Comparison of 2-Dimensional Stencil Application Methods

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

Devin Guo
Department of Computing Sciences
Coastal Carolina University
Conway, SC, USA
ddguo@coastal.edu

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

Abstract—**This paper presents a comprehensive evaluation of parallel implementations of a 2D 9-point stencil computation, comparing four distinct approaches: POSIX threads (Pthreads), OpenMP, Message Passing Interface (MPI), and a hybrid MPI+OpenMP implementation. The stencil computation, which updates array elements based on neighboring values, is 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, evaluating execution time, speedup, and parallel efficiency using up to 16 processors on the Expanse high-performance computing system.**

Our analysis reveals distinct performance characteristics for each implementation method. The shared-memory approaches (Pthreads and OpenMP) demonstrated excellent performance for single-node execution, with computational speedups reaching approximately 15x with 16 threads. The pure MPI implementation showed strong scaling across distributed memory, achieving up to 8.29x computational speedup with 16 processes for the largest matrix size, though with increased communication overhead. The hybrid MPI+OpenMP implementation offered the best balance of performance and scalability, particularly effective for larger problem sizes where it could leverage both inter-node and intra-node parallelism.

The study provides detailed breakdowns of computation time versus overhead costs across all implementations, helping to identify the optimal approach for different problem sizes and thread configurations. Our results demonstrate how overhead costs from thread management, synchronization, and inter-process communication affect the scalability of each implementation, providing insights into the trade-offs between different parallel programming models for stencil computations.

Index Terms—**parallel computing, stencil computation, OpenMP, POSIX threads, MPI, hybrid parallelization, performance analysis, high-performance computing, parallel efficiency, distributed computing**

I. Introduction

In high-performance computing, stencil operations represent a fundamental class of algorithms essential for scientific simulations, image processing, and numerical analysis. The 9-point stencil, in particular, is widely used for its balance of computational accuracy and complexity, making it an ideal candidate for studying different parallel programming paradigms. As problem sizes grow and computational demands increase, understanding the trade-offs between various parallelization strategies becomes crucial for optimal performance.

A. Background and Motivation

Stencil computations involve updating array elements based on neighboring values, following a fixed pattern. In a 9-point stencil, each element is updated using its eight immediate neighbors and itself, creating a computational pattern that exhibits both spatial locality and data dependencies. These characteristics make stencil operations interesting from a parallelization perspective, as they require careful consideration of data sharing, synchronization, and communication strategies.

Modern high-performance computing systems often feature complex architectures combining multiple nodes (distributed memory) with multi-core processors (shared memory) within each node. This heterogeneous environment presents opportunities and challenges for parallel programming, leading to our investigation of four distinct implementation approaches:

- **POSIX Threads (Pthreads):** A low-level threading API offering fine-grained control over thread management and synchronization
- **OpenMP:** A directive-based parallel programming model that simplifies shared-memory parallelization
- **Message Passing Interface (MPI):** A standardized message-passing system for distributed memory parallelism
- **Hybrid MPI+OpenMP:** A combined approach leveraging both distributed and shared memory parallelization

B. Project Objectives

The main objectives of this study are:

- Implement and analyze the 9-point stencil using four different parallel programming approaches
- Compare performance characteristics across varying problem sizes and processor counts
- Evaluate the impact of communication overhead, synchronization costs, and memory access patterns
- Identify optimal implementation strategies for different computational scenarios
- Understand the scalability limitations and trade-offs of each approach

C. Implementation Overview

Each implementation approach presents unique challenges and opportunities:

Shared Memory Implementations:

- *Pthreads:* Requires explicit thread management and synchronization but offers maximum control
- *OpenMP:* Provides high-level abstractions for parallelization with simpler code structure

Distributed Memory Implementation:

- *Pure MPI:* Enables scaling across multiple nodes but introduces communication overhead
- *Domain decomposition:* Matrix rows distributed across processes with halo exchange

Hybrid Implementation:

- *MPI+OpenMP:* Combines distributed memory scaling with shared memory efficiency
- *Two-level parallelism:* Process-level distribution with thread-level parallelization

D. Experimental Methodology

Our experiments were conducted on the Expanse supercomputer at the San Diego Supercomputer Center, using matrix sizes ranging from 5000×5000 to 40000×40000 elements. We tested various configurations of threads and processes, measuring:

- Overall execution time
- Computation time
- Communication and synchronization overhead
- Speedup and efficiency metrics
- Experimentally determined serial fraction

E. Paper Organization

The remainder of this paper is organized as follows: Section 2 details the experimental procedure and system configuration. Section 3 examines the implementation details of each approach. Section 4 presents and analyzes the performance results. Section 5 concludes with our findings and recommendations. If you would like to see all the outputs, please visit the Appendix.

This comprehensive study provides insights into the effectiveness of different parallel programming models for stencil computations, helping developers choose the most appropriate approach based on their specific requirements and available computing resources.

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, testing parameters, and the process of gathering and analyzing performance metrics across all implementations.

A. Experimental Setup

All experiments were conducted on the Expanse system at the San Diego Supercomputer Center. The compute nodes 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
- Network: Mellanox HDR-100 InfiniBand

[1]

For the distributed memory tests (MPI and Hybrid), we utilized multiple compute nodes, each containing the above specifications. The nodes were interconnected using Mellanox HDR-100 InfiniBand, providing high-bandwidth, low-latency communication between processes.

