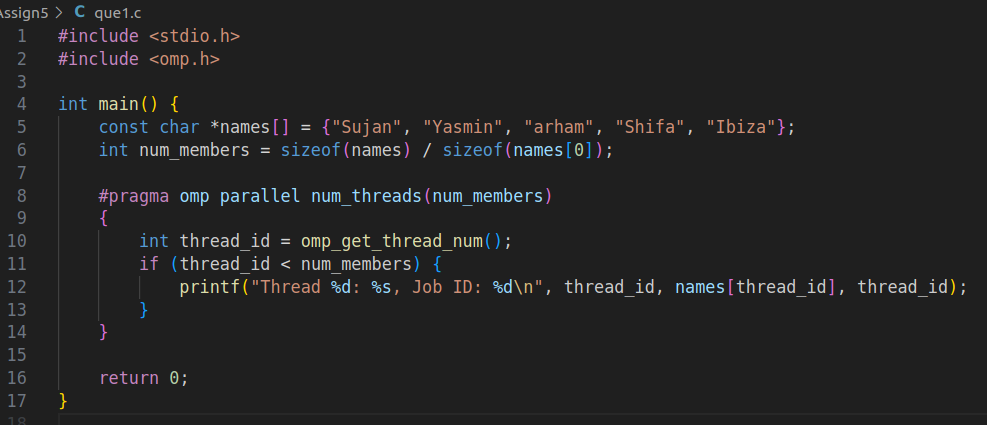
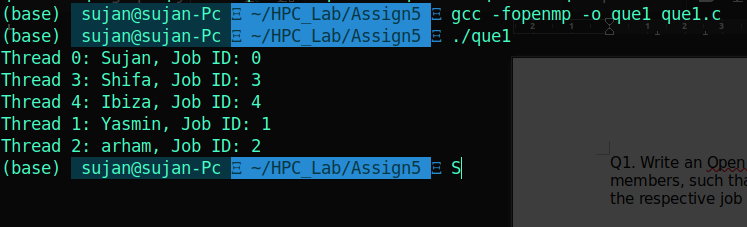
Q1. Write an OpenMP program such that, it should print the name of your family members, such that the names should come from different threads/cores. Also print the respective job id.

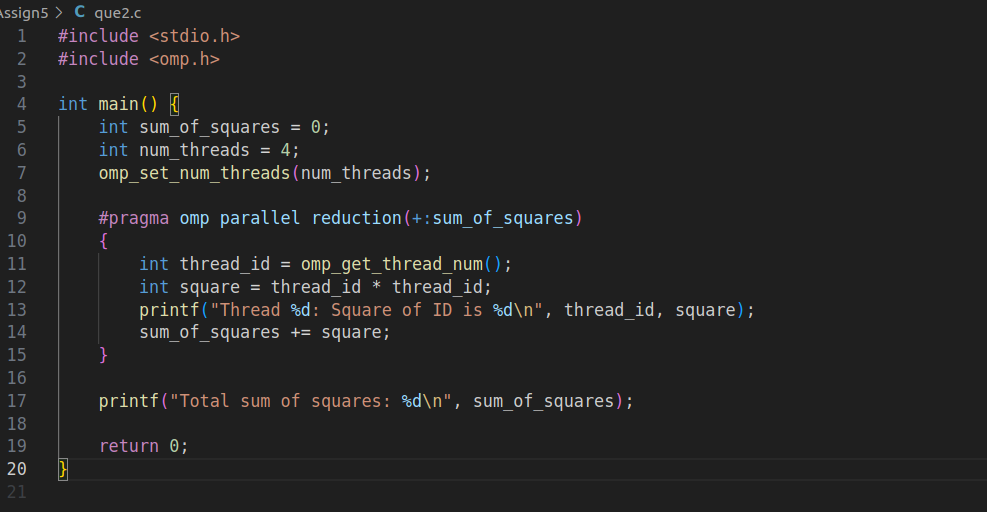
Code:



