P-Thread and OpenMP Implementation of a 9-Point Stencil Operation

Michael Dandrea
Department of Computing Sciences
Coastal Carolina University
Conway, SC, USA
mdandrea@coastal.edu

***Abstract*—This project implements and analyzes parallel versions of a 9-point stencil operation from the previos project using Pthreads and OpenMP. The stencil operation was implemented on matrices with varying problem sizes (5K × 5K to 40K × 40K) and thread counts (1-16 threads). Both implementations were evaluated on the Expanse supercomputer to compare their performance, including execution time, speedup, efficiency, and experimental serial fraction.**

I. Introduction

A. Project Objectives

This project implements and analyzes a 9-point stencil computation using two different parallel programming approaches: Pthreads and OpenMP. The main objectives are:

- Implement the stencil operation using both Pthreads and OpenMP
- Compare performance between the two implementations
- Analyze scalability with different matrix sizes and thread counts
- Understand the trade-offs between implementation complexity and performance

B. Methodology

The implementations are tested on the Expanse supercomputer with matrix sizes ranging from 5,000 × 5,000 to 40,000 × 40,000 elements, using 1 to 16 threads. Performance metrics including execution time, speedup, efficiency, and experimental serial fraction are measured and analyzed to compare the effectiveness of each approach.

C. Implementation Approaches

The key challenge in parallelizing the stencil operation was dividing the work efficiently among threads while maintaining correct boundary conditions and synchronization. Both parallel implementations share a similar strategy but differ in their approach to thread management.

For the Pthread version, thread management was implemented explicitly. Each thread is assigned a portion of the grid rows using block decomposition. This was achieved using the BLOCK_LOW and BLOCK_HIGH macros to calculate the starting and ending rows for each thread:

```
start_row = BLOCK_LOW(thread_id, num_threads,
total_rows);
end_row = BLOCK_HIGH(thread_id, num_threads,
total_rows);
```

A barrier synchronization was necessary after each iteration to ensure all threads completed their computations before moving to the next iteration. Two arrays (current and temporary) are used to store the grid states, swapping them after each iteration to avoid data conflicts.

The OpenMP version simplifies this process by using directive-based parallelization. The main computation loop is parallelized using:

```
#pragma omp parallel for
for (int i = 1; i < rows - 1; i++) {
    for (int j = 1; j < cols - 1; j++) {
        // stencil computation
    }
}
```

This directive automatically handles the work distribution and synchronization that we implemented manually in the Pthread version. Boundary conditions are updated after each iteration in both implementations to maintain the left and right walls at 1.0 and the top and bottom walls at 0.0.

The remainder of this report is organized as follows: Section 2 provides background information, Section 3 details the implementations, Section 4 presents the results and analysis, and Section 5 concludes with our findings.

II. The 9-Point Stencil Operation

A. Computational Pattern

The 9-point stencil updates each grid point using its current value and eight neighbors:

NW	N	NE
W	C	E
SW	S	SE

Fig. 1. 9-point stencil pattern showing center point (C) and its eight neighbors

The new value is calculated as:

$$value = \frac{C + N + S + E + W + NE + NW + SE + SW}{9}$$

B. Implementation Approaches

Both parallel implementations share key characteristics:

- Two-array approach to avoid data conflicts
- Special handling of boundary conditions
- Thread synchronization after each iteration
- Block decomposition for work distribution

C. Boundary Conditions

The implementation maintains specific boundary values:

- Left and right walls fixed at 1.0
- Top and bottom walls fixed at 0.0
- Interior points updated through stencil computation

III. Pthread Implementation

A. Work Distribution

In the Pthread implementation, each thread is assigned a specific chunk of rows in the matrix using a technique called block decomposition. This way, every thread knows exactly which rows it's responsible for. We calculate the starting and ending row for each thread based on its ID and the total number of threads:

```
int start_row = BLOCK_LOW(thread_id,
global_num_threads, global_rows);
int end_row = BLOCK_HIGH(thread_id,
global_num_threads, global_rows);

// Adjust for boundaries
if (start_row == 0)
    start_row = 1;
if (end_row == global_rows - 1)
    end_row = global_rows - 2;
```

This approach ensures each thread works on a fair portion of the matrix without overlapping with others. We adjust the boundaries here so that threads don't accidentally overwrite the boundary rows, focusing only on the interior cells.

