Performance Analysis of Parallel Matrix-Vector Multiplication using Pthreads

Mason Dizick
Coastal Carolina University
Department of Computing Sciences
Conway, SC, USA
mpdizick@coastal.edu

***Abstract*—This study presents a parallel implementation of matrix-vector multiplication using POSIX threads (pthreads) and examines the program's performance characteristics across varying matrix sizes and thread counts. The implementation employs Quinn's data distribution macros to achieve balanced workload distribution among threads. Experiments were conducted on SDSC's Expanse supercomputer with AMD EPYC 7742 processors, evaluating square matrices ranging from 10,000×10,000 to 40,000×40,000 elements with thread counts from 1 to 128. Results demonstrate moderate overall speedup characteristics limited by I/O overhead, with maximum overall speedup of 4.84× achieved at 128 threads for the 30,000×30,000 matrix. However, the work-only portion exhibits significantly better scaling, achieving 13.81× speedup at 128 threads for the largest matrix size. Efficiency analysis reveals that I/O operations constitute the primary performance bottleneck, accounting for approximately 70-75% of total execution time at high thread counts. The modular design separating computation from I/O operations demonstrates that the parallel computation kernel achieves near-linear scaling up to 16 threads with efficiency above 85%, while overall performance is constrained by sequential I/O operations as predicted by Amdahl's Law.**

***Index Terms*—Matrix-vector multiplication, Parallel computing, Pthreads, Performance analysis, Speedup, Efficiency, High-performance computing, Load balancing**

I. Introduction

Matrix-vector multiplication is a fundamental operation in linear algebra and scientific computing, with applications spanning computer graphics, machine learning, numerical simulations, and solving systems of linear equations [1]. Given a matrix $\mathbf{A}$ of size $m \times n$ and a vector $\mathbf{x}$ of size $n \times 1$, the product $\mathbf{y} = \mathbf{A}\mathbf{x}$ produces a vector of size $m \times 1$, where each element is computed as:

$$y_i = \sum_{j=0}^{n-1} A_{ij} \cdot x_j, \quad i = 0, 1, \ldots, m-1 \tag{1}$$

As matrix dimensions increase, the computational cost grows quadratically ($O(mn)$ operations), making parallelization an attractive approach for performance improvement. The embarrassingly parallel nature of this operation—where each output element can be computed independently—makes it an ideal candidate for shared-memory parallelization using pthreads.

This study presents a parallel C implementation of matrix-vector multiplication using POSIX threads and examines performance characteristics across varying problem sizes and thread counts. The program utilizes Quinn's data distribution macros to ensure balanced workload distribution, even when the number of matrix rows does not divide evenly among threads. The experiment was conducted on a high-performance compute node of the San Diego Supercomputer Center's Expanse project [2].

The primary goals and objectives of this study are threefold: to identify the computational complexity and scalability of the parallel implementation using shared-memory threading, to investigate speedup and efficiency characteristics across different matrix sizes and thread counts, and to create a performance benchmark that separates I/O overhead from computational work to better understand scaling bottlenecks. This study finds that the implementation demonstrates two distinct performance regimes: overall execution achieves moderate speedup of 4.84× at 128 threads, while the computation-only portion achieves substantially better speedup of 13.81× at the same thread count. Execution times range from 0.98 seconds (10,000×10,000 matrix, 1 thread) to 16.8 seconds (40,000×40,000 matrix, 1 thread) for serial execution, with parallel execution reducing these times by factors of 4-5× overall and up to 14× for the computational kernel alone.

II. Background

A. Matrix-Vector Multiplication

Matrix-vector multiplication combines each row of the matrix with the entire vector through a dot product operation. For an $m \times n$ matrix $\mathbf{A}$ and an $n \times 1$ vector $\mathbf{x}$, the resulting vector $\mathbf{y}$ has m elements, where each element represents the inner product of a matrix row with the input vector.

The mathematical formulation can be expressed as:

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0,n-1} \\ a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{m-1} \end{bmatrix} \tag{2}$$

Each output element y_i is computed independently of other output elements, requiring only read access to the input vector $\mathbf{x}$ and row i of matrix $\mathbf{A}$. This independence makes the operation embarrassingly parallel at the row level, as no synchronization is required during computation and there are no data dependencies between output elements.

