# Exploring Multi-Threaded and Distributed Computing Approaches for a Nine-Point Stencil Program

Christopher Grigsby
*Department of Computing Sciences*
*Coastal Carolina University*
Conway, SC, USA
cwgrigsby@coastal.edu

Jordan Drakos
*Department of Computing Sciences*
*Coastal Carolina University*
Conway, SC, USA
jdrakos@coastal.edu

***Abstract*—In this paper, we present a comparative analysis of the performance of a nine-point iterative stencil program for simulating heat transference on a metal plate. We implement the program using a serial version, a pthread version, and an MPI version, and compare their respective execution times on a shared-memory system and a distributed-memory cluster. The program involves a double representation of temperature, where 1.0 represents the hottest and 0.0 represents the coldest, and a specific boundary condition where the first and last columns are set to 1.0 and the first and last rows are set to 0.0. At each iteration, the program uses a calculation involving the eight neighboring points to update the temperature at each point. We evaluate the performance of the three versions of the program in terms of speedup, scalability, and efficiency, and present the results of our experiments in tables and graphs. Our findings show that the parallel versions achieve significant speedup over the serial version, and that the MPI version performs better than the pthread version on the distributed-memory cluster. Our study demonstrates the effectiveness of parallel computing techniques in accelerating iterative stencil programs and provides insights into the trade-offs between different parallelization approaches.**

***Index Terms*—reliability, performance, HPC, simulation**

## I. Introduction

Iterative stencil programs are a class of numerical algorithms widely used in scientific computing to simulate physical phenomena, such as heat transfer, fluid dynamics, and electromagnetic fields. These programs involve updating the values of a two-dimensional grid of cells, where each cell depends on the values of its neighboring cells, based on a mathematical formula. The iterative stencil approach lends itself naturally to parallelization, as each cell can be updated independently of the others. In this paper, we investigate the performance of a nine-point iterative stencil program for simulating heat transfer on a metal plate, using a double representation of temperature, where 1.0 represents the hottest and 0.0 represents the coldest. The program employs a specific boundary condition, where the first and last columns are set to 1.0 and the first and last rows are set to 0.0. We implement the program using a serial version, a pthread version, and an MPI version, and evaluate their performance on a shared-memory system and a distributed-memory cluster. Our goal is to compare the speedup, scalability, and efficiency of the three versions of the program and to provide insights into the trade-offs between different parallelization approaches. We present the results of our experiments in tables and graphs and analyze their implications for scientific computing applications.

This paper is organized as follows. This paper is organized as follows. The software implementation is discussed in Section II, including details on the input file format, algorithms used, and other relevant design aspects.

In Section III, we outline the experiments performed to evaluate the enhancements, including information on the hardware used. The results of the experiments are presented and discussed in Section IV.

Finally, we conclude the paper in Section V with a summary of our findings and suggestions for future work.

## II. Design

In this section, we discuss the design of the nine-point stencil algorithm, which is used to approximate the solution to an equation on a two-dimensional matrix. The algorithm computes an updated value for each point on the matrix based on the average of its surrounding eight neighbors, as well as its current value. Notably, the equation doesn't update the points in the first and last rows and columns of the matrix. We present three different implementations of the algorithm: a serial implementation (II-C), a parallel implementation using pthreads (II-D), and a parallel implementation using MPI (II-E). We compare their performance and discuss the trade-offs between the different approaches.

### A. Input File Format

The input to the program is a binary file in .dat format that contains a two-dimensional array of floating point numbers representing the initial state of the system. The file is created by another piece of the program, where the user specifies the number of rows and columns in the array.

Each value in the array is represented by 8 bytes in little-endian format, and the values are stored in row-major order.

That is, the first row of the array is stored first, followed by the second row, and so on.

To read in the data, the program first reads in the number of rows and columns from the file. It then allocates memory for the two-dimensional array and reads in the data from the file. Once the data has been read in, the computation can begin.

### B. Output File Format

The program produces two outputs: a binary file in .dat format that contains a two-dimensional array of floating point numbers representing the final state of the system after a user-specified number of iterations and an image stack in .raw format that contains a visual representation of the system at each iteration. The image stack file can then be visualized using ParaView [1] in 2D to analyze and confirm the data.

**Algorithm 1** Nine-Point Stencil (Serial)

```
Read x from file
Create array newx
for n ← 0 to numiterations do
  for i ← 1 to rows − 1 do
    for j ← 1 to cols − 1 do
      newx[i][j] = (x[i-1][j-1] + x[i-1][j] + x[i-1][j+1] +
      x[i][j+1] + x[i+1][j+1] + x[i+1][j] + x[i+1][j-1] +
      x[i][j-1] + x[i][j]) / 9.0
    end for
  end for
  tmp ← x
  x ← newx
  newx ← tmp
end for
```

### C. Serial Implementation

In the serial implementation, the algorithm runs sequentially on a single processor. The algorithm follows the nine-point stencil method, where each matrix point is updated based on the values of its neighboring matrix points. The serial implementation is shown in Algorithm 1.