B. Stencil Computation

Once a thread has its assigned rows, it performs the stencil computation—a 9-point averaging operation where each cell is updated based on the values of its eight neighboring cells. This averaging smooths out the matrix data, with each thread updating only its assigned section:

```
for (int i = start_row; i <= end_row; i++) {
    for (int j = 1; j < global_cols - 1; j++) {
        global_temp[i][j] = (global_data[i-1][j-1]+
        global_data[i-1][j]+global_data[i-1][j+1]+
        global_data[i][j+1]+global_data[i+1][j+1]+
        global_data[i+1][j]+global_data[i+1][j-1]+
        global_data[i][j-1]+global_data[i][j])/9.0;
    }
}
```

In this nested loop, each thread moves through its assigned rows and columns. For every interior cell, it calculates the new value by averaging the cell itself and its eight neighbors. The new values are saved in a temporary matrix, `global_temp`, so that we don't overwrite the data before the whole computation is complete.

C. Thread Synchronization

Since all threads need to finish computing their assigned rows before moving on, we use a custom barrier function to sync them up. This barrier acts as a checkpoint, ensuring no thread moves forward until all threads reach this point:

```
typedef struct pthread_barrier {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    volatile uint32_t flag;
    size_t count;
    size_t num;
} my_barrier_t;

int my_pthread_barrier_init(my_barrier_t *bar,
int attr, int num);
int my_pthread_barrier_wait(my_barrier_t *bar);
int my_pthread_barrier_destroy(my_barrier_t *bar
);
```

This barrier was provided to ensure it works smoothly on macOS, which can sometimes handle standard Pthread barriers differently. To avoid name conflicts with existing Pthread functions, we prefixed the function names with `my_`. Additionally, we added a `my_pthread_barrier_destroy` function to clean up resources and prevent memory leaks. The barrier ensures all threads wait until everyone has completed their work for the current iteration before proceeding to the next.

D. Boundary Updates

After each iteration, we need to make sure that the boundary conditions stay intact. For example, we may want the boundary cells to hold specific values (like wall conditions in a physical simulation). To handle this, we update the boundary rows and columns separately after the main computation:

```
// Update left and right walls
for (int i = 0; i < global_rows; i++) {
    global_temp[i][0] = global_data[i][0];
    global_temp[i][global_cols-1] =
        global_data[i][global_cols-1];
}
```

In this example, we update the left and right walls by copying values directly from `global_data` to `global_temp`. This way, boundary cells remain consistent with the conditions we want, while the interior cells reflect the new, averaged values.

IV. OpenMP Implementation

A. Work Distribution

OpenMP makes parallelizing loops easy with simple compiler directives, which take care of thread management for us. For the main computation, we use `#pragma omp parallel for`, which automatically distributes rows across available threads. Each thread then calculates the average of neighboring cells for every interior cell in its assigned rows, updating `temp[i][j]` based on the values in `data`. Here, we start from `i = 1` to `rows - 1` and `j = 1` to `cols - 1` to focus on interior cells and leave out the boundaries:

```
#pragma omp parallel for
for (int i = 1; i < rows - 1; i++) {
    for (int j = 1; j < cols - 1; j++) {
        temp[i][j] = (data[i-1][j-1] +
        data[i-1][j] + data[i-1][j+1] +
        data[i][j+1] + data[i+1][j+1] +
        data[i+1][j] + data[i+1][j-1] +
        data[i][j+1] + data[i][j]) / 9.0;
    }
}
```

B. Boundary Updates

To keep boundary values intact, we update them separately. While the main calculation only handles the interior cells, the edges need to be processed to keep their values consistent (e.g., simulating wall conditions). Here, OpenMP lets us run these updates in parallel too, so each boundary cell can be updated quickly:

```
#pragma omp parallel for
for (int i = 0; i < rows; i++) {
    temp[i][0] = data[i][0];// Left wall
    temp[i][cols-1] = data[i][cols-1];// Right
wall
}

#pragma omp parallel for
for (int j = 0; j < cols; j++) {
    temp[0][j] = data[0][j];// Top wall
    temp[rows-1][j] = data[rows-1][j];// Bottom
wall
}
```

C. Array Management

At the end of each iteration, we need to swap `data` and `temp` so that the updated values in `temp` are ready for the next round. This is a quick operation because we're just swapping pointers, not the entire arrays:

```
// Swap arrays for next iteration
double **swap = data;
data = temp;
temp = swap;
```

D. Key Differences from Pthreads

Using OpenMP here has some big advantages over Pthreads:

- **Automatic Work Distribution**: OpenMP handles dividing work among threads, so we don't have to manually assign specific rows or cells to each thread.
- **Built-in Synchronization**: OpenMP automatically syncs threads at the end of a `parallel for` loop, making sure all threads are finished before moving forward. This removes the need for manual barriers or joining threads.
- **Simpler Syntax**: OpenMP's `#pragma` directives make it easy to add, remove, or adjust parallel code without a lot of extra lines, which keeps things readable.
- **Thread Management**: OpenMP handles thread creation and cleanup for us, so we don't have to manage that ourselves. This not only reduces code but also lowers the risk of threading errors.

