THE ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING

Parallel and Distributed Computing

Parallel and Distributed Computing PROJECT II

ID:202112141

Name: Imad Salamah

Section: -1-

Total	/100

Good Luck!

Mr. Hussein Younis

Introduction

This project implements matrix transposition using three methods: sequential (basic loops), parallel with Pthreads, and parallel with OpenMP. Matrix transposition involves swapping rows with columns and is ideal for parallelization because each element can be processed independently, with no need for synchronization.

The goal is to compare the performance of these methods across different matrix sizes and thread counts, and to observe the speedup gained from parallel execution.

Sequential Implementation

In this phase, we wrote a C++ program to implement matrix transposition using a sequential approach.

We used two matrices: A (original data) and B (result after transposition).
The operation was performed using two nested for loops, where each value from A[i][j] is assigned to B[j][i].

The matrix A was filled with random values using the rand() function, and we measured the execution time using the chrono library.

At the end, we added a simple validation function that compares matrix B with the expected result to ensure correctness.

Parallelization Strategy

In the OpenMP-based implementation of the matrix transposition algorithm, parallelization was achieved using compiler directives, without the need to manually manage threads.

The core transposition logic—swapping `A[i][j]` to `B[j][i]`—was parallelized using the `#pragma omp parallel for`directive. This allowed the outer loop (over rows) to be divided among multiple threads. Since each thread accesses different memory locations, the operation is inherently thread-safe and does not require synchronization.

The number of threads was controlled using the `num_threads(NUM_THREADS)` clause, and performance was measured across varying thread counts.

Compared to the Pthreads implementation, OpenMP offered a simpler and more readable solution with minimal code changes. It allowed efficient parallelism with significantly less overhead and boilerplate code, making it suitable for quick and scalable parallel development.

Experiments

Hardware Specifications

The performance experiments were conducted using a macOS system. With hardware as follows:

Machine (macOS):

- Processor: Intel(R) Core(TM) i7-8559U CPU @ 2.70GHz
- Physical Cores: 4
- Logical Threads: 8
- Host RAM: 16 GB

Input Sizes and Thread Counts Tested

To thoroughly evaluate the performance and scalability of the sequential and parallel implementations of the matrix transposition algorithm, we designed a set of experiments involving multiple matrix sizes and a range of thread counts.

The selected matrix sizes were:

- **1000×1000** – representing a small-scale matrix
- **2000×2000**, **4000×4000**, **8000×8000** – representing medium to large sizes
- **16000×16000**, **32000×32000** – representing very large matrices that challenge memory usage and parallel efficiency

These sizes were chosen to investigate how the algorithm performs under increasing computational load and memory requirements. For each matrix size, the transposition was carried out using the parallel implementation with varying numbers of threads to assess the impact of multithreading on performance.

The number of threads used in testing were:

- **1 thread** – serving as a baseline for comparison with the sequential version
- **2 threads**
- **4 threads**
- **8 threads**
- **16 threads**

For every combination of matrix size and thread count, we measured both the **execution time** and the resulting **speedup**, calculated as:

Speedup=Sequential TimeParallel TimeSpeedup=Parallel TimeSequential Time

Each experiment was repeated several times to ensure consistency, and the average values were recorded. These values were then visualized using two comparative charts:

1. **Speedup vs. Thread Count**
2. – to observe how the speedup scales with parallelism
3. **Execution Time vs. Thread Count** – to illustrate the actual runtime trends of different matrix sizes

Results

The following charts summarize the results of these performance tests.

