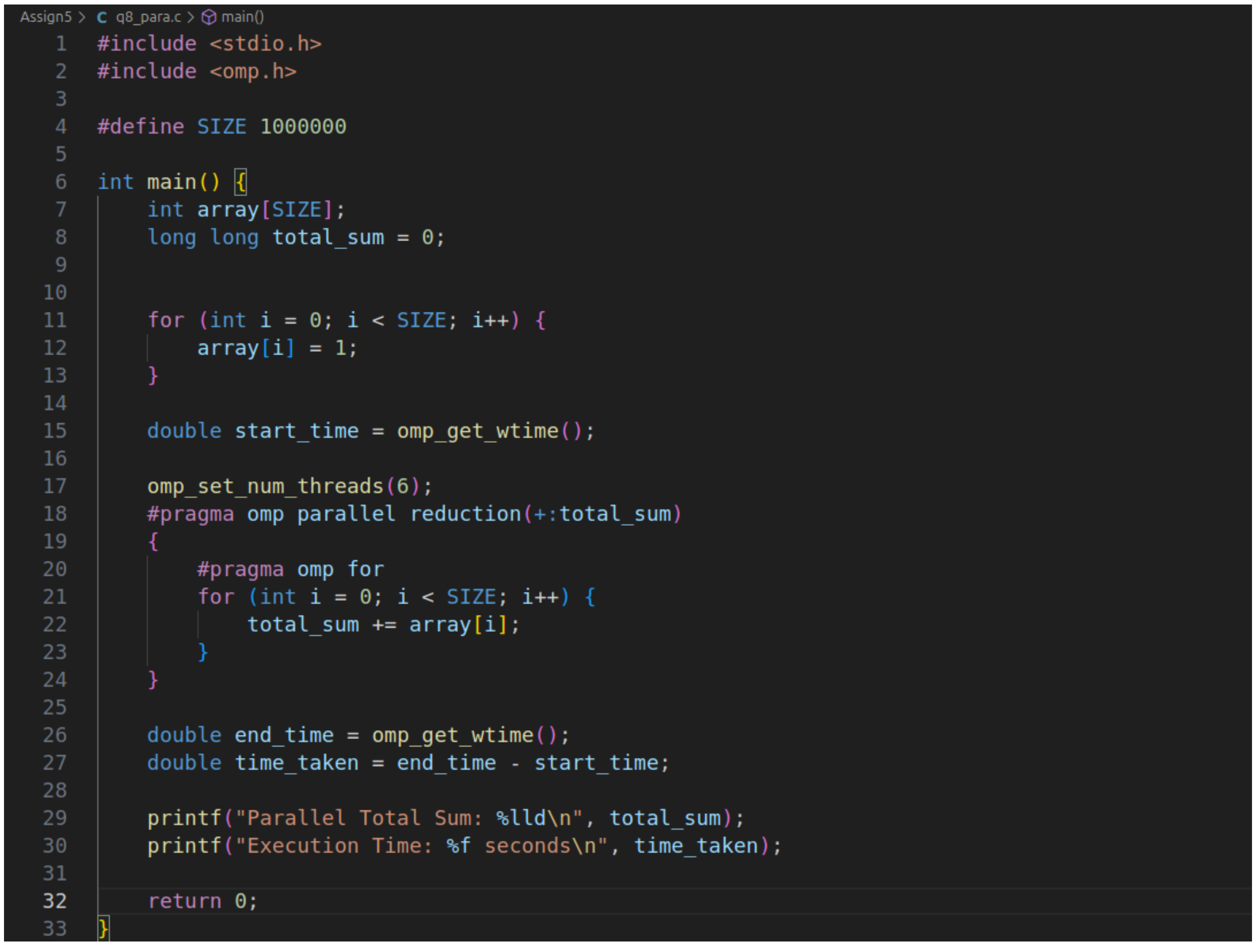
```c
#include <stdio.h>
#include <time.h>

#define SIZE 1000000

int main() {
    int array[SIZE];
    long long total_sum = 0;


    for (int i = 0; i < SIZE; i++) {
        array[i] = 1;
    }

    clock_t start = clock();

    for (int i = 0; i < SIZE; i++) {
        total_sum += array[i];
    }

    clock_t end = clock();

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

    printf("Serial Total Sum: %lld\n", total_sum);
    printf("Execution Time: %f seconds\n", time_taken);

    return 0;
}
```

Practical No 5



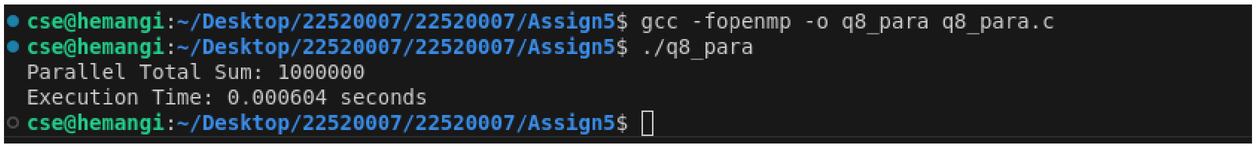
```
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ gcc -fopenmp -o q8_seq q8_seq.c
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q8_seq
Serial Total Sum: 1000000
Execution Time: 0.001653 seconds
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ 
```

Parallel :



Assign5 > q8_para.c > main()

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

#define SIZE 1000000

int main() {
    int array[SIZE];
    long long total_sum = 0;


    for (int i = 0; i < SIZE; i++) {
        array[i] = 1;
    }

    double start_time = omp_get_wtime();

    omp_set_num_threads(6);
    #pragma omp parallel reduction(+:total_sum)
    {
        #pragma omp for
        for (int i = 0; i < SIZE; i++) {
            total_sum += array[i];
        }
    }

    double end_time = omp_get_wtime();
    double time_taken = end_time - start_time;

    printf("Parallel Total Sum: %lld\n", total_sum);
    printf("Execution Time: %f seconds\n", time_taken);

    return 0;
}
```



```
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ gcc -fopenmp -o q8_para q8_para.c
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ ./q8_para
Parallel Total Sum: 1000000
Execution Time: 0.000604 seconds
cse@hemangi:~/Desktop/22520007/22520007/Assign5$ 
```