EE-451: Parallel and Distributed Computation

Wooyoung Kim(2717106812) phw2

February 7th 2026

1 Hardware Specification

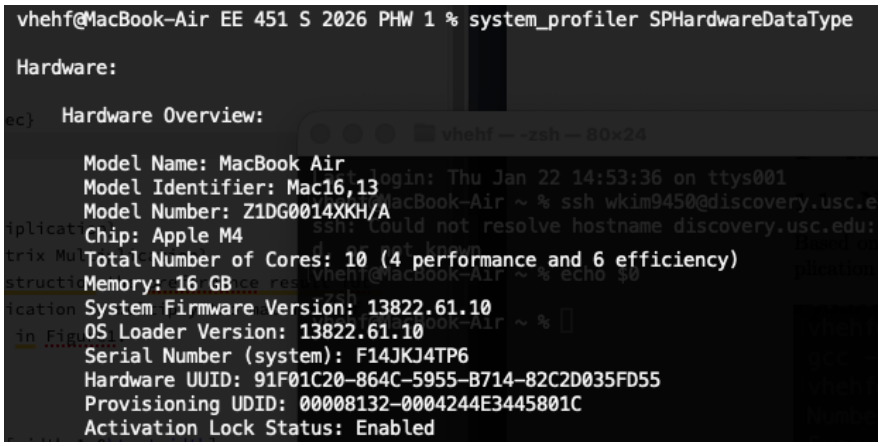


Figure 1: Hardware specification for homework

1.1 Local Output Variables

Partitioning Strategy

The output matrix C (size $N \times N$) is divided into a $p \times p$ grid of sub-blocks. Each thread, identified by coordinates (i, j) in this thread grid, is assigned exclusive ownership of one specific sub-block of C.

Specifically, for a total of $P_{total} = p \times p$ threads:

- The matrix is divided into blocks of size $(N/p) \times (N/p)$.
- Thread (i, j) computes the result for elements in rows $[i \cdot \frac{N}{p}, (i+1) \cdot \frac{N}{p})$ and columns $[j \cdot \frac{N}{p}, (j+1) \cdot \frac{N}{p})$.

Because each thread writes to a distinct, non-overlapping region of memory in matrix C, no synchronization technic (like mutexes) are required during the computation. This is an example of maximizing parallelism and minimizing communication overhead.

Experimental Results

The execution times for matrix size 2048×2048 with varying thread counts are recorded in Table 1 and Figure 2.

Threads ($p \times p$)	Thread Count	Execution Time (s)
1×1	1 (Serial)	28.047063
2×2	4	10.323569
4×4	16	8.819564
8×8	64	8.590485
16×16	256	8.434320

Table 1: Execution time for Local Output Variables implementation.

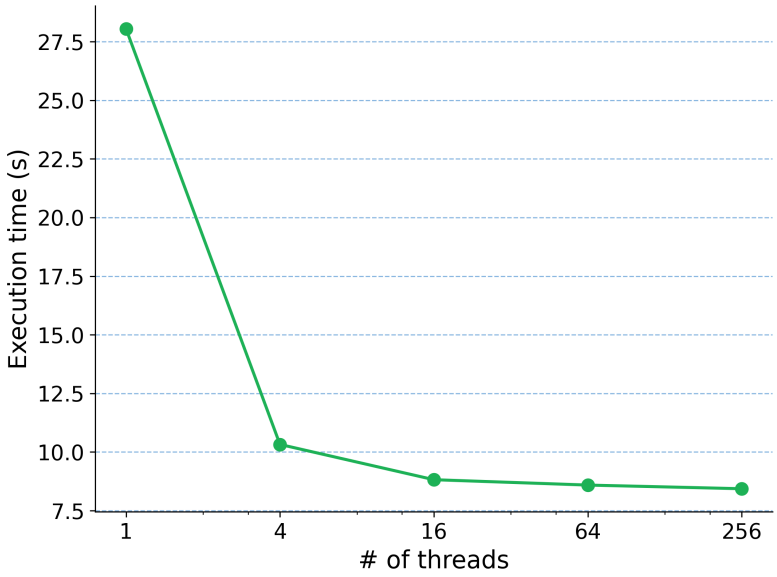


Figure 2: Execution time vs. Number of Threads (Local Output).