B. Implementation Configurations

The experiments were conducted using four distinct implementation approaches:

- **Serial**: Baseline implementation for performance comparison
- **Shared Memory**:
  - Pthreads: 1, 2, 4, 8, and 16 threads
  - OpenMP: 1, 2, 4, 8, and 16 threads
- **Distributed Memory**:
  - Pure MPI: 1, 2, 4, 8, and 16 processes
- **Hybrid**:
  - MPI+OpenMP: Combinations of processes (1-16) and threads (1-128)
  - Process-thread combinations tested: 1×1-128, 2×1-128, 4×1-128, 8×1-128, 16×1-128

[2] [3]

C. Test Parameters

The experiments were conducted using a matrix of test configurations:

- Matrix sizes: 5000×5000, 10000×10000, 20000×20000, and 40000×40000
- Number of iterations: 12 (fixed for all tests)
- Debug levels: 0 (minimal output), 1 (detailed timing), 2 (full matrix output)

D. Data Collection Methodology

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) Communication time measurement (MPI and Hybrid only):
   - Time spent in MPI communication routines

- Includes halo exchange operations
- Measured separately from other overhead

4) Other time calculation:
   - Computed as the difference between total time and (work time + communication time)
   - Represents thread management, synchronization, and I/O overhead

E. Performance Metrics

The following metrics were calculated from the collected timing data:

1) Speedup calculation:

$$S = \frac{T_{serial}}{T_{parallel}} \tag{1}$$

2) Efficiency calculation:

$$E = \frac{S}{p} = \frac{T_{serial}}{p \times T_{parallel}} \tag{2}$$

3) Overall overhead fraction:

$$e_{overall} = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{3}$$

4) Computational overhead fraction:

$$e_{computation} = \frac{\frac{1}{S_{work}} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{4}$$

5) Communication efficiency (MPI and Hybrid):

$$E_{comm} = \frac{T_{computation}}{T_{computation} + T_{communication}} \tag{5}$$

F. Data Processing and Visualization

Performance data was collected in CSV format and processed using Python scripts that generated various visualization plots:

- Execution time plots comparing all four implementations
- Speedup curves showing parallel efficiency
- Efficiency plots demonstrating resource utilization
- Time component breakdowns showing work vs. overhead ratios
- Communication overhead analysis for distributed implementations
- 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.

G. Verification Methodology

To ensure the correctness of all implementations:

1) Results from all 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
5) Halo region exchanges were verified for distributed implementations
6) Process-thread coordination was validated for hybrid implementation

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 parallel configurations.

III. Code

The implementation of the 2D stencil computation was developed in five variants: a serial baseline version, OpenMP, Pthreads, pure MPI, and a hybrid MPI+OpenMP 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:

```c
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. Core Stencil Implementation

B. OpenMP Implementation

The OpenMP version utilizes directive-based parallelization for simplicity and maintainability:

```c
#pragma omp parallel
{
    for(int i = 0; i < iterations; i++) {
        #pragma omp for collapse(2)
            ↪ schedule(static)
        for(int i = 1; i < rows-1; i++) {
            for(int j = 1; j < cols-1;
                ↪ j++) {
                 // stencil computation
```

```
      }
    }

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

Listing 2. OpenMP Implementation

C. Pthreads Implementation

The Pthreads version implements explicit thread management and work distribution:

```
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++) {
        // stencil computation
      }
    }
    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 3. Pthreads Worker Function

D. MPI Implementation

The MPI version distributes matrix rows across processes with halo exchange:

```
void stencil2DMPI(double **subs, double
 ↪ **subs1,
  MPI_Datatype dtype, int m, int n,
     ↪ MPI_Comm comm) {
  int id, p;
  MPI_Comm_size(comm, &p);
  MPI_Comm_rank(comm, &id);

  // Calculate local rows including
     ↪ halo regions
  int halo_top = (id == 0) ? 0 : 1;
  int halo_bottom = (id == p - 1) ? 0
     ↪ : 1;
  int local_rows = BLOCK_SIZE(id, p,
     ↪ m) +
    halo_top + halo_bottom;

  // Perform halo exchange
  MPI_Request requests[4];
  int req_count = 0;

  // Send/receive boundary rows
  if (id > 0) {
    MPI_Isend(subs[1], n,
       ↪ MPI_DOUBLE, id-1, 0,
      comm,
         ↪ &requests[req_count++]);
    MPI_Irecv(subs[0], n,
       ↪ MPI_DOUBLE, id-1, 0,
      comm,
         ↪ &requests[req_count++]);
  }
  if (id < p-1) {
    MPI_Isend(subs[local_rows-2], n,
       ↪ MPI_DOUBLE,
      id+1, 0, comm,
         ↪ &requests[req_count++]);
    MPI_Irecv(subs[local_rows-1], n,
       ↪ MPI_DOUBLE,
      id+1, 0, comm,
         ↪ &requests[req_count++]);
  }

  // Compute interior points while
     ↪ communication happens
  for(int i = 2; i < local_rows-2;
     ↪ i++) {
    for(int j = 1; j < n-1; j++) {
      // stencil computation for
         ↪ interior points
    }
  }

  // Wait for communication to complete
  MPI_Waitall(req_count, requests,
     ↪ MPI_STATUSES_IGNORE);

  // Compute boundary points
  for(int i = 1; i < local_rows-1;
     ↪ i++) {
    if(i == 1 || i == local_rows-2) {
      for(int j = 1; j < n-1; j++)
         ↪ {
        // stencil computation
           ↪ for boundary points
      }
    }
    }
}
```

Listing 4. MPI Implementation

E. Hybrid MPI+OpenMP Implementation

The hybrid implementation combines MPI for inter-node communication with OpenMP for intra-node parallelization:

