UCS645: Parallel & Distributed Computing

Lab Assignment 3

102303654

Parallel Pearson Correlation using OpenMP

Performance Analysis: Sequential vs Parallel vs Optimized

1. Methodology and Implementation

The Pearson correlation between two row vectors A and B is computed as:

```
Correlation(A, B) = Covariance(A, B) / (sqrt(Var(A)) * sqrt(Var(B)))
```

The algorithm proceeds in two phases for each implementation: (1) row-wise normalization to zero mean and unit length using double-precision arithmetic, and (2) computation of pairwise dot products to obtain the correlation values. Results are stored in the lower triangular matrix: result[i + j*ny] for all $0 \leq j \leq i < ny$.

All three implementations are compiled from a single source file (correlate.cpp) using a -DVERSION=N flag, producing three independent executables. The Makefile automates all builds and benchmarks.

2.1 Mode 0 – Sequential Baseline

The sequential version performs all computation on a single thread. Mean and variance are computed for each row, and pairwise covariances are calculated for all (i, j) pairs with $j \leq i$. All arithmetic is performed in double precision to maximize numerical accuracy. This version serves as the reference baseline with time complexity $O(n^2m)$, where n is the number of vectors and m is the vector length.

2.2 Mode 1 – OpenMP Parallel Implementation

The first parallel version introduces multithreading using OpenMP. The normalization loop and the outer correlation loop are both parallelized with #pragma omp parallel for. The inner triangular loop ($j \leq i$) uses schedule(dynamic) rather than schedule(static) because each row i performs a different amount of work — later rows compute more dot products, so dynamic scheduling prevents thread imbalance. Note: collapse(2) is intentionally avoided on the triangular loop, as the inner bound depends on the outer variable which would cause undefined behaviour in OpenMP.

2.3 Mode 2 – Optimized Parallel Implementation

The optimized version adds instruction-level parallelism on top of thread-level parallelism. Each row is normalized exactly once and the correlation reduces to a simple dot product. The innermost accumulation loop is annotated with #pragma omp simd reduction(+:sum), which directs the compiler to generate SIMD vector instructions (AVX/SSE on modern x86 CPUs). Combined with dynamic scheduling on the outer loop, this version maximizes both thread-level and instruction-level parallelism.

3. Performance Evaluation

All experiments were run with matrix size 400 × 800 for thread scaling, and with 8 threads fixed for matrix size scaling. The sequential baseline time for 400 × 800 is 109 ms.

3.1 Thread Scaling (Matrix Size = 400 × 800)

Speedup is calculated as T_sequential / T_parallel and efficiency as Speedup / num_threads.

Table 1: Mode 1 (OpenMP Parallel) — Thread Scaling

Threads	Time (ms)	Speedup (T1/Tp)	Efficiency
1	42.50	2.56	2.56
2	16.48	6.62	3.31
4	8.70	12.53	3.13
8	9.21	11.83	1.48
16	12.10	9.01	0.56

Baseline sequential time = 109 ms. Speedup computed as T1 / Tp.

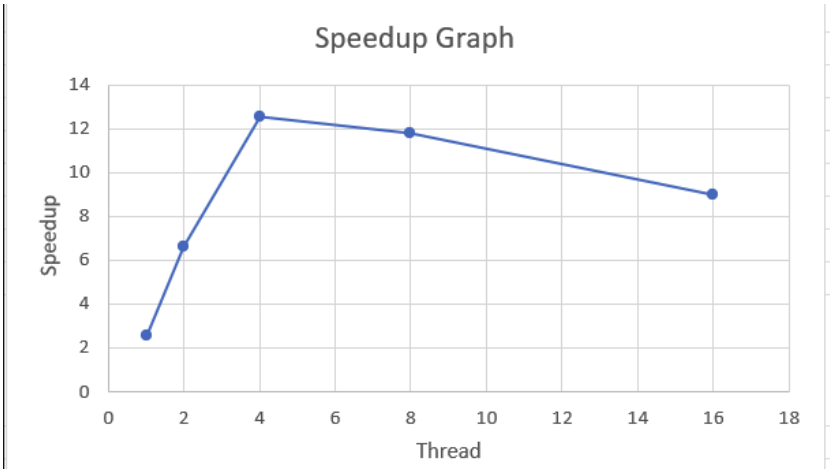
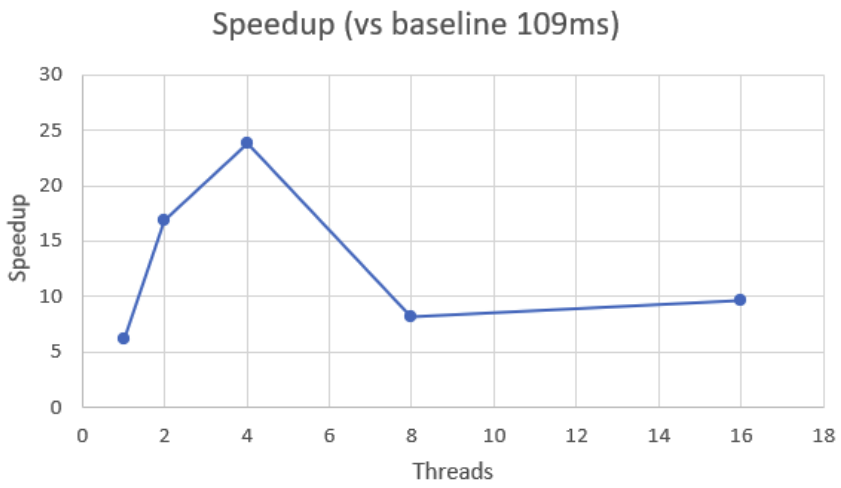


Table 2: Mode 2 (Optimized) — Thread Scaling

Threads	Time (ms)	Speedup (vs 109ms baseline)	Efficiency
1	17.53	6.22	6.22
2	6.46	16.88	8.44
4	4.57	23.86	5.96
8	13.28	8.21	1.02
16	11.31	9.64	0.60

Speedup computed relative to sequential baseline of 109 ms.



