Practical No 5

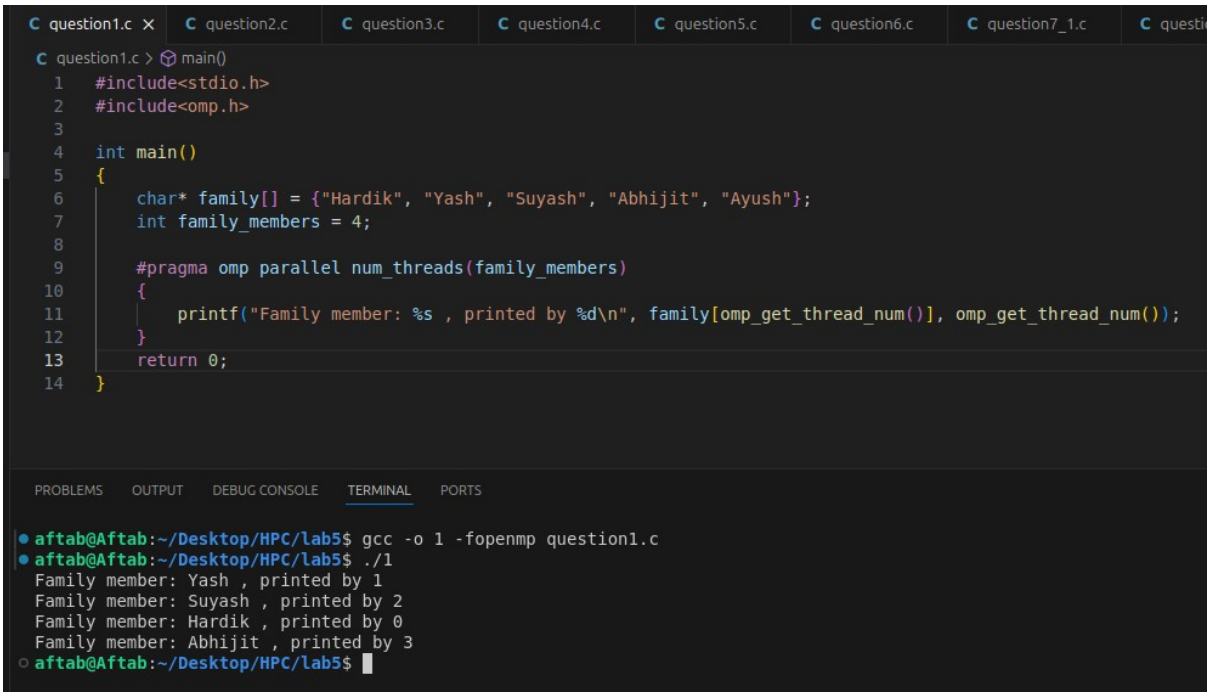
PRN: 22520005

Name: Aftab Imtiyaj Bhadgaonkar

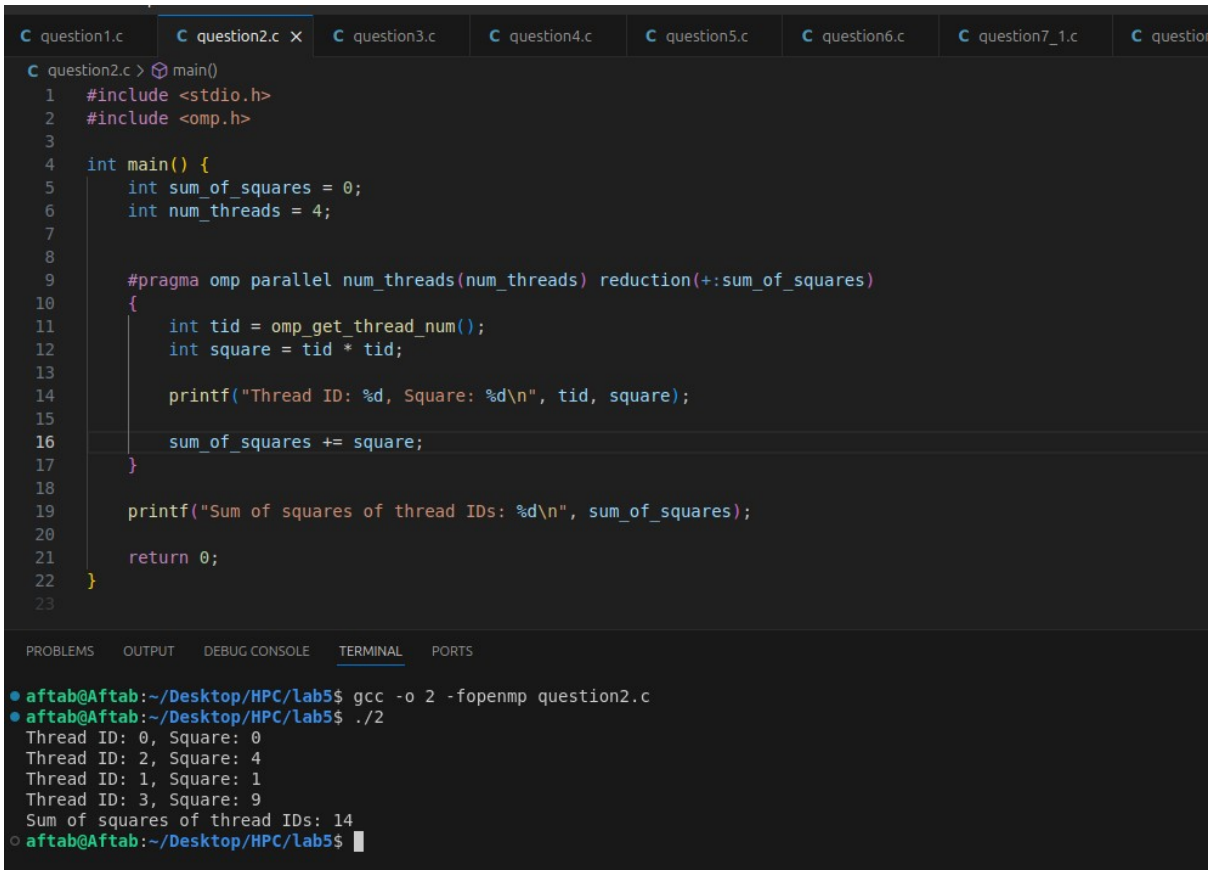
Batch: B6

Course: High Performance Computing Lab

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.