Overall, OpenMP provides a simpler, faster way to parallelize the stencil operation without the detailed setup that Pthreads requires. It's a good fit when you want to focus more on the algorithm and less on managing threads.

V. Gathering Data and Running on Expanse

All data was gathered on the Expanse Supercomputer's compute node. The job was queued using a SLURM script, which ran two python scripts. These python scripts ran both the Pthread version and the OpenMP version of the stencil across all combinations of matrix sizes and thread counts. The scripts also outputted this data to specific csv files depending on which implementation was being tested.

Then, the csv files were taken from Expanse and put onto a local machine that had two additional python scripts to plot the findings as graphs. These scripts produce 20 total plots: 10 for Pthread results and 10 for OpenMP results. These plots include speedup, timing, efficiency and serial fraction based on overall time vs thread count, as well as computation time vs thread count.

VI. Experimentally Determined Serial Fraction

The Karp-Flatt metric, also known as the experimentally determined serial fraction (e), provides valuable insight into parallel program performance beyond traditional speedup and efficiency measurements. This metric helps identify the fundamental limitations of parallel performance by quantifying the portion of a program that remains inherently serial.

A. The Karp-Flatt Equation

The experimentally determined serial fraction is calculated using the formula:

$$e = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}}$$

where:

- e is the experimentally determined serial fraction
- S is the observed speedup
- p is the number of processors (threads in our case)

B. Significance

The Karp-Flatt metric helps distinguish between two primary causes of speedup degradation:

- Inherently serial portions of the program that cannot be parallelized
- Parallel overhead from operations such as:
  - Thread creation and management
  - Synchronization barriers
  - Communication between threads
  - Memory access conflicts

C. Interpretation

When analyzing the Karp-Flatt metric:

- A constant e across different numbers of processors suggests that sequential code is the primary limitation
- An increasing e with more processors indicates that parallel overhead is the dominant factor
- Lower values of e indicate better parallel efficiency

D. Application to Stencil Computation

For our stencil implementation, we calculate two different serial fractions:

- $e_{overall}$ using total execution time, which includes I/O and setup operations
- $e_{computation}$ using only the computation time, focusing on the parallel stencil operation itself

This dual analysis allows us to:

- Isolate the efficiency of the core parallel algorithm
- Identify the impact of necessary serial operations
- Understand the overall scalability limitations of the implementation

The Karp-Flatt metric serves as a valuable tool for analyzing parallel performance, providing insights that complement traditional speedup and efficiency measurements. It helps identify whether performance limitations stem from inherently serial code sections or from the overhead of parallel execution.

VII. Pthread Implementation Results

A. Speedup Analysis

The speedup characteristics of the Pthread implementation were measured in two ways: overall speedup, which includes all program execution time, and computation speedup, which focuses solely on the stencil computation portion.

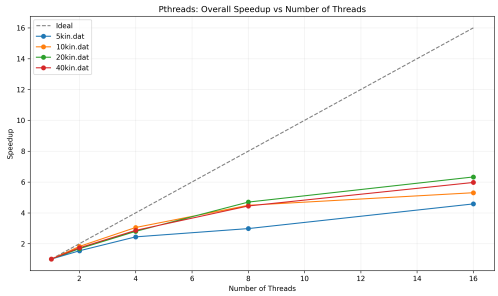

Fig. 2. Overall Speedup vs Number of Threads

1) Overall Speedup: The overall speedup results (Figure 2) demonstrate significant deviation from ideal linear speedup:

- All matrix sizes show sub-linear scaling, with maximum speedups ranging from 4.5x to 6.5x at 16 threads
- Larger matrices (20K, 40K) achieve better speedup, reaching approximately 6.3x with 16 threads
- Medium-sized matrices (10K) show intermediate performance, achieving about 5.5x speedup
- The smallest matrix (5K) exhibits the poorest scaling, reaching only 4.5x speedup
- Performance begins to plateau after 8 threads for all problem sizes

2) Computation Speedup: The computation-only speedup results (Figure 3) show markedly different characteristics:

- Near-ideal scaling is observed for 10K and 20K matrices up to 16 threads, achieving approximately 15.5x speedup
- The 5K matrix shows good scaling but falls short of ideal at higher thread counts, reaching 14.2x speedup

