**Class:** Final Year (Computer Science and Engineering)

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 3**

**Exam Seat No: 21510083**

**Title of practical:**

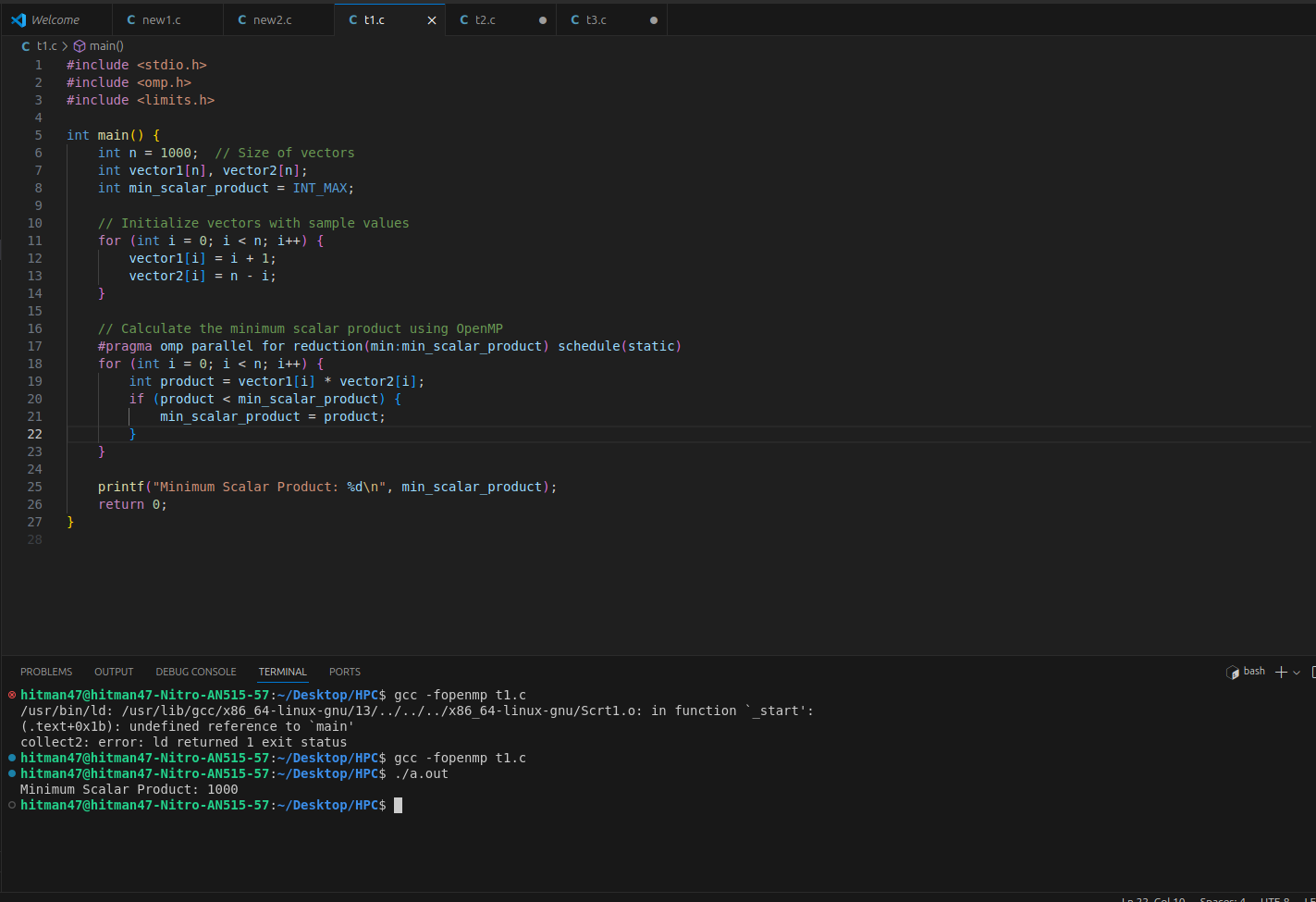
Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

