**High Performance Computing**

**Lab**

**Assignment No. 5**

**Name : Sourabh Shankar Patil**

**PRN : 21510045**

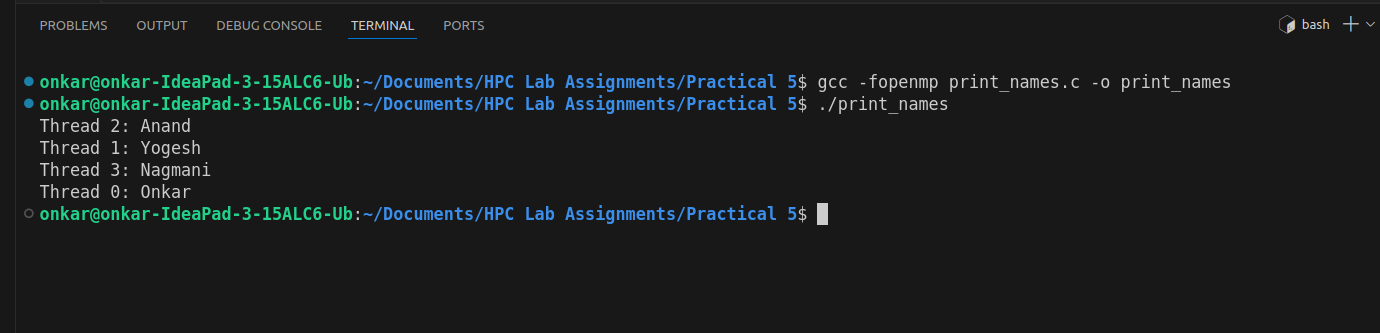
**Batch : B2**

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.

**Ans:**

**Print the Name of Family Members from Different Threads/Cores**

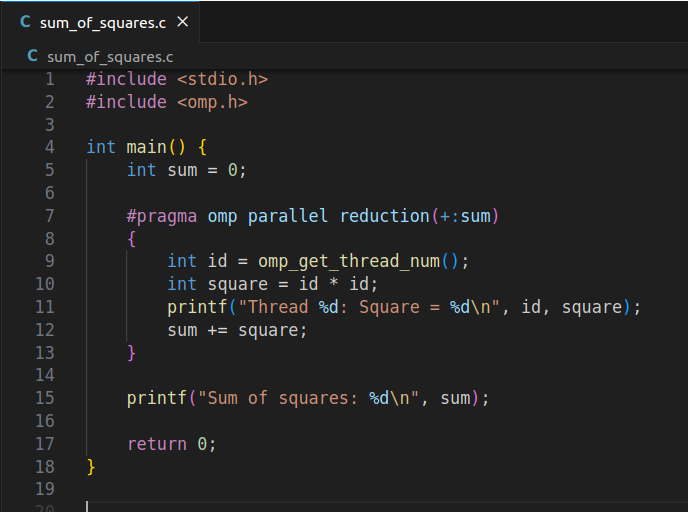


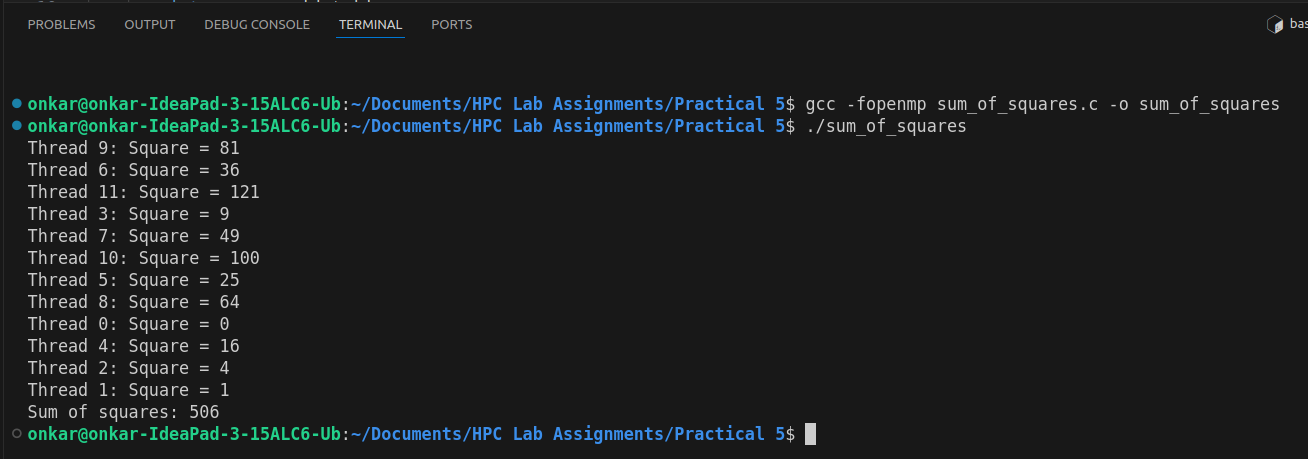


Q2. Write an OpenMP program such that, it should print the sum of square of the thread id’s. Also make sure that, each thread should print the square value of their thread id.