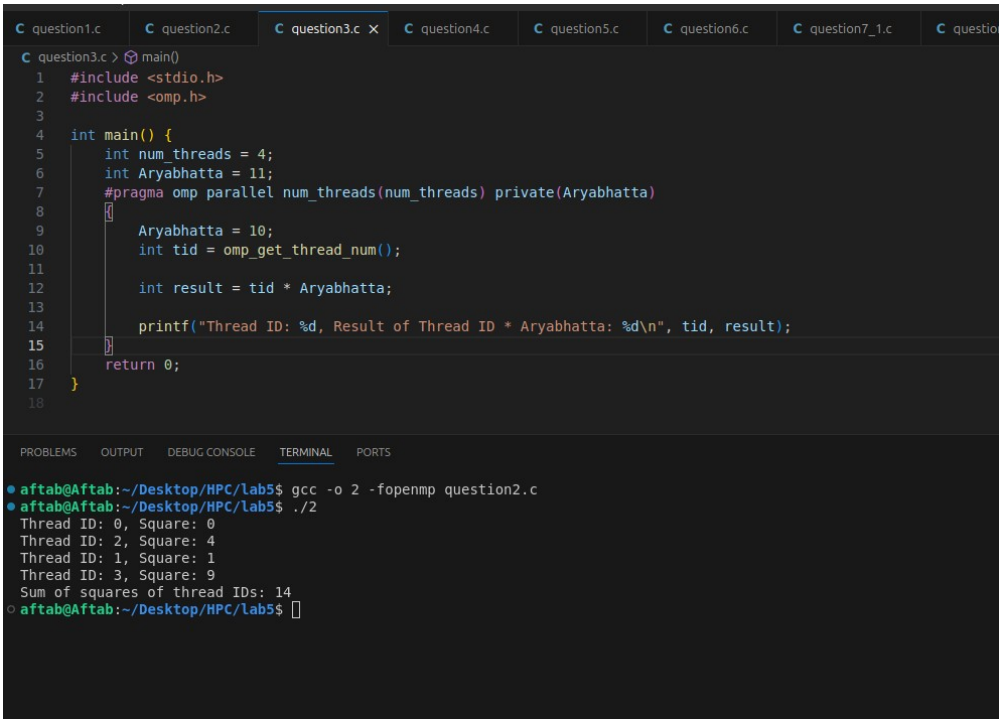


Q2. Write an OpenMP program such that, it should print the sum of square of the thread id's. Also make sure that, each thread should print the square value of their thread id.



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



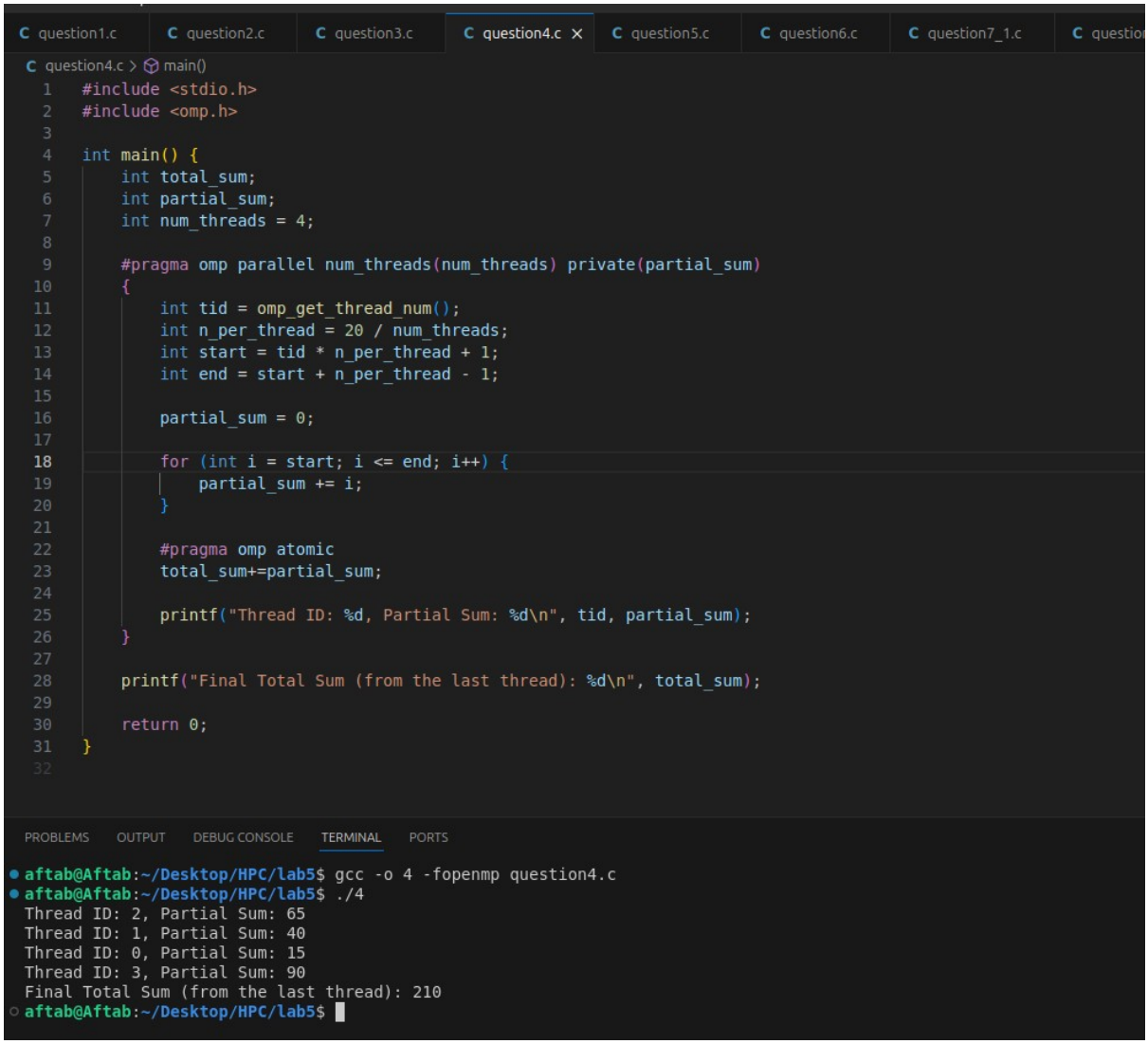
Q4. Write an OpenMP program that calculates the partial sum of the first 20 natural numbers using parallelism. Each thread should compute a portion of the sum by iterating through a loop. Implement the program using the lastprivate clause to ensure that the final total sum is correctly computed and printed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

