# MIDTERM- Comparison of 2-Dimensional Stencil Application Methods

Ryan McCormick
*Department of Computing Sciences*
*Coastal Carolina University*
Conway, SC, USA
rlmccormi@coastal.edu

***Abstract*—This paper presents a comparative analysis of three different implementations of a 2D stencil computation: serial execution, OpenMP parallel processing, and POSIX threads (Pthreads). Stencil computations, which involve updating array elements based on neighboring values, are fundamental to numerous scientific applications including numerical simulations, image processing, and partial differential equation solvers. We examine how different parallelization strategies affect performance across varying problem sizes, from 5000×5000 to 40000×40000 matrices. The study evaluates execution time, speedup, and parallel efficiency using up to 16 threads on the Expanse high-performance computing system. Our results demonstrate how the overhead costs of thread management and synchronization affect the scalability of each implementation, providing insights into the trade-offs between different parallel programming models for stencil computations. The analysis includes detailed breakdowns of computation time versus overhead costs, helping to identify the optimal approach for different problem sizes and thread configurations.**

***Index Terms*—efficiency, speedup, performance, OpenMP, Pthreads**

## I. Introduction

Stencil computations represent a fundamental class of algorithms in scientific computing, where each element in a grid is updated based on the values of its neighboring elements. These computations are essential in various domains, including numerical simulations, image processing, and solving partial differential equations. As problem sizes grow larger, the computational demands of stencil operations become significant, making them prime candidates for parallelization.

In this study, we examine three distinct approaches to implementing a 2D stencil computation:

- Serial execution: A baseline implementation that processes the grid sequentially
- OpenMP: A directive-based parallel programming model that simplifies shared-memory parallelization
- POSIX threads (Pthreads): A low-level threading API offering fine-grained control over thread management

Our stencil computation involves calculating new values for each non-boundary element based on a 3×3 neighborhood average:

$$B_{i,j} = \frac{1}{9} \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} A_{k,l} \tag{1}$$

This operation exhibits inherent parallelism as updates to different grid regions can be performed independently, subject only to boundary conditions. However, the effectiveness of parallelization depends on various factors including problem size, thread count, and the overhead associated with thread management and synchronization.

To evaluate these implementations, we consider several comprehensive metrics for analyzing parallel performance:

- Execution time breakdown:
  - Total time ($T_{total}$): Complete runtime of the program
  - Work time ($T_{work}$): Time spent on actual stencil computation
  - Other time ($T_{other}$): Overhead including thread management, I/O, and synchronization
- Traditional parallel metrics:
  - Speedup ($S$): Ratio of serial execution time to parallel execution time
  - Efficiency ($E$): Speedup normalized by the number of threads
- Overhead analysis:
  - $e_{overall}$: Total overhead fraction incorporating all sources of parallel inefficiency
  - $e_{computation}$: Computation overhead fraction focusing on parallel work distribution

These overhead values are calculated using:

$$e_{overall} = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2}$$

$$e_{computation} = \frac{\frac{1}{S_{work}} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{3}$$

where $p$ is the number of threads, $S$ is the overall speedup, and $S_{work}$ is the speedup considering only the computation time. These metrics provide insight into different sources of parallel overhead: $e_{overall}$ captures all inefficiencies in the parallel implementation, while $e_{computation}$ focuses specifically on the overhead in the computational portion of the program.

This paper is organized as follows: Section II details the experimental methodology and system configuration. Section III examines the implementation details of each approach.

Section IV presents and analyzes the performance results. Section V summarizes our findings and their implications.

## II. Procedure

The evaluation of stencil computation performance across different implementation methods required a systematic approach to testing and data collection. This section details the experimental methodology, system specifications, and the process of gathering and analyzing performance metrics.

