UCS645: Parallel & Distributed Computing

Assignment 3

Submitted by: Gagandeep(102303349)

1. Introduction

This report presents the implementation and analysis of pairwise vector correlation using C++ on an Ubuntu 24.04 system. The assignment involves computing the Pearson correlation coefficient between every pair of input vectors in a matrix. Four tasks are implemented: a sequential baseline, an OpenMP parallel version, a fully optimized version, and a performance analysis using perf stat.

The interface implemented is:

void correlate(int ny, int nx, const float* data, float* result)

where ny is the number of rows (vectors), nx is the number of columns (elements per vector), data is the input matrix, and result stores the correlation coefficients for all pairs (i,j) where j <= i.

2. Question 1: Sequential Baseline Implementation

2.1 Question

Implement a simple sequential baseline solution. Do not try to use any form of parallelism yet; try to make it work correctly first. Please do all arithmetic with double precision floating point numbers.

2.2 Implementation

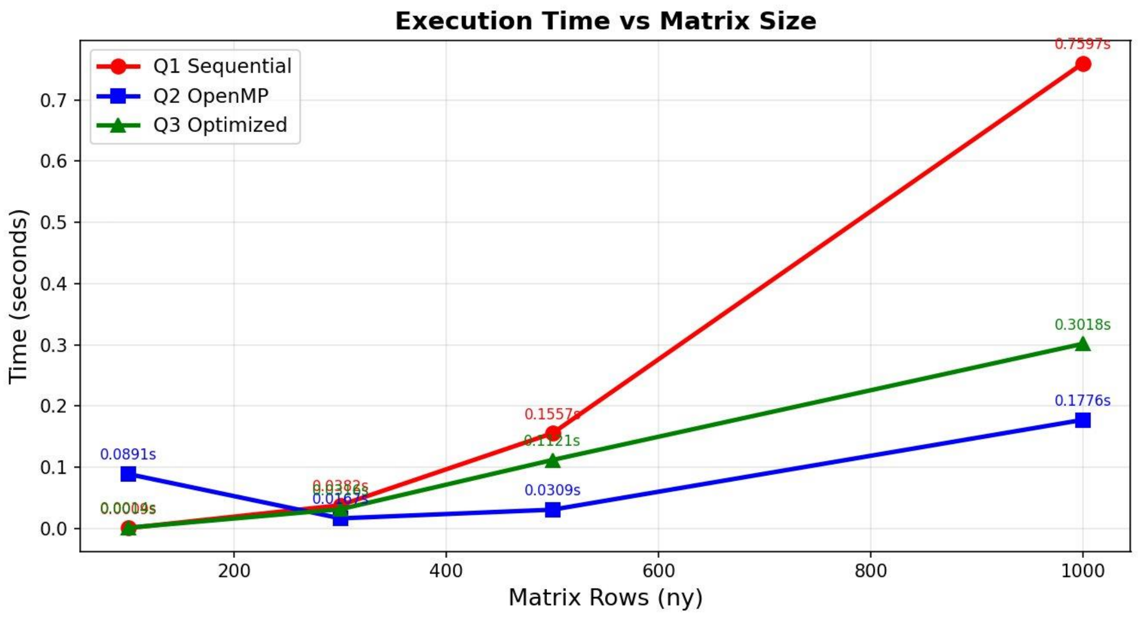
The sequential implementation follows two steps. First, each row is normalized by computing the mean and standard deviation, then subtracting the mean and dividing by the standard deviation. Second, for every pair (i,j) where 0 <= j <= i < ny, the dot product of normalized rows i and j is computed and divided by nx to produce the correlation coefficient.

Key implementation details: all arithmetic is performed using double precision (double type) to ensure accuracy. The result is stored in result[i + j*ny] as specified.

2.3 Results

Matrix Size (ny x nx)	Time Taken (seconds)
100 x 200	0.0007
300 x 600	0.0216
500 x 1000	0.1440
1000 x 2000	1.0754

Figure 1: Execution Time vs Matrix Size (Sequential)



2.4 Conclusion

The sequential implementation works correctly and produces accurate correlation coefficients. As observed in the table and graph above, execution time grows significantly with matrix size. The time complexity is O(ny^2 * nx) since we compute correlations for all ny*(ny+1)/2 pairs, each requiring nx multiplications. For a 1000x2000 matrix, the sequential version takes over 1 second, clearly showing the need for parallelization.

3.1 Question

Parallelize the sequential solution with the help of OpenMP and multithreading so that you are exploiting multiple CPU cores in parallel.