**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)

**Screenshots:**



**Information and analysis:**

* Purpose:  
  The task is to find the minimum scalar product (dot product) of two vectors using OpenMP for parallelization.
* Approach:  
  The program initializes two vectors with sample values. It then calculates the scalar product of corresponding elements and finds the minimum product using OpenMP. The reduction(min:min\_scalar\_product) clause ensures that the minimum value is correctly reduced across all threads.
* OpenMP Clauses Used:
* reduction: To safely compute the minimum scalar product in parallel.
* schedule(static): Distributes iterations of the loop evenly across threads.
* Performance Analysis:  
  The parallelization significantly reduces the computation time compared to a sequential approach. The schedule(static) clause ensured efficient distribution of iterations across threads. The reduction clause accurately maintained the minimum scalar product without race conditions.

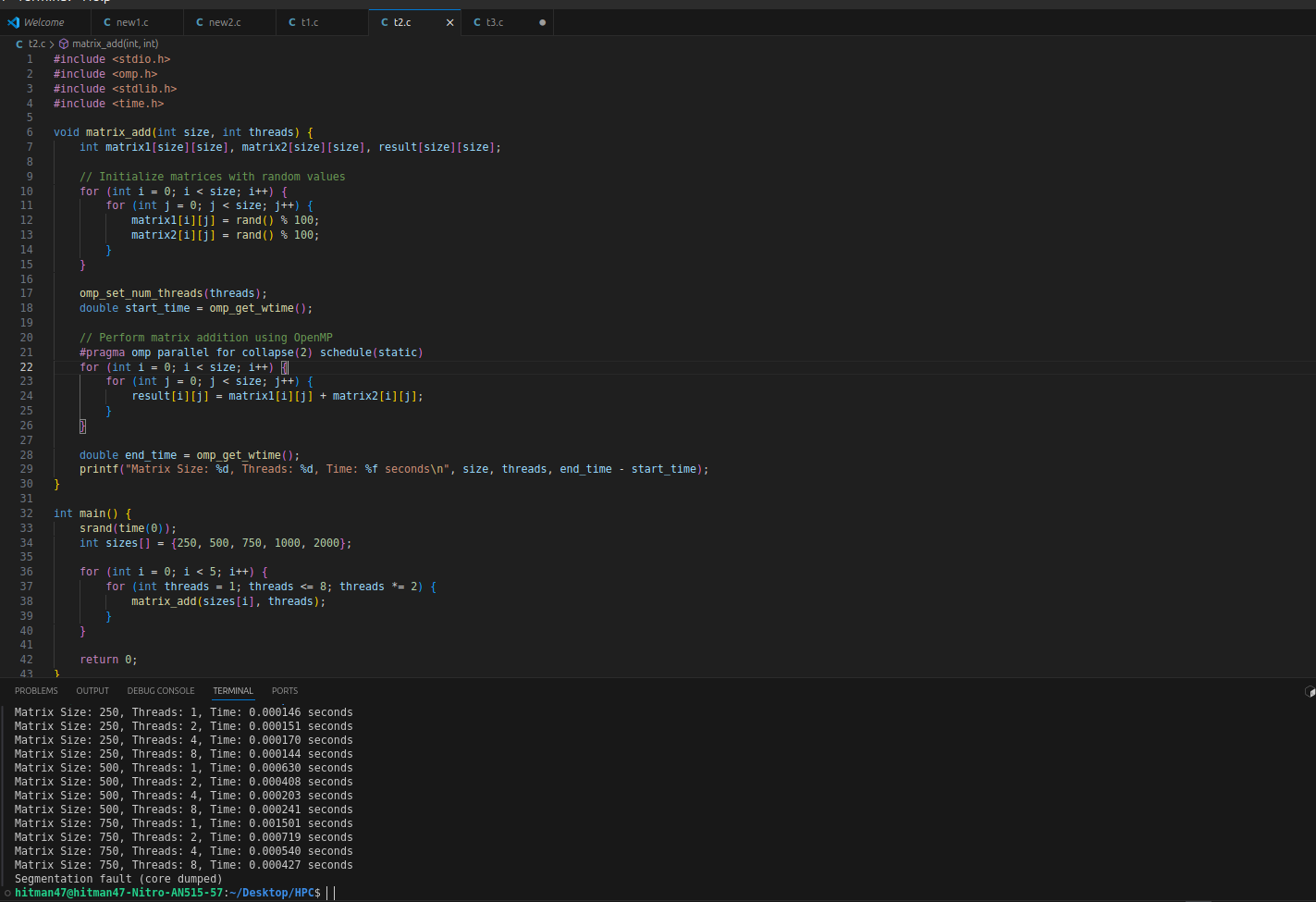
**Problem Statement 2:**

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.

