THE ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING

Parallel and Distributed Computing

Parallel and Distributed Computing PROJECT I

ID:202112141

Name: Imad Salamah

Section: -1-

Total	/100

Good Luck!

Mr. Hussein Younis

Introduction

In this project, we implemented a matrix transposition algorithm using two methods:

- Sequential, using regular loops in C++.
- Parallel, using the Pthreads library to divide the work across multiple threads.

This algorithm was chosen because it is simple in concept, but its implementation on large matrices can take a long time, making it suitable for parallelization.

The matrix transposition process relies on swapping rows with columns. This is an operation in which each element can be executed independently, so there is no need for synchronization between threads and no data conflicts, making it highly suitable for parallel execution.

The goal of the project is to compare the performance of the two versions and measure the time difference between sequential and parallel execution when using matrices of different sizes and different numbers of threads.

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 parallel implementation of the matrix transposition algorithm, the matrix rows were divided equally among multiple threads using the POSIX Threads (Pthreads) library.
Each thread was assigned a specific range of rows to process, calculated by dividing the total number of rows (N) by the number of threads (NUM_THREADS).

The thread then performed the transposition for those rows by copying values from matrix A to their corresponding transposed positions in matrix B.

Since each thread worked on independent rows and wrote to distinct locations in the output matrix, there were no shared writable data or race conditions. As a result, no synchronization mechanisms such as mutexes were required.

To launch the threads, the pthread_create() function was used, passing a dynamically allocated array containing the startRow and endRow values. After all threads were created, the main thread used pthread_join() to wait for them to finish before measuring the final execution time.

This approach was chosen for its simplicity and efficiency, as it ensures balanced workload distribution and allows full utilization of CPU cores with minimal overhead.

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.