For the purpose of our project, the number of rows matches the number of columns, with the number of iterations remaining constant across all matrix sizes. This gives the algorithm a time complexity of $\mathcal{O}(n^2)$ and a space complexity of $\mathcal{O}(n^2)$, as it uses two 2D arrays to store the data: x and newx, each of size n×n. The algorithm also uses a temporary array of the same size, so space complexity is proportional to the square of the input size.

Although the time complexity is feasible for smaller matrices, it can become a bottleneck for larger ones. As the size of the input matrix increases, the time required to complete a single iteration grows quadratically, which can quickly become impractical.

By dividing the matrix into smaller submatrices and assigning each submatrix to a separate process, the computation can be performed in parallel, significantly reducing the time required to complete iteration.

**Algorithm 2** Nine-Point Stencil (Parallel with pthreads)

```
Get numthreads
Read x from file
Create array newx
Calculate number of rows per thread
for n ← 0 to numiterations do
  Do in parallel with pthreads (numthreads)
  for i ← 1 to rows − 1 do
    for j ← 1 to cols − 1 do
      newx[i][j] = (x[i-1][j-1] + x[i-1][j] + x[i-1][j+1] +
      x[i][j+1] + x[i+1][j+1] + x[i+1][j] + x[i+1][j-1] +
      x[i][j-1] + x[i][j]) / 9.0
    end for
  end for
  End parallel
  tmp ← x
  x ← newx
  newx ← tmp
end for
```

### D. Parallel Implementation with pthreads

The pthread implementation of the nine-point stencil algorithm partitions the matrix into subsets that are then processed by multiple threads concurrently. In contrast to the serial algorithm, which updates each point in the matrix sequentially, the pthread implementation parallelizes the innermost loop of the algorithm, allowing for significant speedup in computation times.

In the parallel implementation presented in Algorithm 2, the main thread creates a specified number of worker threads and divides the matrix points evenly among them. Each worker thread updates a subset of the matrix points according to the nine-point stencil algorithm. This division of labor helps to minimize thread synchronization overhead and maximize parallelism. After all worker threads have completed their computations, the main thread updates the entire matrix with the new values.

The use of pthreads provides fine-grained control over the parallelization of the algorithm, allowing for efficient utilization of shared memory architectures. By distributing the workload across multiple threads, we can better exploit the processing power of modern CPUs and achieve faster computation times. This is particularly important for large matrices, where the time complexity of the serial algorithm can become a bottleneck.

The effectiveness of the pthreads implementation relies on the ability to evenly divide the matrix points among the worker threads, as unbalanced workloads can result in performance degradation due to load imbalance. To address this issue, various load balancing techniques can be employed, such as dynamic load balancing, which adjusts the workload assigned to each thread based on the progress of the computation.

**Algorithm 3** Nine-Point Stencil (Parallel with MPI)

```
Initialize MPI
Get numprocs and myrank
Read x from file
Create array newx
Calculate number of rows per process
for n ← 0 to numiterations do
  Do in parallel with MPI (numprocs)
  for i ← 1 to rows − 1 do
    for j ← 1 to cols − 1 do
      newx[i][j] = (x[i-1][j-1] + x[i-1][j] + x[i-1][j+1] +
      x[i][j+1] + x[i+1][j+1] + x[i+1][j] + x[i+1][j-1] +
      x[i][j-1] + x[i][j]) / 9.0
    end for
  end for
  End parallel
  tmp ← x
  x ← newx
  newx ← tmp
end for
Finalize MPI
```

### E. Parallel Implementation with MPI

The MPI implementation of the nine-point stencil algorithm is achieved by dividing the matrix points into subsets, which are then processed by multiple processes in parallel. In the parallel implementation shown in Algorithm 3, the innermost loop of the algorithm is parallelized with each process responsible for updating a subset of the matrix points. The use of MPI allows for communication between the processes and enables the algorithm to be run on distributed memory structures.

To parallelize the inner loop with MPI, the main process creates a specified number of worker processes and divides the matrix points evenly among them. Each worker process updates a subset of matrix points according to the nine-point stencil algorithm. The communication between the processes is handled using MPI communication functions such as `MPI_Isend` and `MPI_Irecv`.

After all worker processes have completed their computations, the main process collects the updated matrix points from each worker process using MPI communication functions and updates the entire matrix with the new values.

The use of MPI allows for the algorithm to be run on distributed memory architectures and can provide good performance for large-scale simulations. However, the communication overhead can limit the scalability of the algorithm on large-scale systems, and careful consideration must be taken when designing the communication pattern to minimize overhead.

## III. Experimentation

We conducted our simulations using the San Diego Supercomputer Center's Expanse supercomputer, which was made available through our university's access to the system. Our simulations were run using three different resource configurations: serial, pthread, and MPI. For the serial simulations, we used a single node with one CPU core, and allocated 64GB of memory. For the pthread simulations, we also used a single node, but allocated 16 CPU cores with a total of 64GB of memory. Finally, for the MPI simulations, we used a single node with 16 CPU cores, and allocated 64GB of memory. These configurations allowed us to explore the performance characteristics of our simulations under different compute loads and parallelization strategies.

