EE-451: Parallel and Distributed Computation

Wooyoung Kim(2717106812) phw3

February 27th 2026

1 Hardware Specification

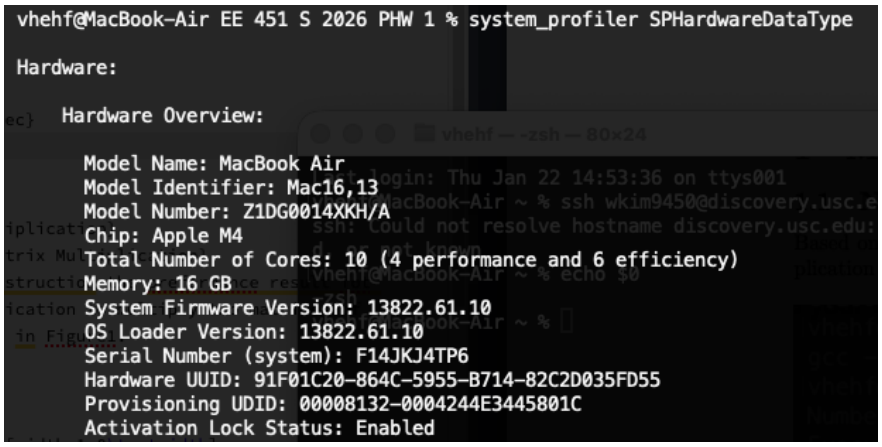

Figure 1: Hardware specification for homework

2 Parallel Matrix Multiplication

2.1 Execution Time

Table 1 shows the execution times for multiplying two 2048×2048 matrices across different OpenMP thread counts ($p \times p$). Version A represents the naive parallelization, Version B is the blocked parallelization (block size $b = 8$), and Version C is the parallelized naive multiplication with a transposed B matrix.

Threads (p^2)	**Version A (Naive)**	**Version B (Blocked)**	**Version C (Transposed)**
1	252.09 s	16.71 s	12.56 s
4	15.63 s	4.62 s	3.45 s
16	11.74 s	3.16 s	2.16 s
64	11.34 s	3.64 s	2.11 s
256	11.83 s	3.51 s	2.11 s

Table 1: Execution time for $N = 2048$ under various thread counts.

2.2 Observation

As illustrated in Figure 2, the scaling behavior exhibits distinct characteristics. Strong scaling is observed primarily when transitioning from 1 to 16 threads. Beyond 16 threads, the execution time sharply stabilizes.

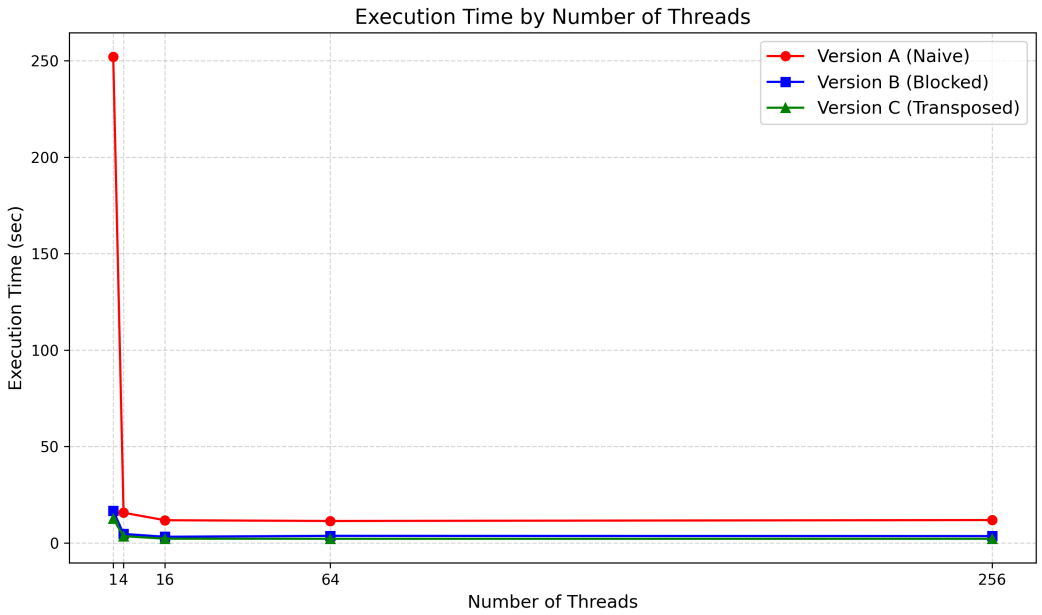

Figure 2: Execution time scaling with respect to the number of threads for Versions A, B, and C.