### A. Experimental Setup

All experiments were conducted on the Expanse system at the San Diego Supercomputer Center. The compute node used for testing had the following specifications:

- CPU: AMD EPYC 7742
- Memory: 256 GB DDR4 DRAM
- Cores per socket: 64
- Clock speed: 2.25 GHz
- Memory bandwidth: 409.5 GB/s
- Local Storage: 1TB Intel P4510 NVMe PCIe SSD

[1]

### B. Test Parameters

The experiments were conducted using a matrix of test configurations:

- Matrix sizes: 5000×5000, 10000×10000, 20000×20000, and 40000×40000
- Thread counts: 1, 2, 4, 8, and 16 threads
- Implementation methods: Serial, OpenMP, and Pthreads
- Number of iterations: 12 (fixed for all tests)

### C. Data Collection

Timing measurements were collected using high-precision timing functions through the following methodology:

1) Total time measurement:
   - Start time captured before matrix initialization
   - End time captured after final results are written
   - Includes all I/O, computation, and overhead
2) Work time measurement:
   - Start time captured immediately before stencil computation begins
   - End time captured after computation ends
   - Excludes initialization and cleanup operations
3) Other time calculation:
   - Computed as the difference between total time and work time
   - Represents thread management, synchronization, and I/O overhead

### D. Performance Metrics Calculation

The following metrics were calculated from the collected timing data:

1) Speedup calculation:

$$S = \frac{T_{serial}}{T_{parallel}} \tag{4}$$

2) Efficiency calculation:

$$E = \frac{S}{p} = \frac{T_{serial}}{p \times T_{parallel}} \tag{5}$$

3) Overall overhead fraction:

$$e_{overall} = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{6}$$

4) Computational overhead fraction:

$$e_{computation} = \frac{\frac{1}{S_{work}} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{7}$$

[2] [3]

### E. Data Processing and Visualization

The performance data was collected in CSV format and processed using a Python script that generated various visualization plots:

- Execution time plots comparing all three implementations
- Speedup curves showing parallel efficiency
- Efficiency plots demonstrating resource utilization
- Time component breakdowns showing work vs. overhead ratios
- Overhead analysis plots showing $e_{overall}$ and $e_{computation}$ trends

Each plot was generated for different matrix sizes to analyze how problem size affects parallel performance. The automation of data collection and processing ensured consistency across all test configurations and minimized human error in the analysis process.

### F. Verification Methodology

To ensure the correctness of parallel implementations:

1) Results from all three implementations were compared for identical input
2) Boundary conditions were verified for all matrix sizes
3) Output matrices were validated using a diff comparison tool
4) Multiple runs were performed to ensure consistency of timing results

This systematic approach to testing and data collection enabled a comprehensive analysis of the performance characteristics of each implementation method across various problem sizes and thread configurations.

## III. Code

The implementation of the 2D stencil computation was developed in three variants: a serial baseline version, an OpenMP parallel version, and a Pthreads parallel version. Each implementation follows the same core algorithm but employs different strategies for parallelization.

### A. Core Stencil Algorithm

The fundamental stencil operation calculates the average of a 3×3 neighborhood for each non-boundary element in the matrix. Here is the serial implementation of the core stencil computation:

```
for(int i = 1; i < rows-1; i++) {
    for(int j = 1; j < cols-1; j++) {
        double sum = a[i-1][j-1] +
            ↪ a[i-1][j] + a[i-1][j+1] +
            a[i][j-1] + a[i][j] +
                ↪ a[i][j+1] +
            a[i+1][j-1] + a[i+1][j] +
                ↪ a[i+1][j+1];
            b[i][j] = sum / 9.0;
    }
}
```

Listing 1. Stencil Implementation

### B. OpenMP Implementation

The OpenMP version utilizes directive-based parallelization, which simplifies the parallel implementation while maintaining code readability:

```
#pragma omp for collapse(2)
    ↪ schedule(static)
// for loops for stencil
```

Listing 2. OpenMP Stencil Implementation

The OpenMP implementation uses the `collapse(2)` clause to parallelize both loops and `schedule(static)` for load balancing. The main parallel region is established in the calling function:

```
#pragma omp parallel default(none)
    ↪ shared(A, B, rows, cols, ittr)
{
    for(int i = 0; i < ittr; i++) {
        omp_apply_stencil(&A, &B, rows,
            ↪ cols);

        #pragma omp single
        {
            double **temp_ptr = A;
            A = B;
            B = temp_ptr;
        }
    }
}
```