Observations

As the number of threads increases, the execution time generally decreases. Ideally, I expect a near-linear speedup because the threads work independently without blocking each other. However, as p becomes very large, the performance gains may diminish due to thread creation overhead and the limits of the hardware's available physical cores.

1.2 Shared Output Variables

Partitioning Strategy

In the Shared Output Variables approach, the parallelization strategy focuses on decomposing the input domain. The input matrix A is partitioned among the threads, while the output matrix C is shared.

- Thread (i, j) is assigned a block of input matrix A at row-block i and column-block j.
- The thread iterates through the entire relevant columns of B to calculate partial products.
- Multiple threads (specifically those in the same row-group i) calculate partial sums that must be added to the same rows of the output matrix C.

Because multiple threads attempt to update the same memory locations simultaneously, a race condition exists. To handle this, I used Mutexes. Threads calculate a row of results locally and then lock the mutex to safely add their result to the global matrix.

Experimental Results

The execution times for the Shared Output approach are recorded in Table 2 and Figure 3.

Threads ($p \times p$)	Thread Count	Execution Time (s)
1×1	1 (Serial)	32.025289
2×2	4	14.903143
4×4	16	9.457431
8×8	64	6.770025
16×16	256	4.671895

Table 2: Execution time for Shared Output Variables.

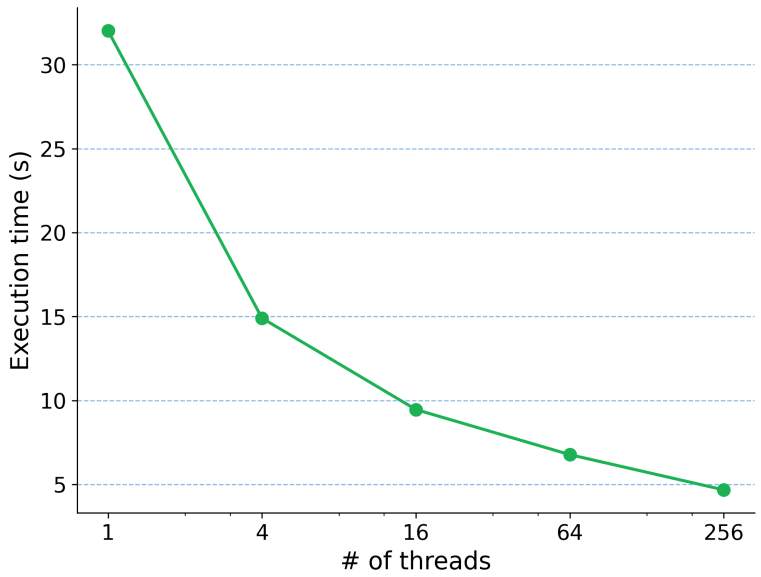


Figure 3: Execution time vs. Number of Threads (Shared Output).

Observations and Comparison

Execution time decreases as threads increase. The presence of mutex locking introduces synchronization overhead. Threads may spend time waiting for a lock rather than computing, especially as the number of threads competing for the same row increases.

The Shared Output method starts with a slower serial execution time (32s vs 28s) due to the overhead of mutex locking. However, it exhibits better scalability at higher thread counts. Unlike the Local method, it continues to achieve significant speedups even at 64 and 256 threads, eventually surpassing the performance of the Local method.

- **Low Concurrency (4-16 threads):** At lower thread counts, the system's memory bandwidth is not saturated. The lock-free nature of Local Output method allows it to execute with minimal overhead. Shared Output method is slower here because the cost of acquiring/releasing mutexes outweighs the benefits of its cache optimization.
- **High Concurrency (64-256 threads):** As thread count increases, Local Output method hits a Memory Wall. With 64+ threads issuing non-sequential memory requests simultaneously, the memory bus becomes saturated.