### A. Serial Experiments

In the serial experiments, we ran our simulations using a single CPU core and allocated 64GB of memory. We used the stencil-2d application, with a range of matrix sizes from 1000x1000 to 16000x16000. We set the number of iterations to 100 for each matrix size.

To create the initial input matrices, we used the make-2d application. We looped over a range of values to create matrices of increasing size, from 1000x1000 to 16000x16000. For each matrix size, we created an input data file using the naming convention i-init.dat, where i is the matrix size.

We then ran the stencil-2d application for each matrix size, using the created input data file, final state file name (final.dat), and summary file name (summary-serial.txt). We recorded the execution time for each matrix size, which we used to analyze the scalability and performance of the serial simulations.

After the simulations were complete, we copied the output data files (input data file, final state file, and summary file) out of the scratch space to our local directory for further analysis.

### B. Pthread Experiments

For the pthread experiments, we used the pth-stencil-2d application and matrix sizes as the serial experiments, but with 1, 2, 4, 8, and 16 threads. We allocated 64GB of memory and 16 CPU cores for all simulations.

We used the provided pthread script to run the simulations, which executed the make-2d application to create the input matrices, and then ran the stencil-2d application with varying numbers of threads for each matrix size. The script recorded the execution time and wrote the final state to a file for each simulation, as well as a summary file for use in generating performance charts.

After the simulations were complete, we copied the output data files (input data file, final state file, and summary file) out of the scratch space to our local directory for further analysis.

### C. MPI Experiments

For the MPI experiments, we used the mpi-stencil-2d application and matrix sizes same as in the serial and pthread experiments, but with 1, 2, 4, 8, and 16 MPI processes. We allocated 64GB of memory and 16 CPU cores for all simulations on a single node.

We used the provided MPI script to run the simulations, which first created the input matrices using the make-2d application. For each matrix size, we created an input data file

using the naming convention i-init.dat, where i is the matrix size.

Next, we ran the mpi-stencil-2d application for each matrix size, using the created input data file, final state file name (final.dat), summary file name (summary-mpi.txt), and varying numbers of MPI processes (1, 2, 4, 8, and 16) for each matrix size. The script recorded the execution time and wrote the final state to a file for each simulation, as well as a summary file for use in generating performance charts.

After the simulations were complete, we copied the output data files (input data file, final state file, and summary file) out of the scratch space to our local directory for further analysis. We then analyzed the execution time and scalability of the MPI simulations and compared them to the results from the serial and pthread simulations.

## IV. Results

In this section, we present the results of our experiments using three different implementations: serial, pthreads, and MPI. We start by discussing the results obtained using the serial implementation in Section IV-A, followed by the results obtained using pthreads in Section IV-B and the results obtained using MPI in Section IV-C.

### A. Serial Results

Here we present the results of our experiments using a serial implementation.

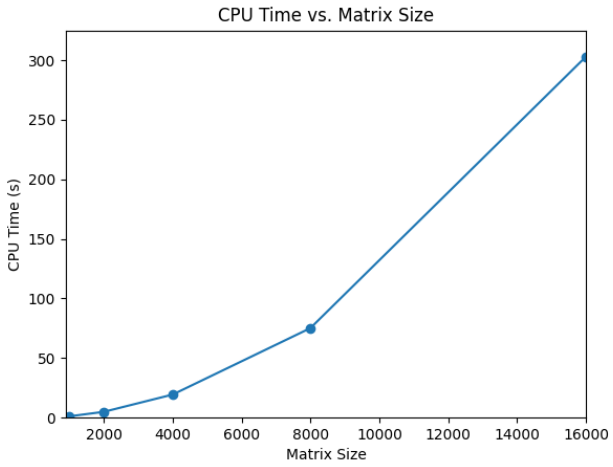



Fig. 1: CPU time of serial implementation for different problem sizes.

Figure 1 shows the CPU times taken by the serial implementation for different problem sizes, while figure 2 shows the overall time taken.

It is worth noting that most of the time spent in the serial implementation comes from CPU time, which reinforces the need for parallel implementations to exploit the available parallelism and reduce the overall execution time.