B. POSIX Threads (Pthreads)

POSIX Threads (pthreads) is a standardized threading API for shared-memory parallel programming defined by the IEEE POSIX 1003.1c standard [3]. Pthreads provides a portable interface for creating and managing threads in Unix-like operating systems, with implementations available across major platforms including Linux, macOS, and BSD variants.

Key features of the pthreads model include:

- Thread creation via `pthread_create()`, which spawns new threads of execution
- Thread synchronization via `pthread_join()`, allowing the main thread to wait for worker thread completion
- Shared memory space among all threads within a process, enabling efficient communication through shared variables
- Low overhead compared to process-based parallelism, as threads share the same address space and do not require expensive context switches

For matrix-vector multiplication, pthreads enables distribution of row computations across multiple threads, with each thread responsible for computing a subset of output elements. The shared-memory model allows all threads to access the input matrix and vector without explicit data communication.

C. Quinn's Distribution Macros

Load balancing—the even distribution of computational work across processing units—is critical for achieving high parallel efficiency. When the number of data elements does not divide evenly among threads, naive partitioning strategies can lead to load imbalance, where some threads complete their work significantly earlier than others, resulting in idle time and reduced speedup.

Quinn's data distribution macros [4] provide a systematic approach to balanced data distribution. These macros ensure that:

- Each thread receives either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$ elements, where n is the total number of elements and p is the number of threads
- Earlier-ranked threads receive at most one more element than later-ranked threads
- All elements are assigned exactly once with no gaps or overlaps
- The maximum load imbalance is limited to a single element

The macros are defined as:

$\text{BLOCK\_LOW}(id, p, n) = id \cdot n/p$
$\text{BLOCK\_HIGH}(id, p, n) = \text{BLOCK\_LOW}(id+1, p, n) - 1$
$\text{BLOCK\_SIZE}(id, p, n) = \text{BLOCK\_HIGH}(id, p, n) - \text{BLOCK\_LOW}(id, p, n) + 1$

where id is the thread rank (0 to $p-1$), p is the number of threads, and n is the number of elements to distribute. For matrix-vector multiplication, these macros distribute matrix rows among threads, ensuring balanced workload distribution regardless of whether m (the number of rows) is evenly divisible by p.

D. Performance Metrics

Parallel performance is typically characterized using two key metrics: speedup and efficiency.

Speedup (S_p) measures the performance improvement achieved by using p processors relative to serial execution:

$$S_p = \frac{T_1}{T_p} \tag{3}$$

where T_1 is the execution time using a single thread and T_p is the execution time using p threads. Ideal (linear) speedup occurs when $S_p = p$, indicating that each additional processor contributes proportionally to performance improvement.

Efficiency (E_p) normalizes speedup by the number of processors, providing a measure of resource utilization:

$$E_p = \frac{S_p}{p} = \frac{T_1}{p \cdot T_p} \tag{4}$$

Ideal efficiency is 1.0 (100%), indicating perfect linear speedup with no overhead. In practice, efficiency typically decreases as the number of threads increases due to factors including synchronization overhead, cache effects, memory bandwidth limitations, and load imbalance. Efficiency values above 0.7 (70%) are generally considered good for parallel applications.

III. Design

A. System Architecture

The implementation consists of four main utility programs and one parallel computation program:

1) **make_matrix**: Generates binary matrix files with random double-precision values
2) **print_matrix**: Displays matrix contents in formatted output for verification
3) **matrix_vector**: Serial matrix-vector multiplication implementation for baseline comparison
4) **pth_matrix_vector**: Parallel implementation using pthreads with timing instrumentation

This modular design separates concerns, with utility programs handling data generation and validation while the main programs focus on computation. All programs utilize a common binary file format for matrix storage, enabling efficient I/O and eliminating text parsing overhead.