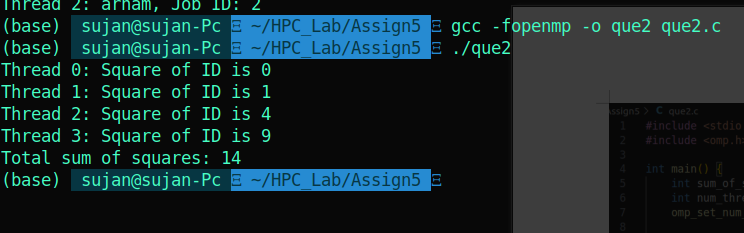


Q2. Write an OpenMP program such that, it should print the sum of square of the thread id’s. Also make sure that, each thread should print the square value of their thread id.

Code:



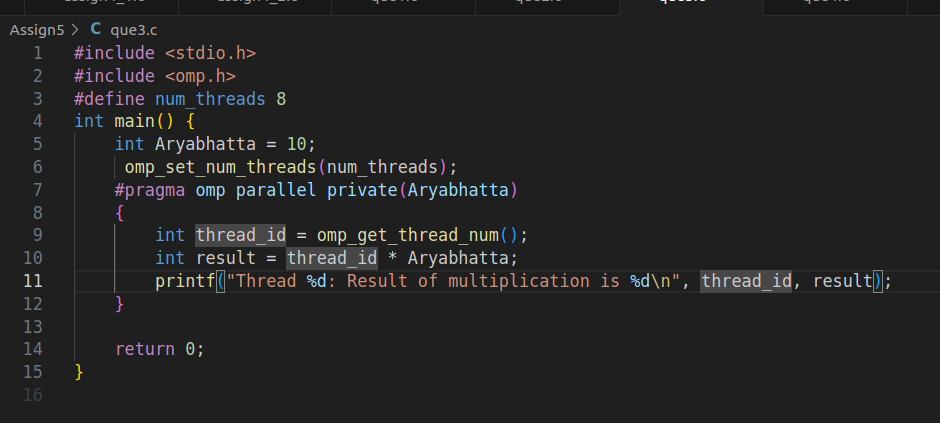
Output:



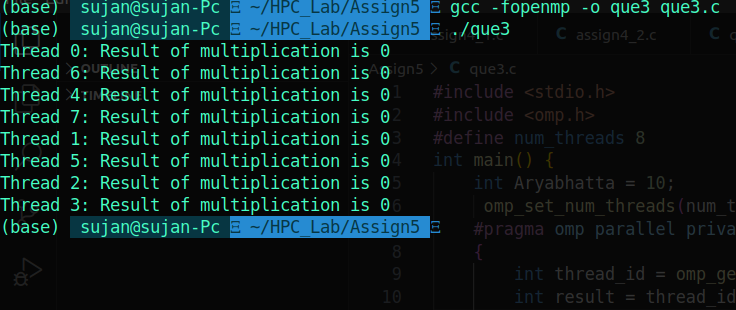
Q3. Consider a variable called “Aryabhatta” declared as 10 (i.e int Arbhatta=10).Write an OpenMP program which should print the result of multiplication of thread id and value of the above variable.

Note\*: The variable “Aryabhatta” should be declared as private

Code:



Output:



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.

