2D Parallel Heat Transfer Simulation

Justin LaForge
Student at Coastal Carolina Univ.
Conway, SC
jmlaforge@coastal.edu

Megan Hickman Fulp
Coastal Carolina Univ. Prof
Conway, SC
mlhickman@coastal.edu

Kyle Wallace
Student at Coastal Carolina Univ.
Conway, SC
kwwallace@coastal.edu

***Abstract*—This paper explores the parallelization of a heat transfer simulation program using PThreads OpenMP, and MPI to improve runtime of heavily computational code. This project starts by creating individual programs for each form of parallelization and one large program, which utilizes all forms, and then recording data such as overall time, work time, and overhead time.**

I. Introduction

Heat transfer is a basic physical process that happens in many natural and man-made systems. Examples include heat moving through solid materials, cooling in electronic devices, and the way temperature spreads through the air or water in the environment. Simulating heat transfer on a computer helps us understand how heat behaves in different situations, especially when it's too hard or impossible to solve by hand.

In this project, we simulate heat transfer through a two-dimensional square surface. The left and right sides of the grid are kept hot at a constant temperature, while the top and bottom edges are kept cold. Over time, the heat flows from the hot side towards the cold ones.

To simulate the temperature changes, we used a technique called a nine-point stencil. This means that each point in the grid is updated based on the values of its eight neighboring points plus itself. This method helps give a smoother and more accurate result than simpler methods that use fewer neighbors. The simulation runs through many steps, showing how the heat spreads over time.

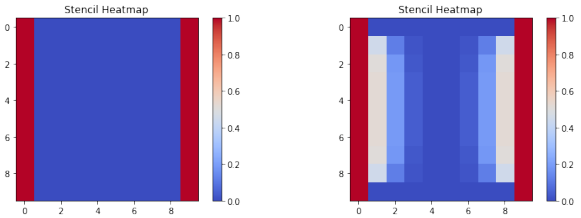
We used several ways to implement this simulation. Those being Serial (one processor), and the rest using different parallel programming models: Pthreads, OpenMP, MPI, and a hybrid including the previously mentioned parallel programming models. All versions were tested using a fixed grid of size (5000×5000 to make performance comparisons fair. The goal is to see how fast each version runs and how well they scale with more threads or processes, as well as analyzing the speedup and efficiency of each parallel version compared to the serial baseline. This helps us determine how well the program uses its available hardware resources and how much benefit we gain from the parallelization of the program. We will visualize the results through performance plots, which will help guide the choice between parallel methods for stencil-based scientific computing.

II. Background

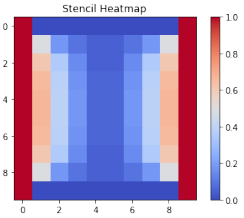
the Simulation of heat transfer is a common task in scientific computing and engineering. simulating heat transfer helps researchers analyze heat distribution in real-world applications such as material science (e.g., thermal conductivity of composites), meteorology (e.g., temperature distribution in the atmosphere), and electronics (e.g., heat dissipation in microchips).

To visualize how heat flows across the grid over time, we generated a series of heat map images that represent the temperature transfer at different stages of the simulation.

Figure 1 shows three snapshots from the simulation: the initial state with hot and cold edges, an intermediate state as heat begins to spread, and the final steady-state distribution where equilibrium is reached.



(a) Initial state (b) Midway



(c) Final state

Fig. 1: Heat map snapshots at different time steps. Red = high temperature, blue = low temperature.

in this project, we use a nine-point stencil to approximate the temperature at each point on a 2D grid. A stencil is a pattern that determines how the value at a specific grid cell is calculated based on its neighbors. The nine-point stencil incorporates its eight neighbors unlike a five point stencil where it only uses its four neighbors. making the resulting calculation more accurate.

To better understand this, consider the following illustrations of a five-point and nine-point stencil pattern, where X represents the grid point being updated:

Five-point stencil:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & X & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Nine-point stencil:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & X & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

While the five-point stencil requires less work, the nine-point stencil provides a more accurate representation of heat transfer.