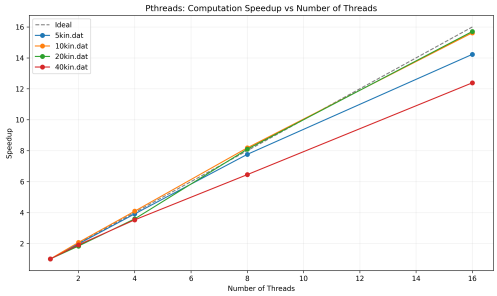

Fig. 3. Computation Speedup vs Number of Threads

- The 40K matrix demonstrates the poorest computation speedup, achieving only 12.4x at 16 threads
- All problem sizes maintain relatively linear scaling up to 16 threads, unlike the overall speedup
- Medium-sized matrices (10K, 20K) actually slightly exceed ideal speedup at some thread counts

3) Speedup Comparison: The differences between overall and computation speedup reveals a few things:

- The difference between overall and computation speedup (e.g., 6.3x vs 15.5x for 20K) shows significant impact from non-computational overhead
- I/O operations and thread management overhead significantly limit overall program scalability
- The stencil computation parallelizes extremely well, as shown by the near-ideal computation speedup

These results indicate that while the stencil computation itself parallelizes very well, the overall program performance is limited by non-computational factors such as I/O operations, thread management overhead, and synchronization costs.

B. Efficiency Analysis

The parallel efficiency of the Pthread implementation was evaluated through two metrics: overall efficiency and computational efficiency. These measurements provide insight into how effectively the implementation utilizes additional threads.

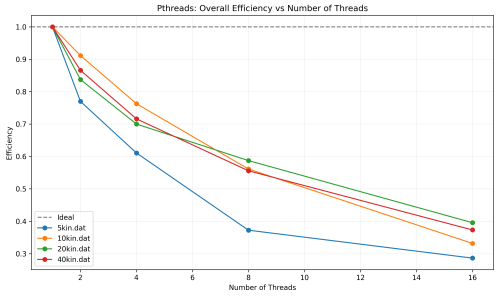

Fig. 4. Overall Efficiency vs Number of Threads

1) Overall Efficiency: The overall efficiency results (Figure 4) demonstrate a consistent decrease in efficiency as thread count increases:

- At 2 threads, initial efficiencies range from 0.77 (5K) to 0.91 (10K)

- All matrix sizes show steep efficiency decline with increasing thread count
- By 16 threads, efficiencies drop to:
  - about 0.3 for 5K matrix (poorest scaling)
  - about 0.35 for 10K matrix
  - 0.40 for 20K matrix (best scaling)
  - about 0.37 for 40K matrix
- Larger matrices (20K, 40K) maintain better efficiency up to 8 threads
- The efficiency curves show steeper decline in the 2-8 thread range compared to 8-16 threads

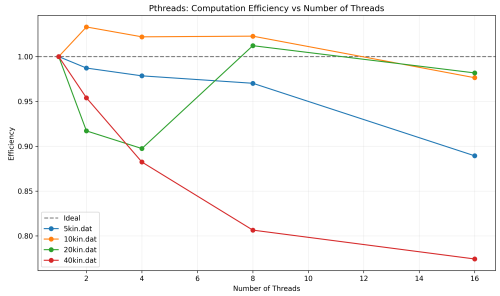

Fig. 5. Computation Efficiency vs Number of Threads

2) Computation Efficiency: The computation-only efficiency results (Figure 5) show markedly different characteristics:

- 10K matrix has above ideal efficiency up to 8 threads
- 20K matrix shows interesting behavior:
  - Initial drop to 0.90 at 4 threads
  - Increases to above ideal efficiency at 8 threads
  - Gradual decline to 0.97 at 16 threads
- 5K matrix maintains relatively high efficiency but shows steady decline
- 40K matrix has the worst computational efficiency, dropping to 0.77 at 16 threads
- Most matrix sizes maintain above 0.90 efficiency even at 16 threads

3) Efficiency Comparison: The contrast between overall and computational efficiency reveals important implementation characteristics:

- Computational efficiency is significantly higher than overall efficiency across all matrix sizes
- Medium-sized matrices (10K, 20K) show the best computational efficiency
- The largest matrix (40K) shows very bad efficiency degradation
- Non-computational overheads significantly impact overall efficiency while the core computation remains highly efficient

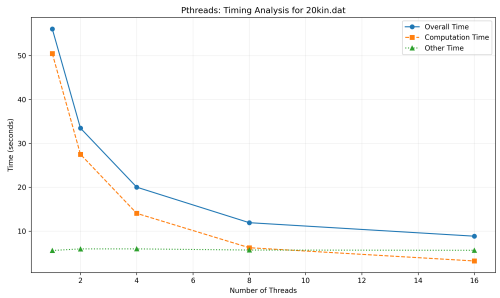

Fig. 6. Timing Analysis for 20k Matrix

C. Timing Analysis: 20K × 20K Matrix

Since the 20k matrix looks to be the ”best” overall, we’ll look at the results of its timing. Three timing components were

measured: overall execution time, computation time, and other time (overhead), as shown by Figure 6.