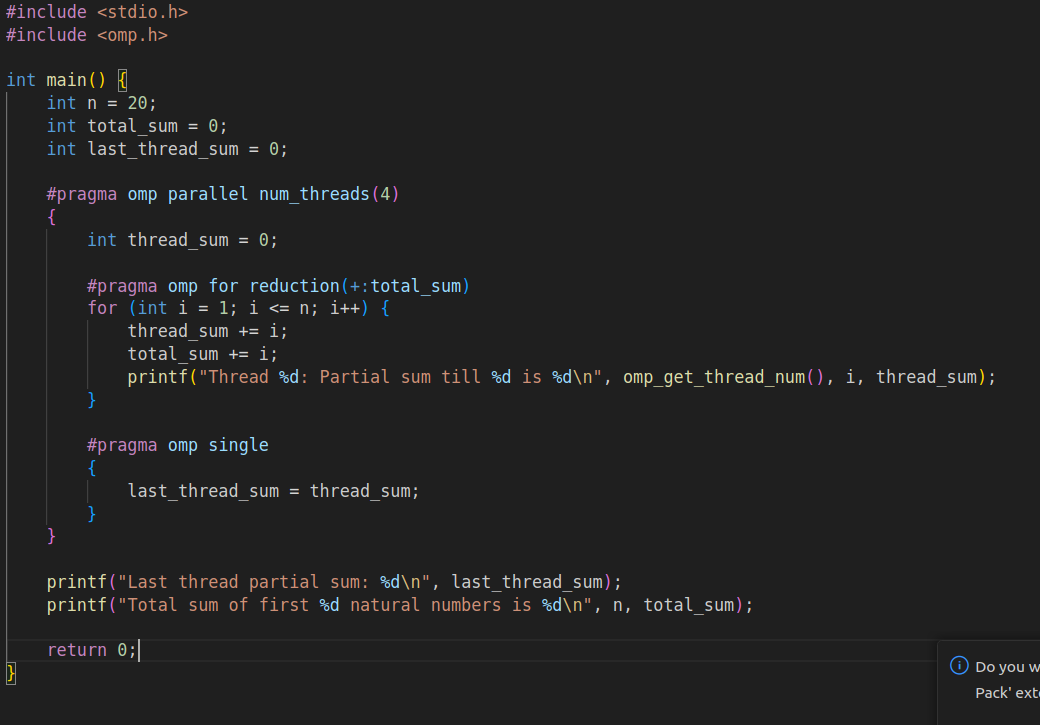
Hint:

1.Utilize OpenMP directives to parallelize the summation process.

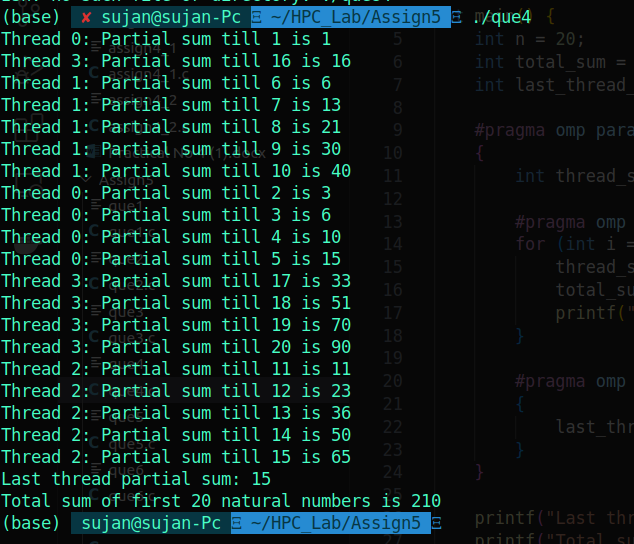
2.Ensure that each thread has its private copy of partial sum.

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.

Code:



Output:



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.