In this project, we explore different models of parallel computing:

- **Pthreads**: A threading API that allows programs to manage multiple concurrent execution flows.
- **OpenMP**: An API that enables shared-memory parallel programming
- **MPI (Message Passing Interface)**: enables efficient communication between processes in distributed-memory systems
- **Hybrid Approach**: A combination of MPI, OpenMP, and Pthreads to leverage both shared and distributed memory architectures.

By applying these parallel programming models, we aim to reduce the overall computation time and analyze how well each model utilizes system resources. Understanding these trade-offs is crucial for high-performance computing applications that involve large-scale simulations.

III. Design

The project started with creating a program, make-2d.c, which generates a stencil matrix, i.e an nxn matrix with 1's on the left and right most columns and 0's everywhere else. This program takes command line arguments to allow the user to decide the size of the matrix and the output file name. Next print-2d.c was implemented to print a given stencil matrix to the screen in a nice format to debug and determine the accuracy of the results.

To simulate a heat transfer stencil program in parallel, five programs were made, one program for the serial version of the stencil simulation, parallel versions for Pthreads, OpenMP, MPI, and one program that utilizes all of these forms of parallelization. The following programs were made:

- stencil-2d.c: Serial version of stencil simulation
- stencil-2d-pth.c: Parallel stencil simulation using Pthreads.
- stencil-2d-omp.c: Parallel stencil simulation using OpenMP.
- stencil-2d-mpi.c: Parallel stencil simulation using MPI.
- stencil-2d-hybrid.c: Merged parallel stencil simulation using MPI, OpenMP, and Pthreads.

A. Serial Implementation

The serial code for the heat transfer simulation will define the process for the rest of the programs and will work as the golden standard that we compare with the other results.

This program creates a copy of the stencil matrix and then uses it to index through all indices, avoiding the edges, and performs the 9-Point stencil operation using a double-nested for loop adding the new calculated values to the copy of the matrix, this will avoid overwriting the data we are using to calculate. This double nested for loop is also nested within another for-loop which acts as the iterations, or time-steps for the heat map, making this program a triple nested for loop. This technique works, however it runs slowly, making it a perfect candidate for parallelization. An example of the pseudo-code for serial stencil simulation is as follows:

Algorithm 1 2D Stencil Update

for $o \leftarrow 1$ to n **do**
  for $i \leftarrow 1$ to $rows - 2$ **do**
    for $j \leftarrow 1$ to $cols - 2$ **do**
      $new[i][j] \leftarrow \cdot($
  $old[i-1][j-1] + old[i-1][j] + old[i-1][j+1] +$
  $old[i][j-1] + old[i][j] + old[i][j+1] +$
  $old[i+1][j-1] + old[i+1][j] + old[i+1][j+1]$
  $) \cdot \frac{1}{9}$
    end for
  end for
  Swap(old, new)
end for

After the operation is performed on the matrix, data such as overall time, and work time along with the resulting stencil matrix are all recorded to allow for scientific analysis and data collection.

B. Pthreads Implementation

Our first parallel program is implemented using Pthreads. To use pthreads we must create a separate function which our parallel threads will use to perform their calculations on the stencil. This function will break up the stencil matrix into smaller chunks, or smaller rows, and divide the work among the separate threads, giving each thread it's unique section to work on, and once all threads finish their calculations, the 0th thread will take all the results and put that matrix back together, preparing the program for the next iteration.