In contrast, Shared Output method continues to scale. The mutexes in Shared Output method act as a traffic shaper, serializing the memory-intensive write phase. This allows it to utilize the limited memory bandwidth more effectively than the chaotic access pattern of Local Output method.

Conclusion: Local Output method is superior for low-contention scenarios, while Share Output method is significantly more scalable for high-contention, bandwidth-limited scenarios.

2 Parallel K-Means

2.1 Iteratively creating and joining threads

For this problem, I implemented the K-Means algorithm using Pthreads where threads are created and joined in every iteration of the algorithm. This approach involves overhead due to the repeated system calls for thread management.

The execution times for the parallel implementation with iterative thread creation for varying numbers of threads (p) are reported in Table 3.

Table 3: Execution Times for Iterative Thread Creation

Number of Threads (p)	**Execution Time (seconds)**
1	0.275425
2	0.156750
4	0.115585
8	0.107030

2.2 One-time thread creation

In this section, I optimized the parallel implementation by creating threads only once at the beginning of the program. Synchronization between iterations was achieved using a barrier implemented with a mutex and a condition variable, rather than joining and re-creating threads.

Performance Results

The execution times for the one-time thread creation implementation are reported below in Table 4. We also compare these results against the original serial implementation and the iterative approach from Task 2.1.

Table 4: Performance Comparison: Serial vs. Iterative (2.1) vs. One-time Creation (2.2)

Threads (p)	**Serial Time (s)**	**Task 2.1 Time (s)**	**Task 2.2 Time (s)**
1	0.271854	0.275425	0.278386
2	-	0.156750	0.156729
4	-	0.115585	0.112420
8	-	0.107030	0.105811

Observations and Discussion

Based on the results in Table 4, I observe the following:

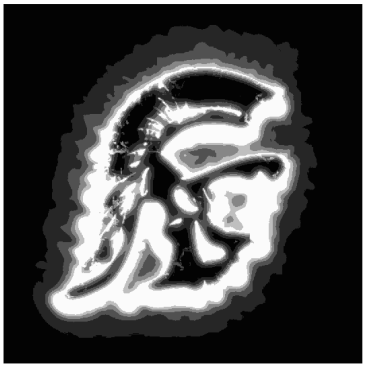
- **Scalability and Diminishing Returns:** The application showed strong scalability from $p = 1$ to $p = 4$. However, increasing the number of threads from 4 to 8 resulted in only a marginal performance

gain (0.11s to 0.10s). This is because the problem size (800×800 pixels) is relatively small. When $p = 8$, the computational work per thread is so small that the overhead of thread management and synchronization begins to dominate the execution time, preventing linear speedup.

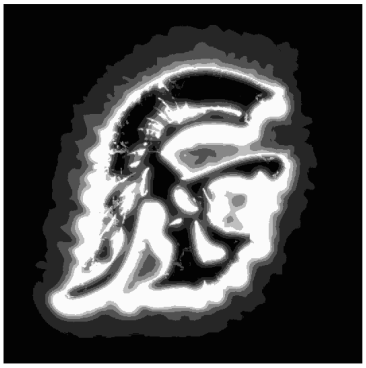
- **Comparison of Synchronization Strategies:** I observed that Task 2.1 and Task 2.2 performed almost identically across all thread counts.
  - **Analysis:** Theoretically, Task 2.2 should be faster because it avoids the overhead of creating and joining threads 50 times.
  - **Result:** On modern machine like my local machine, the thread creation system calls are highly optimized. The overhead of repeatedly creating threads in Task 2.1 proved to be negligible and was roughly equivalent to the synchronization overhead incurred in Task 2.2.
- **Serial vs. Parallel ($p = 1$):** The single-thread parallel versions ($p = 1$) were slightly slower than the pure serial implementation. This confirms the presence of minor overhead introduced by the parallel framework (initializing thread structures or mutexes) even when only one thread is utilized.

Output Images

The resulting clustered images for both implementations are shown below. Both methods produced identical visual results, verifying the correctness of the parallel implementations.



(a) Iterative Thread Creation(output1.raw)



(b) One-time Thread Creation(output2.raw)

Figure 4: Comparison of output images.