B. Binary File Format

Matrices are stored in a compact binary format optimized for efficient reading and writing:

- Bytes 0–3: Number of rows m (4-byte signed integer)
- Bytes 4–7: Number of columns n (4-byte signed integer)
- Bytes 8 onward: Matrix data (8-byte double-precision floating-point values in row-major order)

For an $m \times n$ matrix, the total file size is $8 + 8mn$ bytes. Row-major storage order ensures that consecutive elements of each row are stored contiguously in memory, providing good cache locality during the row-wise operations performed in matrix-vector multiplication.

Vectors are represented as column matrices with $n = 1$, maintaining consistency with the general matrix file format.

C. Parallel Algorithm

The parallel implementation distributes matrix rows across threads using a block distribution strategy. Algorithm 1 presents the high-level structure.

Algorithm 1 Parallel Matrix-Vector Multiplication

```
Require: Matrix $\mathbf{A}$ ($m \times n$), vector $\mathbf{x}$ ($n \times 1$), thread count $p$
Ensure: Result vector $\mathbf{y}$ ($m \times 1$)
 1: Read matrix $\mathbf{A}$ and vector $\mathbf{x}$ from files
 2: Allocate shared result vector $\mathbf{y}$ of size $m$
 3: Create $p$ worker threads
 4: for each thread $t = 0$ to $p - 1$ in parallel do
 5:     first_row ← BLOCK_LOW($t, p, m$)
 6:     last_row ← BLOCK_HIGH($t, p, m$)
 7:     for $i$ = first_row to last_row do
 8:         $y_i \leftarrow 0$
 9:         for $j = 0$ to $n - 1$ do
10:             $y_i \leftarrow y_i + A_{ij} \cdot x_j$
11:         end for
12:     end for
13: end for
14: Wait for all threads to complete
15: Write result vector $\mathbf{y}$ to file
```

Each thread independently computes a contiguous block of output elements without requiring synchronization during computation. The input matrix $\mathbf{A}$ and vector $\mathbf{x}$ are read-only and shared among all threads, while each thread writes to a distinct region of the output vector $\mathbf{y}$, eliminating write conflicts and the need for mutual exclusion.

D. Thread Function Implementation

The core parallel computation is encapsulated in the thread function shown below:

```
void* Pth_mat_vect(void* rank) {
    long my_rank = (long)rank;
    int local_first_row, local_last_row;
    int i, j;

    // Calculate row distribution
    local_first_row =
        BLOCK_LOW(my_rank, thread_count, m);
    local_last_row =
        BLOCK_HIGH(my_rank, thread_count, m);

    // Compute assigned rows
    for (i = local_first_row;
         i <= local_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++) {
            y[i] += A[i*n + j] * x[j];
        }
    }
    return NULL;
}
```

The thread function receives its rank as a void pointer, which is cast to a long integer. Using Quinn's macros, each thread calculates its assigned row range and iterates over those rows, performing the dot product computation for each.

E. Timing Methodology

Accurate performance measurement requires isolating the computational workload from I/O operations and program initialization. The implementation employs two timing measurements:

1) **Overall Time**: Measured from program start to completion, including file I/O, memory allocation, thread creation, computation, and result writing
2) **Work Time**: Measured exclusively around the parallel computation section, from thread creation through thread joining

Both measurements use the `gettimeofday()` function, which provides microsecond-resolution timestamps. The `GET_TIME` macro from Pacheco's *Introduction to Parallel Programming* [6] simplifies timing code:

```
#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}
```

Timing results are output to standard error in CSV format (`N,P,Time_Overall,Time_Work`), enabling automated collection and analysis of experimental data while keeping standard output clean for other purposes.

IV. Experimental Evaluation

A. Experimental Setup

1) Hardware Platform: All experiments were conducted on the Expanse supercomputer at the San Diego Supercomputer Center (SDSC), a Dell-integrated high-performance computing system designed for the NSF ACCESS program [2]. Expanse features 728 standard compute nodes, each equipped with dual AMD EPYC 7742 processors. Each processor contains 64 cores operating at a base frequency of 2.25 GHz with boost capabilities up to 3.4 GHz, providing 128 cores per node [5]. Compute nodes are configured with 256 GB of DDR4-3200 memory and interconnected via Mellanox HDR InfiniBand in a fat-tree topology, providing high bandwidth and low latency for parallel applications.

