Parallel Matrix-Matrix Multiplication: Design and Performance Evaluation on Expanse HPC Cluster

Cihan Yildirim
Department of Computer Science
Coastal Carolina University
Conway, SC
cnyildiri@coastal.edu

***Abstract*—This paper presents the design, implementation, and performance evaluation of a parallel matrix-matrix multiplication algorithm using Pthreads on the Expanse HPC cluster. Building upon previous work on matrix-vector multiplication, this study extends the approach to the more computationally intensive matrix-matrix multiplication. We compare serial and parallel implementations across varying matrix sizes and thread counts, reporting overall, computation-only, and I/O times as well as speedup and efficiency metrics.**

***Index Terms*—Parallel Computing, Matrix-Matrix Multiplication, Pthreads, HPC, Performance Evaluation**

I. Introduction

Matrix-matrix multiplication is a core operation in many scientific and engineering applications. Unlike matrix-vector multiplication, which is relatively straightforward to parallelize, matrix-matrix multiplication increases both the computational load and the complexity of memory management. This work evaluates a parallel implementation using Pthreads on the Expanse HPC cluster and contrasts it with the serial version. This document also discuss the key differences from our previous matrix-vector multiplication assignment.

II. Background

Matrix-matrix multiplication computes the product of an $m \times n$ matrix A and an $n \times q$ matrix B to produce a result matrix C of dimensions $m \times q$. The standard algorithm uses three nested loops:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}, \quad \text{for } i = 1, 2, \ldots, m \text{ and } j = 1, 2, \ldots, q.$$

This naive algorithm performs $O(m \times n \times q)$ arithmetic operations, which, for square matrices of size $N \times N$, results in a computational complexity of $O(N^3)$. When N is large, the number of operations can grow very quickly, leading to high computational demands.

Compared to matrix-vector multiplication—which involves only $O(m \times n)$ work and typically exhibits a more straightforward memory access pattern—matrix-matrix multiplication requires the calculation of a dot product for every pair of a row from A and a column from B. This not only increases the number of computations but also intensifies the pressure on memory bandwidth and cache utilization. The larger data movement can result in performance bottlenecks if memory hierarchy optimizations, such as blocking or tiling, are not employed.

Moreover, matrix-matrix multiplication is a cornerstone operation in many scientific and engineering applications including numerical simulations, graphics transformations, and machine learning. Its inherent parallelism—since the computation of each element in C is largely independent—makes it an ideal candidate for parallel computing. Shared-memory multithreading using Pthreads allows for the division of the workload across multiple processor cores, potentially reducing overall execution time.

However, achieving efficient parallelization requires careful consideration of load balancing and memory access patterns. In contrast to matrix-vector multiplication, where each output element is computed from a single row and a vector, matrix-matrix multiplication involves the combined use of two large matrices, making cache optimization and synchronization strategies critical for scaling performance.

In summary, while the naive three-loop algorithm for matrix-matrix multiplication is simple and intuitive, its computational and memory challenges necessitate the use of parallel computing strategies to achieve practical performance on large-scale problems.

III. Design

The implementation is divided into several modules, each responsible for a specific task:

A. File Structure and Data Format

The codebase includes:

- **make_matrix.c** – Generates random matrices in binary format.
- **print_matrix.c** – Reads and prints matrices from binary files.
- **matrix_matrix.c** – Serial matrix-matrix multiplication implementation.
- **pth_matrix_matrix.c** – Parallel matrix-matrix multiplication using Pthreads.
- **sbatch.bash** – Batch script to run experiments on the Expanse cluster.
- **plot_results.py** – Python script to process results and generate performance plots.

Matrices are stored in binary format: the first two integers specify the dimensions, followed by the matrix elements in row-major order.

B. Parallelization Strategy

In the serial implementation, three nested loops compute each element of C. In the parallel version, the rows of C are divided among threads using macros such as `BLOCK_LOW` and `BLOCK_HIGH` to ensure balanced workload distribution. Each thread independently computes its assigned rows, thus reducing overall computation time while mitigating the increased complexity compared to matrix-vector multiplication.

C. Differences from Matrix-Vector Multiplication

While matrix-vector multiplication involves less data movement and lower computational complexity, matrix-matrix multiplication requires managing additional loops and significantly larger data sets. This not only increases computation time but also intensifies memory bandwidth and cache utilization challenges. Our design addresses these issues through careful work partitioning and memory management optimizations.