Algorithm 2 Pthread Stencil Worker Thread

$local\_start \leftarrow$ BLOCK_LOW$(id, num_threads, rows - 2) + 1$
$local\_end \leftarrow$ BLOCK_HIGH$(id, num_threads, rows - 2) + 1$
for $iter \leftarrow 1$ to n **do**
  for $i \leftarrow local\_start$ to $local\_end$ **do**
    for $j \leftarrow 1$ to $cols - 2$ **do**
      $new[i][j] \leftarrow ($
    $old[i-1][j-1] + old[i-1][j] + old[i-1][j+1] +$
    $old[i][j-1] + old[i][j] + old[i][j+1] +$
    $old[i+1][j-1] + old[i+1][j] + old[i+1][j+1]$
    $) \cdot \frac{1}{9}$
    end for
  end for
  PTHREAD_BARRIER_WAIT(barrier)
  SWAP(old, new)
  PTHREAD_BARRIER_WAIT(barrier)
  Update local matrix pointers
end for

The macros, BLOCK_LOW and BLOCK_HIGH, calculate the thread's assigned row range, adjusting for the region each thread is responsible for. Synchronization between threads is enforced using pthread_barrier_wait to ensure all threads finish their computations before swapping the input matrix and output matrix pointers. This guarantees consistent memory state across threads. This structure succeeds in parallelizing the stencil program while maintaining correctness in a shared memory environment.

C. OpenMP Implementation

Algorithm 3 OpenMP Stencil Code

Similar to the Pthreads implementation, OpenMP separates the matrix into different portions to split among threadas, however does so in a more abstract manner using #pragma statements from the OpenMP library as follows:

#pragma omp parallel
for $o \leftarrow 1$ to n **do**
  #pragma omp for collapse(2)
  for $i \leftarrow 1$ to $rows - 2$ **do**
    for $j \leftarrow 1$ to $cols - 2$ **do**
      $new[i][j] \leftarrow \cdot($
    $old[i-1][j-1] + old[i-1][j] + old[i-1][j+1] +$
    $old[i][j-1] + old[i][j] + old[i][j+1] +$
    $old[i+1][j-1] + old[i+1][j] + old[i+1][j+1]$
    $) \cdot \frac{1}{9}$
    end for
  end for
  #pragma omp single
  Swap(old, new)
end for

This OpenMP based stencil code performs the 9-point stencil operation on a 2D matrix over multiple iterations. The #pragma omp parallel directive creates a team of threads. For each iteration, the #pragma omp for collapse(2) directive parallelizes the nested for loops over matrix indices i and j, allowing multiple threads to compute different portions of the matrix simultaneously. To prevent race conditions during the matrix swapping step, the #pragma omp single directive ensures that only one thread swaps the old matrix and new matrix pointers. This version of the stencil code parallelizes the algorithm using OpenMP while ensuring race conditions to not occur and data remains accurate.

D. MPI Implementation

The MPI Implementation of our stencil program gets a little more complex as MPI breaks the code down into different processes that run on different cores as opposed to threads. To do this we use the MPI functions such as MPI_Init to establish connections between the different cores and generate an "MPI_COMM_WORLD", and then using functions like MPI_Send, MPI_Recv, MPI_Gather and MPI_Scatter to communicate data between the cores. The program first generates a connection between the cores and establishes how many cores and processes we will be running. It then uses MPI_Bcast to share the rows and cols information among the other processes. The program then splits the matrix into different subsections to send out to the other processes, using MPI_Scatter, to perform their calculations and then use MPI_Gather to pull the data back together in the end. Using MPI_Barrier when needed to ensure all other processes have finished computing their parts before the others move onto the next iterations.

Algorithm 4 Simplified MPI Matrix Averaging

Initialize MPI
Get `rank`, `size` from MPI
if rank == 0 **then**
  Read input matrix from file
end if
MPI_Bcast matrix dimensions to all processes
Compute local row count for each process
MPI_Scatter portions of global matrix to each process
Copy data to local working matrix
for each iteration **do**
  Exchange ghost rows with neighbors (non-blocking)
  Wait for all communication to complete
  for each local row (excluding halos) **do**
    if global row is first or last **then**
      Skip
    else
      for each column (excluding borders) **do**
        Compute 9-Point stencil process
      end for
    end if
  end for
  Swap matrix pointers
end for
MPI_Gather all local sub matrices to root process
Finalize MPI