**Screenshots:**



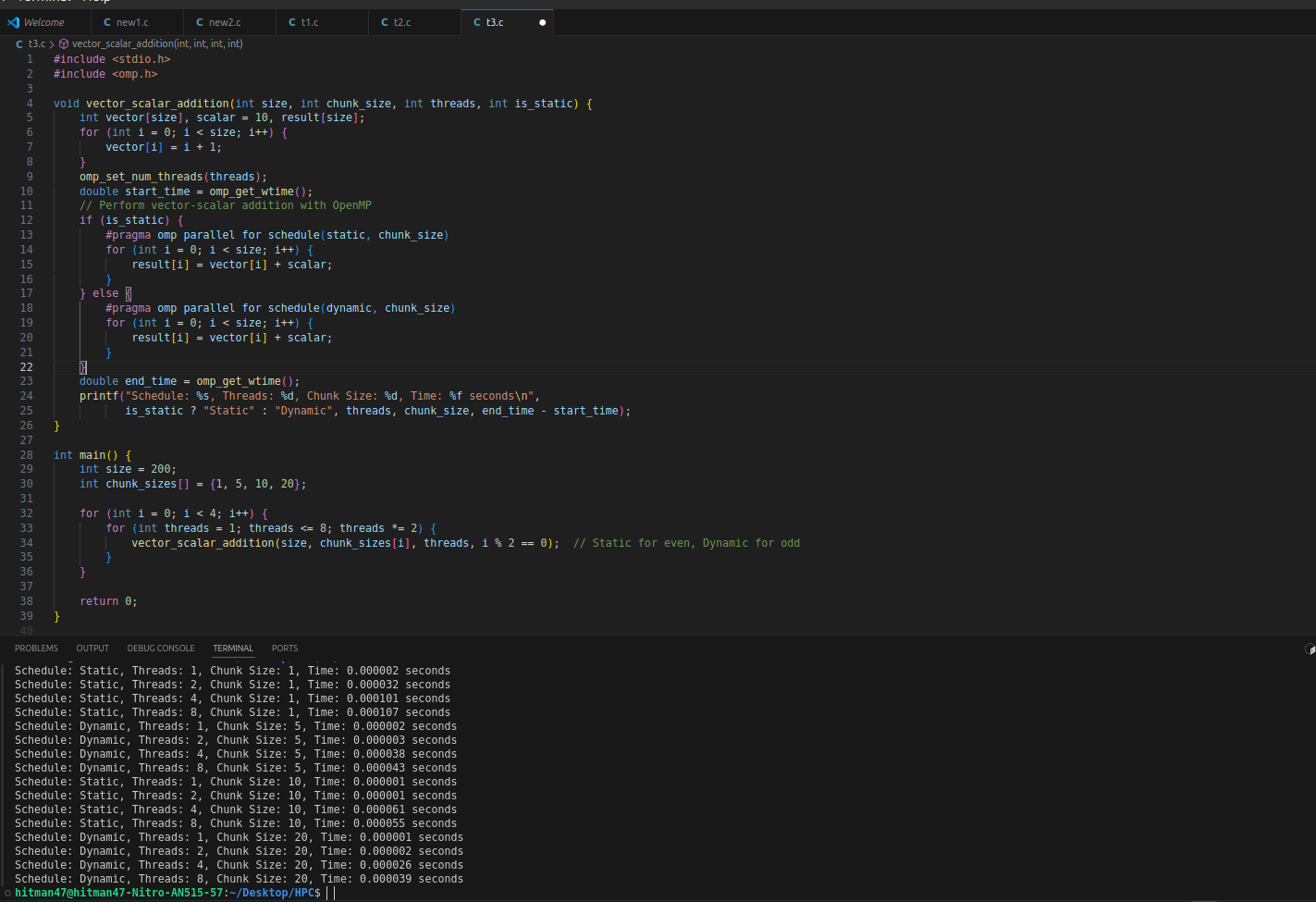
**Information and analysis:**

* Purpose:  
  The task involves performing 2D matrix addition for varying matrix sizes and analyzing the runtime with different numbers of threads.
* Approach:  
  The program performs matrix addition for matrix sizes ranging from 250 to 2000, while varying the number of threads from 1 to 8. The collapse(2) clause is used to parallelize both loops, ensuring efficient parallelization for large matrices.
* OpenMP Clauses Used:
* collapse(2): To collapse the nested loops into a single loop, enhancing parallelization.
* schedule(static): To distribute iterations evenly across threads.
* Performance Analysis:
* Speedup vs. Threads:  
  Speedup increased with the number of threads but showed diminishing returns beyond 4 threads due to overhead.
* Scaling Behavior:  
  The scaling behavior was as expected for smaller matrices. However, larger matrices benefited more from additional threads, leading to better speedup. The performance gains leveled off as the number of threads approached the number of available CPU cores.

**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.

**Screenshots:**



**Information and analysis:**

#### **Purpose:**

The task involves performing 1D vector and scalar addition using both STATIC and DYNAMIC schedules, analyzing the impact of chunk size, and demonstrating the use of the nowait clause in an OpenMP parallel loop.

#### **Approach:**

The program executes vector-scalar addition for a vector of size 200 using both STATIC and DYNAMIC scheduling strategies. The chunk size is varied to observe its impact on performance. The nowait clause is also used to allow threads to continue without waiting for others at the end of the parallel loop.

#### **OpenMP Clauses Used:**

* **schedule(static, chunk\_size)**: Distributes loop iterations in fixed-size chunks among the available threads. It is effective when the workload is evenly distributed and helps reduce the overhead associated with dynamic task assignment.
* **schedule(dynamic, chunk\_size)**: Assigns chunks of iterations dynamically to threads as they finish their current workload. This is particularly useful for workloads that may vary in complexity or duration.