1) Sequential Performance Baseline: At single-thread execution:

- Overall execution time: 55 seconds
- Computation time: 50 seconds
- Other time: 5 seconds
- Overhead represents approximately 9% of total execution time

2) Parallel Scaling Behavior: As thread count increases, the timing component change:

Computation Time:

- Drops sharply from 50 to 27 seconds with 2 threads
- Further reduces to 14 seconds with 4 threads
- Continues declining to approximately 3 seconds with 16 threads
- Shows consistent improvement with additional threads

Other Time (Overhead):

- Remains consistent around 5-6 seconds across all thread counts
- Shows slight variation but nothing massive
- Becomes a larger proportion of total time as computation time decreases

Overall Time:

- Follows a pattern similar to computation time but with an offset due to constant overhead
- Decreases from 55 to 33 seconds with 2 threads
- Reaches about 20 seconds with 4 threads
- Diminishing returns start showing after 8 threads
- Reaches a final execution time of about 9 seconds with 16 threads

3) Performance Analysis: The timing results for the 20K matrix reveal several key characteristics:

- The computational portion scales well, showing nearly linear reduction with thread count
- The constant overhead time becomes increasingly significant at higher thread counts:
  - Represents 9% of total time with 1 thread
  - Increases to 15% with 4 threads
  - Becomes approximately 55% of total time with 16 threads
- The intersection of computation and other time occurs around 8 threads, marking a transition point where overhead becomes as significant as computation

D. Serial Fraction Analysis

The experimentally determined serial fraction provides insight into the fundamental limitations of the parallel implementation. Analysis of both overall and computation-specific serial fractions reveals distinct patterns across different problem sizes.

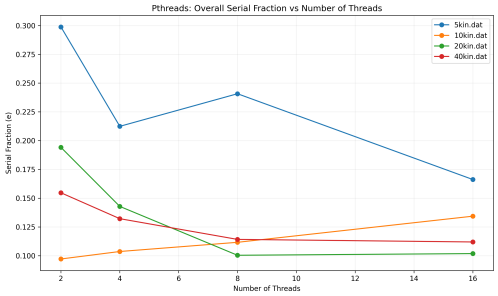

Fig. 7. Overall Serial Fraction vs Number of Threads

1) Overall Serial Fraction: The overall serial fraction results (Figure 7) show varied behavior across matrix sizes:

5K Matrix:

- Exhibits highest initial serial fraction (0.30)
- Shows significant initial decrease to 0.21 at 4 threads
- Slight increase between 4-8 threads
- Gradual decline to 0.17 by 16 threads
- Consistently shows highest serial fraction among all sizes

Medium Sizes (10K, 20K):

- 20K starts at 0.19 and decreases steadily to 0.10
- 10K shows unique behavior, starting low (0.10) and remaining relatively constant
- Both converge to similar values (0.10-0.13) at higher thread counts

40K Matrix:

- Initial serial fraction of 0.15
- Steady decrease to 0.11 at 16 threads
- Shows most stable behavior among all sizes

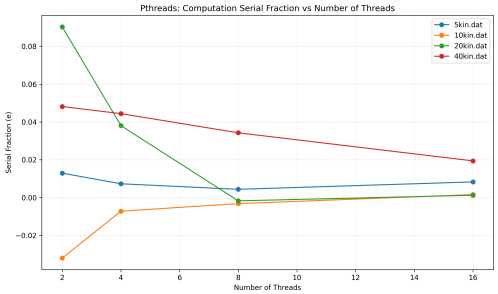

Fig. 8. Computation Serial Fraction vs Number of Threads

2) Computation Serial Fraction: The computation-only serial fraction (Figure 8) demonstrates markedly different characteristics:

Unusual Patterns:

- 10K matrix shows negative serial fraction at low thread counts, indicating superlinear speedup
- 20K matrix exhibits highest initial serial fraction (0.09) but rapid improvement
- Most sizes converge to near-zero serial fraction at higher thread counts

Size-Specific Behavior:

- 5K matrix maintains low, stable serial fraction (0.01-0.02)
- 40K shows gradual improvement from 0.05 to 0.02
- 20K demonstrates most dramatic improvement, from 0.09 to near-zero