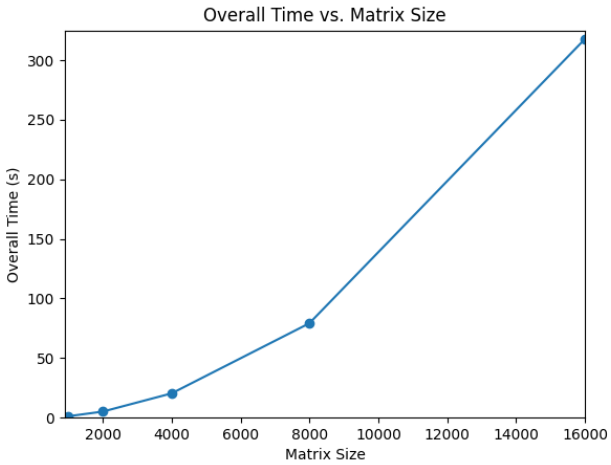
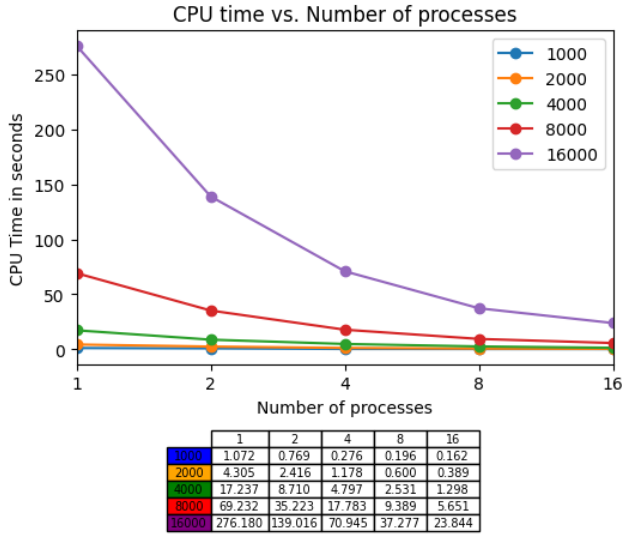



Fig. 2: Overall time of serial implementation for different problem sizes.

### B. Pthread Results

Here we present the results of our experiments using pthreads.




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.072 | 0.769 | 0.276 | 0.196 | 0.162 |
| 2000 | 4.305 | 2.416 | 1.178 | 0.600 | 0.389 |
| 4000 | 17.237 | 8.710 | 4.797 | 2.531 | 1.298 |
| 8000 | 69.232 | 35.223 | 17.783 | 9.389 | 5.651 |
| 16000 | 276.180 | 139.016 | 70.945 | 37.277 | 23.844 |

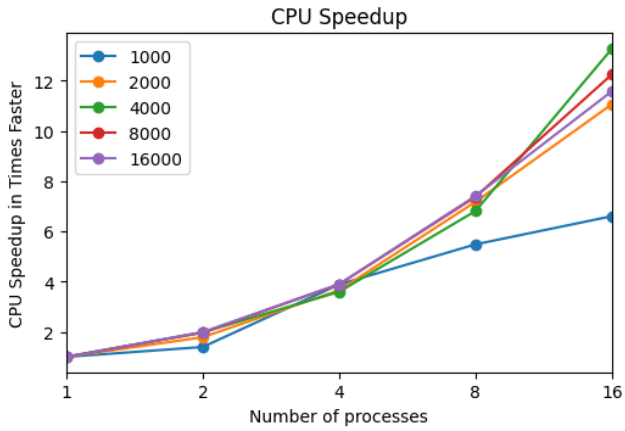
Fig. 3: CPU time of pthreads implementation for different problem sizes and thread counts.

Figure 3 shows the CPU times taken by the pthread implementation for different numbers of threads.

The speedup achieved by the pthread implementation relative to the serial implementation is calculated using the following formula:

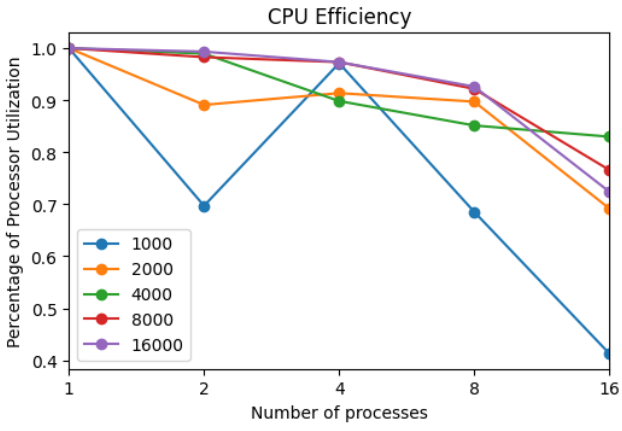
$$S_p = \frac{T_s}{T_p} \tag{1}$$

where $S_p$ is the speedup, $T_s$ is the time taken by the serial implementation, and $T_p$ is the time taken by the pthread implementation [2].




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.000 | 1.394 | 3.884 | 5.482 | 6.604 |
| 2000 | 1.000 | 1.782 | 3.654 | 7.177 | 11.066 |
| 4000 | 1.000 | 1.979 | 3.593 | 6.811 | 13.277 |
| 8000 | 1.000 | 1.966 | 3.893 | 7.374 | 12.252 |
| 16000 | 1.000 | 1.987 | 3.893 | 7.409 | 11.583 |

Fig. 4: CPU speedup of pthreads implementation for different problem sizes and thread counts.




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.000 | 0.697 | 0.971 | 0.685 | 0.413 |
| 2000 | 1.000 | 0.891 | 0.914 | 0.897 | 0.692 |
| 4000 | 1.000 | 0.989 | 0.898 | 0.851 | 0.830 |
| 8000 | 1.000 | 0.983 | 0.973 | 0.922 | 0.766 |
| 16000 | 1.000 | 0.993 | 0.973 | 0.926 | 0.724 |

Fig. 5: CPU efficiency of pthreads implementation for different problem sizes and thread counts.