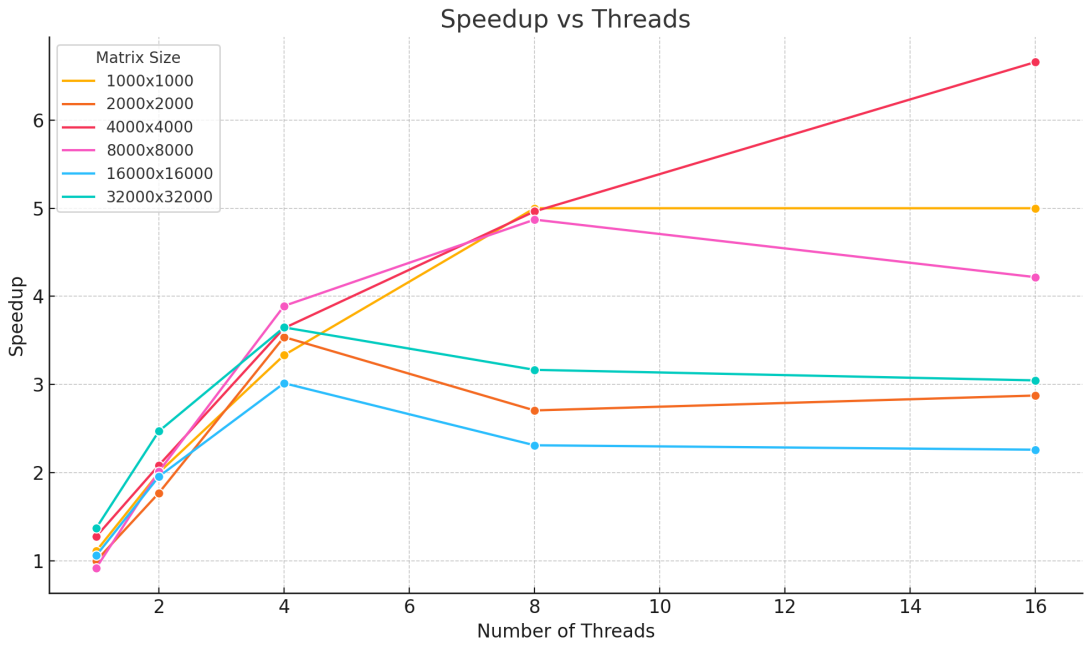


Figure 1:Speedup vs. Thread Count

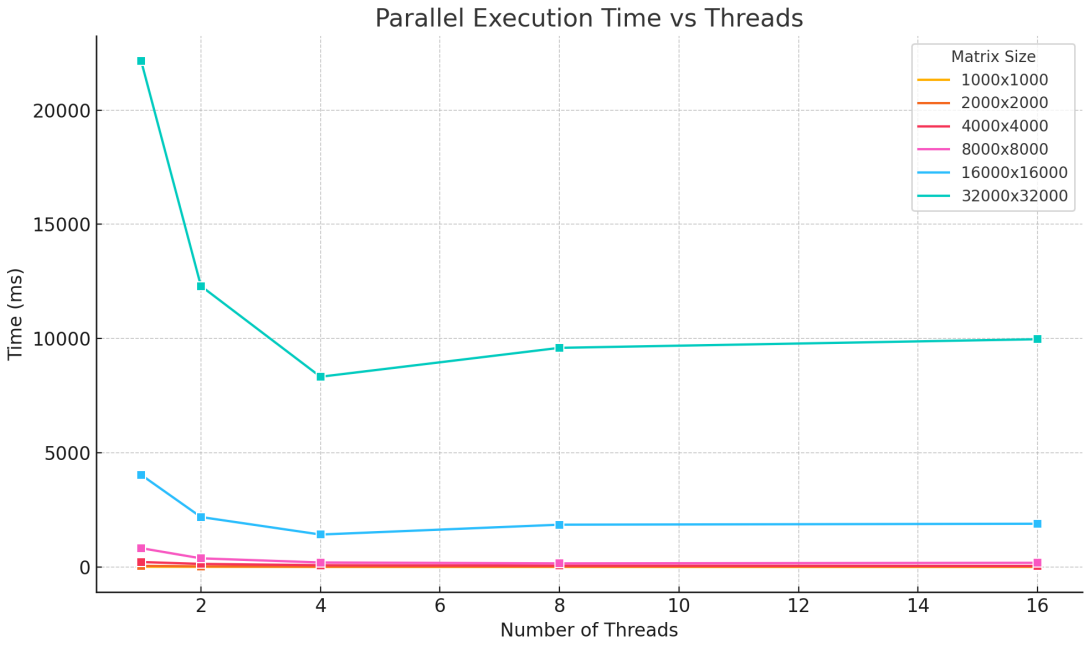


Figure 2:Execution Time vs. Thread Count

Matrix Size	Threads	Sequential Time (ms)	Parallel Time (ms)	Speedup
1000x1000	1	10	9	1.111111111
1000x1000	2	10	5	2
1000x1000	4	10	3	3.333333333
1000x1000	8	10	2	5
1000x1000	16	10	2	5
2000x2000	1	46	46	1
2000x2000	2	46	26	1.769230769
2000x2000	4	46	13	3.538461538
2000x2000	8	46	17	2.705882353
2000x2000	16	46	16	2.875
4000x4000	1	273	214	1.275700935
4000x4000	2	273	131	2.083969466
4000x4000	4	273	75	3.64
4000x4000	8	273	55	4.963636364
4000x4000	16	273	41	6.658536585
8000x8000	1	755	821	0.919610231
8000x8000	2	755	375	2.013333333
8000x8000	4	755	194	3.891752577
8000x8000	8	755	155	4.870967742
8000x8000	16	755	179	4.217877095
16000x16000	1	4272	4030	1.060049628
16000x16000	2	4272	2184	1.956043956
16000x16000	4	4272	1417	3.014820042
16000x16000	8	4272	1849	2.310438075
16000x16000	16	4272	1890	2.26031746
32000x32000	1	30373	22148	1.37136536
32000x32000	2	30373	12298	2.469751179
32000x32000	4	30373	8326	3.647970214
32000x32000	8	30373	9592	3.166492911
32000x32000	16	30373	9969	3.046744909

Table 1:result data

Discussion

The experimental results show that parallel execution becomes more effective as the matrix size increases. For small matrices like 1000×1000, the speedup was minimal due to overhead. However, for larger sizes such as 8000×8000 and 16000×16000, the parallel version achieved noticeable speedup, especially up to 8 threads.

The highest speedup was observed around 4× to 6× with 8 or 16 threads, but performance gains started to level off or slightly decrease after 8 threads due to overhead and memory limitations.

Execution time consistently decreased as thread count increased, but the benefit varied by matrix size. For very large matrices (32000×32000), parallel execution time was reduced by more than half compared to the sequential version.

Overall, the results confirm that multithreading improves performance for large data sizes, but excessive threading can reduce efficiency.

