Parallel Matrix-Matrix Multiplication

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

***Abstract*—This paper presents an extensive study on the design, implementation, and performance evaluation of parallel matrix-matrix multiplication algorithms on the Expanse HPC cluster. Building on a previous matrix-matrix multiplication work, we now focus on the more computationally demanding matrix-matrix multiplication. Our study details two parallelization approaches: a low-level threading model using Pthreads and a higher-level directive-based model using OpenMP. In addition to describing the underlying matrix-matrix routines and their inherent computational challenges, we discuss the nuances of thread creation, load balancing, and memory management. Comprehensive experimental evaluations are provided for various matrix sizes and thread counts, with detailed discussions on overall, computation-only, and I/O times, as well as speedup and efficiency metrics. Our results not only validate the effectiveness of parallelization but also highlight critical challenges such as synchronization overhead and memory bandwidth limitations.**

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

I. Introduction

Matrix-matrix multiplication is a fundamental kernel in scientific computing, computer graphics, and machine learning. As problem sizes increase, the cubic complexity inherent in the naive algorithm imposes severe computational challenges. In high-performance computing (HPC) environments, parallelizing this operation is crucial to reducing execution times and enhancing resource utilization [1] [2].

This paper expands our earlier work on matrix-matrix multiplication by addressing matrix-matrix multiplication, which, due to its three nested loop structure, demands far more attention to both computational load and memory management. Two parallel programming paradigms are explored:

- **Pthreads:** A low-level threading model that gives fine-grained control over thread creation and synchronization.
- **OpenMP:** A higher-level, directive-based approach that simplifies parallel loop execution.

We compare these parallel implementations with the serial version on the Expanse HPC cluster, highlighting both the benefits and the limitations (e.g., overheads, synchronization delays, memory bandwidth saturation) inherent to parallel computing [4]. The remainder of the paper is organized as follows: Section II provides background on matrix-matrix routines and the parallel frameworks used; Section III details the design and implementation; Section IV presents the experimental evaluation; and Section V concludes with a discussion of future work.

II. Background

This section provides a comprehensive overview of the computational kernels and parallelization techniques used in this work.

A. Matrix-Matrix Multiplication Routines

Matrix-matrix multiplication involves computing the product of an $m \times n$ matrix A and an $n \times q$ matrix B to yield an $m \times q$ matrix C. The standard algorithm uses three nested loops:

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

For square matrices of size $N \times N$, the computational complexity is $O(N^3)$. Each element of C is computed as a dot product of a row of A and a column of B, leading to large numbers of arithmetic operations and data accesses. The naive algorithm, while simple, suffers from poor cache performance and high memory traffic if not optimized with techniques such as blocking or tiling [2]. Blocking divides the matrices into sub-blocks to improve cache locality, though in our study we focus on straightforward parallelization methods to better isolate the benefits of multithreading.

In the standard matrix-matrix multiplication expression

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj},$$

the variables represent the following:

- A: The first input matrix of dimensions $m \times n$. Here,
  - m denotes the number of rows in A,
  - n denotes the number of columns in A.
- B: The second input matrix of dimensions $n \times q$. Here,
  - n (which must be equal to the number of columns in A) denotes the number of rows in B,
  - q denotes the number of columns in B.
- C: The output matrix resulting from the multiplication $A \times B$, with dimensions $m \times q$.
- i: The row index used for matrix A (and the corresponding row in C). It ranges from 1 to m.
- j: The column index used for matrix B (and the corresponding column in C). It ranges from 1 to q.
- k: The summation index, which runs from 1 to n, used to compute the dot product between the ith row of A and the jth column of B.

Thus, each element C_{ij} is calculated as:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj},$$

which means that the ith row of A and the jth column of B are multiplied element-wise and the results are summed to produce the corresponding entry in C.

For square matrices of size $N \times N$, where $m = n = q = N$, the total number of arithmetic operations required is on the order of $O(N^3)$.

B. Pthreads

Pthreads (POSIX Threads) is a standardized API for multithreading in C and C++ on Unix-like operating systems [4]. Using Pthreads, a program can create multiple threads that execute concurrently, sharing the same address space. This model is especially useful for operations such as matrix-matrix multiplication where each thread can be assigned a subset of the rows of the output matrix C.