As shown in Figure 4, the speedup increases as the number of threads increases for all problem sizes. This is expected, as more threads allow for more parallelism and therefore faster execution. The speedup achieved by the pthread implementation is significant; however, it is important to note that the speedup is not linear with respect to the number of threads. As shown in the figure, there is a diminishing return as the number of threads increases beyond a certain point. This is due to the overhead of thread creation and synchronization becoming more significant as more threads are used. Therefore, it is important to choose an appropriate number of threads to balance the benefits of parallelism with the overhead of thread management.

The efficiency achieved by the pthread implementation is calculated using the following formula:

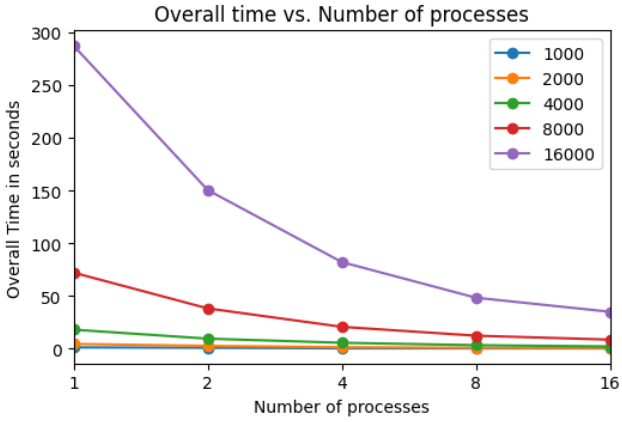
$$E_p = \frac{S_p}{p} \tag{2}$$

where $E_p$ is the efficiency, $S_p$ is the speedup, and $p$ is the number of pthreads used [2].

Figure 5 shows the CPU efficiency achieved by the pthread implementation for different problem sizes and thread counts.

As shown in Figure 5, the efficiency decreases as the number of threads increases for all problem sizes. This is because the overhead of thread creation and synchronization becomes more significant as more threads are used, and therefore a greater proportion of the potential speedup is lost.

Overall, the pthread implementation shows significant speedup over the serial implementation, especially for larger problem sizes and with an appropriate number of threads chosen. However, it is important to carefully balance the benefits of parallelism with the overhead of thread management in order to achieve optimal performance.

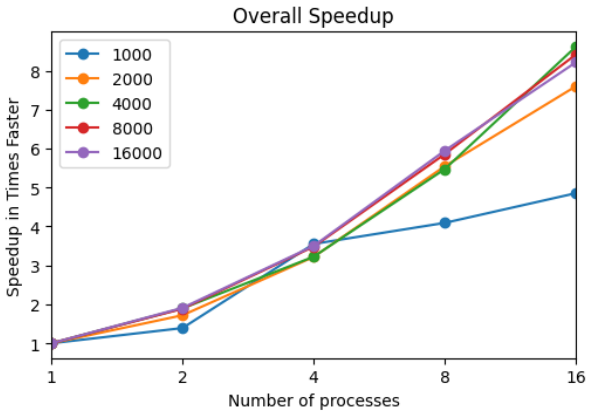



| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.205 | 0.866 | 0.339 | 0.295 | 0.248 |
| 2000 | 4.527 | 2.632 | 1.407 | 0.816 | 0.596 |
| 4000 | 18.016 | 9.501 | 5.590 | 3.293 | 2.091 |
| 8000 | 72.081 | 38.191 | 20.705 | 12.308 | 8.566 |
| 16000 | 287.283 | 150.407 | 82.103 | 48.316 | 34.973 |

Fig. 6: Overall execution times of pthreads implementation for different problem sizes and thread counts.

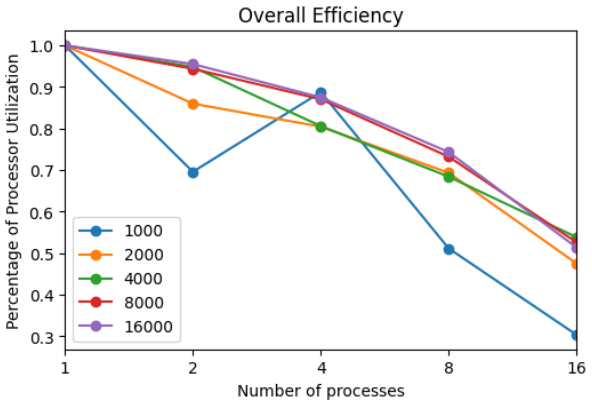
As seen in Figure 6, the trend in overall execution times across various problem sizes and thread counts is approximately the same as the trend for the CPU time. Because the program spends most of its time performing the stencil operation, this is to be expected.

Based on the observation that the trend in overall execution times across various problem sizes and thread counts is similar to the trend for the CPU time, it can be assumed that the program's rate of speedup and efficiency will also follow this trend.




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.000 | 1.391 | 3.550 | 4.091 | 4.853 |
| 2000 | 1.000 | 1.720 | 3.218 | 5.546 | 7.599 |
| 4000 | 1.000 | 1.896 | 3.223 | 5.472 | 8.615 |
| 8000 | 1.000 | 1.887 | 3.481 | 5.856 | 8.414 |
| 16000 | 1.000 | 1.910 | 3.499 | 5.946 | 8.214 |

(a) Overall speedup of pthreads implementation for different problem sizes and thread counts.