```
void stencil2D_MPI_OMP(double **subs,
    ↪ double **subs1,
    MPI_Datatype dtype, int m, int n,
        ↪ MPI_Comm comm) {
    int id, p;
    MPI_Comm_rank(comm, &id);
    MPI_Comm_size(comm, &p);

    // Calculate local rows with halos
    int halo_top = (id == 0) ? 0 : 1;
    int halo_bottom = (id == p - 1) ? 0
        ↪ : 1;
    int local_rows = BLOCK_SIZE(id, p,
        ↪ m) +
        halo_top + halo_bottom;

    // Perform halo exchange
    exchange_row_striped_values(&subs,
        ↪ dtype, m, n, comm);

    // OpenMP parallel region for
        ↪ computation
    #pragma omp parallel
    {
        // Thread teams work on local
            ↪ rows
        #pragma omp for
        for(int i = 1; i < local_rows-1;
            ↪ i++) {
            for(int j = 1; j < n-1; j++)
                ↪ {
                // stencil computation
            }
        }
    }
}
```

Listing 5. Hybrid MPI+OpenMP Implementation

F. Performance Timing

High-precision timing measurements were implemented for all versions:

```
// Start overall timing
GET_TIME(overall_start);

// Initialize data structures
GET_TIME(start_work_time);

// Perform stencil computation
GET_TIME(end_work_time);

// For MPI versions, measure
    ↪ communication
GET_TIME(start_comm_time);

// Perform halo exchange
GET_TIME(end_comm_time);

// Cleanup
GET_TIME(end_time);