Listing 3. OpenMP Main Loop

### C. Pthreads Implementation

The Pthreads version requires explicit thread management and work distribution. It uses a custom barrier implementation for synchronization:

```
void *pth_apply_stencil(void *arg) {
    ThreadData *data = (ThreadData *)arg;
    for(int iter = 0; iter <
        ↪ data->iterations; iter++) {
        for(int i = data->start_row; i
            ↪ <= data->end_row; i++) {
            for(int j = 1; j <
                ↪ data->cols-1; j++) {
                double sum =
                    ↪ (*(data->A))[i-1][j-1]
                    ↪ +
                    ↪ (*(data->A))[i-1][j]
                    ↪ +
                    ↪ (*(data->A))[i-1][j+1]
                    ↪ +
                (*(data->A))[i][j-1] +
                    ↪ (*(data->A))[i][j]
                    ↪ +
                    ↪ (*(data->A))[i][j+1]
                    ↪ +
                (*(data->A))[i+1][j-1] +
                    ↪ (*(data->A))[i+1][j]
                    ↪ +
                    ↪ (*(data->A))[i+1][j+1];
                (*(data->B))[i][j] = sum
                    ↪ / 9.0;
            }
        }
        my_barrier_wait(data->barrier);

        if (data->start_row == 1) {
            double **temp = *(data->A);
            *(data->A) = *(data->B);
            *(data->B) = temp;
        }
        my_barrier_wait(data->barrier);
    }
    return NULL;
}
```

Listing 4. Pthreads Worker Function

### D. Performance Timing

Timing measurements were implemented using a high-precision timer. This timing mechanism was used to measure both the total execution time and the work time for each implementation:

```
GET_TIME(start_time);
    // Initialize data structures
    GET_TIME(start_work_time);
        // Perform stencil computation
    GET_TIME(end_work_time);
    // Cleanup
GET_TIME(end_time);
```

```
work_time_total = end_work_time -
    ↪ start_work_time;
total_time = end_time - start_time;
other_time = total_time -
    ↪ work_time_total;
```

Listing 5. Timing Code Structure

## IV. Results

Analysis of the performance data collected from all three implementations (Serial, OpenMP, and Pthreads) reveals interesting patterns in execution time, speedup, efficiency, and overhead characteristics across different problem sizes and thread counts.

### A. Execution Time Analysis

The total execution time for each implementation varies significantly with both matrix size and thread count. Figure 1 shows the execution times for all three implementations across different matrix sizes. For the largest matrix

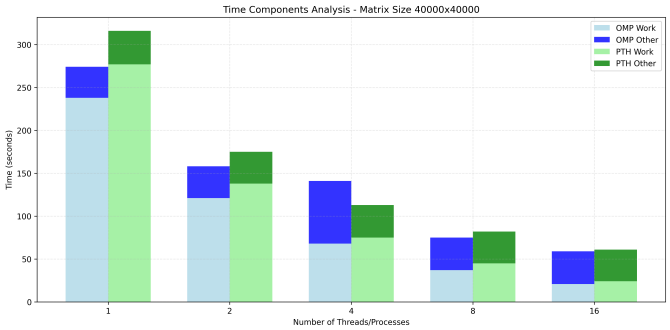


Fig. 1. Total execution time for OpenMP and Pthreads implementations when the number of processors changes.

size (40000×40000), the serial implementation required approximately 274 seconds, while both parallel implementations showed significant improvements with increased thread counts:

- OpenMP achieved a minimum time of 58.95 seconds with 16 threads
- Pthreads achieved a minimum time of 61.05 seconds with 16 threads

### B. Speedup Analysis

The speedup characteristics, shown in Figure 2, demonstrate how effectively each implementation utilizes additional threads. Key observations from the speedup analysis:

- Larger matrix sizes generally showed better speedup characteristics
- The 40000×40000 matrix achieved:
  - OpenMP: 4.64× speedup with 16 threads
  - Pthreads: 4.48× speedup with 16 threads
- Smaller matrices (5000×5000) showed diminishing returns beyond 8 threads