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.000 | 0.696 | 0.888 | 0.511 | 0.303 |
| 2000 | 1.000 | 0.860 | 0.804 | 0.693 | 0.475 |
| 4000 | 1.000 | 0.948 | 0.806 | 0.684 | 0.538 |
| 8000 | 1.000 | 0.944 | 0.870 | 0.732 | 0.526 |
| 16000 | 1.000 | 0.955 | 0.875 | 0.743 | 0.513 |

(b) Overall efficiency of pthreads implementation for different problem sizes and thread counts.

Fig. 7: Overall performance of pthreads implementation for different problem sizes and thread counts.

The fluctuation in efficiency, seen in both Figure 5 and Figure 7b may be due to contention for shared resources such as memory or caches, which can cause threads to wait for each other and reduce efficiency. Additionally, the overhead of thread creation and synchronization may also be a factor. It is possible that the specific hardware configuration for the SDSC Expanse supercomputer may also be a contributing factor.

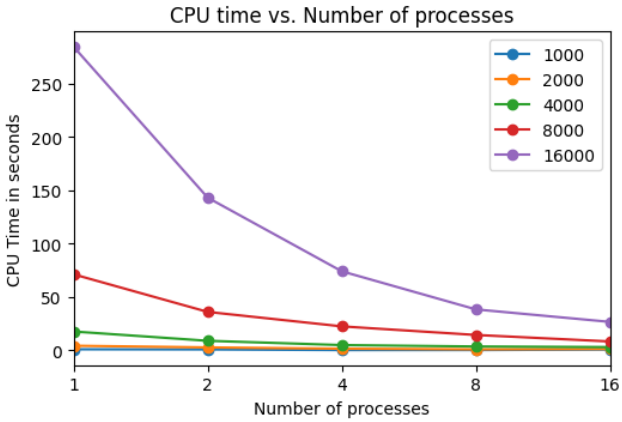
If one were to further investigate the cause of the fluctuation, it could be beneficial to analyze the memory access patterns of the program and explore the specific hardware characteristics of the supercomputer. This could potentially provide insights into the source of the fluctuation and help identify any areas for optimization. Specifically, examining the memory access patterns can help identify potential issues with data locality or cache thrashing, which may be causing the contention. Additionally, examining the hardware characteristics such as cache size, interconnect speed, and memory bandwidth can help identify potential bottlenecks. Profiling tools such as perf may also be useful in identifying potential issues with memory access patterns or contention.

The results demonstrate the effectiveness of pthreads for parallelizing stencil computations, with the pthread implementation achieving significant speedup compared to the serial implementation. Speedup generally increased with the number of threads and problem size, but with diminishing returns indicating that further increases in the number of threads may not provide proportional performance improvements. While efficiency was highest when using larger problem sizes, a decreasing trend was observed as the number of threads increased.

It's important to note that various factors may have contributed to this trend, including contention for shared resources like memory or caches, and the overhead of thread creation and synchronization.

### C. MPI Results

Here we present the results of our experiments using MPI.



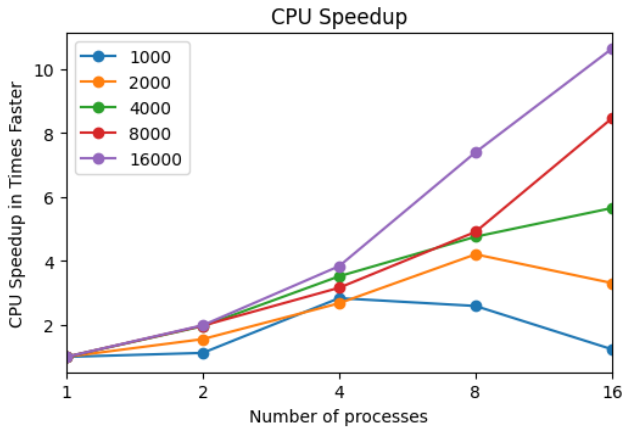

| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.105 | 0.982 | 0.389 | 0.426 | 0.892 |
| 2000 | 4.434 | 2.851 | 1.657 | 1.054 | 1.340 |
| 4000 | 17.770 | 9.060 | 5.049 | 3.736 | 3.144 |
| 8000 | 71.306 | 36.111 | 22.560 | 14.528 | 8.431 |
| 16000 | 284.350 | 143.072 | 74.122 | 38.451 | 26.740 |

Fig. 8: CPU times of MPI implementation for different problem sizes and number of processes.