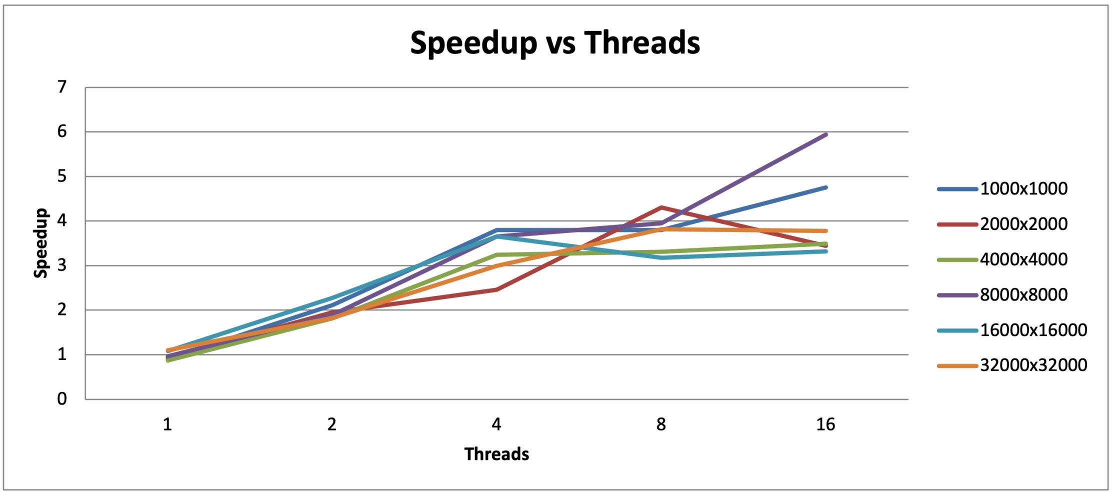


Figure 1:Speedup vs. Thread Count

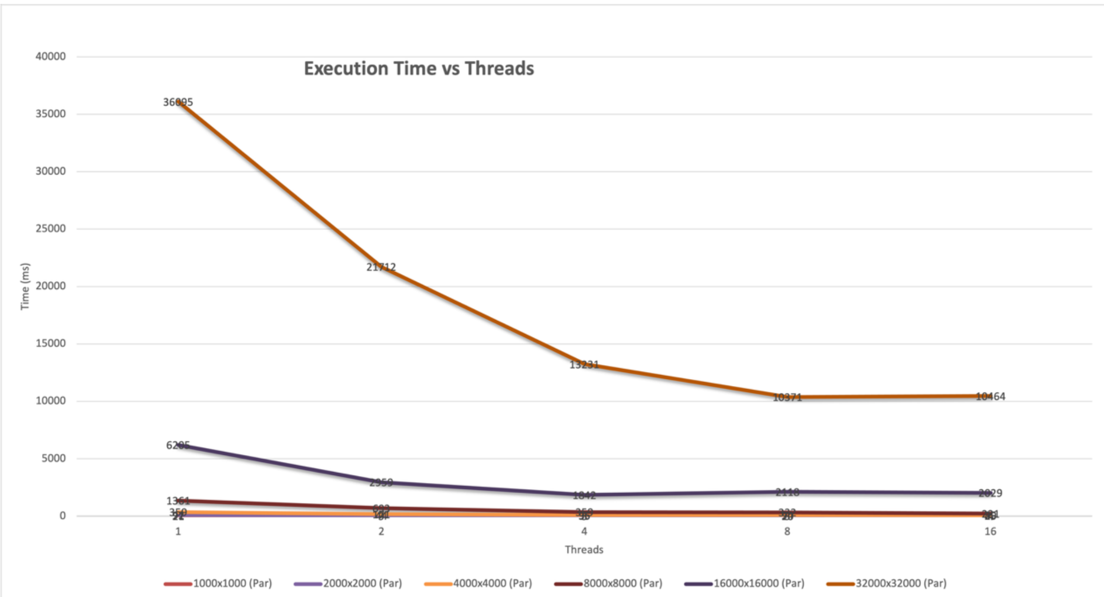


Figure 2:Execution Time vs. Thread Count

Matrix Size	Threads	Sequential Time (ms)	Parallel Time (ms)	Speedup
1000x1000	1	19	21	0.904761905
1000x1000	2	19	9	2.111111111
1000x1000	4	19	5	3.8
1000x1000	8	19	5	3.8
1000x1000	16	19	4	4.75
2000x2000	1	86	92	0.934782609
2000x2000	2	86	44	1.954545455
2000x2000	4	86	35	2.457142857
2000x2000	8	86	20	4.3
2000x2000	16	86	25	3.44
4000x4000	1	311	359	0.866295265
4000x4000	2	311	171	1.81871345
4000x4000	4	311	96	3.239583333
4000x4000	8	311	94	3.308510638
4000x4000	16	311	89	3.494382022
8000x8000	1	1312	1361	0.963997061
8000x8000	2	1312	693	1.893217893
8000x8000	4	1312	359	3.6545961
8000x8000	8	1312	332	3.951807229
8000x8000	16	1312	221	5.936651584
16000x16000	1	6729	6205	1.084448026
16000x16000	2	6729	2959	2.274079081
16000x16000	4	6729	1842	3.653094463
16000x16000	8	6729	2118	3.177053824
16000x16000	16	6729	2029	3.316412026
32000x32000	1	39569	36095	1.096246017
32000x32000	2	39569	21712	1.822448416
32000x32000	4	39569	13231	2.99062807
32000x32000	8	39569	10371	3.815350497
32000x32000	16	39569	10464	3.781441131