E. Hybrid Implementation

The Hybrid implementation is the most complex of them them all as it uses a combination of MPI, OpenMP and Pthreads to create one super parallelized program. This program first creates an MPI world to generate a connection between cores and then breaks the rows of the matrix into different sections to send off to different cores, next each core takes its part of the matrix and breaks the rows down even further to be parallelized in OpenMP threads, finally each of those threads will break the columns down into different threads that will use Pthreads functions to finally calculate all the data. In the end all threads put their pieces back together to eventually create the full matrix again before moving onto the next iteration. Visually we can represent this as follows:

Big Matrix:
- MPI process 0 handles rows 0–199
  - OpenMP thread 0 → part of the rows
  - OpenMP thread 1 → part of the rows
    - pthreads split columns
- MPI process 1 handles rows 200–399
  - OpenMP thread 0
  - OpenMP thread 1
    - pthreads split columns

...

A simplified process of the program in pseudo code is as shown in Algorithm 5.

It is important to note that in reality, most hybrid codes only do MPI plus OpenMP because OpenMP already abstracts thread management, making pthreads unnecessary. Adding Pthreads on top is rarely needed but for testing and experimental purposes, this hybrid code utilizes all these forms of parallelization.

This section demonstrated the evolution of the stencil heat transfer simulation from a basic serial implementation to increasingly complex and efficient parallel versions using Pthreads OpenMP and MPI. Beginning with the generation and display of the matrix, we established a consistent structure upon which all simulations operate. The serial version served as the foundation and benchmark, while the Pthreads and OpenMP implementations showcased two approaches to parallelism in shared-memory environments using threads. The MPI implementation introduced distributed memory and interprocess communication using cores instead. Finally, the hybrid approach combined MPI, OpenMP, and Pthreads to maximize parallelization. Together, these implementations highlight the trade offs in performance, complexity, and resource management involved in parallelizing scientific computations.

Algorithm 5 Simplified Hybrid Parallel Stencil Process

```
Initialize MPI
Get rank, size from MPI
MPI_Bcast matrix dimensions to all processes
Compute local row count for each process
MPI_Scatter portions of global matrix to each process
Copy data to local working matrix
for each iteration do
    Exchange ghost rows with neighbors (non-blocking)
    Divide rows even further for OpenMP
    #pragma omp parallel for
    for each local row (excluding halos) do
        pthread_t pthreads[p]
        split [1, cols-2] columns among p threads
        for each thread do
            call pthread function
            Inside pthread function
            for each column (excluding borders) do
                Compute 9-Point stencil process
            end for
        end for
        for each thread do
            Join pthreads
        end for
    end for
    Swap matrix pointers
end for
MPI_Gather all local sub matrices to root process
Finalize MPI
```

IV. Experimental Evaluation

Multiple tests were ran on the matrices, first running the serial code to create the "correct" matrix, and then comparing results of the parallel version to ensure our parallel results are correct. To evaluate each parallel method, we compared their runtime, speedup, and efficiency across all methods (OpenMP, MPI, Pthreads, and Hybrid) and matrix sizes. We wanted to identify which method scaled best with increasing processor could and matrix size. along with determining their performance against the serial baseline.

For each experiment, we recorded the following:

- **T_overall**: Total runtime of the simulation.
- **T_computation**: Time spent actively doing computation.
- **T_other**: Derived as $T\_other = T\_overall - T\_computation$, representing communication, synchronization, and overhead.
- **Speedup**: $S = \frac{T_{serial}}{T_{parallel}}$
- **Efficiency**: $E = \frac{S}{p} \times 100\%$

A. Test Serial Code

Once the programs were finished, we ran tests on the serial algorithm to ensure accuracy. To test our program, we created a 5x5 stencil matrix using the make-2d.c program:

$$\begin{bmatrix} 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \end{bmatrix}$$