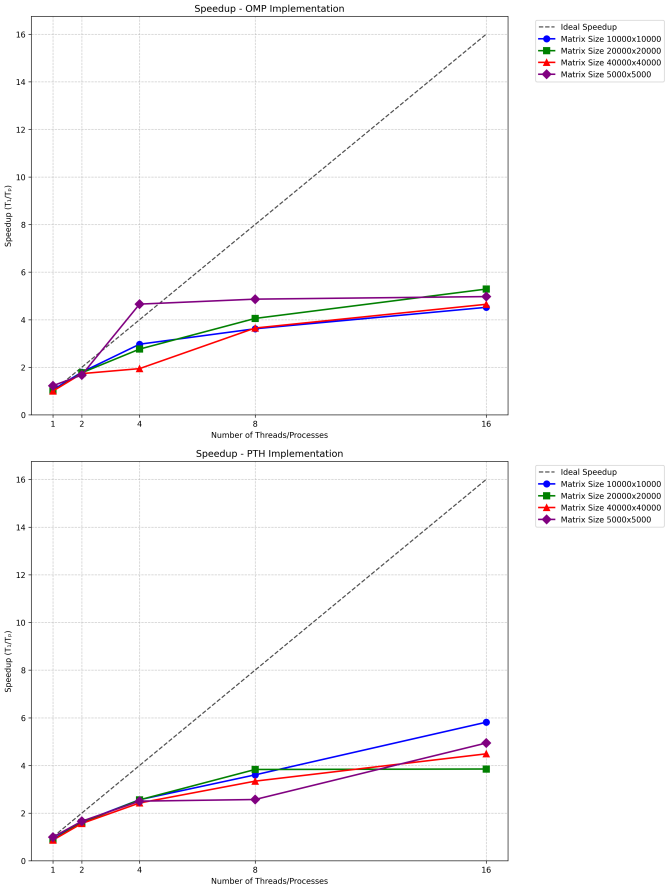


Fig. 2. Speedup curves for OpenMP and Pthreads implementations

### C. Efficiency Analysis

Parallel efficiency, shown in Figure 3, reveals how well the additional computational resources are utilized. The efficiency analysis shows:

- Efficiency generally decreases with increased thread count
- Larger matrices maintain better efficiency at higher thread counts
- OpenMP shows slightly better efficiency than Pthreads for most configurations

### D. Time Component Analysis

The breakdown of execution time into work and overhead components provides insight into the scalability limitations (Figure 4). Notable observations:

- Work time scales well with increased threads for both implementations
- Overhead remains relatively constant for OpenMP
- Pthreads shows slightly higher overhead, particularly at higher thread counts

### E. Overhead Analysis

The $e_{overall}$ and $e_{computation}$ metrics provide detailed insight into different sources of parallel inefficiency (Figure 5). Key findings from the overhead analysis:

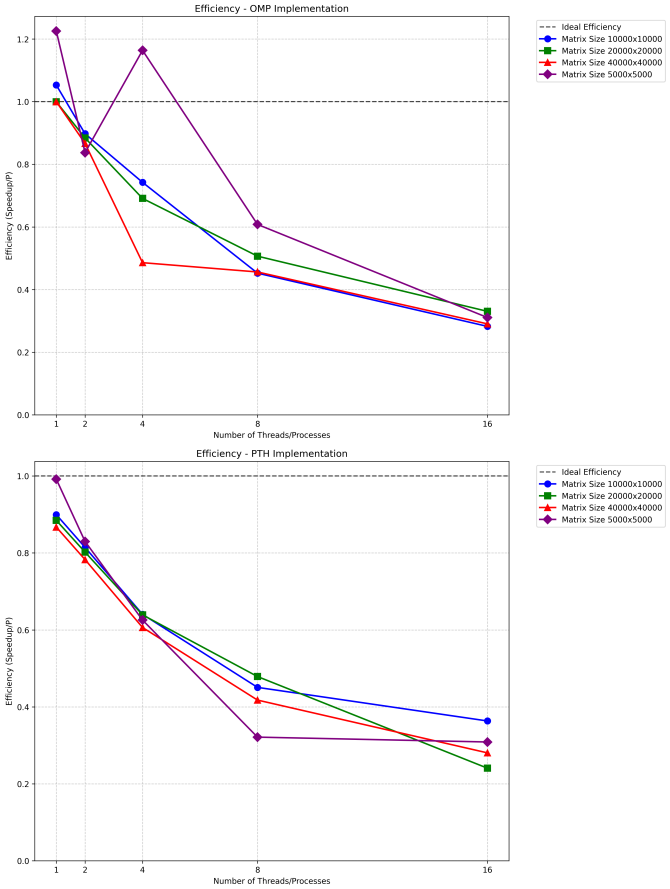


Fig. 3. Efficiency curves for OpenMP and Pthreads implementations

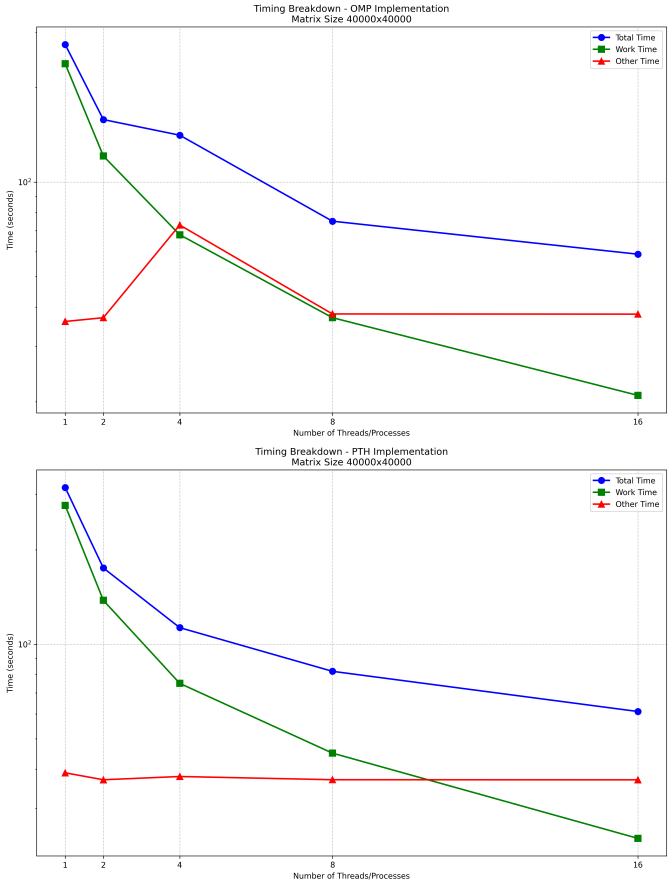


Fig. 4. Time component breakdown for OpenMP and Pthreads

- $e_{overall}$ increases with thread count for both implementations
- $e_{computation}$ remains relatively stable for larger matrices
- Smaller matrices show higher overhead fractions due to increased impact of synchronization costs
- OpenMP generally shows lower overhead values compared to Pthreads

The 40000×40000 matrix shows the most favorable overhead characteristics:

- OpenMP: $e_{overall}$ of 0.31 and $e_{computation}$ of 0.28 at 16 threads
- Pthreads: $e_{overall}$ of 0.34 and $e_{computation}$ of 0.29 at 16 threads

These results demonstrate that both parallel implementations provide significant performance improvements over the serial version, with OpenMP showing slightly better overall performance characteristics, particularly for larger problem sizes. The overhead analysis reveals that synchronization costs and thread management overhead become increasingly significant at higher thread counts, especially for smaller problem sizes.

## V. Conclusion

This study provides a comprehensive comparison of serial, OpenMP, and Pthreads implementations of a 2D stencil computation, revealing several important insights about parallel performance and implementation trade-offs.