3) Comparative Analysis: The contrast between overall and computation serial fractions reveals several key insights:

- Overall serial fraction is consistently higher than computation serial fraction
- The gap between overall and computation metrics indicates significant non-computational overhead
- Medium-sized matrices (10K, 20K) show optimal computational parallelization
- Smaller matrices suffer from higher overall serial fraction due to overhead dominance
- Larger matrices show more stable but higher serial fractions due to memory effects

4) Analysis of Negative Serial Fraction: A particularly interesting phenomenon observed in the computational serial fraction results is the occurrence of negative values, most notably in the 10K matrix case. This unusual behavior warrants detailed examination as it provides insights into the implementation's performance characteristics.

Theoretical Implications:

- A negative serial fraction implies superlinear speedup, where $S(p) > p$
- This means the parallel implementation is performing better than theoretically expected
- Traditional Amdahl's Law suggests this should be impossible, as speedup should be limited by $\frac{1}{s}$, where s is the serial fraction

VIII. OpenMP Implementation Results

A. Speedup Analysis

The OpenMP implementation demonstrates distinct speedup characteristics for both computational and overall performance across different problem sizes.

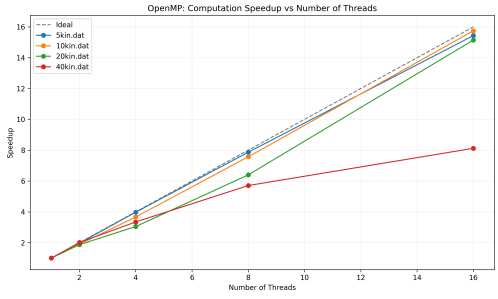

Fig. 9. Overall Speedup vs Number of Threads

1) Computational Speedup: The computational speedup results (Figure 9) show remarkable scaling characteristics:

- **Smaller Matrices (5K, 10K):**
  - Achieve near-ideal scaling up to 16 threads
  - 10K matrix shows best performance, reaching approximately 15.8x speedup
  - 5K matrix follows closely with about 15.2x speedup
- **Medium Size (20K):**
  - Shows slightly lower but still impressive scaling
  - Achieves approximately 15x speedup at 16 threads
  - Maintains consistent scaling throughout thread count range
- **Large Size (40K):**
  - Demonstrates significantly different behavior
  - Limited to approximately 8x speedup at 16 threads
  - Shows early deviation from ideal scaling

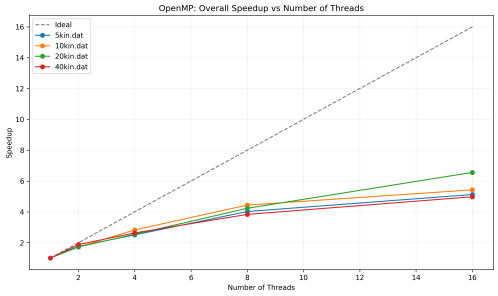

Fig. 10. Overall Speedup vs Number of Threads

2) Overall Speedup: The overall speedup results (Figure 10) reveal more constrained scaling:

- **General Characteristics:**
  - All matrix sizes show sub-linear scaling
  - Maximum speedup limited to 5-6.5x range at 16 threads
  - Initial scaling similar across all sizes up to 4 threads
- **Size-Specific Performance:**
  - 20K matrix achieves best overall speedup (6.5x at 16 threads)
  - 10K and 5K matrices show similar patterns, reaching 5.5x
  - 40K matrix demonstrates lowest overall speedup at 5x

B. OpenMP Efficiency Analysis

The efficiency measurements for the OpenMP implementation reveal distinct patterns between computational and overall performance across different problem sizes.

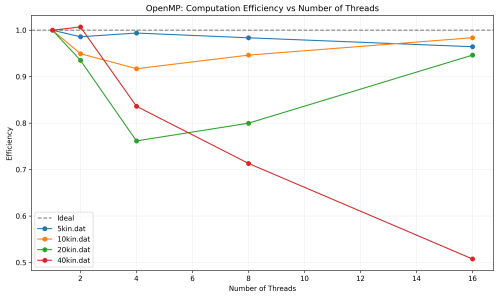

Fig. 11. Computation Efficiency vs Number of Threads

1) Computational Efficiency: The computational efficiency results (Figure 11) show varying behaviors across matrix sizes:

Small Matrix (5K):

- Maintains consistently high efficiency (¿0.95) up to 8 threads
- Gradual decline to approximately 0.95 at 16 threads
- Most stable efficiency curve among all sizes