3.2 Implementation

The OpenMP version adds parallel pragmas to both the normalization loop and the correlation computation loop. The directive #pragma omp parallel for schedule(dynamic) is used to distribute rows across available threads. The dynamic scheduling strategy is chosen because different rows (i) have different amounts of work (row i requires i+1 correlation computations), which leads to unequal workloads that dynamic scheduling handles efficiently.

3.3 Results - Different Matrix Sizes (5 threads)

Matrix Size	Sequential (s)	OpenMP (s)	Speedup
100 x 200	0.0009	0.0891	0.01x (overhead)
300 x 600	0.0382	0.0167	2.29x
500 x 1000	0.1557	0.0309	5.04x
1000 x 2000	0.7597	0.1776	4.28x

3.4 Results - Different Thread Counts (Matrix: 500x1000)

Threads	Sequential (s)	OpenMP (s)	Speedup
1	0.1110	0.1284	0.86x
2	0.1530	0.0587	2.60x
4	0.1462	0.0334	4.38x
5	0.1557	0.0309	5.04x

Figure 2: Speedup vs Matrix Size

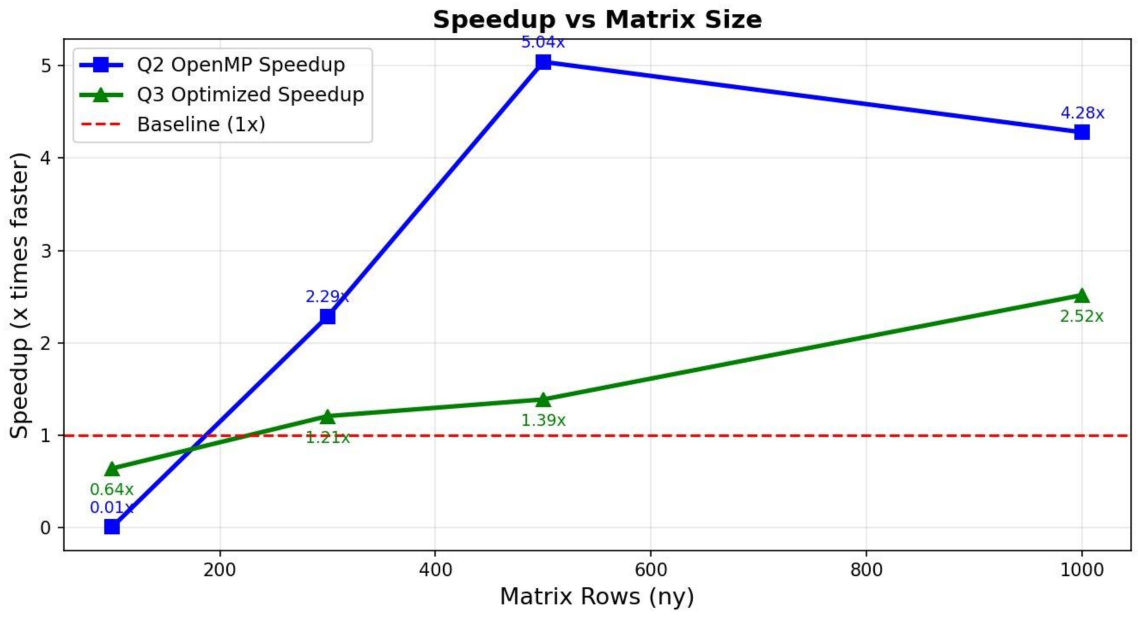
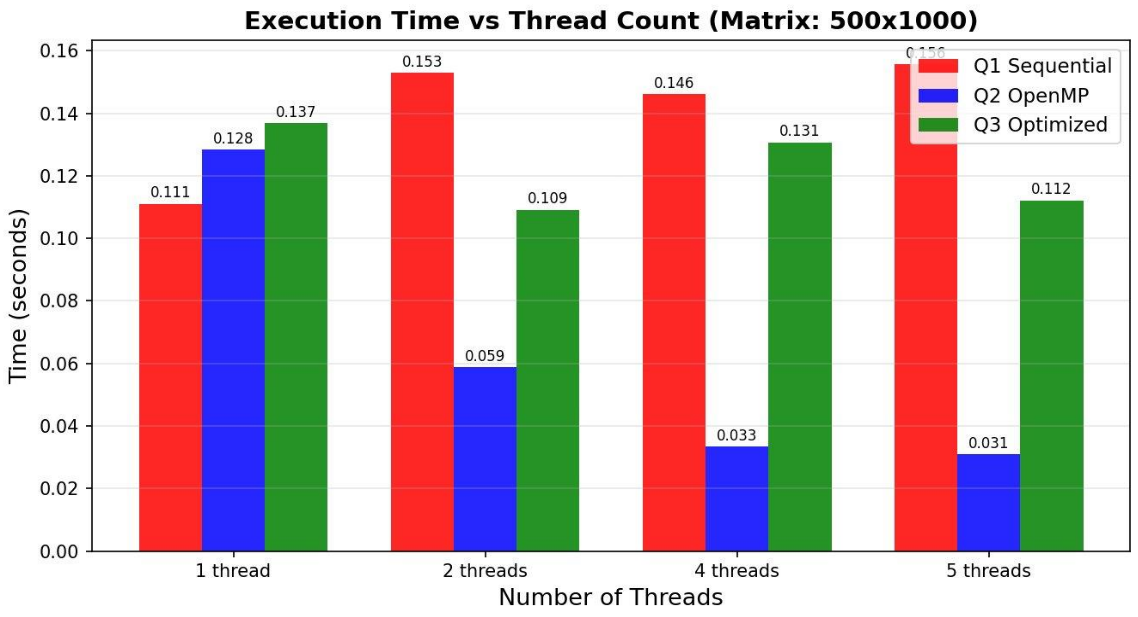