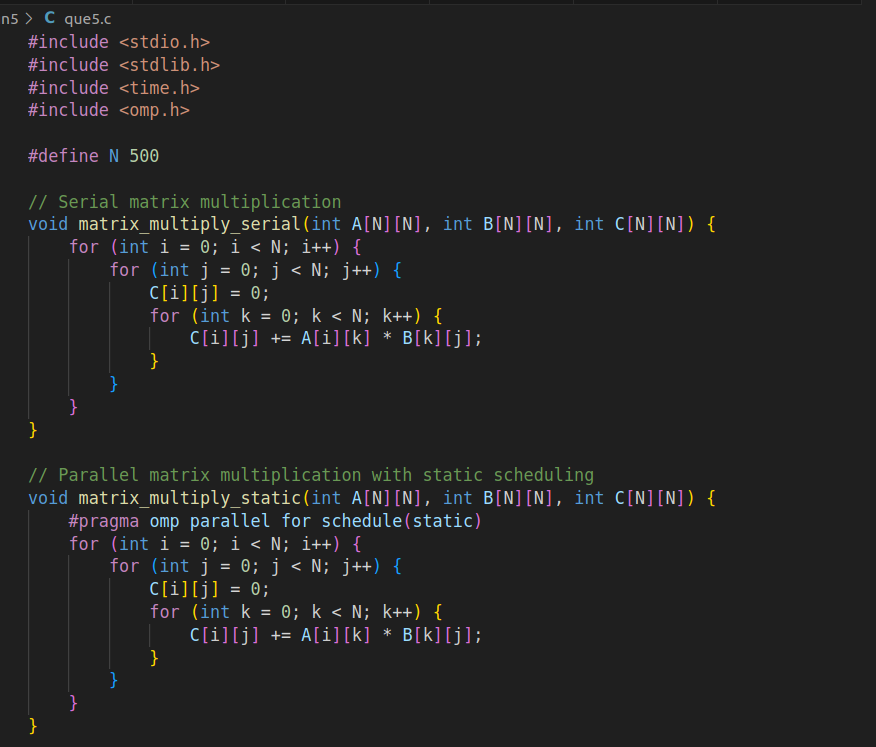
**Note\*:**

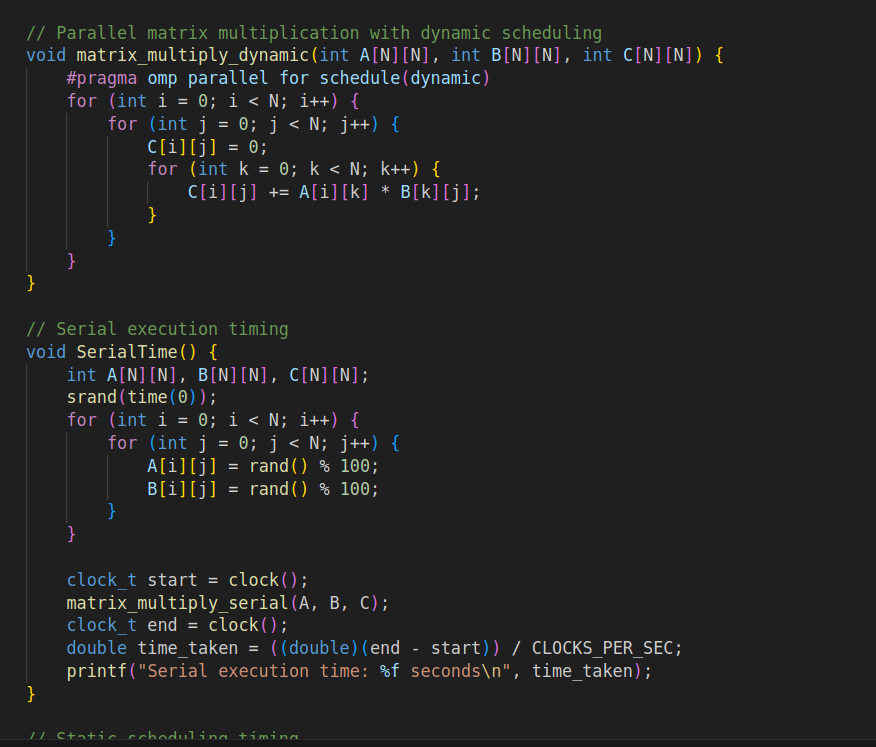
* Implement a serial version of matrix multiplication in C/C++.
* Parallelize the matrix multiplication using OpenMP with static scheduling.
* Parallelize the matrix multiplication using OpenMP with dynamic scheduling.
* Measure the execution time of each parallelized version for various matrix sizes.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