Table 1:result data

Discussion

matrix size increases. For smaller matrices such as 1000×1000, the speedup was modest due to thread management overhead outweighing computation time. However, as matrix dimensions grew—especially 8000×8000 and beyond—noticeable performance gains were achieved.

The maximum speedup reached nearly 6× at 16 threads for the 8000×8000 matrix. However, for many sizes, the improvement began to plateau or slightly decrease beyond 8 threads, due to overhead, thread contention, or memory bandwidth limitations.

As seen in the execution time chart, the parallel runtime consistently decreased with more threads, particularly for larger matrices like 32000×32000, where time dropped from 39500 ms to 10400 ms. This proves that multithreading can cut execution time by over 70%, especially on large datasets.

Despite that, adding more threads does not always yield better performance. For some matrix sizes, performance slightly declined at 16 threads compared to 8, reinforcing that optimal thread count varies depending on workload.

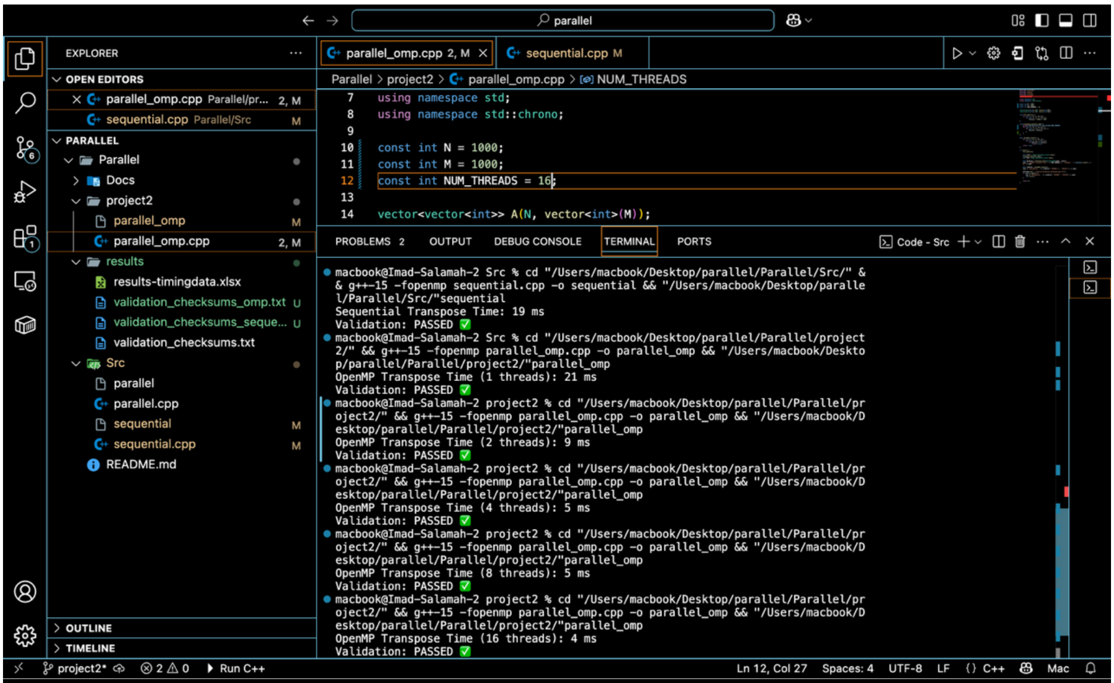
Overall, the results confirm that OpenMP-based parallelization offers significant performance benefits for computationally heavy operations like matrix transposition.