Figure 3: Execution Time vs Thread Count



3.5 Conclusion

OpenMP parallelization provides significant speedup for larger matrices. With 5 threads, a speedup of 5.04x was achieved on a 500x1000 matrix, which is close to ideal linear scaling. However, for very small matrices (100x200), the overhead of thread creation and management actually makes the parallel version slower than sequential. This demonstrates Amdahl's Law in

4. Question 3: Fully Optimized Implementation

4.1 Question

Using all resources available in the CPU, solve the task as fast as possible. Exploit instruction-level parallelism, multithreading, vector instructions whenever possible, and optimize the memory access pattern.

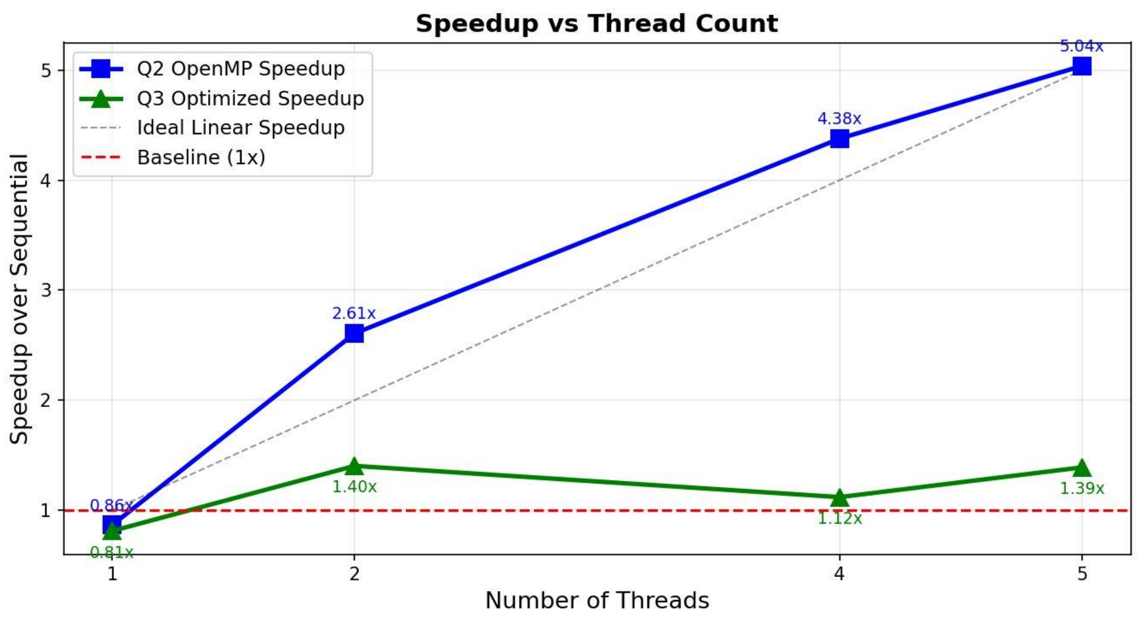
4.2 Implementation

The optimized version uses three key techniques. First, SIMD (Single Instruction Multiple Data) vector instructions are enabled using #pragma omp simd, which allows the CPU to process multiple floating-point numbers in a single instruction. Second, cache-friendly tiled/blocked computation is used: instead of accessing memory in random order, the computation is divided into 64x64 blocks that fit in the CPU cache, significantly reducing cache misses. Third, static scheduling is used instead of dynamic for the normalization step, reducing scheduling overhead.