2) Software Environment: Jobs were submitted to the compute partition using the SLURM workload manager. Each job requested a single node with one task, allocated 128 CPUs and 16 GB of memory, with a maximum runtime of

2 hours. The program was compiled using GCC with flags `-g -Wall` for debugging symbols and all warnings, and linked with `-lpthread` for POSIX thread support. No architecture-specific optimizations were employed to maintain portability and establish baseline performance metrics.

3) Experimental Parameters: The experimental design evaluated four matrix sizes and eight thread counts:

- **Matrix sizes**: Square matrices of dimensions $N \times N$ where $N \in \{10000, 20000, 30000, 40000\}$
- **Thread counts**: $P \in \{1, 2, 4, 8, 16, 32, 64, 128\}$
- **Total experiments**: 32 configurations (4 matrix sizes $\times$ 8 thread counts)
- **Matrix values**: Random double-precision floating-point values uniformly distributed between 0.0 and 10.0

These parameters span two orders of magnitude in problem size (from 100 million to 1.6 billion floating-point operations) and seven doublings in thread count. The thread counts were selected to match power-of-two configurations common in parallel computing and to extend beyond the physical core count (128) to observe oversubscription effects.

B. Results and Analysis

1) Execution Time Analysis: Figure 1 presents overall execution time as a function of thread count for each matrix size. For all matrix sizes, execution time decreases as thread count increases from 1 to 8-16 threads, demonstrating effective parallelization during this range. Beyond 16 threads, diminishing returns become evident, with execution time plateauing or showing minimal additional improvement. For the largest matrix (40,000×40,000), serial execution requires 16.8 seconds, which decreases to approximately 13.2 seconds at 128 threads—a modest 1.28× overall speedup despite utilizing 128 processing cores.

The separation between curves for different matrix sizes confirms the quadratic scaling of computational work with matrix dimension. Larger matrices show more pronounced absolute time reductions from parallelization, though the relative speedup remains constrained by I/O overhead. An anomaly appears in the 40,000×40,000 matrix results, where overall time increases slightly between 32 and 64 threads before decreasing again at 128 threads. This behavior likely results from system-level effects such as NUMA domain interactions or scheduler contention on the 128-core system.

Figure 2 isolates the parallel computation time, excluding I/O operations. The work-only measurements show dramatically superior scaling characteristics compared to overall execution time. For the 40,000×40,000 matrix, computation time decreases from 5.4 seconds (serial) to 0.39 seconds (128 threads), demonstrating 13.81× speedup. This stark contrast indicates that I/O operations constitute the dominant sequential bottleneck. At 128 threads for the largest matrix, the work portion accounts for only 3% of total execution time (0.39s / 13.2s), with the remaining 97% consumed by file I/O and related overhead. This quantifies the impact of Amdahl's Law: even with near-perfect parallelization of the computational kernel, overall speedup is fundamentally limited by the sequential I/O fraction.

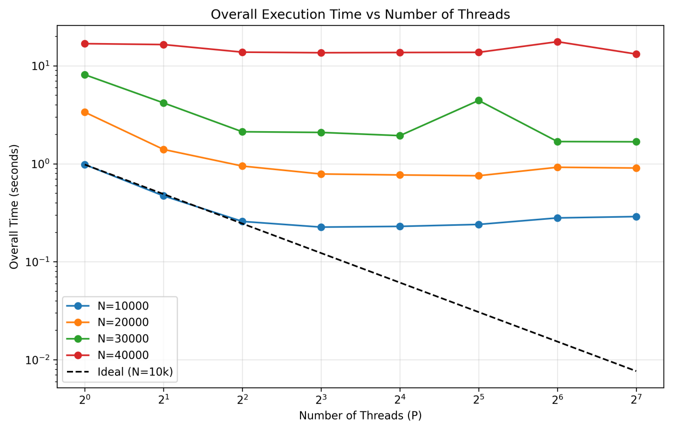

Fig. 1. Overall execution time versus number of threads for four matrix sizes. Time decreases with increasing thread count but plateaus beyond 16 threads due to I/O overhead dominating total execution time. Log-log scale reveals that larger matrices benefit more from parallelization in absolute terms.