// Calculate timing components
work_time = end_work_time -
    ↪ start_work_time;
comm_time = end_comm_time -
    ↪ start_comm_time;
other_time = total_time - (work_time +
    ↪ comm_time);
```

Listing 6. Timing Implementation

The hybrid implementation required careful coordination between MPI processes and OpenMP threads:

- MPI operations performed outside OpenMP parallel regions
- Thread-safe MPI initialization using MPI_Init_thread
- Proper nesting of OpenMP directives within MPI processes
- Efficient workload distribution across both processes and threads

Key implementation considerations for the distributed versions included:

- Efficient halo region exchange using non-blocking MPI communication
- Overlap of computation and communication where possible
- Process grid creation and topology management
- Load balancing across processes and threads
- Memory management for distributed arrays

G. Heat Transfer: Visualization of Stencil Matrix Evolution

Heat transfer analysis is fundamental across numerous scientific and engineering applications, including:

- Thermal management in electronic devices and processors
- Building energy efficiency and HVAC system design
- Materials processing and manufacturing
- Climate and weather modeling
- Aerospace thermal protection systems
- Nuclear reactor cooling systems
- Biomedical applications such as thermal therapy

Understanding and accurately modeling heat diffusion is crucial for:

- Predicting system behavior under various thermal conditions

- Optimizing design parameters for thermal efficiency
- Ensuring safety in high-temperature applications
- Developing cooling strategies for critical systems
- Validating thermal management solutions

Figures 1, 2, 3, and 4 show the evolution of the matrix values over the course of the stencil computations, illustrating the diffusion or smoothing effect achieved by the 9-point stencil operation. The images are of a 100 by 100 matrix over 2000 iterations.

The visualization demonstrates how the 9-point stencil computation effectively models heat diffusion in a 2D plane. In this simulation:

- The colors represent temperature values, with warmer colors (red) indicating higher temperatures and cooler colors (blue) indicating lower temperatures
- The initial state (Figure 1) shows distinct regions of high and low temperatures, representing localized heat sources and sinks
- As iterations progress (Figures 2 and 3), we observe the gradual diffusion of heat across the matrix, with sharp temperature gradients becoming smoother
- The final state (Figure 4) demonstrates the system approaching thermal equilibrium, where temperature differences have been largely averaged out through the diffusion process

This visualization serves multiple purposes:

- Validates the correctness of our parallel implementations by showing physically realistic heat diffusion behavior
- Illustrates the practical application of stencil computations in physics simulations