Key aspects of Pthreads include:

- **Thread Creation and Termination:** Functions like `pthread_create()` initiate new threads, and `pthread_join()` waits for threads to finish.
- **Synchronization:** Pthreads offer primitives such as mutexes and condition variables to prevent race conditions and ensure proper data synchronization.
- **Work Division:** In our implementation, macros (e.g., `BLOCK_LOW` and `BLOCK_HIGH`) are used to evenly partition the matrix rows among available threads.

The advantage of Pthreads is the fine-grained control it offers; however, it requires explicit management of thread lifecycles and careful attention to synchronization to avoid overheads that can diminish performance.

C. OpenMP

OpenMP is an API that supports multi-platform shared-memory parallel programming in C, C++, and Fortran. It relies on compiler directives (pragmas) to instruct the compiler to execute loops and sections of code in parallel [2]. OpenMP abstracts much of the complexity of thread management, allowing developers to parallelize code with minimal modifications.

Important features of OpenMP include:

- **Parallel Regions:** Directives such as `#pragma omp parallel for` automatically divide loop iterations among threads.
- **Thread Management:** OpenMP internally manages thread creation, scheduling, and termination, reducing the need for explicit calls.
- **Scheduling:** It offers different scheduling strategies (static, dynamic, guided) to balance load across threads.
- **Scalability:** Although simpler to use, the performance gains from OpenMP depend on factors such as the overhead of parallel region creation and memory bandwidth limitations.

In our OpenMP implementation, a similar division of work is applied as with Pthreads, but with less code overhead. The ease of use and rapid development cycle make OpenMP an attractive option for many shared-memory applications.

III. Design and Implementation

This section describes the overall architecture of our matrix multiplication programs and details the implementations for both the serial and parallel versions.

A. File Structure and Data Format

The project is organized into several modules:

- **make_matrix.c:** Generates random matrices stored in binary format. The file begins with two integers representing the matrix dimensions, followed by matrix elements in row-major order.
- **matrix_matrix.c:** Implements the serial matrix-matrix multiplication using the naive three-loop algorithm .
- **pth_matrix_matrix.c:** Contains the Pthreads implementation, where the matrix C is partitioned by rows across multiple threads. Each thread independently computes its assigned rows using block partitioning macros from `quinn_macros.h` .
- **omp_matrix_matrix.c:** Provides an OpenMP-based parallel version where loop parallelization is handled by the OpenMP runtime. The directive `#pragma omp parallel for` simplifies work distribution .
- **print_matrix.c:** Utility to read and print a matrix from a binary file for verification purposes.
- **sbatch.bash:** A batch script for executing experiments on the Expanse HPC cluster, automating runs for various matrix sizes and thread counts.
- **result.py:** Python scripts for processing experiment results and generating performance plots.

Both approaches aim to reduce the overall compute time by dividing the work among multiple cores. However, the effectiveness of parallelization is limited by factors such as thread overhead, memory contention, and the non-parallelizable portions of the code as described by Amdahl's Law [4].

B. Parallelization Strategy

For the serial version, the standard triple-loop algorithm computes every element of the result matrix C. In contrast, the parallel implementations employ two distinct approaches:

1) **Pthreads Implementation:**
   - The rows of C are divided among threads based on the total number of rows and the thread count.
   - Each thread is responsible for a contiguous block of rows.
   - The macros `BLOCK_LOW` and `BLOCK_HIGH` (defined in `quinn_macros.h`) ensure that work is evenly distributed.
   - Threads are created using `pthread_create()` and synchronized with `pthread_join()` to ensure that all computations are complete before the result is written to disk.

The relevant portions of the Pthreads implementation are shown below. In this code, the function `Pth_mat_mult` is the thread routine. Each thread calculates a block of rows of the result matrix C by first determining its starting and ending row indices using the macros `BLOCK_LOW` and `BLOCK_HIGH`. Then, it performs the standard triple-loop multiplication for its assigned rows.

```
    #Word thread has been changed to "t" for pseudo
void *Pth_mat_mult(void *rank) {
    long my_rank = (long) rank;
    int my_first_row = BLOCK_LOW(rank, t, m);
    int my_last_row  = BLOCK_HIGH(rank, t, m);

    for(int i=my_first_row;i<=my_last_row;i++){
        for (int j = 0; j < q; j++) {
            double sum = 0.0;
            for (int k = 0; k < n; k++) {
                sum+= A[i*n+k]*B[k*q+j];
            }
            C[i * q + j] = sum;
        }
    }
    return NULL;
}

for (t= 0; t < tcount; t++)
    p_create(&t_handles[t],NULL,Pth_mat_mult,(void*)t);

for (t = 0; t < tcount; t++)
    pthread_join(t_handles[t], NULL);
```