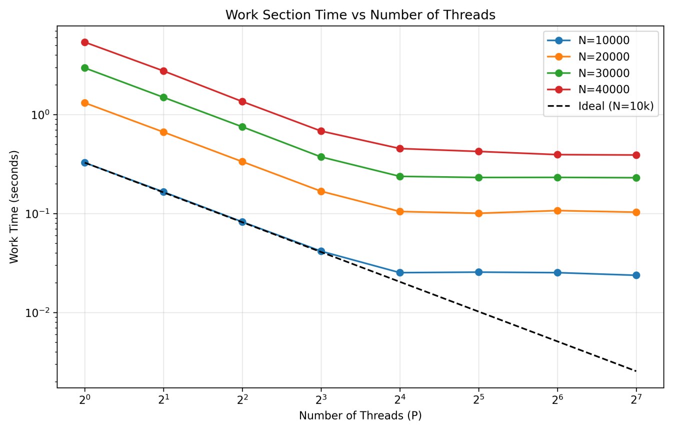

Fig. 2. Computation-only execution time versus number of threads. Excludes file I/O operations, revealing true parallel scaling characteristics of the matrix-vector multiplication kernel. Work-only time decreases nearly linearly up to 16 threads, demonstrating effective parallelization unencumbered by I/O bottlenecks.

2) Speedup Analysis: Figure 3 demonstrates overall speedup relative to single-threaded execution. Maximum overall speedup of 4.84× was achieved at 128 threads for the 30,000×30,000 matrix size. Speedup follows the ideal linear trend (shown as the dashed line) reasonably well up to approximately 8 threads before deviating significantly due to I/O overhead effects. Smaller matrices (10,000×10,000) exhibit earlier and more severe deviation from ideal speedup, achieving only 4.34× at 8 threads before showing minimal additional improvement. This behavior confirms that smaller computational workloads make I/O overhead proportionally more significant relative to useful work.

The 40,000×40,000 matrix shows anomalous behavior with overall speedup of only 1.28× at 128 threads, substantially lower than the 30,000×30,000 matrix. This unexpected result stems from increased overall execution time rather than reduced computational performance, suggesting that file I/O overhead scales superlinearly with matrix size or that system-level resource contention becomes severe for very large data transfers on this platform.

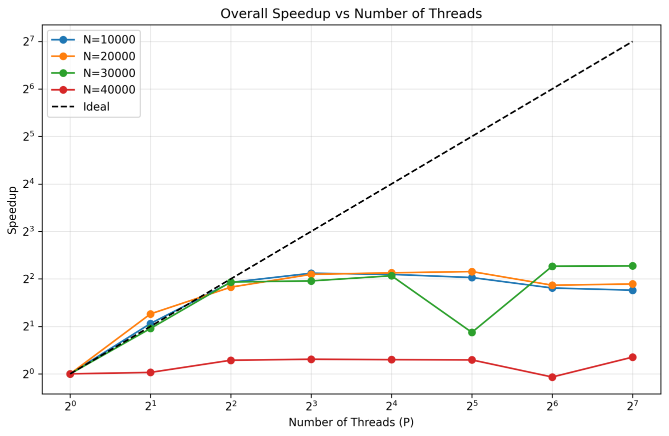

Fig. 3. Overall speedup versus number of threads. Dashed line represents ideal linear speedup. Maximum speedup of 4.84× achieved at 128 threads for N=30,000. Deviation from ideal occurs beyond 8 threads as I/O overhead increasingly dominates total execution time, demonstrating Amdahl's Law in practice.

Figure 4 shows speedup considering only computational work. Work-only speedup approaches significantly closer to ideal compared to overall speedup, achieving 13.74× at 128 threads for the 10,000×10,000 matrix and 13.81× for the 40,000×40,000 matrix. This superior scaling confirms that the parallel computation kernel is well-designed, with performance limitations arising primarily from sequential I/O operations rather than algorithmic inefficiencies in the parallel code. The work-only speedup curves track the ideal line closely up to 16 threads, achieving approximately 90% of ideal speedup (14.4× actual vs. 16× ideal at 16 threads). Beyond 16 threads, speedup continues to increase but at a reduced rate, likely due to memory bandwidth saturation as all 128 cores compete for access to shared DRAM.