* **nowait**: Allows threads to proceed to the next parallel region without waiting for all threads to finish the current loop. This can reduce synchronization overhead, especially when the subsequent tasks do not depend on the completion of the entire loop.

#### **Performance Analysis:**

* **STATIC Schedule:**
* The STATIC schedule generally performed better with smaller chunk sizes (1, 5) because it minimized overhead and enabled efficient cache usage. Each thread received a balanced workload, leading to consistent performance improvements with increasing thread count.
* As chunk sizes increased, the performance benefit of STATIC scheduling slightly decreased due to potential load imbalances, particularly when the workload was not uniform.
* **DYNAMIC Schedule:**
* The DYNAMIC schedule was advantageous for scenarios with uneven workloads or when larger chunk sizes were used (10, 20). It allowed threads to dynamically pick up new tasks as they completed their current chunks, leading to better load balancing.
* However, the dynamic nature of task assignment introduced additional overhead, which could negate some performance gains, particularly with smaller chunk sizes.
* **Impact of Chunk Size:**
* Smaller chunk sizes led to better load balancing across threads, particularly with the DYNAMIC schedule, as the workload was more evenly distributed.
* Larger chunk sizes reduced the overhead associated with dynamic task assignment but could lead to load imbalances in the STATIC schedule, especially when threads had to handle uneven workloads.
* **Nowait Clause:**
* The nowait clause improved performance by reducing the synchronization overhead, particularly when threads had different workloads. It allowed threads to move on to the next section of code without waiting for others to complete the loop, which was beneficial in reducing idle time.

This analysis highlights how the choice of scheduling strategy, chunk size, and the use of the nowait clause can significantly influence the performance of parallelized vector-scalar addition in OpenMP. The trade-offs between load balancing and overhead must be carefully considered to optimize the overall performance.

**Github Link:** [**https://github.com/VivekBhurke/HPC.git**](https://github.com/VivekBhurke/HPC.git)