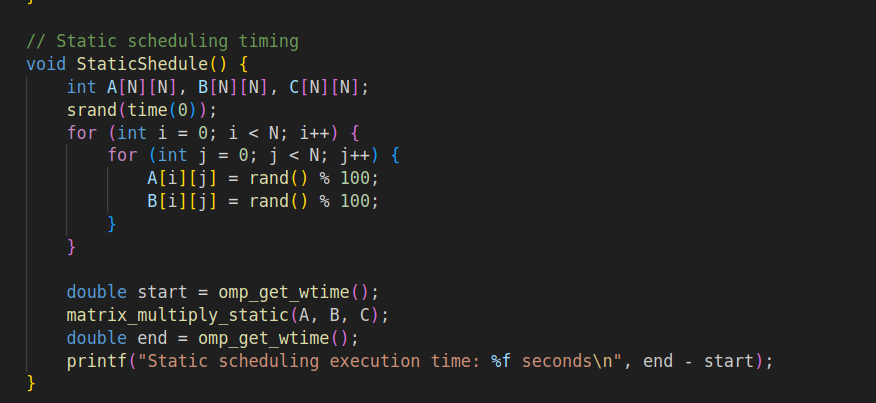
→

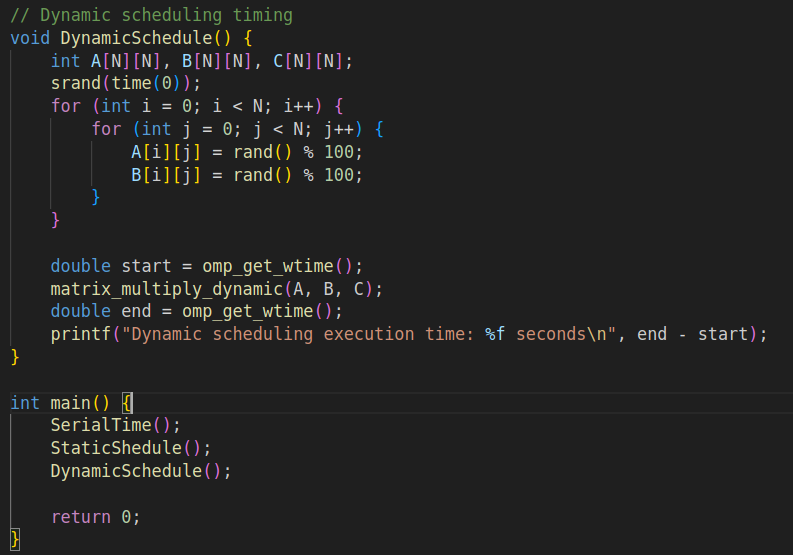
* **Static Scheduling**: Allocates iterations to threads in a fixed, predictable pattern. It works well when the computation load is evenly distributed among iterations.
* **Dynamic Scheduling**: Distributes iterations to threads as they become available. It’s useful when the workload varies significantly between iterations.

Code:

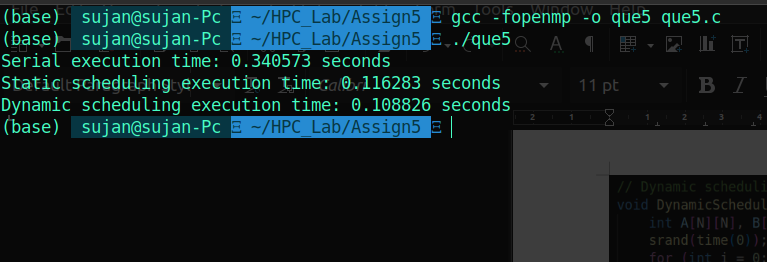








Output:

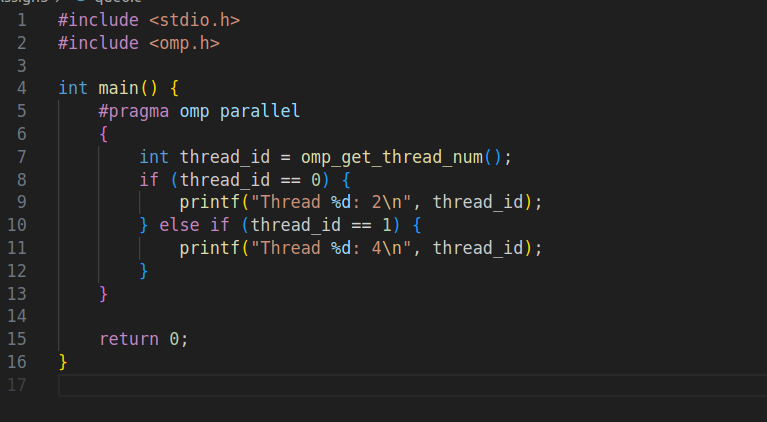


Analysis:

The reduction in execution time indicates that parallel processing with OpenMP is beneficial for matrix multiplication. The dynamic scheduling’s slight edge over static scheduling might be due to better load balancing across threads, which is particularly useful if the workload is not evenly distributed.

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

Code:

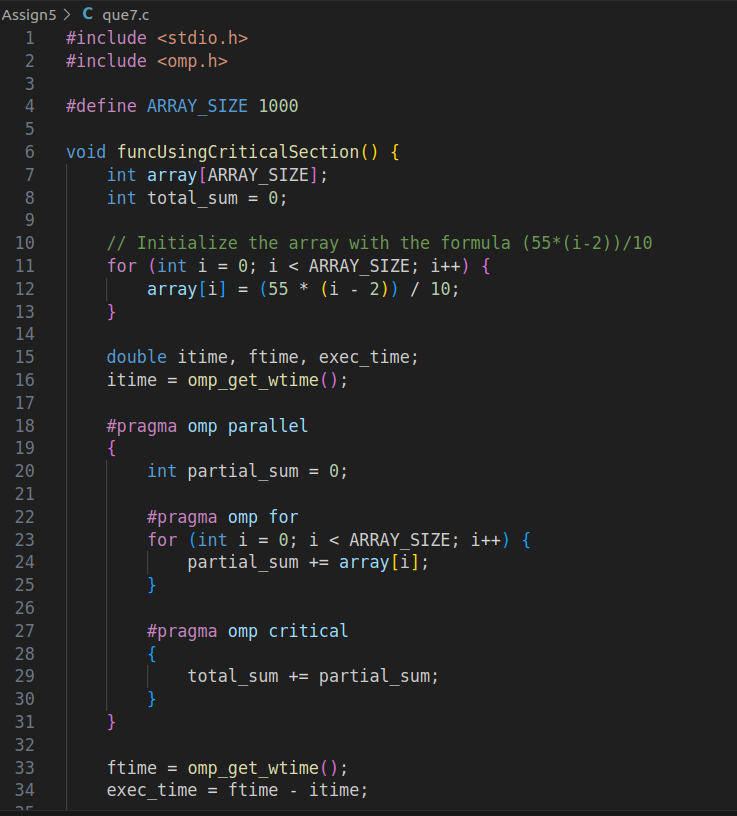
Output:

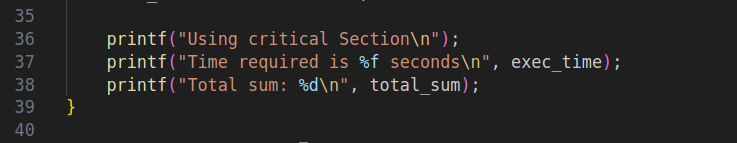


Q7. Consider a scenario where you have a shared variable total\_sum that needs to be updated concurrently by multiple threads in a parallel program. However, concurrent updates to this variable can result in data races and incorrect results. Your task is to modify the program to ensure correct synchronization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