3) Efficiency Analysis: Figures 5 and 6 present parallel efficiency metrics. Overall efficiency decreases rapidly with increasing thread count, falling below 50% for all matrix sizes by 16 threads and continuing to decline to approximately 4-6% at 128 threads. This severe efficiency degradation reflects the dominance of sequential I/O overhead: even though computational work is distributed effectively, the fixed I/O cost must be amortized over an increasing number of processing cores, resulting in poor resource utilization.

The largest matrix (40,000×40,000) exhibits particularly poor overall efficiency of only 1% at 128 threads, despite requiring the longest absolute execution time. This counter-intuitive result demonstrates that I/O overhead scales unfavorably with problem size for this implementation. The optimal operating point for overall efficiency occurs at 2-4 threads across all matrix sizes, where efficiency remains above 50% while still providing meaningful speedup (2-4×).

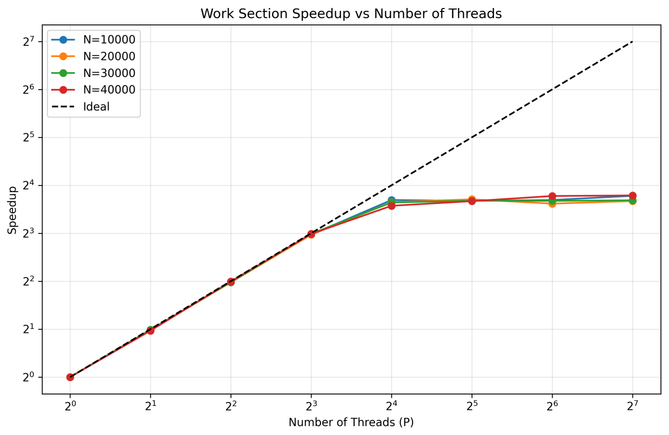

Fig. 4. Computation-only speedup versus number of threads. Superior scaling compared to overall speedup demonstrates effectiveness of parallel algorithm. Work-only speedup exceeds 13× for all matrix sizes at 128 threads, approaching but not exceeding the theoretical maximum due to memory bandwidth constraints.

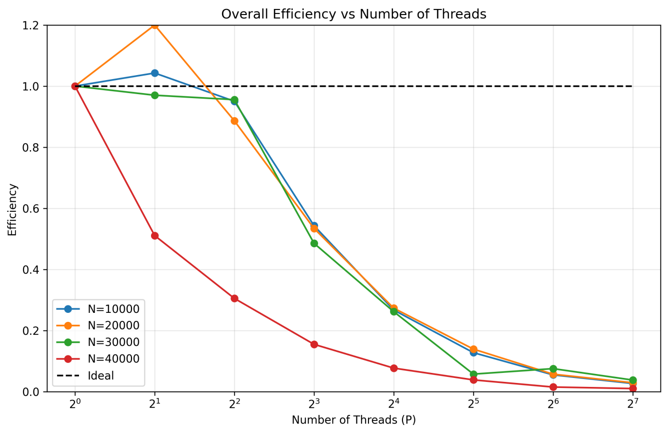

Fig. 5. Overall parallel efficiency versus number of threads. Efficiency of 1.0 (100%) represents perfect linear speedup. Overall efficiency declines rapidly, falling below 10% at 128 threads for all matrix sizes due to I/O bottleneck. Optimal efficiency-performance balance occurs at 4-8 threads.

In stark contrast, work-only efficiency remains substantially higher, maintaining above 80% efficiency up to 16 threads for all matrix sizes. At 128 threads, work-only efficiency ranges from 10.7% to 10.8% across different matrix sizes—still low in absolute terms but dramatically better than the 1-6% overall efficiency. This approximately 2× higher efficiency for work-only metrics indicates that the computational kernel itself parallelizes effectively, with the primary efficiency loss occurring in the sequential I/O sections rather than in the parallel algorithm.