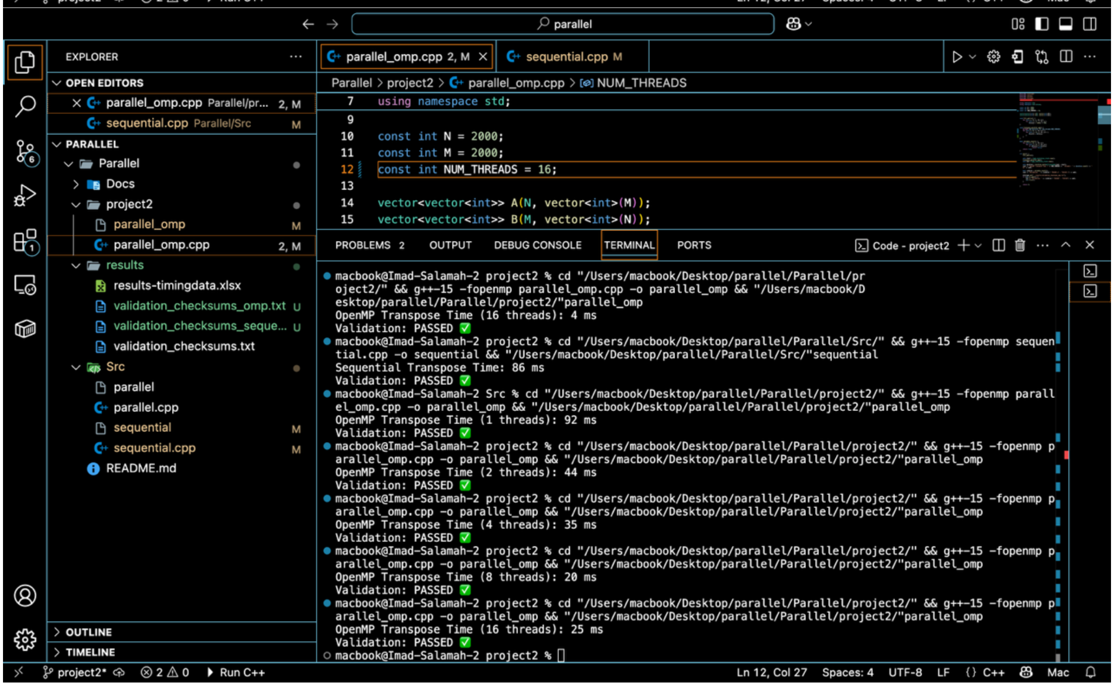
Conclusion

This project demonstrated the effectiveness of parallel programming in accelerating matrix transposition. By comparing sequential, Pthreads, and OpenMP implementations, we observed significant performance improvements—especially with large matrices and up to 8 threads. OpenMP provided a simple and efficient way to parallelize the algorithm, confirming that choosing the right thread count is key to maximizing speedup with minimal overhead.