In this implementation, after reading the matrices and allocating the result matrix C, the program creates multiple threads. Each thread executes `Pth_mat_mult()`, which computes a portion of C independently. Once all threads finish their work (ensured by `pthread_join()`), the main thread writes the final result to a file.

2) **OpenMP Implementation:**
   - The OpenMP version uses the `#pragma omp parallel for` directive to automatically split the outer loop across threads.
   - The runtime environment manages thread creation, scheduling, and synchronization, greatly simplifying the parallel code.
   - The number of threads is set by `omp_set_num_threads()` based on the command-line input.

The core of the OpenMP-based matrix-matrix multiplication is illustrated below. Here, the directive `#pragma omp parallel for schedule(static)` is applied to the outer loop (iterating over rows), allowing the iterations to be distributed evenly across the available threads. OpenMP handles thread management and synchronization, reducing the programming overhead compared to manual thread management with Pthreads.

```
#include <omp.h>
...
int thread_count = atoi(argv[4]);
omp_set_num_threads(thread_count);

#pragma omp parallel for schedule(static)
for (int i = 0; i < m; i++) {
    for (int j = 0; j < q; j++) {
        double sum = 0.0;
        for (int k = 0; k < n; k++) {
            sum += A[i * n + k] * B[k * q + j];
        }
        C[i * q + j] = sum;
    }
}
```

In the OpenMP version, after the matrices are loaded and the result matrix C is allocated, the number of threads is set using `omp_set_num_threads()`. The `parallel for` directive then divides the work of processing each row of the output matrix C among the threads automatically. The simplicity of this approach lies in its minimal code modifications relative to the serial version, as OpenMP abstracts much of the thread management and synchronization.

Both approaches aim to reduce the overall compute time by dividing the work among multiple cores. However, the effectiveness of parallelization is limited by factors such as thread overhead, memory contention, and the non-parallelizable portions of the code as described by Amdahl's Law [4].

C. Differences from Matrix-Vector Multiplication

Matrix-vector multiplication typically requires only $O(m \times n)$ operations with a simpler memory access pattern, since each output element is computed from one row of A and one element of the vector. In contrast, matrix-matrix multiplication involves a dot product between a row of A and a column of B for each element of C, resulting in $O(N^3)$ complexity for square matrices. This increase in computational and memory requirements necessitates more advanced load balancing and cache optimization strategies. Our design reflects these challenges through explicit work partitioning and efficient memory access patterns.

IV. Experimental Evaluation

In this section, we detail the experimental setup, methodology, and results obtained from running the serial, Pthreads, and OpenMP implementations on the Expanse HPC cluster.

A. Setup and Methodology

Experiments were conducted on Expanse, a high-performance computing cluster equipped with AMD EPYC processors and up to 128 cores per node. The software environment was set up as follows:

- **Programming Language:** C (compiled with GCC).
- **Parallel Frameworks:** Pthreads for explicit thread management and OpenMP for directive-based parallelism.
- **Matrix Sizes:** Square matrices with dimensions $N \in \{256, 512, 1024, 2048, 4096\}$ (and 8192 where feasible).
- **Thread Configurations:** Thread counts ranging from 1 to 128.

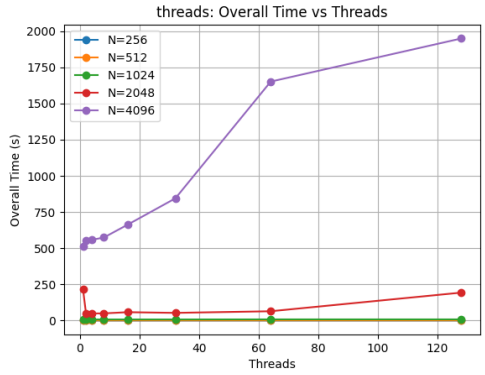

Fig. 1. Overall Time vs. number of threads for the Pthreads implementation across various matrix sizes.