IV. Experimental Evaluation

A. Setup

Experiments were conducted on the Expanse HPC cluster with the following configuration:

- **Programming Language:** C with Pthreads.
- **Compiler:** GCC (compiled on Expanse to ensure compatibility with GLIBC, typically version 2.17).
- **Hardware:** Expanse compute nodes with AMD EPYC processors, 128 cores per node, and 256GB RAM.
- **Datasets:** Square matrices (i.e., $N \times N$) for $N \in \{256, 512, 1024, 2048, 4096\}$ (and 8192 if feasible).
- **Thread Configurations:** $P \in \{1, 2, 4, 8, 16, 32, 64, 128\}$.
- **Metrics Measured:** Overall execution time, computation-only time, I/O time, speedup, and efficiency.

An automated batch script (`sbatch.bash`) was used to generate matrices, run both serial and parallel implementations, and log results in CSV format.

Performance plots generated from the experimental data illustrate these trends and provide insights into the scalability and limitations of the parallel implementation.

B. Setup

All experiments were conducted on the Expanse HPC cluster at SDSC. Our goal was to evaluate both the serial and parallel (Pthreads) implementations of matrix-matrix multiplication for square matrices of sizes 256×256, 512×512, 1024×1024, and, where possible, 4096×4096 and 8192×8192. We varied the number of threads over $P \in \{1, 2, 4, 8, 16, 32, 64, 128\}$. Each experiment recorded:

- **Overall Time:** Total wall-clock time, including input/output (I/O) and computation.
- **Compute Time:** Time spent exclusively on numerical multiplication, excluding I/O.

We then derived:

- **Speedup:** Speedup $= \dfrac{T_{\text{serial}}}{T_{\text{parallel}}}$, comparing parallel time against the serial baseline.
- **Efficiency:** Efficiency $= \dfrac{\text{Speedup}}{P}$, which indicates how effectively the threads are utilized.

C. Evaluation Results

Figures 1 through 2 present our primary performance metrics:

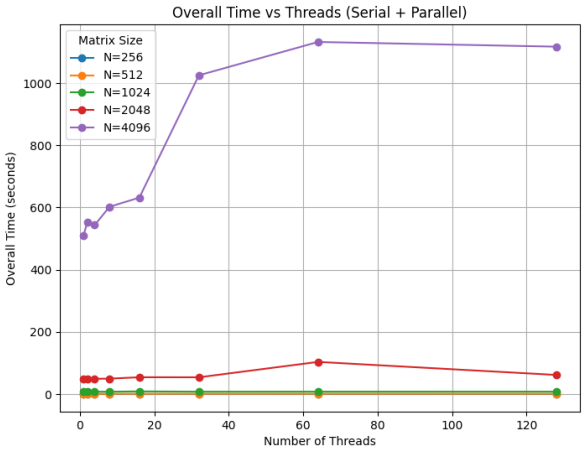

Fig. 1. Overall execution time (including I/O) vs. number of threads for various matrix sizes.

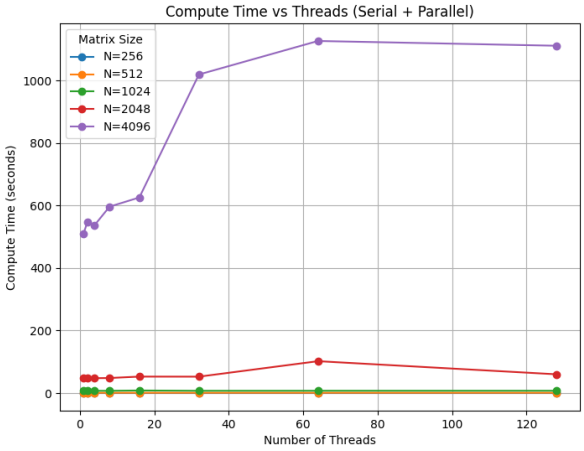

Fig. 2. Speedup (based on overall time) vs. number of threads for various matrix sizes.

1) Overall Time and Compute Time: As shown in Figures 1 and 2, increasing the number of threads reduces both the overall execution time and the pure compute time. However, beyond a certain threshold, speedup gains diminish due to thread synchronization overheads and memory bandwidth saturation. Larger matrices see more noticeable improvements from parallelization, but also encounter greater memory traffic.