**Ans:**

**Sum of Squares of Thread Ids**



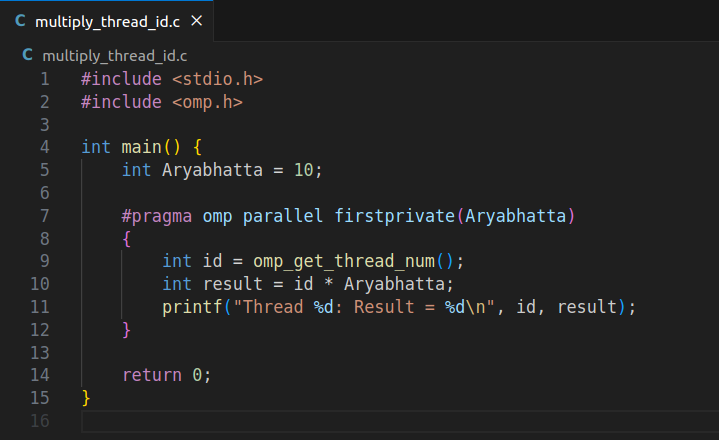


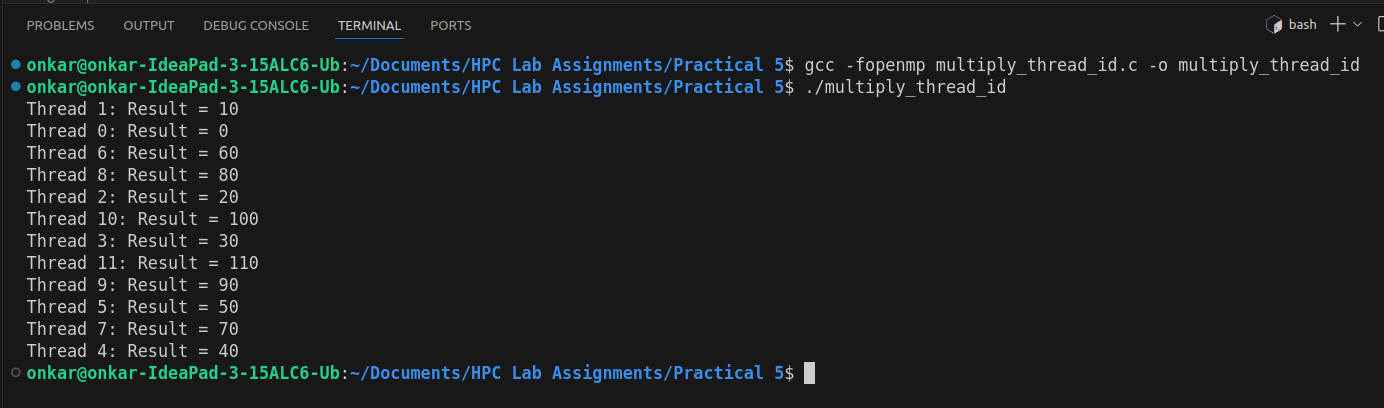
Q3. Consider a variable called “Aryabhatta” declared as 10 (i.e int Arbhatta=10).Write an OpenMP program which should print the result of multiplication of thread id and value of the above variable.

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

**Ans:**

### Multiplication of Thread ID with Private Variable If you want each thread to have its own copy of Aryabhatta with an initial value of 10, you can use the firstprivate clause instead of private. The firstprivate clause ensures that each thread's private copy of the variable is initialized with the value it had before the parallel region.





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.

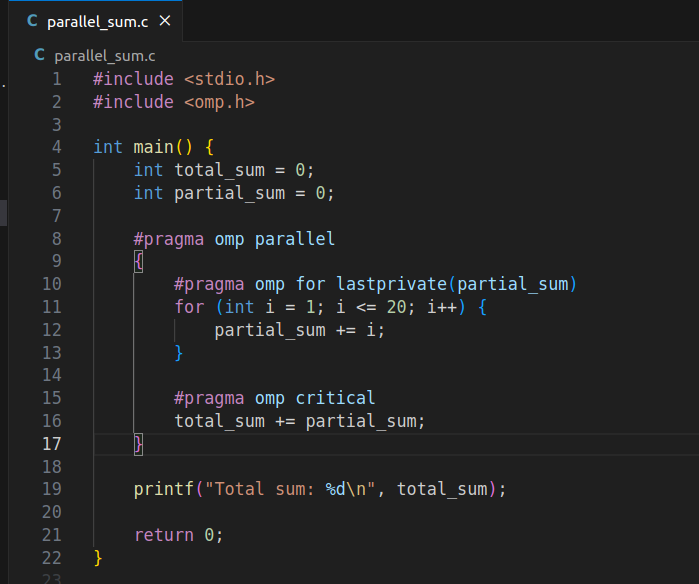
2. Ensure that each thread has its private copy of partial sum.

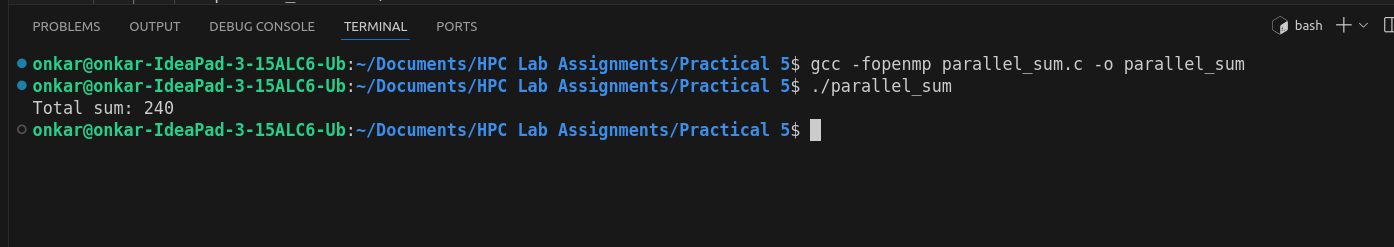
3. Use the lastprivate clause to assign the value of the last thread's partial sum to the final total sum after the parallel region.