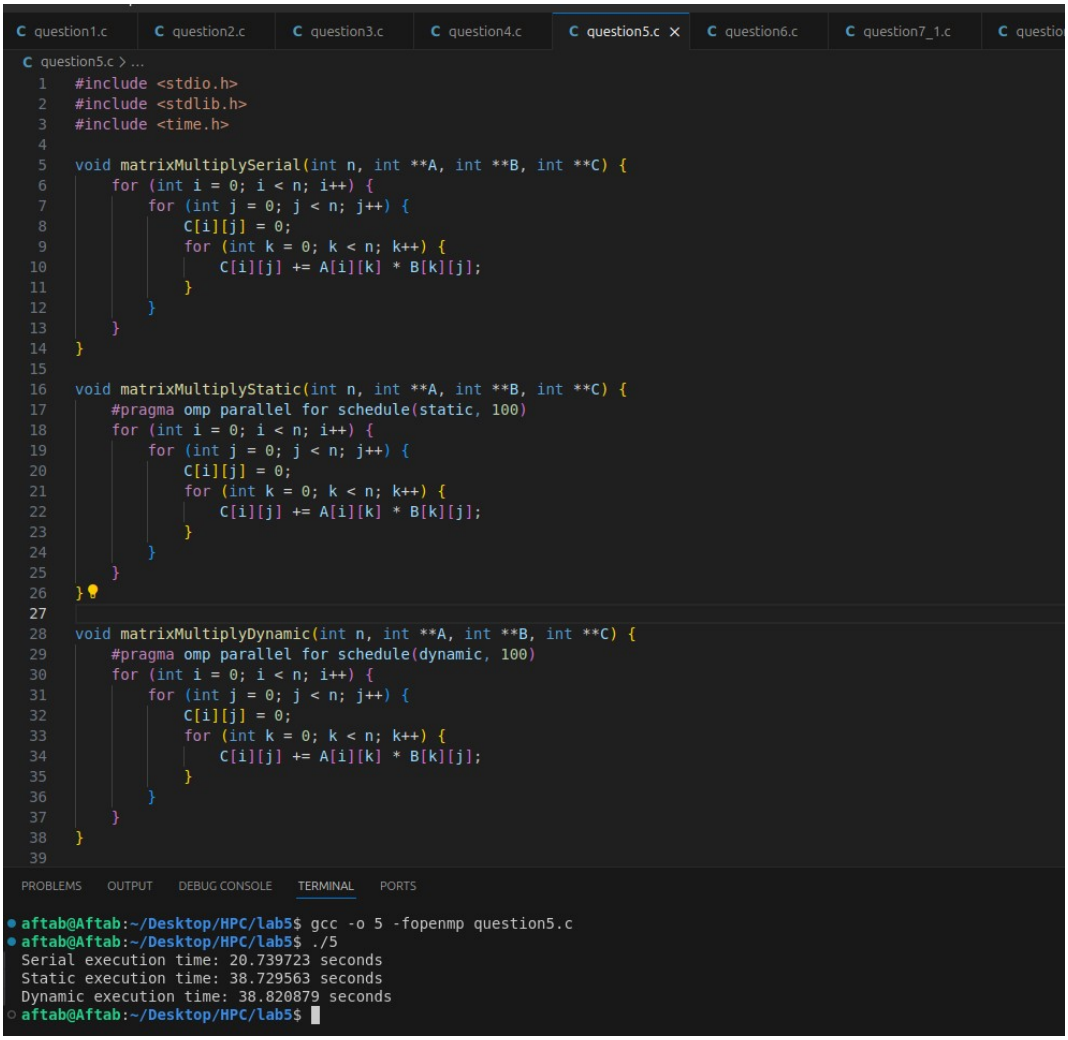
3.Use the lastprivate clause to assign the value of the last thread's partial sum to the final total sum after the parallel region.