- Demonstrates how local interactions (9-point neighborhood) lead to global system evolution
- Shows the importance of efficient parallel implementation for such iterative computations, as real-world applications often require much larger matrices and more iterations

The smooth transition between states and the preservation of conservation laws (total heat content) across iterations also serves as a quality metric for our implementations, confirming that our parallel approaches maintain numerical stability and accuracy while improving computational performance. For real-world applications, these simulations often need to handle much larger domains and longer time scales, making efficient parallel implementation crucial for practical use.

IV. Results

Analysis of performance data collected from all implementations (Serial, Pthreads, OpenMP, MPI, and MPI+OpenMP hybrid) reveals distinct patterns in execution time, speedup, efficiency, and overhead characteristics across different problem sizes and parallel configurations. Through detailed visualization and analysis of our performance data, we identified several key performance trends and relationships.

A. Performance Visualization Analysis

Our analysis utilized multiple visualization approaches to understand the performance characteristics:

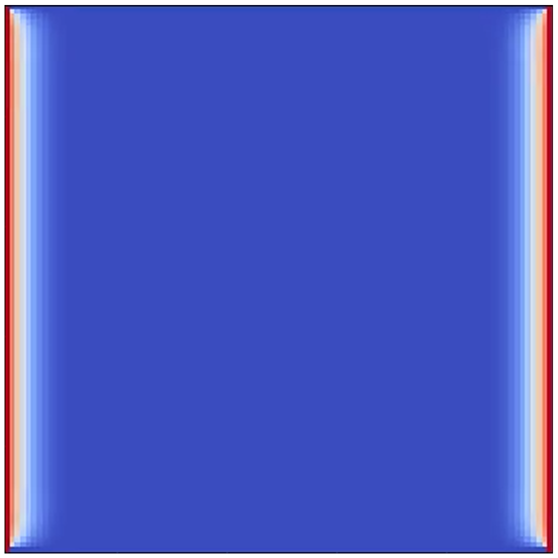

Fig. 1. Initial state of the matrix.

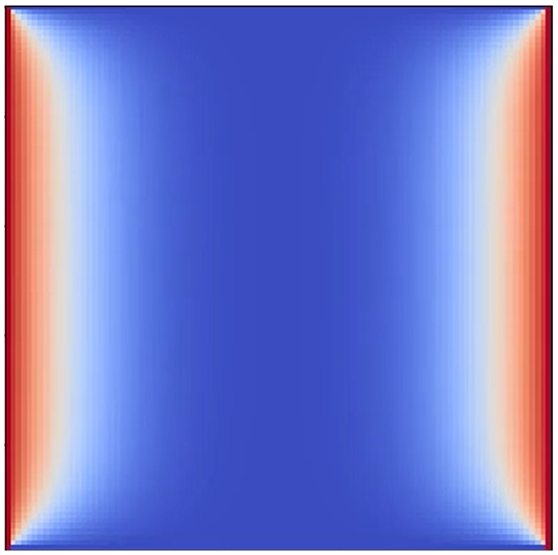

Fig. 2. Intermediate state of the matrix after several iterations.

- **Time Components Analysis**:
  - Bar plots showing computation vs. overhead time
  - Distinct color coding for work and other time components
  - Comparison across all implementation types
  - Clear visualization of overhead impact

B. Performance Metric Correlations

Our visualization analysis revealed several important correlations:

- **Time vs. Thread Count**:
  - Non-linear improvement with thread increase
  - Sharp efficiency drop beyond 32 threads per process
  - Implementation-specific scaling patterns

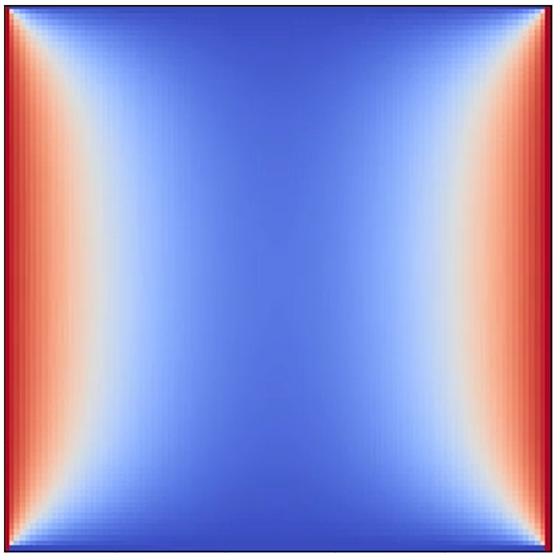

Fig. 3. Intermediate state of the matrix after several more iterations.

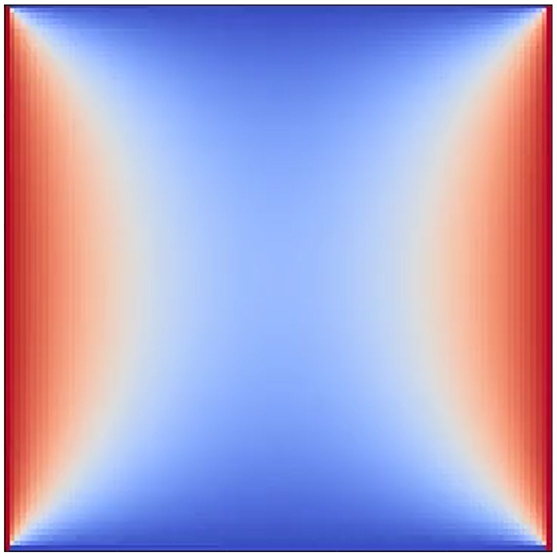

Fig. 4. Final state of the matrix after all iterations.

    * Clear performance plateaus
  - **Communication vs. Computation**:
    * Clear trade-offs between message size and frequency
    * Implementation-specific communication patterns
    * Impact of process/thread configuration
    * Optimal balance points

These visualizations and correlations provided crucial insights for understanding the performance characteristics of each implementation approach and guided our recommendations for optimal configuration choices.

- **Speedup Analysis**:
  - Logarithmic scale comparisons
  - Ideal speedup reference lines
  - Implementation-specific scaling patterns

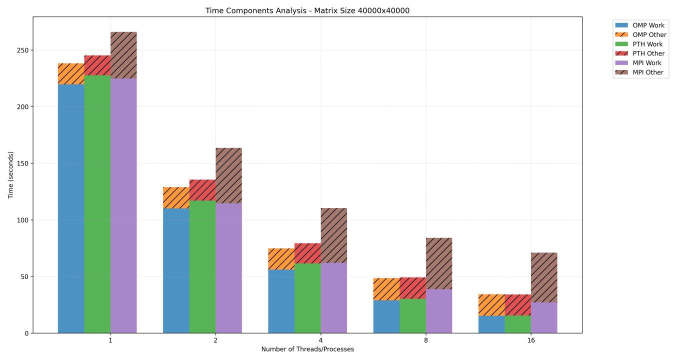

Fig. 5. Time Componets for MPI, OpenMP, and PThreads