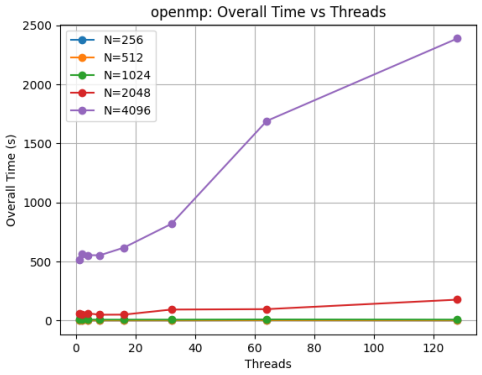

Fig. 4. Overall Time vs. number of threads for the OpenMP implementation across various matrix sizes.

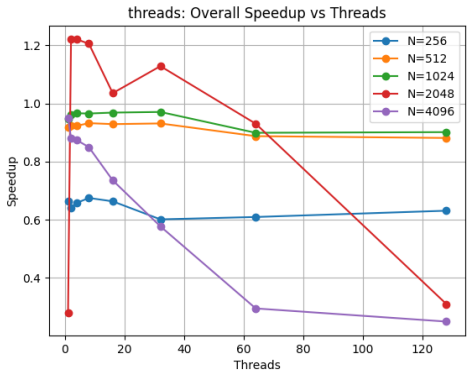

Fig. 2. Overall Speedup vs. number of threads for the Pthreads implementation across various matrix sizes.

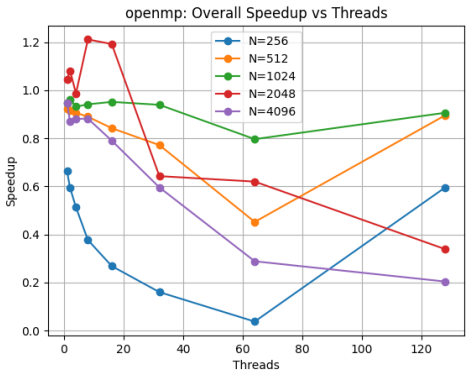

Fig. 5. Overall Speedup vs. number of threads for the OpenMP implementation across various matrix sizes.

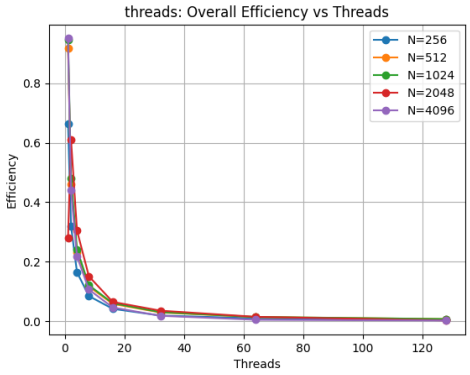

Fig. 3. Overall Efficiency vs. number of threads for the Pthreads implementation across various matrix sizes.

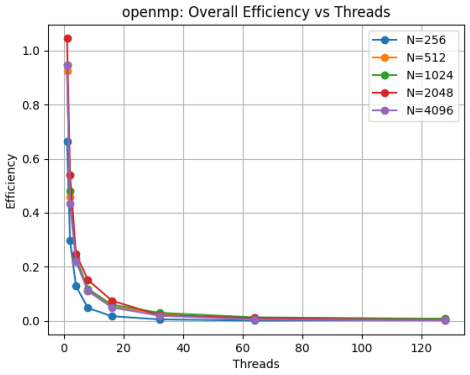

Fig. 6. Overall Efficiency vs. number of threads for the OpenMP implementation across various matrix sizes.

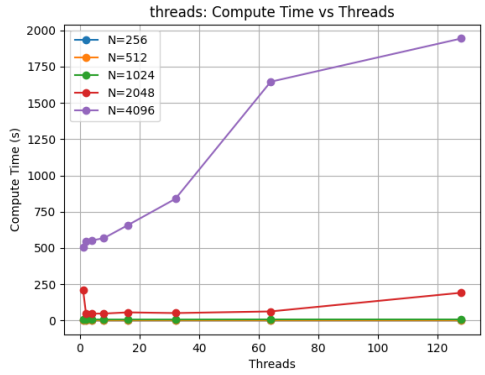

Fig. 7. Compute Time vs. number of threads for the Pthreads implementation across various matrix sizes.