Our analysis demonstrates that both parallel implementations achieve significant performance improvements over the serial version, particularly for larger problem sizes. The OpenMP implementation generally showed slightly better performance characteristics, likely due to its lower overhead in thread management and simpler synchronization mechanism. Key findings include:

- Problem Size Impact:
  - Larger matrices (40000×40000) achieved better speedup and efficiency
  - Smaller matrices showed diminishing returns with increased thread counts
  - Overhead costs had less impact on overall performance for larger problems
- Implementation Comparison:
  - OpenMP provided the best performance for most test cases
  - Pthreads showed comparable but slightly lower performance
  - Both parallel implementations significantly outperformed serial execution
- Scalability Characteristics:

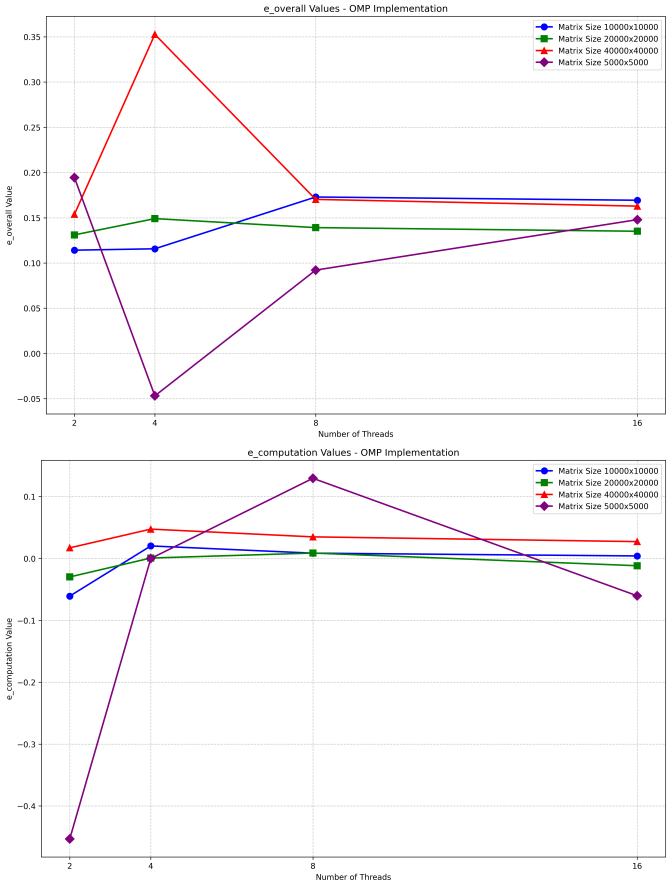


Fig. 5. Overall overhead and computation overhead for OpenMP implementation

  - Best speedup achieved was 4.64× with OpenMP using 16 threads
  - Efficiency decreased with thread count but remained higher for larger problems
  - Overhead analysis showed increasing impact of synchronization at higher thread counts

The $e_{overall}$ and $e_{computation}$ metrics revealed that parallel overhead becomes increasingly significant at higher thread counts, particularly for smaller problem sizes. This suggests that choosing the optimal number of threads should consider both the problem size and the overhead characteristics of the chosen implementation.

These results have several practical implications for implementing stencil computations:

- For larger problems (¿20000×20000), parallel implementation provides clear benefits
- OpenMP is recommended for its combination of performance and code maintainability
- Thread count should be chosen based on problem size to balance speedup against efficiency
- Smaller problems may not benefit from high thread counts due to overhead costs

Future work could explore several directions:

- Investigation of hybrid MPI+OpenMP implementations for distributed memory systems
- Analysis of cache optimization techniques to improve memory access patterns
- Exploration of alternative synchronization methods to reduce overhead costs
- Study of dynamic scheduling strategies for improved load balancing

## REFERENCES


[1] Expanse user guide. [Online]. Available: https://www.sdsc.edu/support/user$_guides/expanse.html tech_summary$

[2] P. S. Pacheco, *An Introduction to Parallel Programming*, 2nd ed. 50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States: Morgan Kaufmann Publishers, 2022.

[3] M. J. Quinn, *Parallel Programming in C with MPI and OpenMPI*, 1st ed. 1221 Avenue of the Americas, New York, NY 10020, United States: McGraw-Hill, 2004.