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*:

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



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

question1.c | question2.c | question3.c | question4.c | question5.c | question6.c

question6.c > main()

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

void printSeries(int start, int step, int count) {
    for (int i = 0; i < count; i++) {
        printf("%d ", start + i * step);
    }
    printf("\n");
}

int main() {
    int count = 10;

    #pragma omp parallel sections
    {
        #pragma omp section
        {
            printf("Series of 2: ");
            printSeries(2, 2, count);
        }

        #pragma omp section
        {
            printf("Series of 4: ");
            printSeries(4, 4, count);
        }
    }

    return 0;
}
```

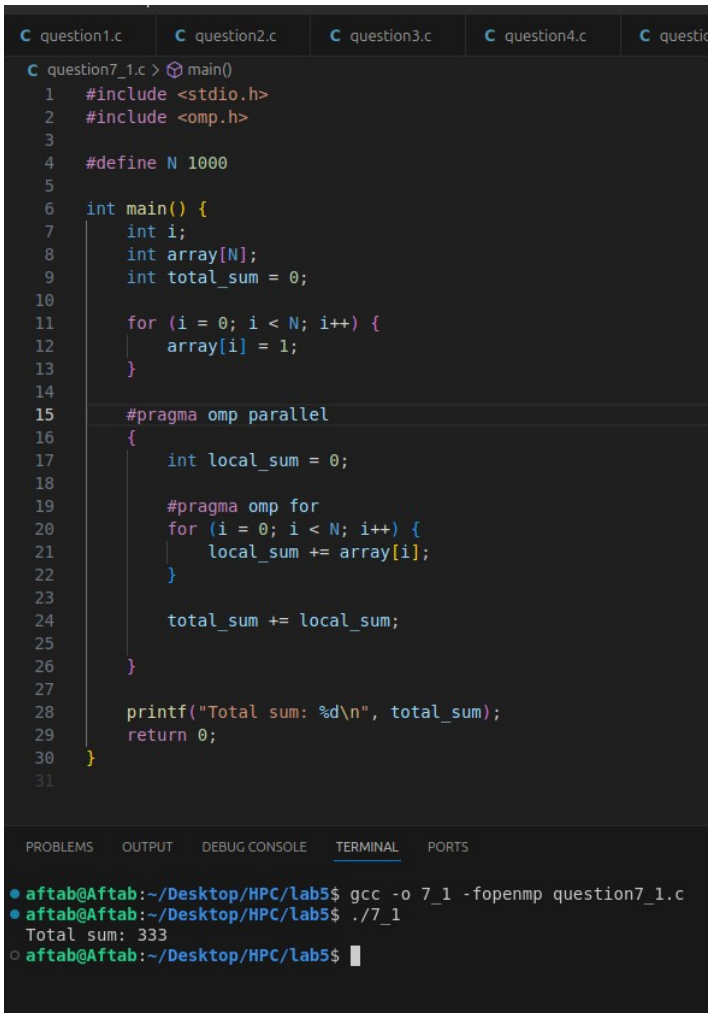
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
aftab@Aftab:~/Desktop/HPC/lab5$ gcc -o 6 -fopenmp question6.c
aftab@Aftab:~/Desktop/HPC/lab5$ ./6
Series of 2: 2 4 6 8 10 12 14 16 18 20
Series of 4: 4 8 12 16 20 24 28 32 36 40
aftab@Aftab:~/Desktop/HPC/lab5$
```