Medium Matrices (10K, 20K):

- 10K shows initial drop to 0.92 at 4 threads, then recovers to 0.98 at 16 threads
- 20K exhibits interesting behavior:
  - Sharp initial decline to 0.76 at 4 threads
  - Strong recovery trend thereafter
  - Reaches 0.94 efficiency at 16 threads

Large Matrix (40K):

- Shows initial high efficiency (1.0) at 2 threads
- Steady degradation with increasing thread count
- Drops significantly to 0.51 at 16 threads
- Demonstrates worst scaling among all sizes

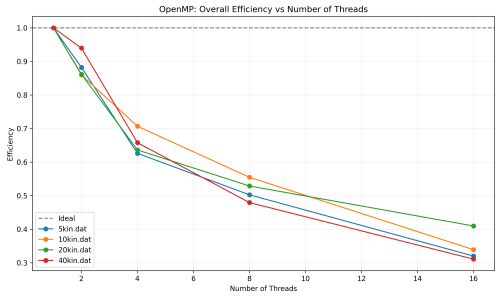

Fig. 12. Overall Efficiency vs Number of Threads

2) Overall Efficiency: The overall efficiency results (Figure 12) demonstrate more consistent degradation patterns:

General Trends:

- All matrix sizes show monotonic efficiency decline
- Initial efficiencies near ideal (0.85-0.94) at 2 threads
- Convergence to 0.31-0.41 range at 16 threads

Size-Specific Behavior:

- 20K matrix maintains best efficiency at high thread counts (0.41 at 16 threads)
- 40K matrix shows steepest efficiency decline
- 5K and 10K matrices follow similar degradation patterns
- Efficiency curves show steeper decline in 2-8 thread range compared to 8-16 threads

C. Timing Analysis: 20K × 20K Matrix (OpenMP)

The timing analysis for the 20K matrix using OpenMP reveals detailed performance characteristics across different thread counts. The analysis examines three key timing components: overall execution time, computation time, and other time (overhead).

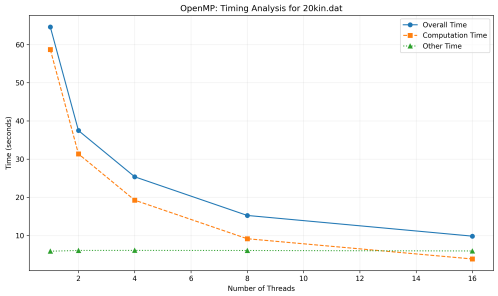

Fig. 13. Timing Analysis for 20k Matrix

1) Sequential Baseline: At single-thread execution:

- Overall execution time: 65 seconds
- Computation time: 58 seconds
- Other time: 6 seconds
- Initial overhead represents approximately 9.2% of total execution time

2) Scaling Behavior: **Computation Time (Orange Line):**

- Dramatic initial reduction from 58 to 31 seconds with 2 threads
- Further decrease to 19 seconds with 4 threads
- Continues improving to approximately 3 seconds at 16 threads
- Shows consistent improvement across thread counts

Other Time (Green Line):

- Remains remarkably consistent around 5-6 seconds
- Shows minimal variation across thread counts
- Demonstrates independence from parallelization
- Becomes proportionally more significant at higher thread counts

Overall Time (Blue Line):

- Reduces from 65 to 37 seconds with 2 threads
- Reaches approximately 25 seconds with 4 threads
- Continues to improve to about 10 seconds with 16 threads
- Shows diminishing returns after 8 threads

3) Performance Analysis: The timing relationships reveal several key characteristics:

- **Overhead Impact:**
  - Initial overhead ratio: 9.2% at 1 thread
  - Increases to 20% at 4 threads
  - Reaches approximately 50% at 16 threads
  - Becomes increasingly dominant factor at higher thread counts
- **Scaling Efficiency:**
  - Computation time scales nearly linearly up to 8 threads
  - Overhead remains constant regardless of thread count
  - Intersection of computation and other time occurs near 16 threads
  - Overall scaling limited by fixed overhead costs
- **OpenMP-Specific Characteristics:**

- Efficient thread management shown by smooth scaling curve
- Consistent overhead suggests effective runtime system
- Good load balancing indicated by predictable performance improvements
- Minimal thread management overhead compared to problem size

D. OpenMP Serial Fraction Analysis

The experimentally determined serial fraction for the OpenMP implementation shows distinct patterns between computational and overall measurements, providing insights into the implementation's fundamental performance limitations.