Conclusion

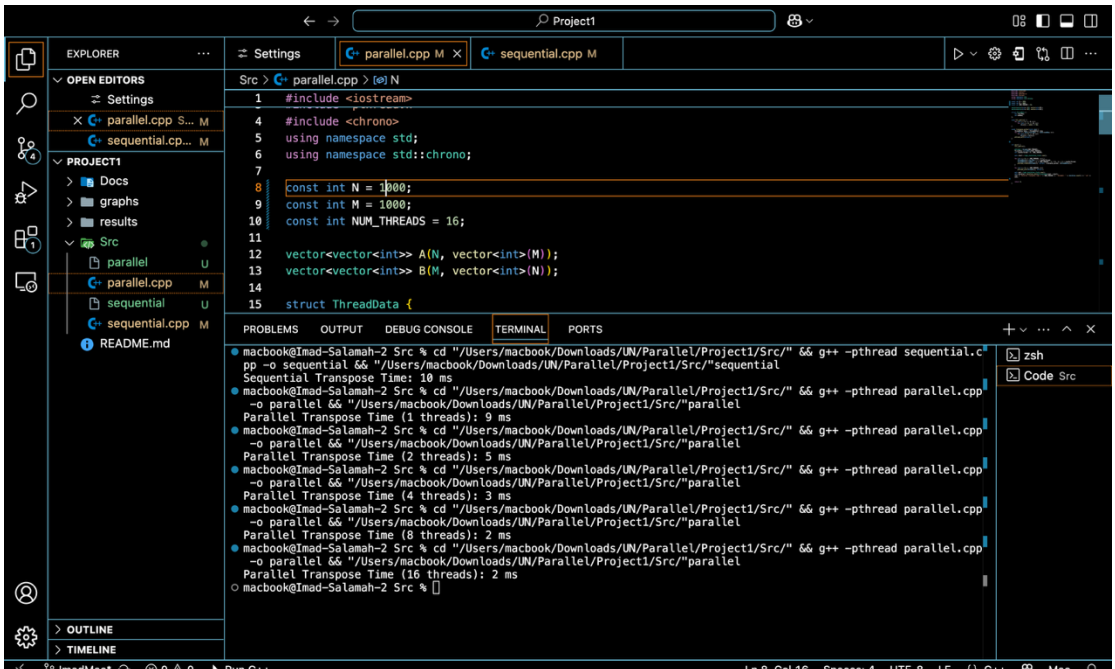
This project explored the implementation and performance of a matrix transposition algorithm using both sequential and parallel approaches. Through extensive testing with varying matrix sizes and thread counts, we observed clear improvements in execution time when applying multithreading—particularly for large data sizes.

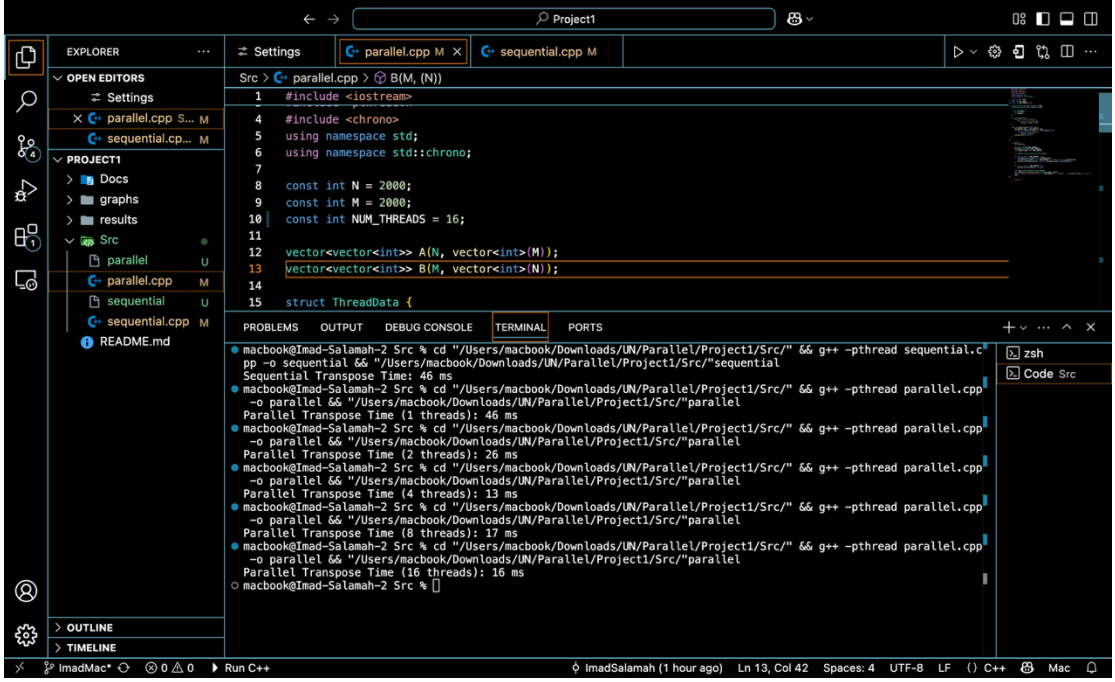
The experimental results confirmed that parallelism significantly reduces processing time and achieves notable speedup, especially up to 8 threads. However, excessive threading beyond this point can introduce overhead and memory limitations that reduce overall efficiency.