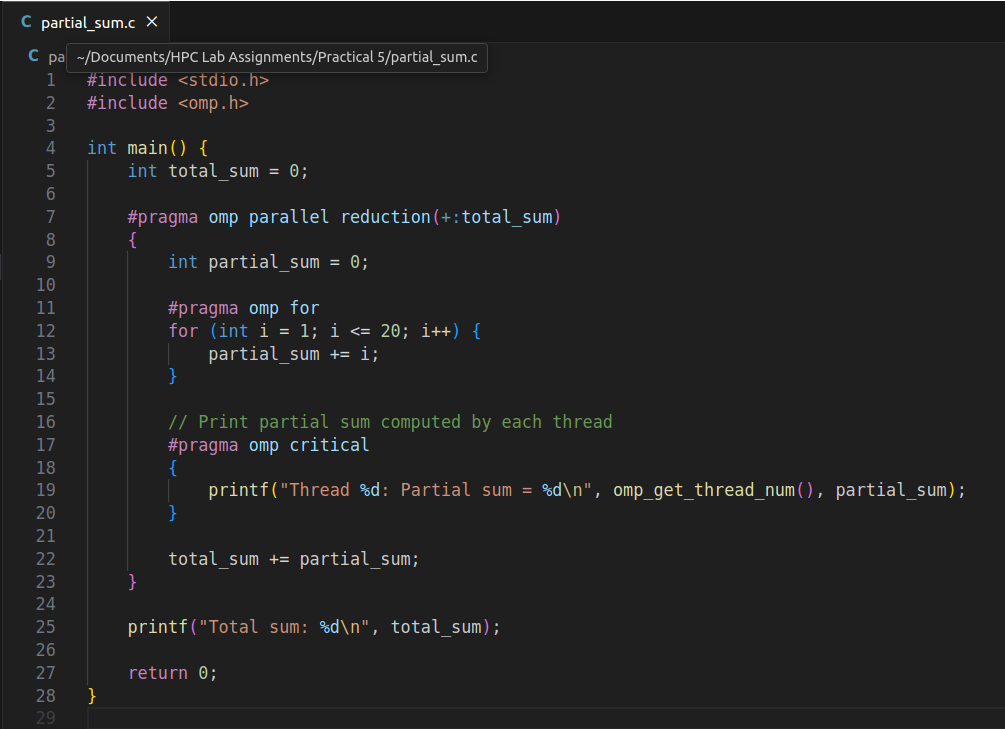
**Ans:**

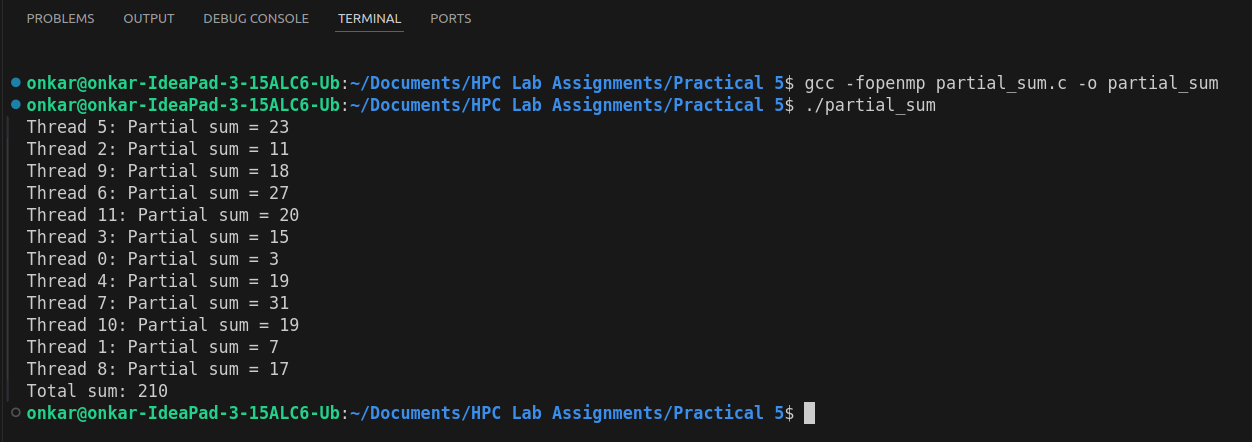
**Parallel Sum of First 20 Natural Numbers Using ‘lastprivate’**

reduction(+:sum): This clause tells OpenMP to create a private copy of sum for each thread. After the parallel region, these private sums are automatically added together to produce the final value of sum.