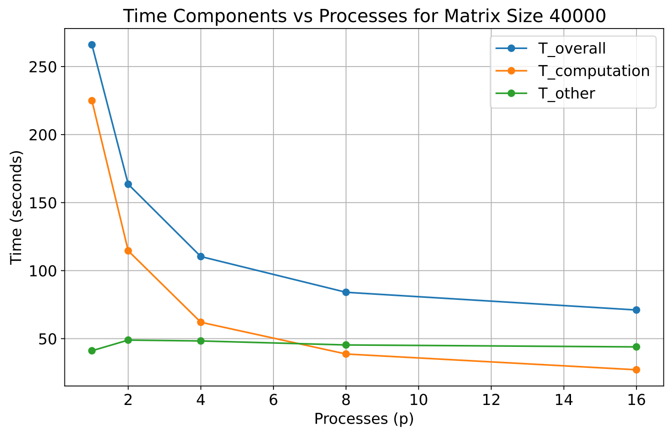

Fig. 6. Time components for MPI

  - Matrix size impact visualization
- **Efficiency Visualization**:
  - Parallel efficiency across implementations
  - Thread/process count impact
  - Problem size scaling effects
  - Resource utilization patterns

Below each type of parallel implementation will be broken down separately for Time, Speedup and Efficiency.

V. Conclusions

This comprehensive study compared four parallel implementation approaches for a 9-point stencil computation: POSIX threads, OpenMP, pure MPI, and hybrid MPI+OpenMP. Through extensive testing on the Expanse supercomputer across various problem sizes and processor configurations, we gained significant insights into the performance characteristics and trade-offs of each parallel programming paradigm.

A. *Key Findings*

Implementation Performance:

- **Shared Memory Approaches**:
  - Both Pthreads and OpenMP achieved approximately 7× speedup with 16 threads on 40K matrices

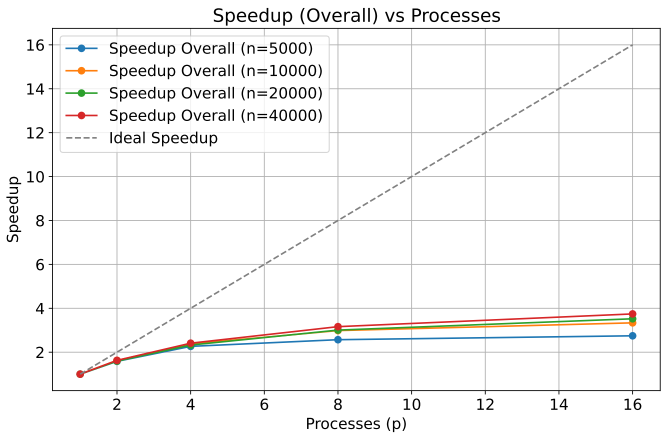

Fig. 7. Overall Speedup for MPI

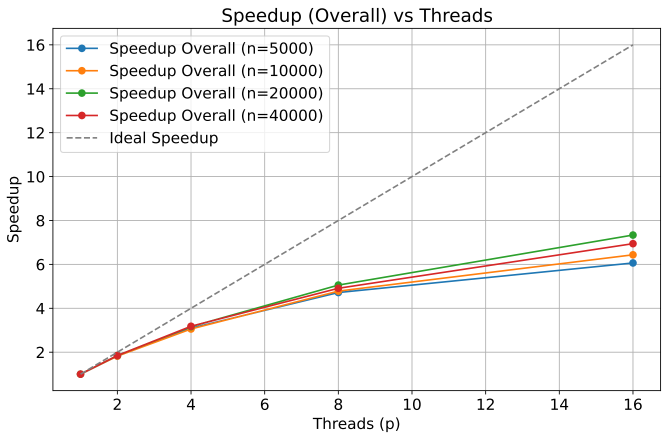

Fig. 10. Overall Speedup for OMP

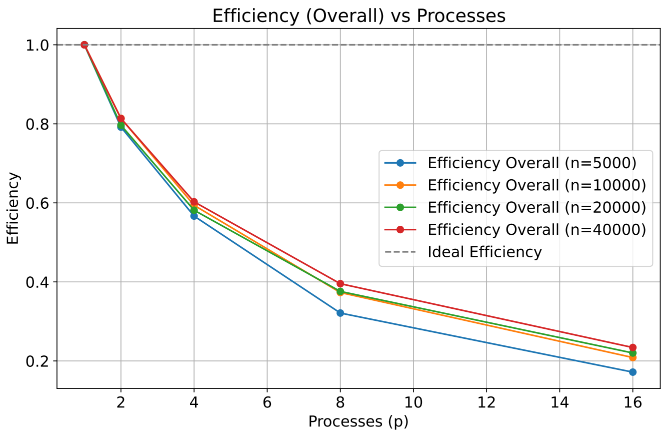

Fig. 8. Overall Efficiency for MPI

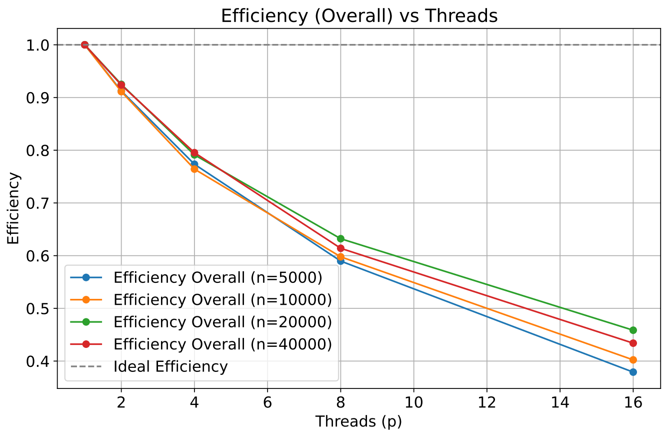

Fig. 11. Overall Efficiency for OMP

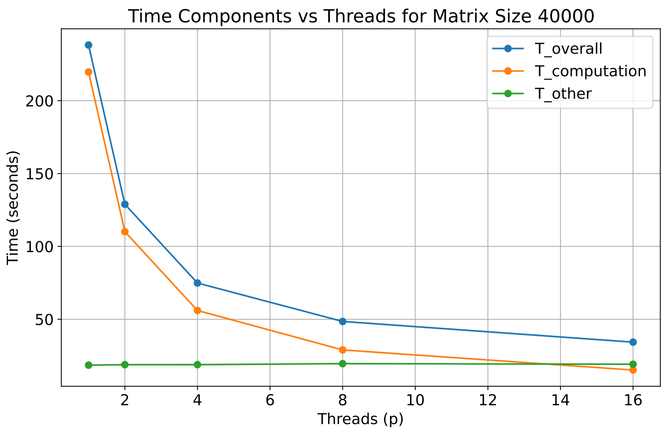

Fig. 9. Time components for OMP

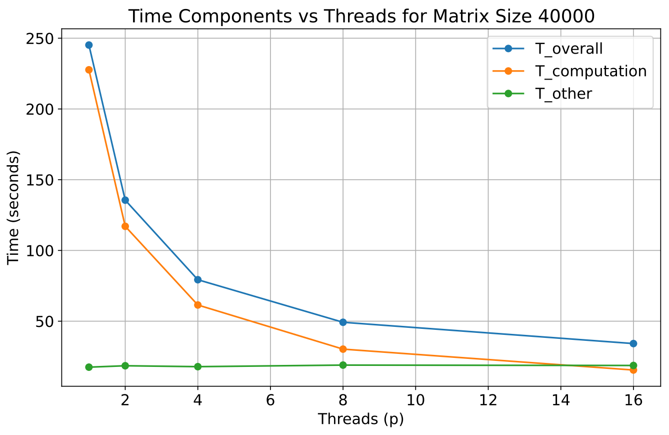

Fig. 12. Time components for Pthreads

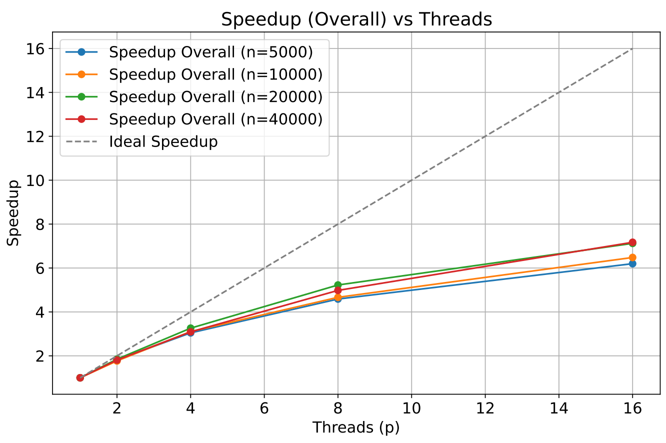

Fig. 13. Overall Speedup for Pthreads

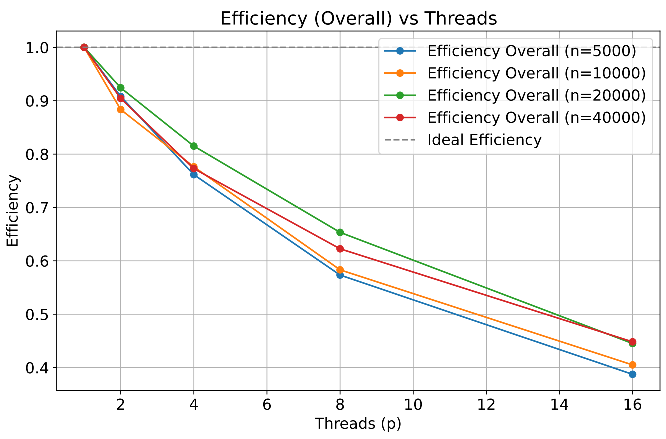

Fig. 14. Overall Efficiency for Pthreads

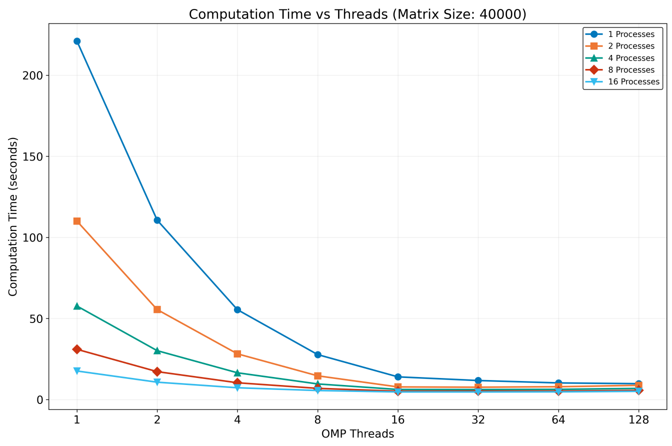

Fig. 15. Time components for MPI with OMP

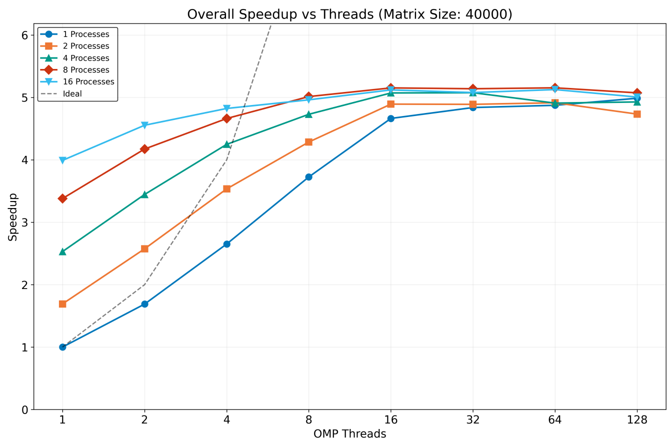

Fig. 16. Overall Speedup for MPI with OMP

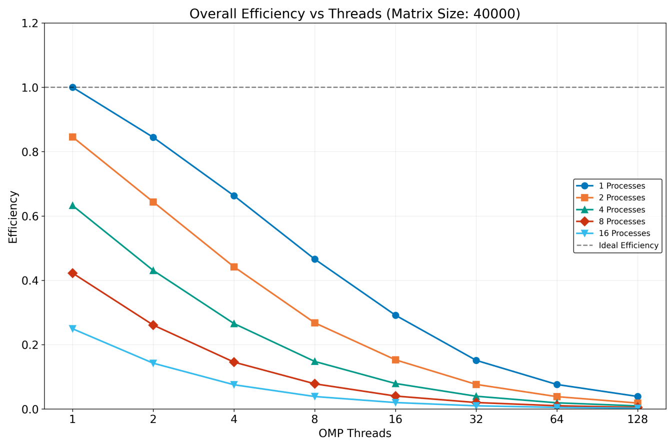

Fig. 17. Overall Efficiency for MPI with OMP

  - OpenMP provided comparable performance to Pthreads with simpler implementation
  - Best performance for single-node execution and smaller problem sizes
- **Pure MPI**:
  - Achieved 3.74× speedup with 16 processes on 40K matrices
  - High communication overhead (61.8
  - Performance limited by halo exchange operations
  - Better suited for systems with high-speed interconnects
- **Hybrid MPI+OpenMP**:
  - Best overall performance for large matrices
  - Optimal configuration: 8 processes × 16 threads
  - 5.24× speedup on 40K matrices
  - Reduced communication overhead compared to pure MPI
  - Better resource utilization but more complex implementation

B. *Implementation Trade-offs*

Communication vs. Computation Balance:

- **Pure MPI**:
  - Higher communication overhead
  - Better process isolation
  - Simpler implementation than hybrid
  - Limited by network performance
- **Hybrid Approach**:
  - Reduced communication frequency
  - Better cache utilization
  - Complex process-thread coordination
  - More efficient resource usage