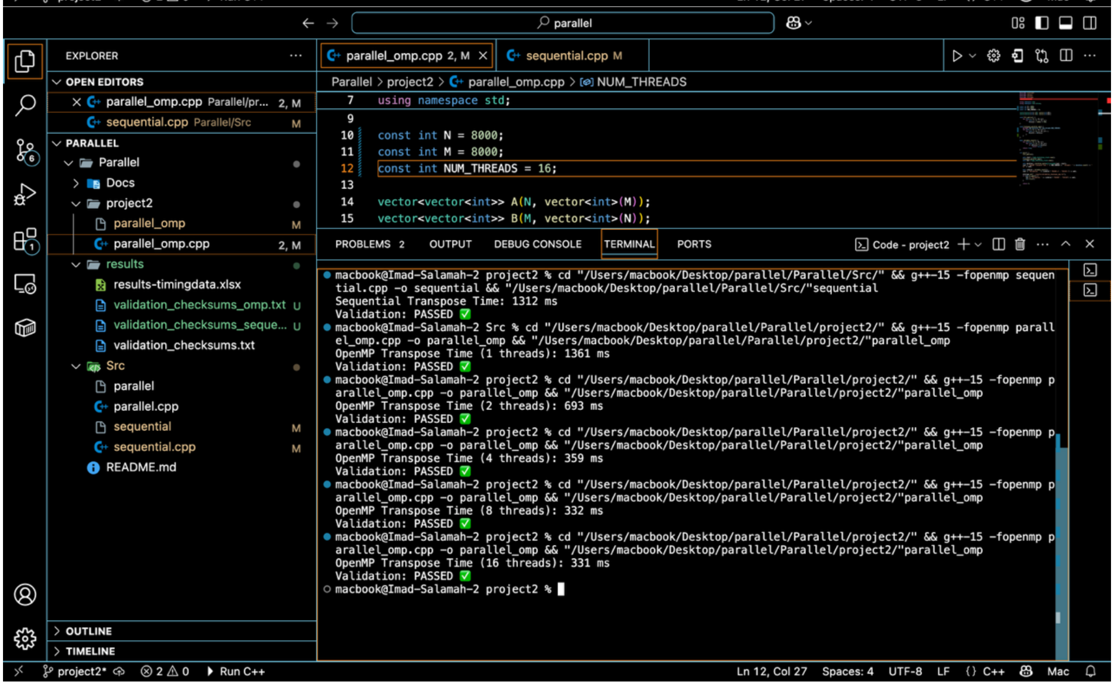
** This project benefited from the use of ChatGPT to clarify multithreading concepts, structure the C++ code for matrix transposition.