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:**

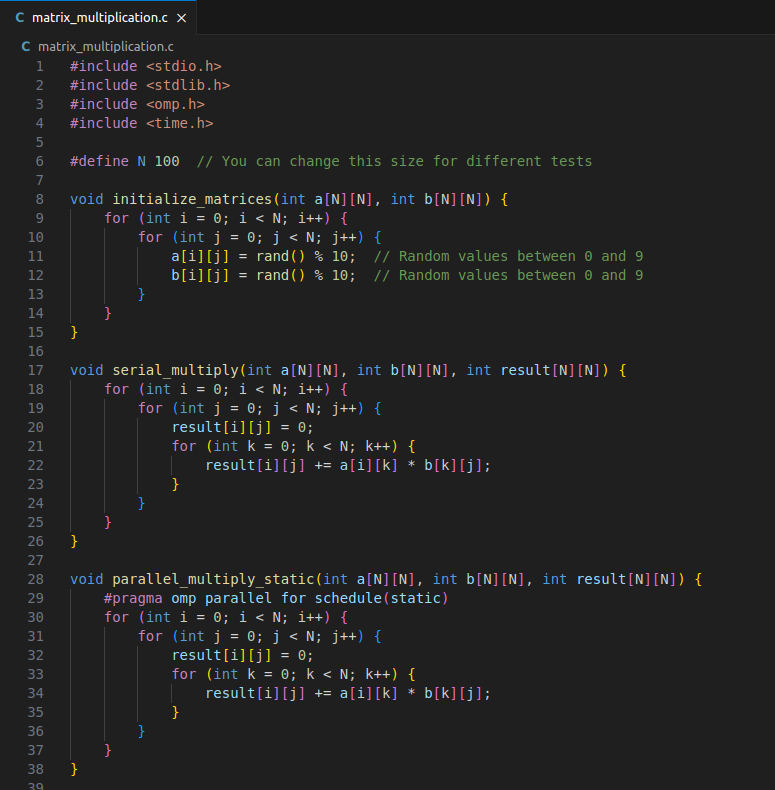
* Implement a serial version of matrix multiplication in C/C++.
* Parallelize the matrix multiplication using OpenMP with static scheduling.
* Parallelize the matrix multiplication using OpenMP with dynamic scheduling.
* Measure the execution time of each parallelized version for various matrix sizes.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

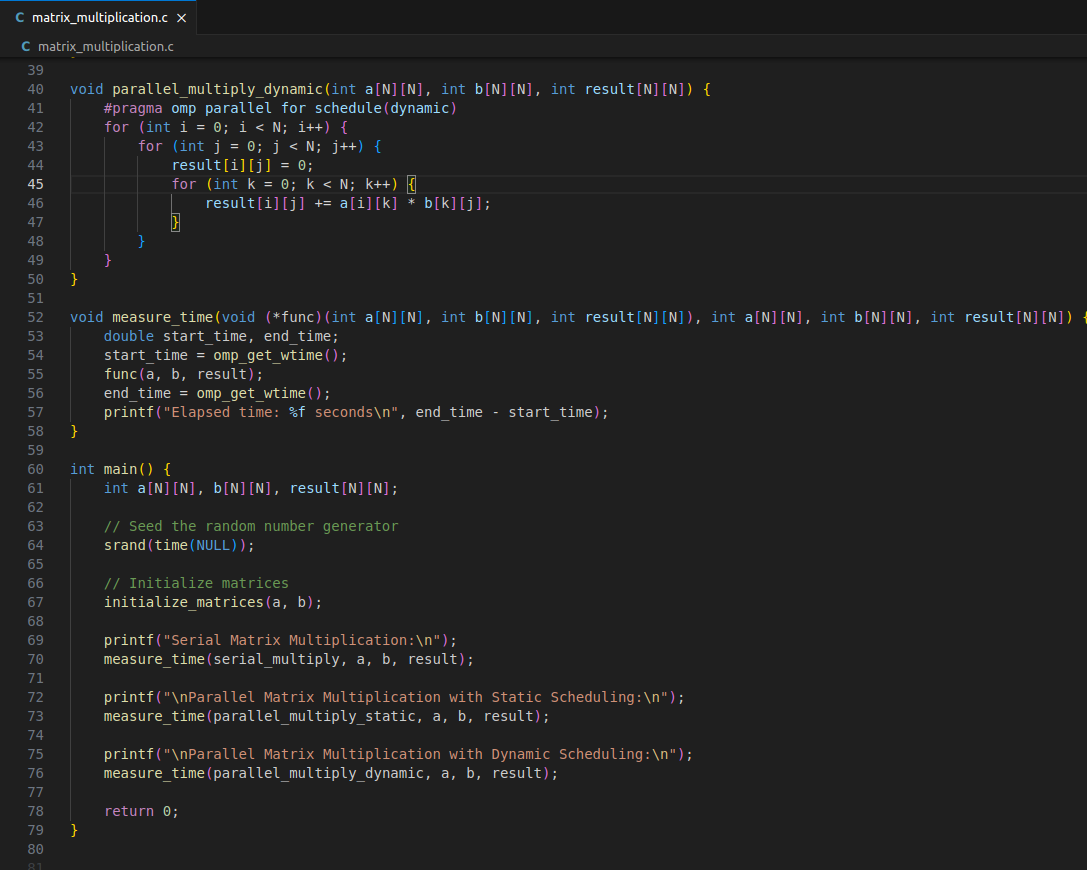
**Ans:**

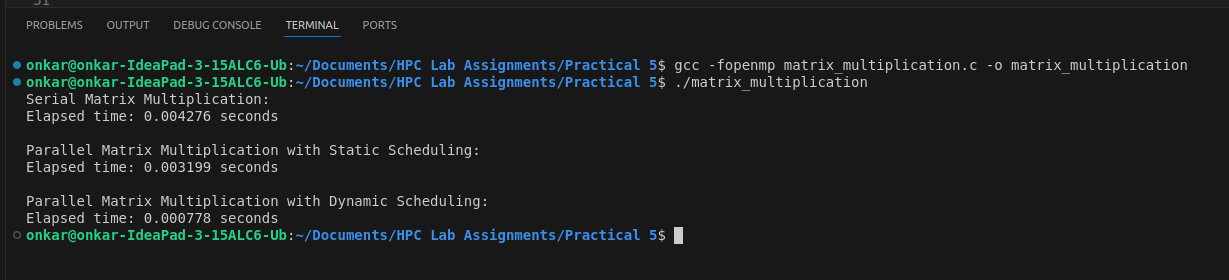
**Matrix Multiplication with Static and Dynamic Scheduling**