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.



question1.c | question2.c | question3.c | question4.c | questi

question7_2.c > main()

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

#define N 1000

int main() {
    int i;
    int array[N];
    int total_sum = 0;

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

    #pragma omp parallel
    {
        int local_sum = 0;

        #pragma omp for
        for (i = 0; i < N; i++)
        {
            local_sum += array[i];
        }

        #pragma omp critical
        total_sum += local_sum;
    }

    printf("Total sum: %d\n", total_sum);
    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
aftab@Aftab:~/Desktop/HPC/lab5$ gcc -o 7_2 -fopenmp question7_2.c
aftab@Aftab:~/Desktop/HPC/lab5$ ./7_2
Total sum: 1000
aftab@Aftab:~/Desktop/HPC/lab5$
```

question1.c | question2.c | question3.c | question4.c | questi

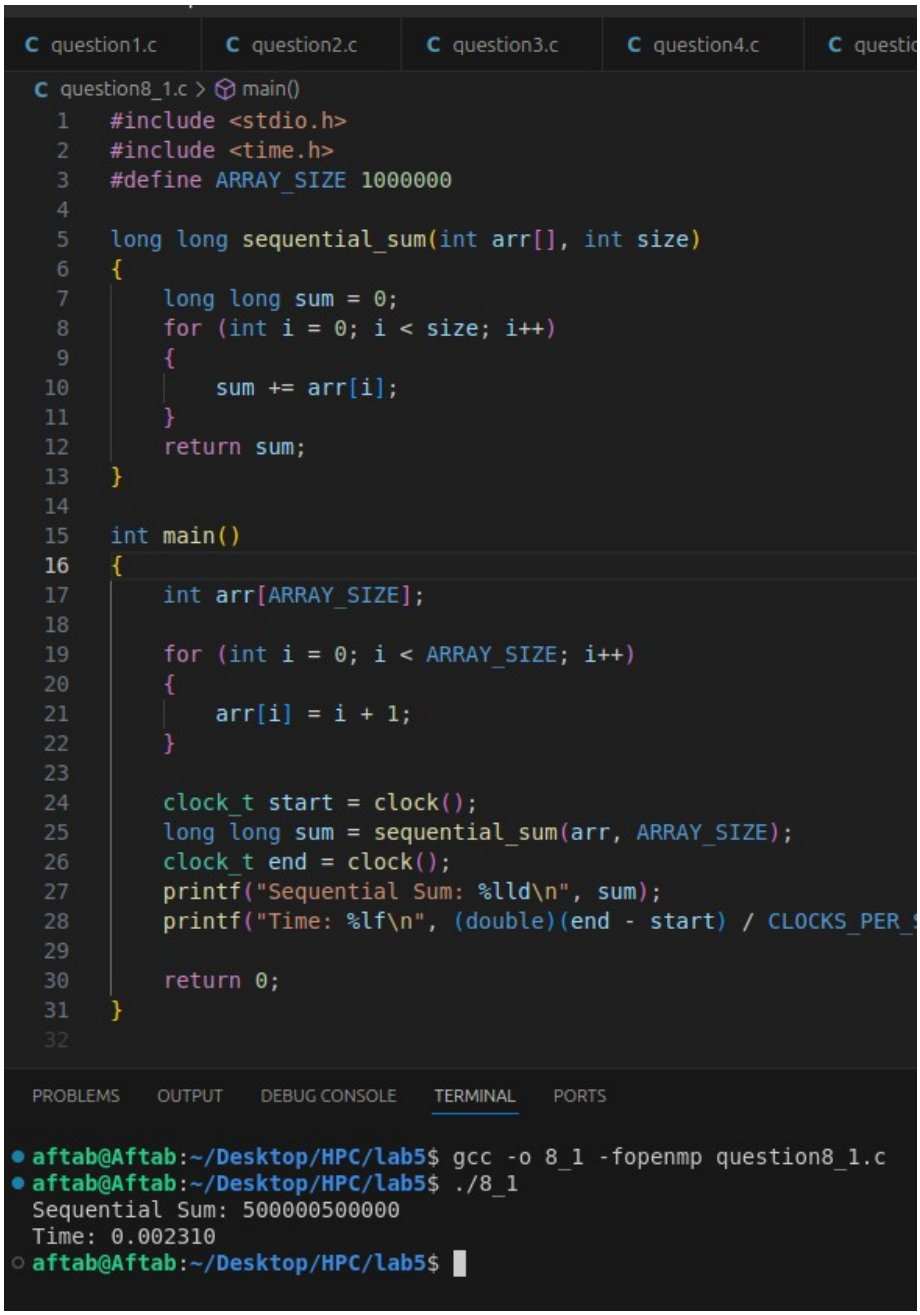
question7_3.c > ...

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

#define N 1000

int main() {
    int i;
    int array[N];
    int total_sum = 0;

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

    #pragma omp parallel
    {
        int local_sum = 0;

        #pragma omp for
        for (i = 0; i < N; i++) {
            local_sum += array[i];
        }

        #pragma omp atomic