Screenshots of Code Execution

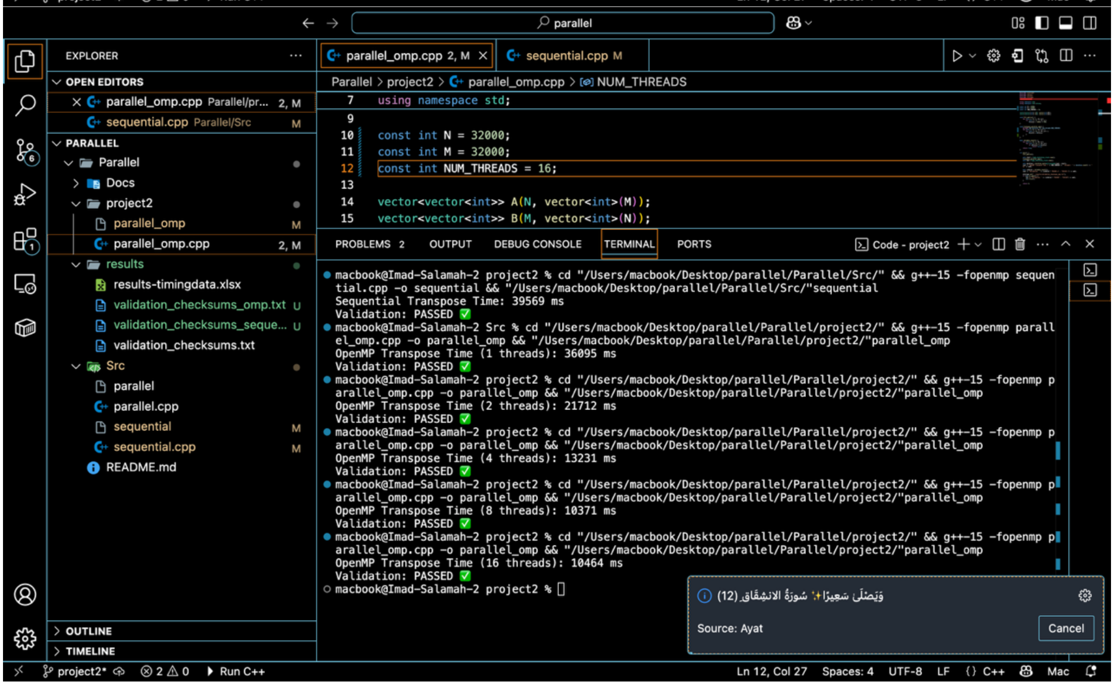





parallel_omp.cpp 2, M
sequential.cpp M
Parallel > project2 > parallel_omp.cpp > NUM_THREADS
using namespace std;
const int N = 4000;
const int M = 4000;
const int NUM_THREADS = 16;
vector<vector<int>> A(N, vector<int>(M));
vector<vector<int>> B(M, vector<int>(N));
PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
macbook@Imad-Salamah-2 project2 % cd "/Users/macbook/Desktop/parallel/Parallel/Src/" && g++-15 -fopenmp sequential.cpp -o sequential && "/Users/macbook/Desktop/parallel/Parallel/Src/"sequential
Sequential Transpose Time: 311 ms
Validation: PASSED
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Desktop/parallel/Parallel/project2/" && g++-15 -fopenmp parallel_omp.cpp -o parallel_omp && "/Users/macbook/Desktop/parallel/Parallel/project2/"parallel_omp
OpenMP Transpose Time (1 threads): 359 ms
Validation: PASSED
OpenMP Transpose Time (2 threads): 171 ms
Validation: PASSED
OpenMP Transpose Time (4 threads): 96 ms
Validation: PASSED
OpenMP Transpose Time (8 threads): 94 ms
Validation: PASSED
OpenMP Transpose Time (16 threads): 89 ms
Validation: PASSED
macbook@Imad-Salamah-2 project2 %


parallel_omp.cpp 2, M
sequential.cpp M
Parallel > project2 > parallel_omp.cpp > NUM_THREADS
using namespace std;
const int N = 8000;
const int M = 8000;
const int NUM_THREADS = 16;
vector<vector<int>> A(N, vector<int>(M));
vector<vector<int>> B(M, vector<int>(N));
PROBLEMS 2
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
macbook@Imad-Salamah-2 project2 % cd "/Users/macbook/Desktop/parallel/Parallel/Src/" && g++-15 -fopenmp sequential.cpp -o sequential && "/Users/macbook/Desktop/parallel/Parallel/Src/"sequential
Sequential Transpose Time: 1312 ms
Validation: PASSED
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Desktop/parallel/Parallel/project2/" && g++-15 -fopenmp parallel_omp.cpp -o parallel_omp && "/Users/macbook/Desktop/parallel/Parallel/project2/"parallel_omp
OpenMP Transpose Time (1 threads): 1361 ms
Validation: PASSED
OpenMP Transpose Time (2 threads): 693 ms
Validation: PASSED
OpenMP Transpose Time (4 threads): 359 ms
Validation: PASSED
OpenMP Transpose Time (8 threads): 332 ms
Validation: PASSED
OpenMP Transpose Time (16 threads): 331 ms
Validation: PASSED
macbook@Imad-Salamah-2 project2 %


parallel_omp.cpp
sequential.cpp
using namespace std;
const int N = 16000;
const int M = 16000;
const int NUM_THREADS = 16;
vector<vector<int>> A(N, vector<int>(M));
vector<vector<int>> B(M, vector<int>(N));
Sequential Transpose Time: 6729 ms
Validation: PASSED
OpenMP Transpose Time (1 threads): 6205 ms
Validation: PASSED
OpenMP Transpose Time (2 threads): 2959 ms
Validation: PASSED
OpenMP Transpose Time (4 threads): 1842 ms
Validation: PASSED
OpenMP Transpose Time (8 threads): 2118 ms
Validation: PASSED
OpenMP Transpose Time (16 threads): 2029 ms
Validation: PASSED


parallel_omp.cpp
sequential.cpp
using namespace std;
const int N = 32000;
const int M = 32000;
const int NUM_THREADS = 16;
vector<vector<int>> A(N, vector<int>(M));
vector<vector<int>> B(M, vector<int>(N));
Sequential Transpose Time: 39569 ms
Validation: PASSED
OpenMP Transpose Time (1 threads): 36095 ms
Validation: PASSED
OpenMP Transpose Time (2 threads): 21712 ms
Validation: PASSED
OpenMP Transpose Time (4 threads): 13231 ms
Validation: PASSED
OpenMP Transpose Time (8 threads): 10371 ms
Validation: PASSED
OpenMP Transpose Time (16 threads): 10464 ms
Validation: PASSED

Tools and Resources Used

Tool/Software	Purpose
G++	(sequential and parallel)Compiling C++ code
VSCode	Writing and editing C++ code
GitHub	Hosting the project repository
Excel	Recording execution times and calculating speedup
macOS Terminal	Running and testing the program locally