This task involves three parts:

1. Serial Matrix Multiplication
2. Parallel Matrix Multiplication with Static Scheduling
3. Parallel Matrix Multiplication with Dynamic Scheduling





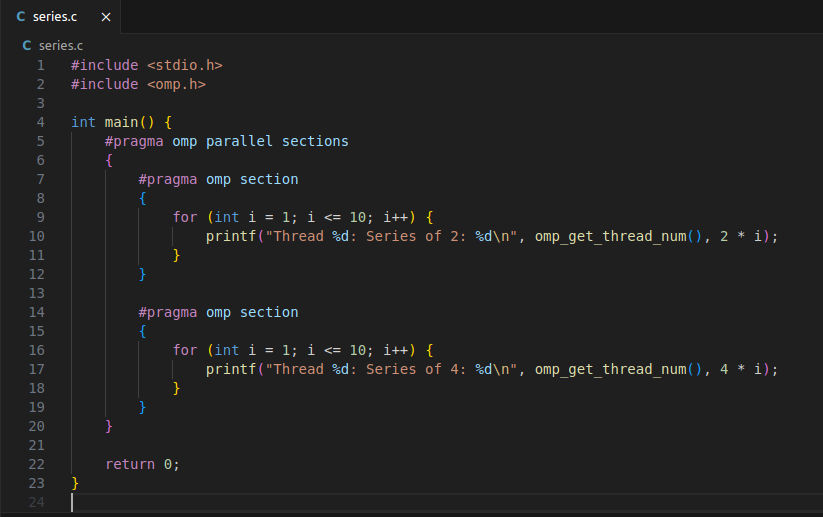
  
  
**Advantages and Disadvantages:**

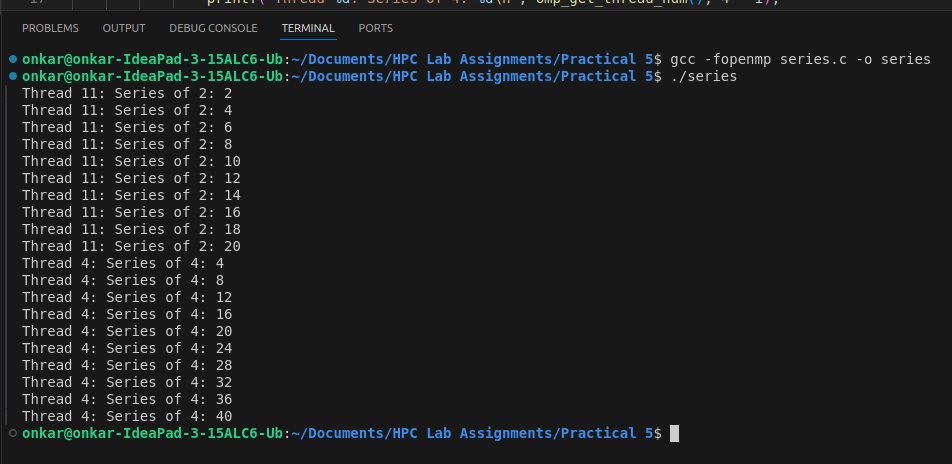
* Static Scheduling:
  + Advantages: Simple and efficient for tasks of uniform size and predictable workload.
  + Disadvantages: Less flexible with load balancing; might not perform well if the workload varies significantly.
* Dynamic Scheduling:
  + Advantages: Better load balancing as threads dynamically take on new chunks of work, leading to more even distribution of workload.
  + Disadvantages: Overhead of managing tasks dynamically might lead to performance loss compared to static scheduling.

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

**Ans:**

**Series of 2 and 4 in Different Threads**





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.