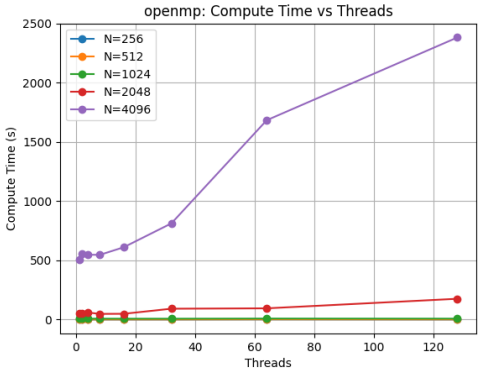

Fig. 10. Compute Time vs. number of threads for the OpenMP implementation across various matrix sizes.

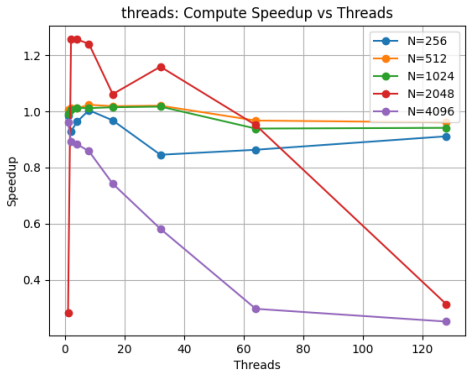

Fig. 8. Compute Speedup vs. number of threads for the Pthreads implementation across various matrix sizes.

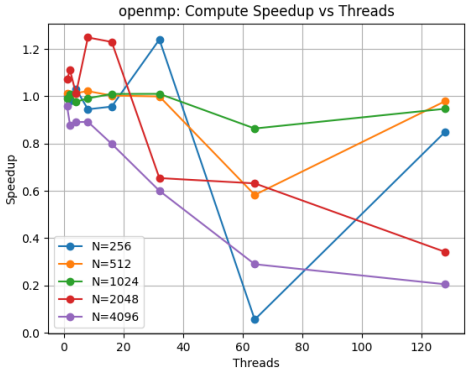

Fig. 11. Compute Speedup vs. number of threads for the OpenMP implementation across various matrix sizes.

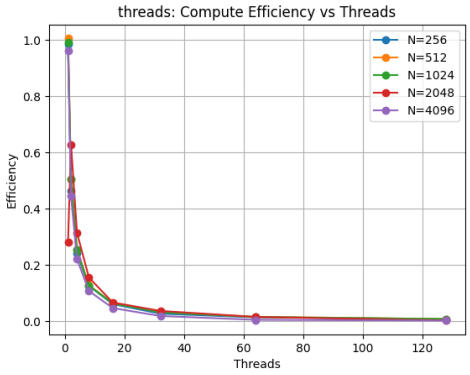

Fig. 9. Compute Efficiency vs. number of threads for the Pthreads implementation across various matrix sizes.

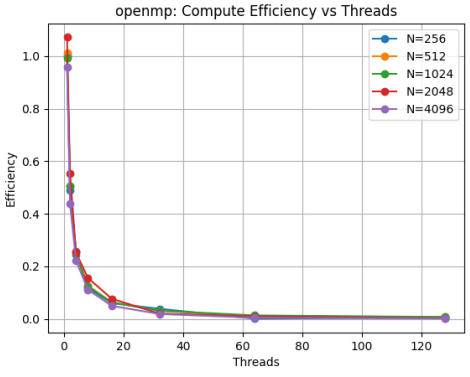

Fig. 12. Compute Efficiency vs. number of threads for the OpenMP implementation across various matrix sizes.

V. Overall Performance

A. Results and Analysis

Performance results are presented in a series of plots generated by Python scripts (`plots_results.py`).

- **Overall Time and Compute Time:** As the number of threads increases, the total workload is divided among more processors, reducing both overall and compute times initially. However, this scaling is not indefinite. With moderate thread counts, the benefits of parallelization are clear because the work per thread is still high relative to the overhead. Beyond this point, two key factors cause diminishing returns:
  - **Synchronization Overhead:** With more threads, the need to coordinate tasks, share results, and maintain data consistency grows. These synchronization points (e.g., barriers, locks) add extra time that does not contribute to actual computation.
  - **Memory Bandwidth Constraints:** When many threads access memory simultaneously, they can saturate the available memory bandwidth. This results in cache misses and delays, as the processors must wait for data to be fetched, reducing the effectiveness of parallel execution.