The matrix has a value of 1.00 on the leftmost and rightmost columns, and 0.00 everywhere else. This setup mimics fixed boundary conditions for heat transfer, where 1 represents the

hot values and 0 represents the cold values. Now we perform the stencil operation on this matrix with 3 iterations or time steps to show how the process effects the matrix:

$$\begin{bmatrix} 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.47 & 0.23 & 0.47 & 1.00 \\ 1.00 & 0.53 & 0.34 & 0.53 & 1.00 \\ 1.00 & 0.47 & 0.23 & 0.47 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \end{bmatrix}$$

After 3 iterations we can see the heat has successfully started transferring to the center of the stencil while the edges remain the same.

After running the serial stencil simulation for 14 iterations, the resulting matrix shows how the interior values have evolved. The center of the matrix now contains gradually increasing floating-point values, reflecting the averaging (diffusion) process from the fixed boundaries inward using a 9-point stencil. The higher values in the center (e.g., 0.68) indicate the accumulation of influence from the boundary cells over successive iterations.

To determine a best fit number of iterations for our program we ran tests to determine which number of iterations would make the serial code for the 40k by 40k matrix run for 4 minutes, and then used this value as our set number of iterations for the rest of the tests. After running the code at 8, 10, 12, and 14 iterations, our resulting number was 14. 2 represents the time it took the serial code to run at 14 iterations.

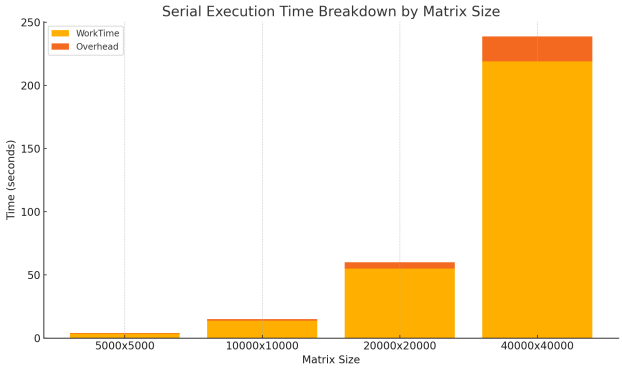


Fig. 2: Serial Time.

As seen in the graph, the 40k matrix is running right at 4 minutes. This helped us choose 14 iterations as the baseline for the rest of the results.

B. Comparing Output Matrices

Before performing large calculations on the parallel versions, we first ran small experiements to ensure they are accurate. Each program, stencil-2d.c, stencil-2d-pth.c, stencil-2d-omp.c, stencil-2d-mpi.c, stencil-2d-hybrid.c, performed their calculations on a 5x5 matrix using 14 threads, and here were the results:

Serial Output:

$$\begin{bmatrix} 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.58 & 0.38 & 0.58 & 1.00 \\ 1.00 & 0.68 & 0.56 & 0.68 & 1.00 \\ 1.00 & 0.58 & 0.38 & 0.58 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \end{bmatrix}$$

Pthread Output:

$$\begin{bmatrix} 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.58 & 0.38 & 0.58 & 1.00 \\ 1.00 & 0.68 & 0.56 & 0.68 & 1.00 \\ 1.00 & 0.58 & 0.38 & 0.58 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \end{bmatrix}$$

OpenMP Output:

$$\begin{bmatrix} 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.58 & 0.38 & 0.58 & 1.00 \\ 1.00 & 0.68 & 0.56 & 0.68 & 1.00 \\ 1.00 & 0.58 & 0.38 & 0.58 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \end{bmatrix}$$

MPI Output:

$$\begin{bmatrix} 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.58 & 0.38 & 0.58 & 1.00 \\ 1.00 & 0.68 & 0.56 & 0.68 & 1.00 \\ 1.00 & 0.58 & 0.38 & 0.58 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \end{bmatrix}$$

Hybrid Output:

$$\begin{bmatrix} 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \\ 1.00 & 0.58 & 0.38 & 0.58 & 1.00 \\ 1.00 & 0.68 & 0.56 & 0.68 & 1.00 \\ 1.00 & 0.58 & 0.38 & 0.58 & 1.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 1.00 \end{bmatrix}$$

As we can see all output is equal meaning our parallel programs are working as intended.