Figure 8 shows the CPU times taken by the MPI implementation for different numbers of processes. The speedup achieved by the MPI implementation relative to the serial implementation is calculated using the following formula:

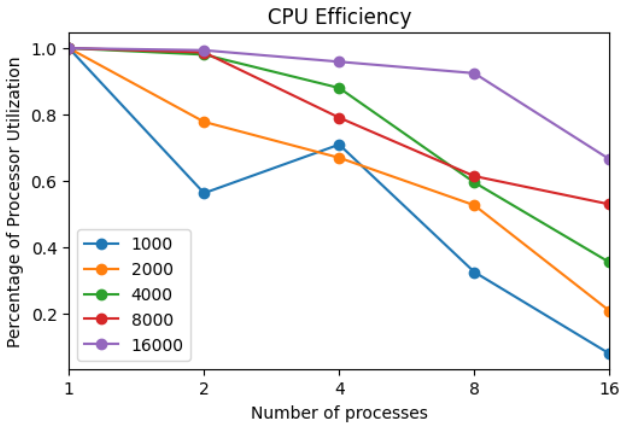
$$S_m = \frac{T_s}{T_m} \tag{3}$$

where $S_m$ is the speedup, $T_s$ is the time taken by the serial implementation, and $T_m$ is the time taken by the MPI implementation [2].




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.000 | 1.124 | 2.836 | 2.591 | 1.238 |
| 2000 | 1.000 | 1.555 | 2.676 | 4.206 | 3.308 |
| 4000 | 1.000 | 1.961 | 3.520 | 4.756 | 5.652 |
| 8000 | 1.000 | 1.975 | 3.161 | 4.908 | 8.457 |
| 16000 | 1.000 | 1.987 | 3.836 | 7.395 | 10.634 |

Fig. 9: CPU speedup of MPI implementation for different problem sizes and processes.




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.000 | 0.562 | 0.709 | 0.324 | 0.077 |
| 2000 | 1.000 | 0.777 | 0.669 | 0.526 | 0.207 |
| 4000 | 1.000 | 0.981 | 0.880 | 0.595 | 0.353 |
| 8000 | 1.000 | 0.987 | 0.790 | 0.614 | 0.529 |
| 16000 | 1.000 | 0.994 | 0.959 | 0.924 | 0.665 |

Fig. 10: Speedup efficiency of MPI implementation for different problem sizes and number of processes.

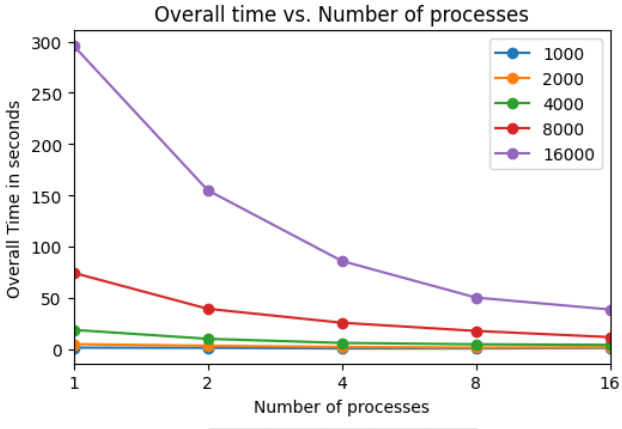
As shown in Figure 9, the speedup increases as the number of processes increases for all problem sizes. This is expected, as more processes allow for more parallelism and therefore faster execution. The speedup achieved by the MPI implementation is significant; however, it is important to note that the speedup is not linear with respect to the number of processes. As shown in the figure, there is a diminishing return as the number of processes increases beyond a certain point. This is due to factors such as communication overhead becoming more significant as more processes are used. Therefore, it is important to choose an appropriate number of processes to balance the benefits of parallelism with the overhead of communication.

The efficiency achieved by the MPI implementation is calculated using the following formula:

$$E_m = \frac{S_m}{p} \tag{4}$$

where $E_m$ is the efficiency, $S_m$ is the speedup, and $p$ is the number of MPI processes used [2].

As shown in Figure 10, the efficiency of the MPI implementation fluctuates as the number of processes increases for all problem sizes. This is due to various factors such as communication overhead, load balancing, and the nature of the problem being solved. These fluctuations in efficiency are commonly observed in many parallel computing environments, including the SDSC Expanse supercomputer used in our experiments. Therefore, it is important to carefully choose the number of processes and consider the characteristics of the problem being solved to achieve optimal efficiency.

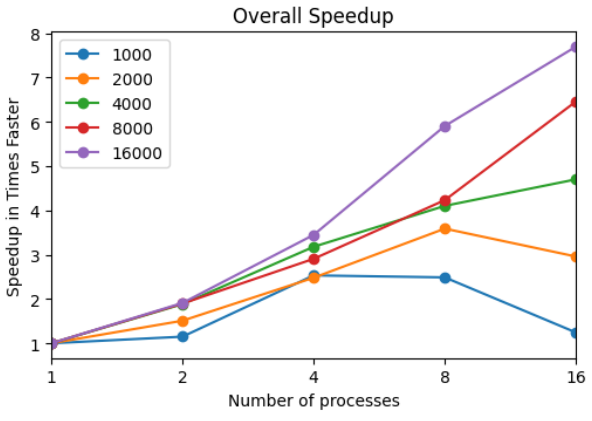



| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.217 | 1.059 | 0.481 | 0.489 | 0.978 |
| 2000 | 4.658 | 3.085 | 1.879 | 1.298 | 1.574 |
| 4000 | 18.561 | 9.843 | 5.839 | 4.524 | 3.948 |
| 8000 | 74.244 | 39.146 | 25.538 | 17.544 | 11.495 |
| 16000 | 295.961 | 154.881 | 85.843 | 50.117 | 38.447 |

Fig. 11: Overall execution times of MPI implementation for different problem sizes and number of processes.