  For small matrix sizes, the overhead of creating and managing threads can be comparable to or even exceed the computation time, so parallelization offers limited benefits. For larger matrices, while there is improved scaling because the computational workload per thread increases, the advantages eventually plateau as synchronization and memory contention become dominant.
- **Speedup:** Near-linear speedup is observed for low thread counts because the parallelizable portion of the code benefits directly from having more processing units. However, as the thread count increases:
  - **Increased Synchronization Costs:** The overhead associated with coordinating a large number of threads grows, reducing the net gain from parallelization.
  - **Memory Contention:** With more threads, there is higher competition for shared memory resources. This competition leads to delays, meaning that the additional threads do not proportionally reduce the computation time.
  - **Amdahl's Law:** Even if the parallel portion scales perfectly, the overall speedup is limited by the serial fraction of the code. As more threads are added, the impact of the non-parallelizable parts becomes more significant, preventing the speedup from scaling linearly.

  Thus, while the initial gains in speedup are promising, the benefits diminish as overheads increase.
- **Efficiency:** Efficiency, defined as speedup divided by the number of threads, is high when a few threads are used because most of the added resources directly contribute to reducing execution time. However, as more threads are employed:
  - **Diminishing Returns:** The proportional benefit of adding extra threads decreases, since the overhead and contention issues start to dominate.
  - **Impact of Serial Code:** The serial fraction of the program (as described by Amdahl's Law) limits the maximum efficiency, causing a rapid decline as the thread count increases.
  - **Resource Underutilization:** As the overhead grows, the additional threads may spend more time waiting or synchronizing than performing useful work, leading to a drop in efficiency.

  In summary, while parallelism can yield substantial benefits, the efficiency drops sharply with high thread counts due to the fixed overheads and the inevitable presence of non-parallelizable code segments.

These figures illustrate these trends in detail. The results validate that while both Pthreads and OpenMP provide significant improvements over the serial implementation, the inherent challenges of data movement and synchronization limit scalability for very high thread counts.

VI. Discussion

The experiments reveal a complex trade-off between computational gains and overhead costs:

- **Thread Overheads:** Pthreads offer fine-grained control but incur higher overhead due to explicit thread management and synchronization. This is evident in the marginal improvements observed for small problem sizes and the eventual performance degradation at very high thread counts.
- **Memory Contention:** As more threads access shared memory, contention and cache misses become significant, particularly for large matrices. This contention limits the achievable speedup and contributes to the rapid decline in efficiency as the number of threads increases.
- **I/O Limitations:** Although the majority of execution time is spent in computation for larger matrices, I/O overhead cannot be ignored, especially in the serial version where file read/write operations contribute a non-negligible portion of the total time. This factor further complicates the direct comparison between serial and parallel implementations.

These figures illustrate these trends in detail. The results validate that while both Pthreads and OpenMP provide significant improvements over the serial implementation, the inherent challenges of data movement and synchronization limit scalability for very high thread counts.

VII. Conclusion

This study presented an in-depth exploration of parallel matrix-matrix multiplication on the Expanse HPC cluster using two different parallel programming paradigms: Pthreads and OpenMP. Our extended report detailed the matrix multiplication routine, explained the mechanisms behind Pthreads and OpenMP, and provided comprehensive experimental results comparing serial and parallel implementations.

Key findings include:

- Parallel implementations significantly reduce compute time for moderate matrix sizes, but benefits diminish for high thread counts due to overheads.
- Pthreads offer fine-grained control at the cost of increased complexity, whereas OpenMP provides a simpler and more concise approach to loop-level parallelism.
- The efficiency of parallel implementations is ultimately constrained by memory bandwidth, synchronization overhead, and the serial fraction of the code.

Future work will explore advanced optimizations such as cache blocking, hybrid MPI+Threads models, and GPU acceleration to further enhance performance on large-scale matrix operations.

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.

[5] W. W. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.

[6] J. Dongarra et al., "An Updated Set of Basic Linear Algebra Subprograms (BLAS)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.