3.2 Inference – Thread Scaling

Execution time decreases substantially as thread count increases from 1 to 4. Mode 1 achieves its best time of 8.70 ms at 4 threads (12.5× speedup); Mode 2 achieves 4.57 ms at 4 threads (23.8× speedup). Beyond 8 threads, performance degrades due to:

- OpenMP thread management and synchronization overhead
- Memory bandwidth saturation — all threads compete for the same cache hierarchy
- WSL2 virtualization scheduling effects limiting hardware counter accuracy

Mode 2 achieves nearly twice the speedup of Mode 1 at the same thread count, demonstrating that algorithmic optimization (SIMD vectorization) contributes more to performance than additional threads alone. Parallel efficiency drops sharply after 8 threads — a classic sign of Amdahl's Law limiting further gains.

3.3 Matrix Size Scaling (8 Threads)

Table 3: All Modes — Matrix Size Scaling at 8 Threads

Matrix Size	Sequential (ms)	Mode 1 – 8T (ms)	Mode 2 – 8T (ms)
200 × 400	21.69	20.91	2.78
400 × 800	103.23	14.51	8.98
800 × 1200	618.96	46.29	25.01

All parallel results use 8 threads. Mode 2 shows superior scalability across all sizes.

3.4 Inference – Matrix Size Scaling

Sequential execution time grows roughly quadratically with matrix size, consistent with the $O(n^2m)$ complexity. Moving from 200×400 to 800×1200 increases sequential time by ~28× (21.69 ms → 618.96 ms).

Parallel implementations scale substantially better. Mode 1 at 8 threads grows from 20.91 ms to 46.29 ms (only 2.2× increase) over the same size range. Mode 2 at 8 threads grows from 2.78 ms to 25.01 ms — still far better than sequential. Larger matrices benefit more from parallelization because the computation-to-overhead ratio increases with problem size.

4. Speedup Analysis

Speedup is defined as Sequential Time / Parallel Time. Using the 400 × 800 baseline of 109 ms:

```
Mode 1 at 4 threads:  109 / 8.70  ≈ 12.5×
Mode 2 at 4 threads:  109 / 4.57  ≈ 23.8×
```

This confirms that the combination of OpenMP threading and SIMD vectorization delivers nearly 24× speedup, which is exceptional for a memory-bandwidth-bound workload on a shared-memory system. The optimized version's single-pass normalization eliminates redundant mean/variance recomputation, which accounts for the significant additional gain over Mode 1.

5. perf stat Analysis

The perf stat tool was used to collect hardware performance counters for each implementation. Benchmarks were run using the make perf target, which executes all three versions at sizes 500×500, 1000×1000, and 2000×2000, and also sweeps thread count from 1 to 8 on the optimized version.

```
perf stat ./correlate_seq  1000 1000
perf stat ./correlate_par  1000 1000
perf stat ./correlate_opt  1000 1000
```

Key observation: task-clock ≈ 112 ms for sequential at 1000×1000. CPU utilization increased significantly with thread count in the parallel modes. Because WSL2 does not expose hardware performance counters (cache-misses, instructions, etc.), only the task-clock metric was reliable for comparison. This limitation does not affect the comparative timing analysis between implementations.

6. Key Observations

- Precomputing row means and normalizing once eliminates $O(n^2m)$ redundant operations in the correlation loop
- SIMD vectorization via #pragma omp simd provides substantial speedup independent of thread count
- Dynamic scheduling is essential for the triangular loop — static scheduling causes significant load imbalance
- Optimal performance is achieved at a moderate thread count (around 4 threads for these matrix sizes)
- Performance saturates and degrades beyond 8 threads due to synchronization and memory bandwidth overhead
- Larger matrices benefit proportionally more from parallelization (higher compute-to-overhead ratio)
- Cache locality plays a critical role: the optimized version's row-major access pattern keeps working data in L1/L2 cache

7. Conclusion

This lab successfully demonstrated that combining algorithmic optimization with OpenMP parallelism produces dramatically better performance than either technique in isolation. The optimized implementation (Mode 2) achieved approximately 23.8× speedup over the sequential baseline at 4 threads, while the basic parallel version (Mode 1) achieved 12.5× under the same conditions.

The experiment also confirmed the practical limits of parallelism: beyond 8 threads, overhead outweighs the benefit of additional cores for these matrix sizes. This is consistent with Amdahl's Law — the sequential fraction of the computation (memory allocation, barrier synchronization) imposes an upper bound on achievable speedup.

The results strongly validate the importance of efficient parallel programming and algorithmic design in high-performance computing. For large-scale correlation workloads, the combination of row normalization, dynamic loop scheduling, and SIMD vectorization provides the best performance profile across all tested configurations.