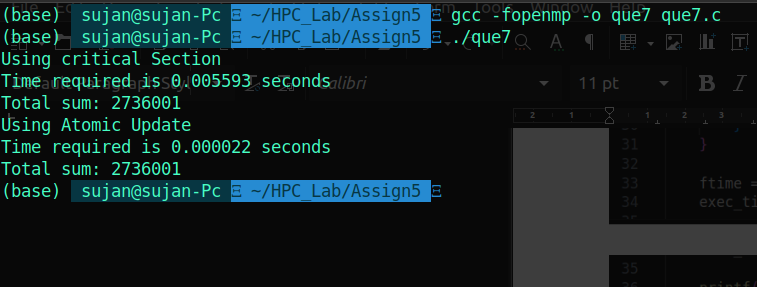
**Note\*:**

* 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.





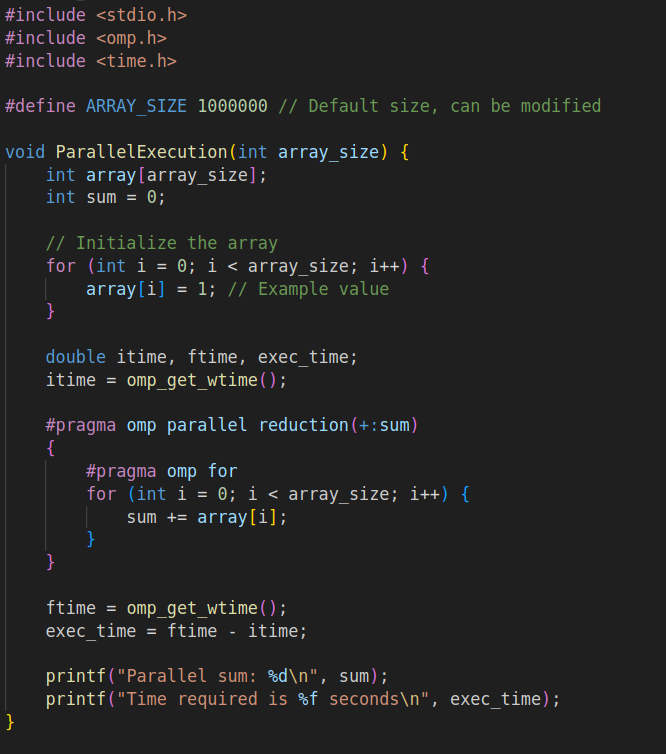
Output:

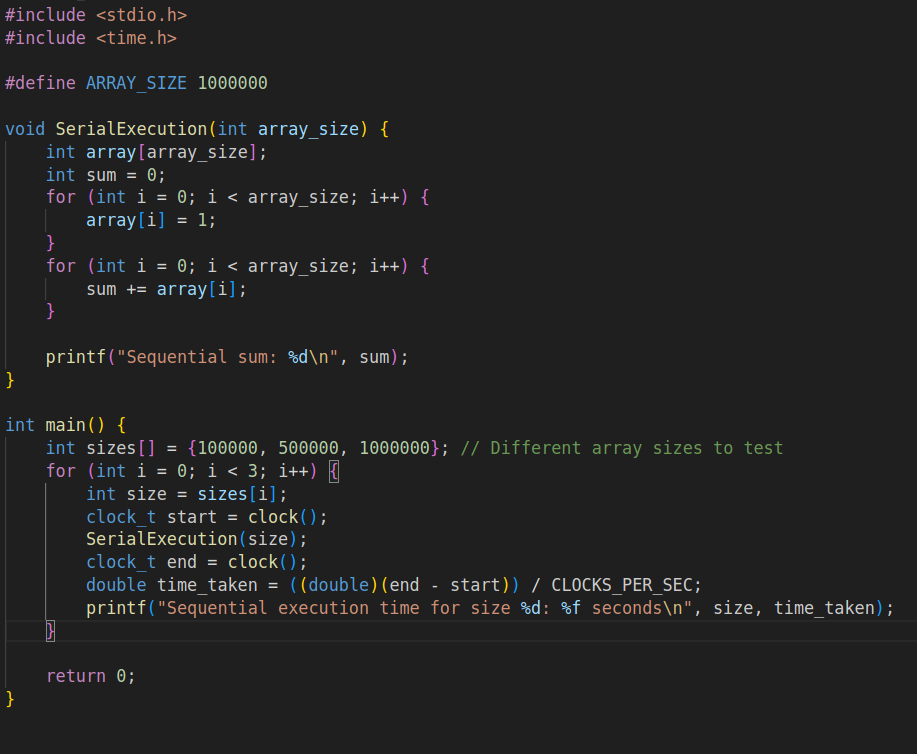


Q8. Consider a scenario where you have a large array of integers, and you need to find the sum of all its elements in parallel using OpenMP. The array is shared among multiple threads, and parallelism is needed to expedite the computation process. Your task is to write a parallel program that calculates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

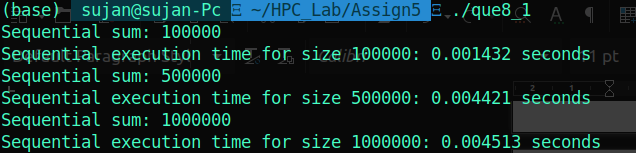
**Note\*:**

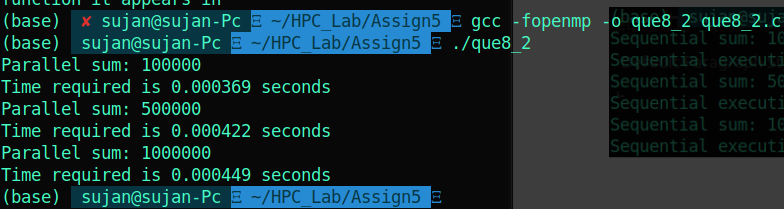
* Implement a sequential version of the program that calculates the sum of all elements in the array without using any parallelism.
* Identify potential bottlenecks and limitations of the sequential implementation in handling large arrays efficiently.
* Modify the program to utilize OpenMP's reduction 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.





Output:





Analysis:

|  |  |  |
| --- | --- | --- |
| Size of array | Serial | Parallel |
| 100000 | 0.001432 | 0.000369 |
| 500000 | 0.004421 | 0.000422 |
| 1000000 | 0.004513 | 0.000449 |

The parallel implementation consistently outperforms the sequential implementation as the array size increases. This is expected as the computational workload increases, making parallel processing more effective.

**Potential Bottlenecks and Limitations of Sequential Implementation**

**Single-Threaded Execution:** The sequential approach is single-threaded, meaning it cannot take advantage of multi-core processors. This limits its performance as all computation is done on a single core.

**Limited by Hardware:** As the array size grows, the sequential implementation does not scale with the number of processors or cores available, leading to inefficient use of modern multi-core processors.

**Single-Core Utilization:** Sequential execution utilizes only one core, leaving other cores idle, which can lead to inefficient resource utilization in a multi-core system.

**Advantages of Using the Reduction Clause for Parallel Summation**

**Performance with Large Arrays:** The parallel approach scales well with larger arrays because the workload is distributed among multiple threads. As the array size grows, the benefits of parallelism become more pronounced, leading to significant reductions in computation time.

**Adaptability:** The parallel implementation can adapt to varying numbers of threads, making it flexible and able to utilize the capabilities of different hardware configurations.

**Resource Utilization:** By leveraging multiple cores, the parallel implementation makes efficient use of available hardware resources, leading to better overall performance and reduced computation times.