C. Recommendations

Based on our findings, we recommend:

- **For Single-Node Systems**:
  - Use OpenMP for simplicity and good performance
  - Consider Pthreads only if fine-grained control is required
  - Optimal for matrices up to 20K×20K
- **For Multi-Node Systems**:
  - Use hybrid MPI+OpenMP for large matrices (¿20K×20K)
  - Consider pure MPI only with high-speed interconnects
  - Target 8-16 processes with 16-32 threads per process
  - Balance process/thread counts based on problem size

D. Future Work

Several directions for future research emerge from this study:

- **Implementation Extensions**:
  - Exploration of GPU acceleration
  - Investigation of adaptive process-thread balancing
  - Development of auto-tuning mechanisms
- **Performance Optimization**:
  - Advanced communication patterns for MPI
  - Improved hybrid scheduling strategies
  - Cache optimization techniques
- **Scaling Studies**:
  - Testing on larger node counts
  - Analysis of energy efficiency
  - Investigation of fault tolerance strategies

This study provides valuable insights for choosing parallel programming models for stencil computations. While shared memory approaches excel in single-node scenarios, hybrid implementations offer the best balance of performance and scalability for large-scale computations. The significant impact of communication overhead in distributed implementations highlights the importance of careful consideration of the underlying hardware architecture and problem characteristics when selecting an implementation strategy.

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.

VI. Appendix

In this appendix, you can view all the data outputs and results that we used through tables. Tables start on next page.

TABLE I
OPENMP IMPLEMENTATION PERFORMANCE RESULTS

Matrix Size	**Thread Count**	**Total Time (s)**	**Compute Time (s)**	**Other Time (s)**
5000	1	3.85	3.44	0.41
	2	2.11	1.72	0.39
	4	1.24	0.86	0.38
	8	0.82	0.43	0.38
	16	0.63	0.22	0.41
10000	1	15.19	13.78	1.41
	2	8.34	6.88	1.46
	4	4.97	3.48	1.49
	8	3.18	1.73	1.44
	16	2.36	0.91	1.45
20000	1	59.91	55.07	4.84
	2	32.38	27.53	4.85
	4	18.93	14.04	4.89
	8	11.85	6.94	4.90
	16	8.17	3.68	4.48
40000	1	238.20	219.69	18.51
	2	128.94	110.11	18.82
	4	74.87	56.02	18.85
	8	48.50	28.96	19.54
	16	34.30	15.17	19.14

TABLE II
PTHREAD IMPLEMENTATION PERFORMANCE RESULTS

Matrix Size	**Thread Count**	**Total Time (s)**	**Compute Time (s)**	**Other Time (s)**
5000	1	3.92	3.55	0.38
	2	2.16	1.78	0.38
	4	1.29	0.89	0.40
	8	0.86	0.45	0.41
	16	0.63	0.23	0.40
10000	1	15.62	14.21	1.42
	2	8.84	7.25	1.59
	4	5.03	3.62	1.42
	8	3.35	1.80	1.55
	16	2.41	0.91	1.50
20000	1	62.65	57.83	4.81
	2	33.89	28.94	4.95
	4	19.22	14.50	4.72
	8	11.99	7.22	4.76
	16	8.80	3.73	5.06
40000	1	245.19	227.68	17.51
	2	135.55	117.05	18.50
	4	79.31	61.45	17.86
	8	49.23	30.26	18.97
	16	34.19	15.49	18.70

TABLE III
MPI IMPLEMENTATION PERFORMANCE RESULTS

Matrix Size	**Process Count**	**Total Time (s)**	**Compute Time (s)**	**Other Time (s)**