This stabilization indicates that the code encounters a memory bandwidth bottleneck. The CPU cores are processing data faster than the main memory can supply it. Furthermore, at 64 and 256 threads, the overhead of managing, scheduling, and synchronizing a large number of OpenMP threads might harm performance.

2.3 Discussion

The data reveals that optimizing memory access patterns yields a much larger performance gain than parallelization alone. Even in the serial execution, Version C was faster than Version A.

- **Version A:** The poor performance across all thread counts is due to cache trashing. 2D arrays was stored in row major order. In the loop, matrix B is accessed column by column, meaning each consecutive read jumps across memory by the size of an entire row (2048×8 bytes). This stride access triggers a cache miss, forcing the CPU to idle while waiting on slow main memory.
- **Version B:** Performance significantly improve over Version A because the 8×8 sub blocks fit entirely within the CPU caches. Once a block is loaded, the data is reused multiple times before eviction, significantly reducing the required memory bandwidth.
- **Version C:** This version achieves the optimal performance baseline. By transposing matrix B, the column-wise accesses become row-wise accesses. The innermost loop now reads both A and B^T sequentially. When one element is fetched, adjacent elements are pulled into the cache line simultaneously, allowing the CPU's hardware to operate efficiently and keeping the processing cores fully fed with data.

3 Estimating π

3.1 Execution Time Report

Table 2 shows the execution times across three different implementations.

Implementation	Execution Time (sec)
Serial	0.193506
Version A (4 threads, `for` directive)	0.058047
Version B (2 threads, `sections` directive)	0.084529

Table 2: Execution times for Serial Version, Version a and Version B.

3.2 Observations

The parallel implementations reduced the total execution time. Version B, utilizing 4 threads with the `for` directive, achieved the fastest execution time. Version B, utilizing 2 threads with the `sections` directive, was also significantly faster than the serial version, though slower than Version A due to having half the number of threads.

4 Sorting

4.1 Execution Time

The performance of Serial Version and OpenMP Version were tested against an array size of $2 \times 1024 \times 1024$ under two distinct initialization conditions. The results are summarized in Table 3.

Initialization	Serial Version	OpenMP Version
`m[i] = size - i`	0.044176 sec	0.023710 sec
`m[i] = rand()`	0.193621 sec	0.150203 sec

Table 3: Execution Times of Quicksort

4.2 Observation

The experimental result confirms that the OpenMP implementation reduces execution time compared to the serial baseline across both data distributions. Utilizing two parallel threads provided a noticeable speedup, validating the efficiency of the divide and conquer parallelization approach.

As observed in the execution times, sorting the random array (`m[i] = rand()`) takes longer than sorting the reverse sorted array (`m[i] = size - i`). This performance difference is counter-intuitive but occurs for two primary reasons:

- **Pivot Selection:** In our Quicksort implementation, the pivot is chosen as the middle element of the array. For a reverse sorted array, the middle element represents the true median value. This guarantees perfectly balanced partitions at every recursive step, resulting in the best case performance for Quicksort, which is exactly balanced tree $O(N \log N)$. For a randomized array, the middle element leads to unbalanced partitions and a deeper recursion tree.
- **CPU Branch Prediction:** Modern processors rely on branch prediction. The pattern of a reverse sorted array allows the CPU to easily predict the outcomes of the `while` loops during the partitioning phase. However, randomized array leads to frequent branch prediction failures. When a prediction failure occurs, the CPU flush its pipeline, which stalls execution and increases the overall time.