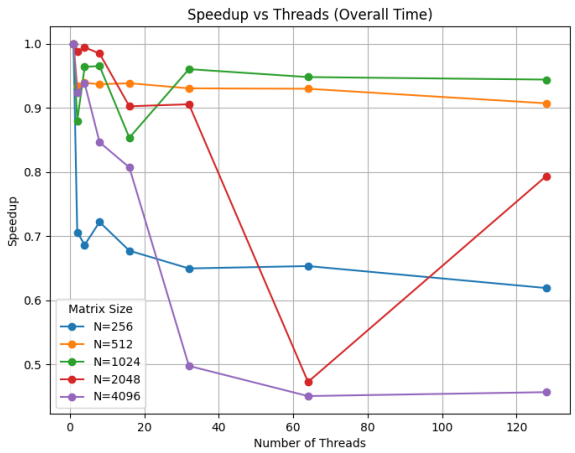

Fig. 3. Efficiency (based on overall time) vs. number of threads for various matrix sizes.

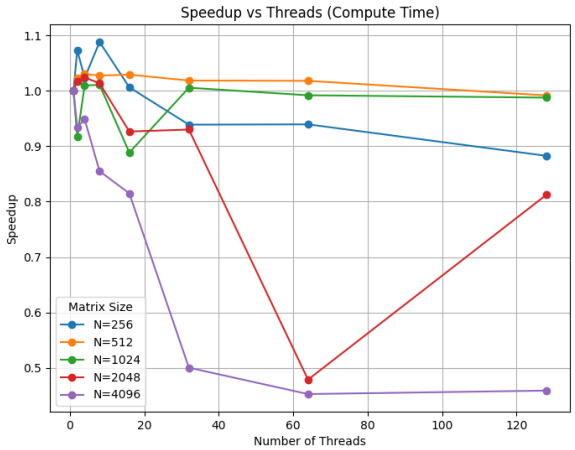

Fig. 4. Speedup (based on compute-only time) vs. number of threads for various matrix sizes.

2) Speedup and Efficiency: Figures 3, 4, 5 and 6 depict the speedup and efficiency for overall time and compute-only time. We observe near-linear speedup for small thread counts, but efficiency declines as P increases. This aligns with Amdahl's Law, which indicates the impact of any non-parallelizable portion of the code becomes more significant at higher thread counts. Figures 3 and 4 illustrate the speedup achieved when increasing the number of threads, for both *compute-only* time and *overall* time. Meanwhile, Figures 5 and 6 present the corresponding efficiency curves.

a) Speedup Trends.: For smaller thread counts $(2, 4, 8)$, we generally observe near-linear speedup for each matrix size, indicating that the parallelization is effective in reducing the total runtime. However, as the thread count continues to increase (beyond 8 or 16 threads), the speedup often *declines* and may even fall below 1.0 for large matrices. This phenomenon suggests that parallel overheads (e.g., thread management, synchronization, memory contention) become increasingly significant and can overshadow the raw parallel gains.

Notably, in Figure 4 (compute-only time), the smaller matrices ($N = 256, 512$) sometimes hover near or slightly above a speedup of 1.0 for a moderate number of threads, then gently taper off. Larger matrices ($N = 2048, 4096$) can experience more dramatic dips, with speedups sometimes dropping below 1.0 for higher thread counts; this often indicates that the memory bandwidth or cache effects are limiting performance gains once many threads are contending for resources. In Figure **??** (*overall* time), the inclusion of I/O or additional overhead tends to reduce the observed speedup further, especially at higher thread counts.

b) Efficiency Trends.: Figures 5 and 6 show that efficiency (speedup/number of threads) starts at 1.0 for a single thread (by definition), then rapidly decreases as the number of threads grows. This drop is a classic illustration of **Amdahl's Law**: even a small fraction of non-parallelizable work, or overhead in setting up and coordinating threads, becomes magnified as more threads are added. Consequently, the portion of time that can be parallelized does not scale perfectly, causing efficiency to diminish with higher P.

For *compute-only* time (Figure 6), the curves remain higher for small P (e.g., 2 or 4 threads), indicating relatively good parallel utilization. As soon as the thread count surpasses this modest range, efficiency tends toward a low fraction (< 0.2 or even lower) for all tested matrix sizes, signifying that overheads outweigh the benefits of additional concurrency. For *overall* time (Figure 5), the efficiency is often even lower because the total runtime includes I/O and other serial portions that do not benefit from parallelization.

c) Interpretation and Amdahl's Law.: These results confirm that for very large thread counts, the non-parallelizable fraction of the workload (and additional overhead) becomes the dominant factor, which is precisely the insight given by Amdahl's Law. While small-scale parallelism (e.g., up to 8 threads) can deliver near-linear speedup in many cases, going beyond that quickly saturates the system's capabilities and may introduce additional overhead (thread management, memory contention, etc.). Consequently, the efficiency curves plummet and overall performance gains flatten or even degrade.