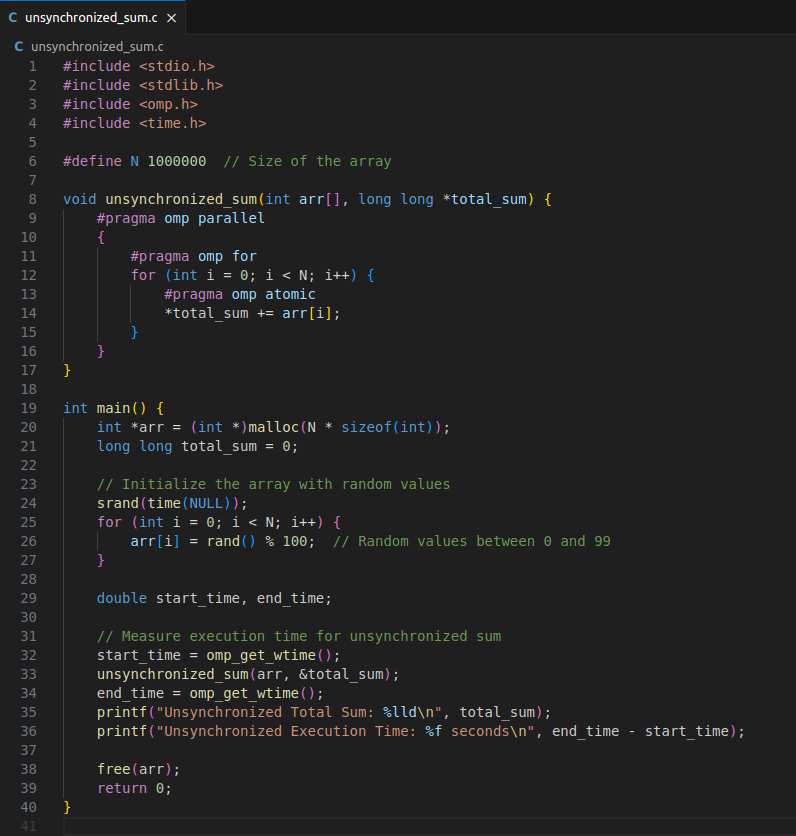
**Note:**

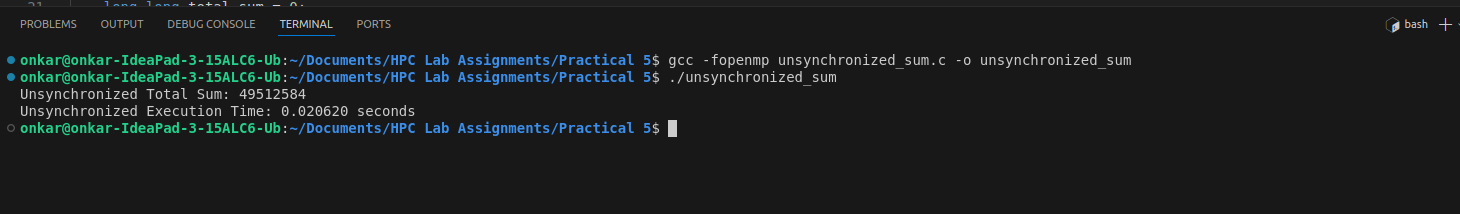
* Implement a simple parallel program in C that initializes an array of integers and calculates the sum of its elements concurrently using OpenMP.
* Identify potential issues with concurrent updates to the total\_sum variable in the parallelized version of the program.
* Modify the program to use OpenMP's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

**Ans:**

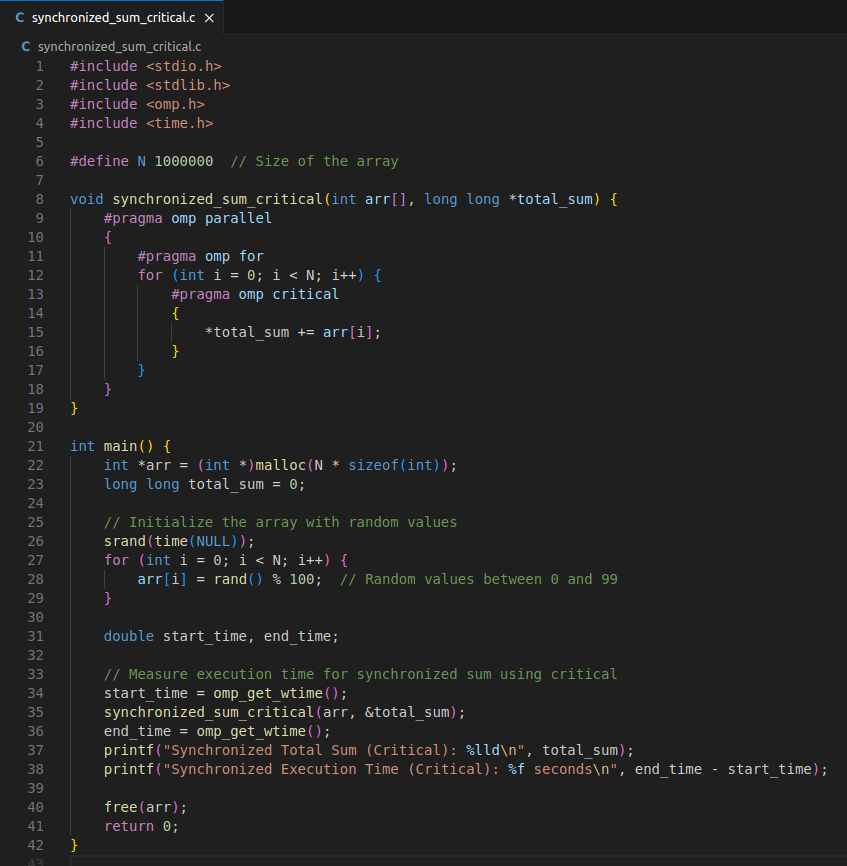
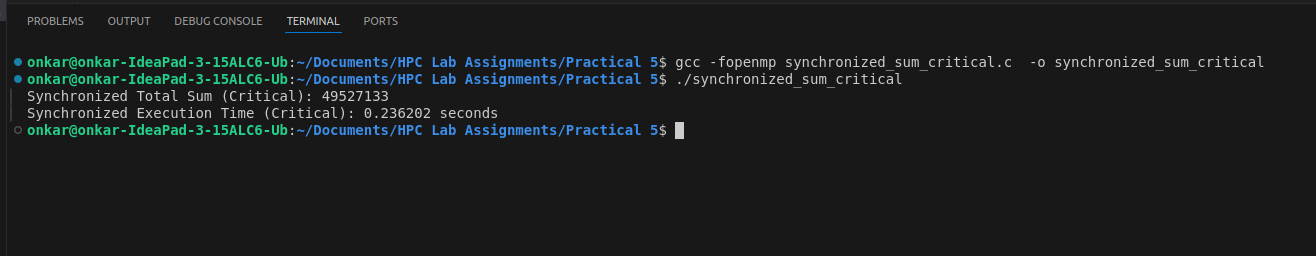
* **critical:** Ensures exclusive access to the critical section but may have higher overhead due to locking.
* **atomic:** Provides a lightweight mechanism for ensuring atomic updates, usually with better performance for simple operations but limited to specific operations like addition.