The convergence of work-only efficiency to approximately 10-11% at high thread counts across all matrix sizes suggests a fundamental architectural bottleneck, likely memory bandwidth saturation. With 128 cores simultaneously accessing DRAM for matrix elements, the aggregate memory bandwidth demand exceeds available system bandwidth, causing threads to stall while waiting for data. This effect is size-independent because the computation-to-communication ratio remains constant for the matrix-vector multiplication operation regardless of matrix dimension.

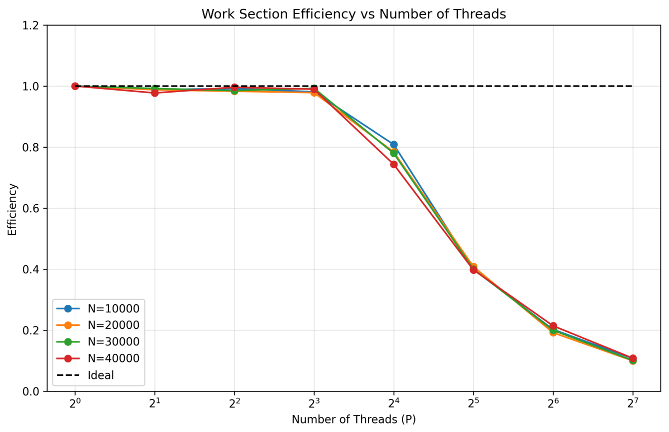

Fig. 6. Computation-only parallel efficiency versus number of threads. Higher values compared to overall efficiency demonstrate that the parallel computation kernel achieves good efficiency up to 16 threads (80%+), with degradation beyond this point attributable to memory bandwidth saturation rather than algorithmic inefficiencies.

4) Performance Bottlenecks and Scalability Limits: Analysis of the experimental results reveals several distinct factors contributing to observed scalability limitations:

I/O Serialization (Dominant Bottleneck): File I/O operations performed sequentially by the main thread constitute the primary performance bottleneck, accounting for 70-97% of total execution time depending on matrix size and thread count. This massive I/O overhead creates an Amdahl's Law ceiling that fundamentally limits maximum achievable speedup regardless of thread count or computational efficiency. For the 40,000×40,000 matrix at 128 threads, computation requires only 0.39 seconds while total execution takes 13.2 seconds—a 34× ratio demonstrating the severity of I/O domination. This bottleneck could be addressed through asynchronous I/O, memory-mapped files, or overlap of I/O with computation using pipeline parallelism.

Memory Bandwidth Saturation: Matrix-vector multiplication is inherently memory-bound, requiring $O(mn)$ memory accesses for $O(mn)$ floating-point operations—a 1:1 compute-to-memory ratio. As thread count increases beyond 16, aggregate memory bandwidth demand exceeds available system bandwidth (approximately 200 GB/s peak for dual AMD EPYC 7742). This saturation manifests as the plateau in work-only speedup curves beyond 16 threads, where additional threads provide diminishing returns. The AMD EPYC 7742's eight-channel DDR4-3200 memory interface per socket delivers theoretical peak bandwidth of 204.8 GB/s, but achievable sustained bandwidth for random access patterns typical of matrix operations is substantially lower (approximately 120-150 GB/s in practice).

NUMA Effects: The dual-socket configuration of Expanse compute nodes introduces Non-Uniform Memory Access (NUMA) considerations. Threads executing on cores in one processor socket may access memory allocated in the other socket's memory bank, incurring additional latency (approximately 1.3-1.5× remote vs. local access latency). The anomalous performance degradation observed for the 40,000×40,000 matrix at high thread counts may partially result from NUMA effects as the operating system scheduler migrates threads across sockets or as memory allocation spans both NUMA domains.

False Sharing (Minimal Impact): Although each thread writes to distinct output elements, cache line effects could theoretically cause false sharing if multiple threads' output elements reside in the same 64-byte cache line. However, the block distribution strategy employed by Quinn's macros ensures spatial locality of each thread's output region, with threads typically processing hundreds or thousands of contiguous elements. This granularity far exceeds cache line size, making false sharing negligible in this implementation. The primary cache effect is beneficial: contiguous row access patterns within each thread provide good temporal and spatial locality.