5000	1	4.27	3.52	0.74
	2	2.69	1.78	0.91
	4	1.88	0.95	0.93
	8	1.66	0.64	1.02
	16	1.55	0.49	1.06
10000	1	16.93	14.11	2.83
	2	10.40	7.13	3.27
	4	7.13	3.82	3.31
	8	5.67	2.41	3.26
	16	5.07	1.68	3.40
20000	1	66.28	56.03	10.25
	2	41.60	28.55	13.06
	4	28.50	15.53	12.97
	8	22.04	9.81	12.23
	16	18.82	6.91	11.91
40000	1	265.99	224.89	41.10
	2	163.49	114.55	48.94
	4	110.34	62.04	48.30
	8	84.07	38.71	45.37
	16	71.05	27.12	43.93

TABLE IV
HYBRID MPI+OPENMP IMPLEMENTATION PERFORMANCE RESULTS (MATRIX SIZE 5000)

Matrix Size	**Process Count**	**Thread Count**	**Total Time (s)**	**Compute Time (s)**	**Other Time (s)**
5000	1	1	4.18	3.44	0.74
	1	2	2.51	1.71	0.80
	1	4	2.10	0.86	1.24
	1	8	1.21	0.43	0.78
	1	16	1.00	0.22	0.78
	2	1	2.58	1.72	0.86
	2	2	1.71	0.87	0.84
	2	4	1.35	0.44	0.91
	2	8	1.09	0.23	0.86
	2	16	0.99	0.13	0.87
	4	1	1.93	0.90	1.03
	4	2	1.44	0.47	0.97
	4	4	1.19	0.26	0.94
	4	8	1.04	0.15	0.89
	4	16	0.98	0.10	0.88
	8	1	1.58	0.49	1.09
	8	2	1.25	0.27	0.97
	8	4	1.14	0.17	0.98
	8	8	1.10	0.11	0.99
	8	16	1.08	0.09	0.99

TABLE V
HYBRID MPI+OPENMP IMPLEMENTATION PERFORMANCE RESULTS (MATRIX SIZE 10000)

Matrix Size	**Process Count**	**Thread Count**	**Total Time (s)**	**Compute Time (s)**	**Other Time (s)**
10000	1	1	16.76	13.70	3.05
	1	2	9.81	6.85	2.96
	1	4	6.47	3.44	3.04
	1	8	4.77	1.72	3.05
	1	16	3.84	0.87	2.98
	2	1	10.05	6.87	3.18
	2	2	6.60	3.48	3.12
	2	4	4.86	1.77	3.09
	2	8	4.04	0.92	3.13
	2	16	3.66	0.49	3.17
	4	1	6.80	3.59	3.21
	4	2	5.08	1.87	3.21
	4	4	4.20	1.03	3.17
	4	8	3.80	0.59	3.21
	4	16	3.58	0.38	3.20
	8	1	5.36	1.94	3.42
	8	2	4.51	1.08	3.43
	8	4	3.98	0.65	3.33
	8	8	3.69	0.44	3.26
	8	16	3.65	0.33	3.32

TABLE VI
HYBRID MPI+OPENMP IMPLEMENTATION PERFORMANCE RESULTS (MATRIX SIZE 20000)

Matrix Size	**Process Count**	**Thread Count**	**Total Time (s)**	**Compute Time (s)**	**Other Time (s)**
20000	1	1	65.89	54.88	11.01
	1	2	38.30	27.54	10.76
	1	4	24.92	13.74	11.19
	1	8	17.72	6.88	10.84
	1	16	14.29	3.44	10.85
	2	1	39.28	27.49	11.79
	2	2	24.69	13.92	10.77
	2	4	18.86	7.09	11.77
	2	8	15.35	3.69	11.66
	2	16	13.65	1.95	11.70
	4	1	26.50	14.35	12.16
	4	2	19.39	7.49	11.90
	4	4	16.00	4.07	11.93
	4	8	14.33	2.38	11.95
	4	16	13.34	1.50	11.84
	8	1	19.67	7.67	11.99
	8	2	16.09	4.24	11.85
	8	4	14.52	2.53	11.99
	8	8	13.72	1.68	12.04
	8	16	14.65	1.24	13.41

TABLE VII
HYBRID MPI+OPENMP IMPLEMENTATION PERFORMANCE RESULTS (MATRIX SIZE 40000)

Matrix Size	**Process Count**	**Thread Count**	**Total Time (s)**	**Compute Time (s)**	**Other Time (s)**
40000	1	1	260.29	221.08	39.21
	1	2	154.12	110.61	43.51
	1	4	98.16	55.40	42.76
	1	8	69.86	27.70	42.16
	1	16	55.82	13.97	41.85
	2	1	153.88	110.05	43.83
	2	2	101.11	55.47	45.64
	2	4	73.64	28.17	45.47
	2	8	60.74	14.61	46.13
	2	16	53.20	7.76	45.44
	4	1	102.87	57.66	45.21
	4	2	75.55	30.13	45.43
	4	4	61.27	16.49	44.79
	4	8	55.03	9.60	45.43
	4	16	51.32	6.19	45.13
	8	1	77.02	30.96	46.05
	8	2	62.37	17.20	45.18
	8	4	55.83	10.33	45.50
	8	8	51.92	6.87	45.04
	8	16	50.52	5.14	45.38