In conclusion, parallel programming offers a powerful method to accelerate computations, but its effectiveness depends on carefully balancing thread count with workload size. The insights gained from this project highlight the importance of optimizing thread usage to achieve maximum performance with minimal resource waste.

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

Screenshots of Code Execution






Project1
EXPLORER
Settings
parallel.cpp M
sequential.cpp M
OPEN EDITORS
PROJECT1
Docs
graphs
results
Src
parallel
parallel.cpp
sequential
sequential.cpp
README.md
#include <iostream>
#include <chrono>
using namespace std;
using namespace std::chrono;
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));
struct ThreadData {
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
Sequential Transpose Time: 273 ms
Parallel Transpose Time (1 threads): 214 ms
Parallel Transpose Time (2 threads): 131 ms
Parallel Transpose Time (4 threads): 75 ms
Parallel Transpose Time (8 threads): 55 ms
Parallel Transpose Time (16 threads): 41 ms
OUTLINE
TIMELINE
Ln 10, Col 27
Spaces: 4
UTF-8
C++
Mac
Project1
const int N = 8000;
const int M = 8000;
Sequential Transpose Time: 755 ms
Parallel Transpose Time (1 threads): 821 ms
Parallel Transpose Time (2 threads): 375 ms
Parallel Transpose Time (4 threads): 194 ms
Parallel Transpose Time (8 threads): 155 ms
Parallel Transpose Time (16 threads): 179 ms


Project1
EXPLORER
Settings
parallel.cpp M
sequential.cpp M
OPEN EDITORS
PROJECT1
Docs
graphs
results
Src
parallel
parallel.cpp
sequential
sequential.cpp
README.md
OUTLINE
TIMELINE
#include <iostream>
#include <chrono>
using namespace std;
using namespace std::chrono;
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));
struct ThreadData {
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread sequential.cpp -o sequential && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"sequential
Sequential Transpose Time: 4272 ms
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
Parallel Transpose Time (1 threads): 4030 ms
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
Parallel Transpose Time (2 threads): 2184 ms
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
Parallel Transpose Time (4 threads): 1417 ms
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
Parallel Transpose Time (8 threads): 1849 ms
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
Parallel Transpose Time (16 threads): 1890 ms
macbook@Imad-Salamah-2 Src %
zsh
Code Src
ImadMac*
Run C++
Ln 9, Col 16
Spaces: 4
UTF-8
LF
C++
Mac
Project1
EXPLORER
Settings
parallel.cpp M
sequential.cpp M
OPEN EDITORS
PROJECT1
Docs
graphs
results
Src
parallel
parallel.cpp
sequential
sequential.cpp
README.md
OUTLINE
TIMELINE
#include <iostream>
#include <vector>
#include <pthread.h>
#include <chrono>
using namespace std;
using namespace std::chrono;
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));
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread sequential.cpp -o sequential && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"sequential
Sequential Transpose Time: 30373 ms
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
Parallel Transpose Time (1 threads): 22148 ms
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
Parallel Transpose Time (2 threads): 12298 ms
cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
Parallel Transpose Time (4 threads): 8326 ms
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
Parallel Transpose Time (8 threads): 9592 ms
macbook@Imad-Salamah-2 Src % cd "/Users/macbook/Downloads/UN/Parallel/Project1/Src/" && g++ -pthread parallel.cpp -o parallel && "/Users/macbook/Downloads/UN/Parallel/Project1/Src/"parallel
Parallel Transpose Time (16 threads): 9969 ms
macbook@Imad-Salamah-2 Src %
zsh
Code Src
ImadMac*
Run C++
Not Committed Yet
Ln 9, Col 21
Spaces: 4
UTF-8
LF
C++
Mac

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