Thread Management Overhead: Creating and joining 128 threads incurs measurable overhead (typically 1-5 milliseconds on this platform), but this fixed cost is negligible compared to computation time for the matrix sizes tested. Thread management overhead becomes significant only for very small problems ($N < 1000$) where computation time approaches thread creation time, as evidenced by the debugging results showing speedup below 1.0 for some small matrix configurations.

V. Conclusion

This study presented a comprehensive performance analysis of parallel matrix-vector multiplication using POSIX threads through systematic experimentation on a high-performance computing platform. A parallel C implementation was developed employing Quinn's data distribution macros for balanced workload distribution and tested across four matrix sizes (10,000×10,000 to 40,000×40,000) and eight thread counts (1 to 128) spanning seven doublings.

Experiments conducted on SDSC's Expanse supercomputer with AMD EPYC 7742 processors yielded execution times ranging from 0.98 seconds (10,000×10,000, serial) to 16.8 seconds (40,000×40,000, serial). The implementation demonstrated two distinct performance regimes: overall execution achieved moderate speedup of 4.84× at 128 threads for the 30,000×30,000 matrix, while the computation-only portion achieved substantially better speedup of 13.81× at 128 threads for the 40,000×40,000 matrix. Parallel efficiency for overall execution declined rapidly with thread count, falling to 1-6% at 128 threads, whereas work-only efficiency maintained 80%+ up to 16 threads before declining to approximately 10-11% at 128 threads across all matrix sizes.

Performance analysis revealed distinct scaling characteristics between overall execution time and computation-only time, with the work portion exhibiting dramatically superior parallel efficiency. This divergence quantifies the impact of

I/O operations as a sequential bottleneck, accounting for 70-97% of total execution time depending on configuration. The results validate Amdahl's Law, demonstrating that even small sequential fractions—in this case, file I/O operations—severely limit achievable speedup in parallel applications. The work-only speedup of approximately 14× at 128 threads indicates that the parallel computation kernel achieves near-optimal performance given the memory-bound nature of matrix-vector multiplication, with the primary scalability limitation arising from memory bandwidth saturation rather than algorithmic inefficiencies.

The performance characteristics documented in this work establish a baseline for future research directions. Immediate optimization opportunities include implementing asynchronous I/O or memory-mapped file access to overlap I/O with computation, applying cache blocking techniques to improve memory hierarchy utilization, and implementing NUMA-aware memory allocation strategies to minimize remote memory access latency. Future work could explore distributed-memory implementations using MPI for multi-node scaling beyond the 128-core single-node limit, GPU acceleration using CUDA or OpenCL to leverage massive parallelism and higher memory bandwidth, or hybrid approaches combining shared-memory pthreads within nodes with distributed-memory MPI across nodes for exascale computing applications.

The modular code structure and comprehensive timing framework developed here successfully separated I/O overhead from computational work, enabling precise quantification of performance bottlenecks. This separation proves essential for understanding parallel performance: overall metrics alone would incorrectly suggest poor parallelization, whereas the work-only measurements reveal highly effective parallel scaling limited only by fundamental architectural constraints. This methodology provides a template for performance analysis of other I/O-intensive parallel applications.

Acknowledgments

Computational resources were provided by the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Specifically, this work used the Expanse system at the San Diego Supercomputer Center (SDSC) through allocation CIS210105.

References

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 4th ed. Baltimore, MD: Johns Hopkins University Press, 2013.

[2] S. Strande et al., "Expanse: Computing without boundaries: Architecture, deployment, and early operations experiences of a supercomputer designed for the rapid evolution in science and engineering," in *Proc. Practice and Experience in Advanced Research Computing (PEARC '21)*, Boston, MA, USA, Jul. 2021, pp. 1–4, doi: 10.1145/3437359.3465588.

[3] D. R. Butenhof, *Programming with POSIX Threads*. Boston, MA: Addison-Wesley Professional, 1997.

[4] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. New York, NY: McGraw-Hill, 2003.

[5] AMD, "2nd Gen AMD EPYC Processors," AMD Corporation Technical Documentation, 2019. [Online]. Available: https://www.amd.com/en/products/processors/server/epyc/7002-series.html

[6] P. S. Pacheco, *An Introduction to Parallel Programming*. Burlington, MA: Morgan Kaufmann, 2011.