        total_sum += local_sum;
    }

    printf("Total sum: %d\n", total_sum);
    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
aftab@Aftab:~/Desktop/HPC/lab5$ gcc -o 7_3 -fopenmp question7_3.c
aftab@Aftab:~/Desktop/HPC/lab5$ ./7_3
Total sum: 1000
aftab@Aftab:~/Desktop/HPC/lab5$
```

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.



question1.c | question2.c | question3.c | question4.c | questi

question8_2.c > parallel_sum(int [], int)

```c
#include <stdio.h>
#include <omp.h>
#include <time.h>
#define ARRAY_SIZE 1000000

long long parallel_sum(int arr[], int size)
{
    long long sum = 0;

#pragma omp parallel for reduction(+ : sum)
    for (int i = 0; i < size; i++)
    {
        sum += arr[i];
    }

    return sum;
}

int main()
{
    int arr[ARRAY_SIZE];

    for (int i = 0; i < ARRAY_SIZE; i++)
    {
        arr[i] = i + 1;
    }

    clock_t start = clock();
    long long sum = parallel_sum(arr, ARRAY_SIZE);
    clock_t end = clock();
    printf("Parallel Sum: %lld\n", sum);
    printf("Time: %lf\n", (double)(end - start) / CLOCKS_PER_

    return 0;
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
aftab@Aftab:~/Desktop/HPC/lab5$ gcc -o 8_2 -fopenmp question8_2.c
aftab@Aftab:~/Desktop/HPC/lab5$ ./8_2
Parallel Sum: 500000500000
Time: 0.074876
aftab@Aftab:~/Desktop/HPC/lab5$
```