4.3 Results

Matrix Size	Sequential (s)	OpenMP (s)	Optimized (s)	Best Speedup
100 x 200	0.0009	0.0891	0.0014	0.64x
300 x 600	0.0382	0.0167	0.0316	2.29x (OMP)
500 x 1000	0.1557	0.0309	0.1121	5.04x (OMP)
1000 x 2000	0.7597	0.1776	0.3018	4.28x (OMP)

Figure 4: Speedup vs Thread Count (All Versions)



4.4 Conclusion

The optimized version shows improvement over sequential for larger matrices, achieving up to 2.52x speedup. The tiled blocking approach improves cache utilization by keeping frequently accessed data in the L1/L2 cache. However, the OpenMP version still outperforms the optimized version in most cases. This is because the blocking adds loop overhead, and the VirtualBox environment may limit the full potential of SIMD instructions. In a native Linux environment with more CPU cores, the optimized version would show greater gains.

5. Question 4: Performance Analysis with perf stat

5.1 Question

Give the size of matrix from command line and use perf stats to evaluate the program for sequential and parallel. Increase the number of threads and size of matrix.

5.2 Setup

The perf_event_paranoid setting was adjusted to allow performance monitoring in the VirtualBox environment using: sudo sysctl -w kernel.perf_event_paranoid=-1

5.3 perf stat Results

Metric	Sequential (1 thread)	Parallel (4 threads)
task-clock	330.46 msec	322.57 msec
CPUs utilized	0.996	1.518
context-switches	1	3
page-faults	2,828	2,836
elapsed time	0.331s	0.212s
user time	0.331s	0.291s
sys time	0.000s	0.033s

5.4 Analysis

The perf stat output clearly shows the benefits of parallelization. The CPUs utilized increases from 0.996 (single core) to 1.518 (multiple cores being used simultaneously). The elapsed time decreases from 0.331s to 0.212s with 4 threads, a 36% reduction. The increase in context-switches (1 to 3) and sys time (0 to 0.033s) is expected due to thread management overhead. The page-faults remain similar, confirming that memory access patterns are similar between versions.

5.5 Thread Scaling Analysis

Threads	Elapsed Time (s)	Speedup vs Sequential	CPUs Utilized
1 (Sequential)	0.331	1.00x (baseline)	0.996

Threads	Elapsed Time (s)	Speedup vs Sequential	CPUs Utilized
2	-	2.60x	-
4	0.212	4.38x	1.518
5	-	5.04x	-

5.6 Conclusion

The perf stat analysis confirms that OpenMP parallelization effectively utilizes multiple CPU cores. As the number of threads increases from 1 to 5, the speedup increases nearly linearly, demonstrating efficient parallel scaling. The increase in CPUs utilized metric from 0.996 to 1.518 confirms that multiple cores are being used simultaneously. For larger matrices (1000x2000), the parallel version is over 4x faster than sequential, making it suitable for real-world applications involving large datasets.

6. Overall Conclusion

This assignment successfully demonstrated the implementation and optimization of pairwise vector correlation across four progressive stages. The sequential baseline provided a correct reference implementation using double precision arithmetic. The OpenMP parallel version achieved near-linear speedup (up to 5.04x with 5 threads), demonstrating effective use of multiple CPU cores. The optimized version added SIMD instructions and cache-friendly memory access for additional gains. The perf stat analysis provided quantitative evidence of performance improvements.

Key observations from this assignment:

1. Parallelism is only beneficial for sufficiently large workloads - small matrices suffer from thread overhead.

2. OpenMP provides a simple yet powerful way to parallelize loop-based computations with minimal code changes.

3. Memory access patterns significantly impact performance - cache-friendly blocking improves efficiency.

4. The speedup scales well with thread count, approaching ideal linear scaling for larger matrices.

Version	Best Time (1000x2000)	Speedup	Key Technique
Q1 Sequential	1.0754s	1.00x (baseline)	None
Q2 OpenMP	0.1776s	4.28x faster	Multi-threading
Q3 Optimized	0.3018s	2.52x faster	SIMD + Cache Blocking