In summary, the experiments show that:

- **Small to moderate** thread counts (≤ 8) often achieve good speedups.
- **High** thread counts (≥ 16) can reduce performance gains significantly because of overheads and partial serialization.
- **Larger** matrix sizes ($N = 2048$ and $N = 4096$) can exhibit *worse scaling* at higher threads if memory or synchronization overhead dominates.

Overall, the balance between computational gains and overhead determines the practical parallel efficiency, with Amdahl's Law providing a useful upper bound on the expected speedup.

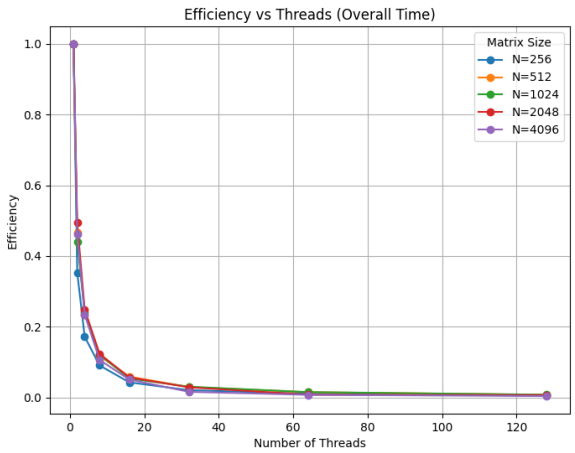

Fig. 5. Efficiency (based on overall time) vs. number of threads for various matrix sizes.

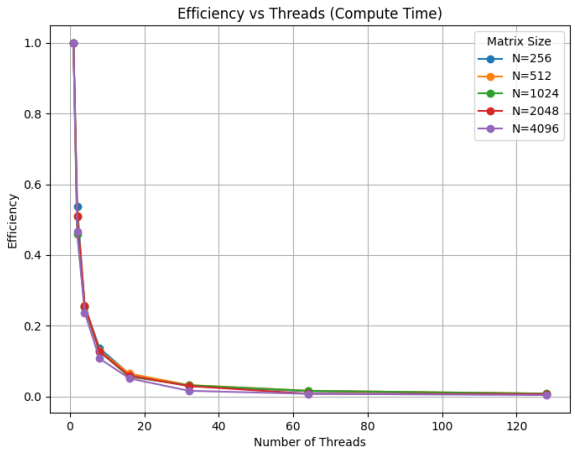

Fig. 6. Efficiency (based on compute-only time) vs. number of threads for various matrix sizes.

3) Memory Constraints for Large Matrices: Although we attempted to include 8192×8192 matrices, running these experiments on Expanse proved difficult. We encountered memory allocation failures and inconsistent performance results, even with increased memory requests in the Slurm scripts. These limitations indicate that further optimizations—such as blocking, tiling, or GPU offloading—may be required to handle extremely large problem sizes reliably on this system.

V. Conclusion

This work presented a parallel matrix-matrix multiplication implementation on the Expanse HPC cluster using Pthreads. The results demonstrate that parallelization significantly reduces computation time for moderate matrix sizes (up to 1024×1024), achieving considerable speedups and high efficiency at lower thread counts. However, as the number of threads grows, synchronization overhead and memory bandwidth constraints limit scalability, aligning with Amdahl's Law.

Attempts to multiply and 8192×8192 matrices revealed critical memory constraints on Expanse, resulting in failed jobs or inconsistent performance. These findings highlight the need for further optimizations, such as tiling or blocking strategies to improve cache locality, and potentially GPU acceleration or distributed-memory (MPI) approaches to handle large-scale problem sizes more effectively.

In summary, while the parallel implementation offers notable speedup for smaller matrices, extending this approach to very large matrices requires additional techniques and resources. Future work will focus on employing advanced parallelization frameworks (e.g., OpenMP, MPI, or hybrid MPI+Threads), exploring GPU offloading, and optimizing memory usage to fully leverage the computational potential of HPC environments like Expanse.

Acknowledgment

This work was conducted as part of CSCI 473 and utilized Expanse HPC resources provided by SDSC. The authors thank their colleagues for valuable discussions and support.

References

[1] D. Patterson and J. Hennessy, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2017.

[2] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*. Pearson, 2003.

[3] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558-565, 1978.

[4] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Pearson, 2004.