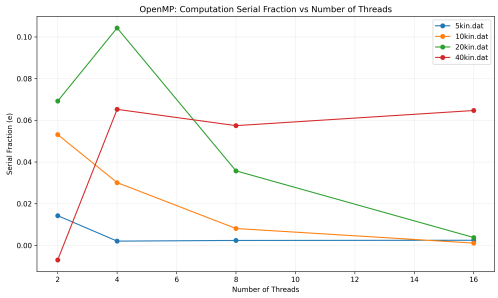

Fig. 14. Computation Serial Fraction vs Number of Threads

1) Computational Serial Fraction: The computational serial fraction (Figure 14) exhibits varied behavior across matrix sizes:

Small Matrix (5K):

- Starts at 0.015 with 2 threads
- Quickly decreases to near zero by 4 threads
- Maintains consistently low serial fraction through 16 threads
- Shows most stable behavior among all sizes

Medium Matrices:

- 10K matrix:
  - Starts at 0.05 and steadily decreases
  - Converges to near zero at higher thread counts
- 20K matrix:
  - Shows distinctive peak (0.105) at 4 threads
  - Demonstrates sharp improvement after peak
  - Converges to near zero by 16 threads

Large Matrix (40K):

- Exhibits unusual initial behavior with negative value at 2 threads
- Jumps to 0.065 at 4 threads
- Maintains relatively stable higher serial fraction (0.06) thereafter

2) Overall Serial Fraction: The overall serial fraction (Figure 2) shows more pronounced effects:

General Characteristics:

- All sizes show peak serial fraction around 4 threads
- Values range from 0.06 to 0.20 across different configurations
- Tendency toward stabilization at higher thread counts

Size-Specific Behavior:

- 5K matrix:
  - Peaks at 0.20 with 4 threads
  - Stabilizes around 0.14 at higher thread counts
- 20K matrix:
  - Shows steady decrease after 4-thread peak
  - Achieves lowest final serial fraction (0.095)
  - Demonstrates best scaling behavior
- 40K matrix:
  - Shows sharp increase from 2 to 4 threads
  - Maintains relatively stable higher values
  - Indicates potential memory-related limitations

3) Analysis of Findings: The serial fraction patterns reveal several key insights:

- **Implementation Characteristics:**
  - OpenMP runtime overhead is most significant at 4 threads
  - Computational portion shows excellent parallelization potential
  - Non-computational overhead dominates overall serial fraction
- **Problem Size Effects:**
  - Medium-sized matrices show optimal overall scaling
  - Larger matrices face increased memory-related serialization
  - Smaller matrices more affected by runtime overhead
- **Performance Implications:**
  - Core computation achieves near-ideal parallelization
  - Overall performance limited by system-level factors
  - Optimal thread count depends on problem size

IX. Conclusion

This study implemented and analyzed parallel versions of a 9-point stencil operation using both POSIX threads and OpenMP. Through extensive testing on the Expanse supercomputer across various problem sizes (5K × 5K to 40K × 40K) and thread counts (1-16), we gained significant insights into the performance characteristics and trade-offs of both parallel programming paradigms.

A. Key Findings

Performance Characteristics:

- Both implementations achieved significant speedup over serial execution, with computational speedups reaching approximately 15x with 16 threads
- Medium-sized matrices (10K, 20K) demonstrated optimal performance characteristics across both implementations
- Overall speedup was consistently lower than computational speedup due to I/O and thread management overhead

- The 20K matrix size emerged as the sweet spot, balancing parallel efficiency with memory hierarchy effects

Implementation Comparisons:

- OpenMP provided simpler implementation with comparable performance to Pthreads
- Pthread implementation offered finer control over thread management and synchronization
- Both implementations showed similar scaling patterns, with performance plateauing after 8 threads
- OpenMP demonstrated more consistent efficiency across different problem sizes

Scaling Limitations:

- Non-computational overhead remained relatively constant across thread counts
- Serial fraction analysis revealed increasing impact of overhead at higher thread counts
- Memory access patterns and cache effects significantly influenced performance
- Problem size played crucial role in determining optimal thread count

B. Implementation Trade-offs

Pthread Advantages:

- Fine-grained control over thread behavior
- Custom synchronization mechanisms
- Direct management of work distribution
- Potential for optimization at thread level

OpenMP Advantages:

- Simpler code structure and maintenance
- Automatic work distribution
- Built-in synchronization mechanisms
- Lower development complexity

C. Performance Implications

The results demonstrate that both implementations effectively parallelize the stencil computation, but face common limitations:

- Fixed overhead costs become increasingly significant at higher thread counts
- Memory access patterns and cache utilization affect scalability
- Problem size significantly influences optimal thread configuration
- System-level constraints ultimately limit achievable speedup