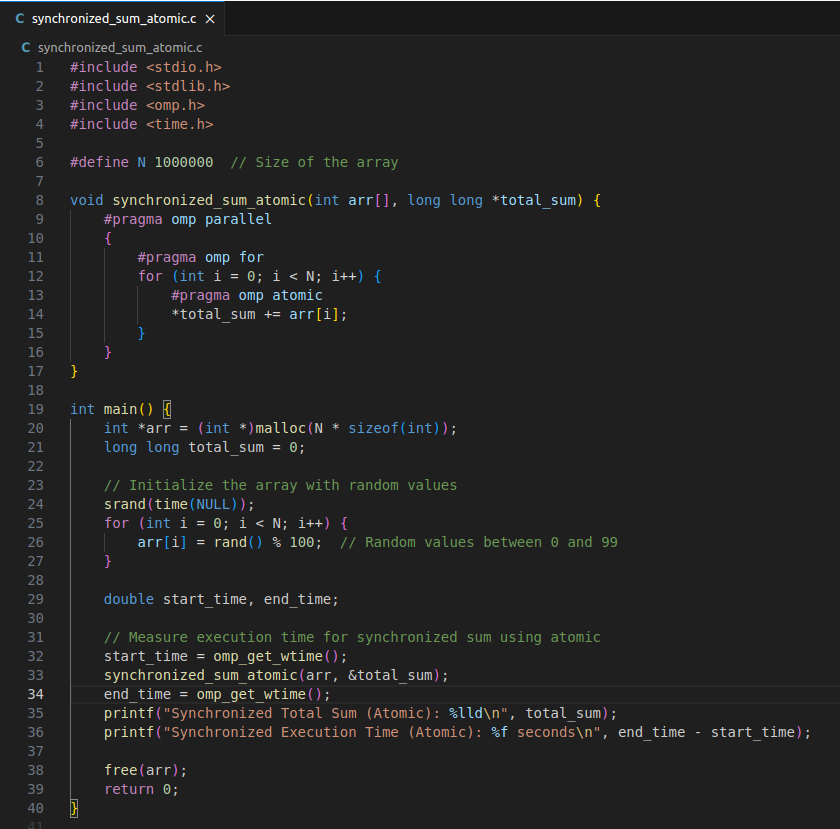
**Parallel Program with Unsynchronized Updates**

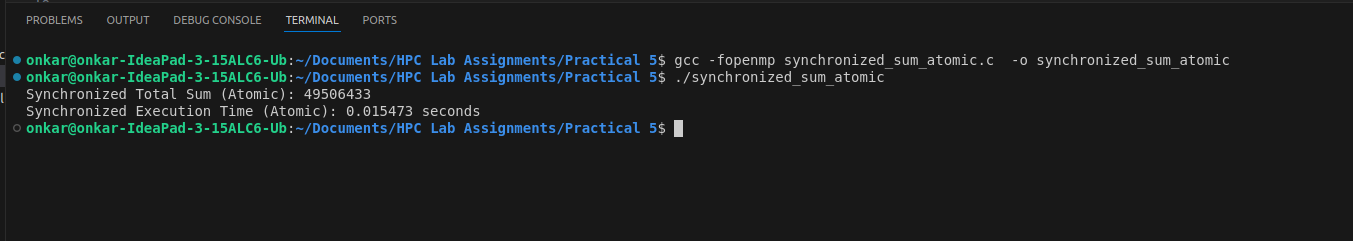




**Parallel Program with Synchronized Updates (Using critical)**

  
  
  
  
  
 **Parallel Program with Synchronized Updates (Using atomic)**



  
Unsynchronized Version: Demonstrates the potential data race by directly updating total\_sum without synchronization. This will often result in incorrect results due to concurrent updates.

* Synchronized Versions:
* Using critical: Ensures that only one thread can update total\_sum at a time, preventing data races but potentially causing contention.
* Using atomic: Provides a more efficient synchronization method than critical for simple operations like addition.

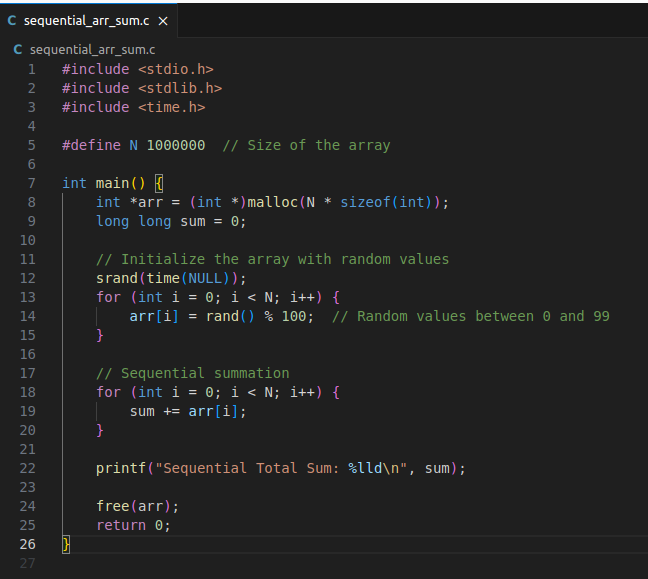
Q8. Consider a scenario where you have a large array of integers, and you need to find the sum of all its elements in parallel using OpenMP. The array is shared among multiple threads, and parallelism is needed to expedite the computation process. Your task is to write a parallel program that calculates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