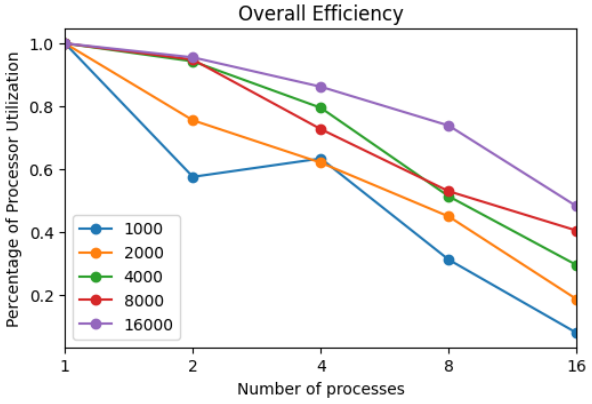
As seen in Figure 11, the trend in overall execution times across various problem sizes and number of is approximately the same as the trend for the CPU time. Because the program spends most of its time performing the stencil operation, this is to be expected.

Based on the observation that the trend in overall execution times across various problem sizes and number of processes is similar to the trend for the CPU time, it can be assumed that the program's rate of speedup and efficiency will also follow this trend.




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.000 | 1.150 | 2.533 | 2.488 | 1.244 |
| 2000 | 1.000 | 1.510 | 2.480 | 3.589 | 2.959 |
| 4000 | 1.000 | 1.886 | 3.179 | 4.102 | 4.701 |
| 8000 | 1.000 | 1.897 | 2.907 | 4.232 | 6.459 |
| 16000 | 1.000 | 1.911 | 3.448 | 5.905 | 7.698 |

(a) Overall speedup of MPI implementation for different problem sizes and number of processes.




| | 1 | 2 | 4 | 8 | 16 |
|---|---|---|---|---|---|
| 1000 | 1.000 | 0.575 | 0.633 | 0.311 | 0.078 |
| 2000 | 1.000 | 0.755 | 0.620 | 0.449 | 0.185 |
| 4000 | 1.000 | 0.943 | 0.795 | 0.513 | 0.294 |
| 8000 | 1.000 | 0.948 | 0.727 | 0.529 | 0.404 |
| 16000 | 1.000 | 0.955 | 0.862 | 0.738 | 0.481 |

(b) Overall efficiency of MPI implementation for different problem sizes and number of processes.

Fig. 12: Overall performance of MPI implementation for different problem sizes and number of processes.

Overall, these results demonstrate the effectiveness of MPI for parallelizing stencil computations, as the MPI implementation achieved significant speedup compared to the serial implementation. The speedup generally increased with the number of processes used and the size of the problem, though diminishing returns were observed. Efficiency was at its highest when using larger problem sizes, but still showed a trend of decreasing as the number of processes increased. It is important to note, however, that various factors, such as those mentioned above, may have contributed to this trend.

## V. Conclusions

In this project, we have implemented and compared two parallelization approaches for the stencil computation: pthreads and MPI. Our experiments were conducted on a shared-memory multicore system and a distributed-memory cluster, respectively.

### A. Pthreads

Our results show that the pthread implementation achieves significant speedup over the serial implementation, especially for larger problem sizes and with an appropriate number of threads chosen. However, it is important to carefully balance the benefits of parallelism with the overhead of thread management in order to achieve optimal performance, particularly given that drops in efficiency and scalability were observed.

### B. MPI

Our results show that the MPI implementation also achieves significant speedup over the serial implementation, especially for larger problem sizes and with an appropriate number of processes chosen. However, communication overhead becomes increasingly important as the number of processes increases, and therefore it is important to use communication-efficient algorithms and minimize the amount of data transferred between processes, particularly given that drops in efficiency and scalability were observed.

Overall, our results demonstrate the effectiveness of both pthreads and MPI in parallelizing the stencil computation, and highlight the importance of carefully balancing parallelism and communication overhead in order to achieve optimal performance.

Regarding the fluctuations in efficiency, and by extension, scalability, it is important to remember that the experiments were being performed on the Expanse supercomputer's shared partition, meaning that the performance of the program could be impacted by factors outside of our control, such as other users running resource-intensive jobs at the same time. The shared partition may also introduce additional overheads due to contention for shared resources like memory and I/O. Therefore, it is possible that the fluctuations in efficiency and scalability were partially caused by factors outside of our control.

Future work could explore other parallelization approaches, such as hybrid MPI and pthreads or OpenMP, and investigate the performance of these approaches on other hardware platforms.

## REFERENCES


[1] Kitware, "Paraview," 2023. [Online]. Available: https://www.paraview.org/

[2] P. Pacheco, *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.