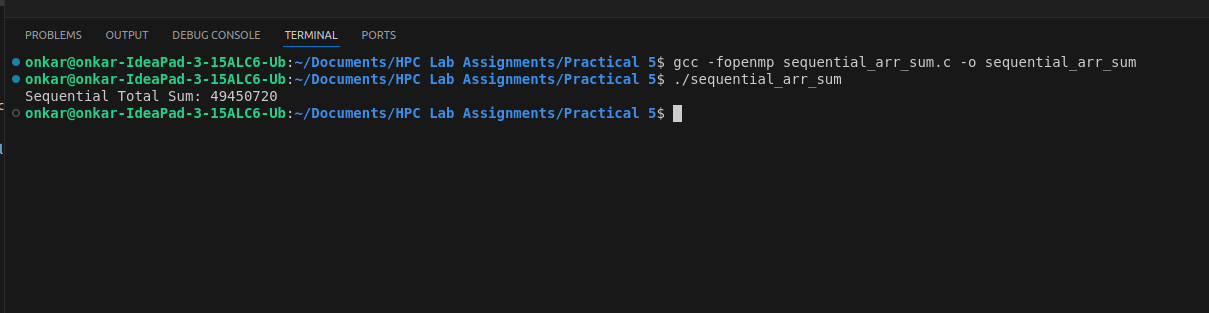
**Note:**

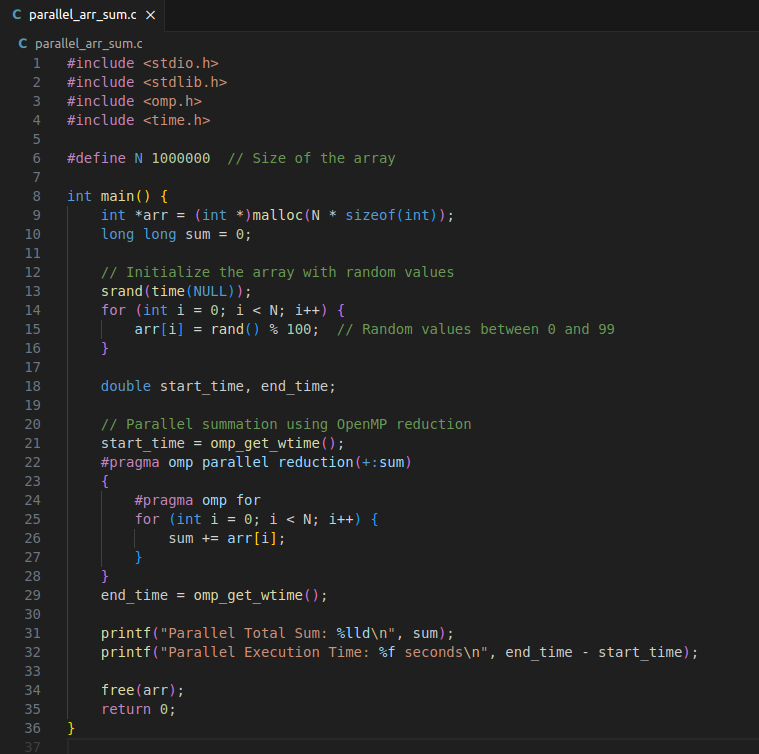
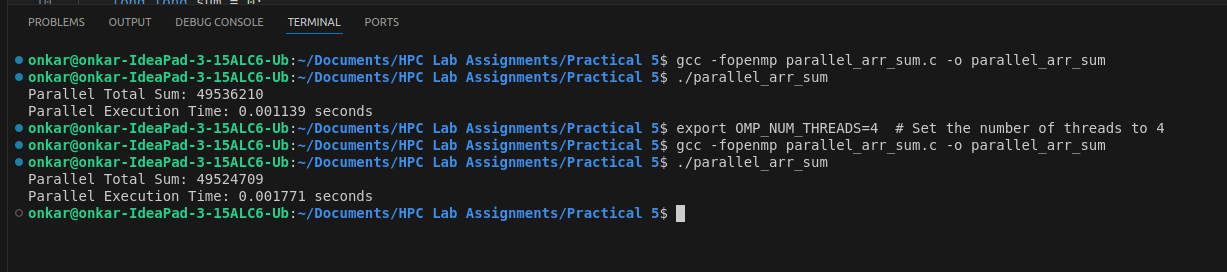
* Implement a sequential version of the program that calculates the sum of all elements in the array without using any parallelism.
* Identify potential bottlenecks and limitations of the sequential implementation in handling large arrays efficiently.
* Modify the program to utilize OpenMP's reduction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

**Ans:**

**Sequential Version**

****

 **Parallel Version with OpenMP Reduction**

****Advantages of reduction**:**

* + Efficiency: The reduction clause provides a way to safely and efficiently perform parallel reductions without explicit locking mechanisms.
  + Scalability: It scales well with the number of threads and can handle large arrays effectively, provided there are sufficient resources.
  + Simplified Code: Using reduction simplifies the code by automatically handling the synchronization and combination of partial results.

**Github Link:**<https://github.com/sourabh-patil-7/HPC-Lab-Assignments/tree/main/Practical%205>