C. Program Runtime

To evaluate the performance of our stencil computation implementations, we compare the overall runtime, computational time, and overhead time across five different versions of the program: Serial, PThreads, OpenMP, MPI, and Hybrid (MPI + OpenMP). The overhead is defined as the difference between the total runtime and the time spent purely on computation, accounting for factors such as thread management, communication, and synchronization.

We conduct this analysis using two matrix sizes—5000×5000 and 10000×10000—to observe how each approach scales with problem size. This comparison highlights not only the raw speed differences but also the efficiency of parallelization strategies and their associated overheads.

Our first data is recorded from the 5000x5000 matrix computations. Figure 3 shows the runtime for the Pthreads program.

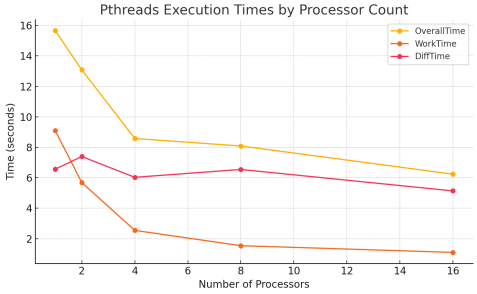


Fig. 3: Pthreads Time.

Our results show data that is quite expected coming from parallelization with pthreads, overhead remains pretty stable throughout the threads and as threads increase the times decrease.

Figure 4 shows the runtime for the OpenMP program.

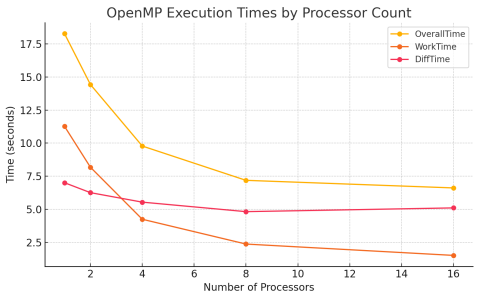


Fig. 4: OpenMP Time.

Similarly, OpenMP runs as expected too, with a steady bit of overhead and decreasing runtime as threads increase.

Figure 5 shows the runtime for the MPI program.

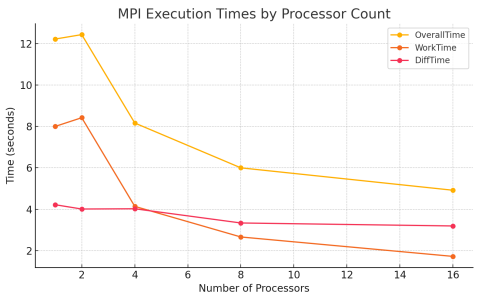


Fig. 5: MPI Time.

MPI gives interesting results as the difference between using 1 thread and 2 threads is actually increasing, meaning that it is better to run MPI only starts improving speed after using 2 threads.

Figure 6 shows the runtime for the Hybrid program.

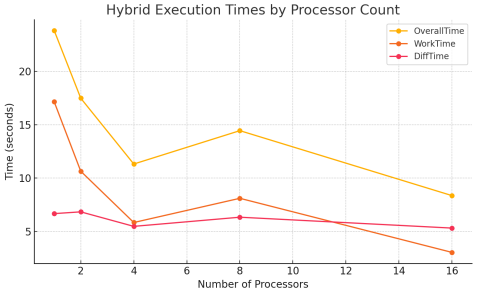


Fig. 6: Hybrid Time.

When looking at our Hybrid runtime, we see it has a significant amount of runtime in the begining while drastically falling, only to rise again around 8 threads.

Finally we created a graph, figure 7, to show the runtime for the overall execution time for all programs.

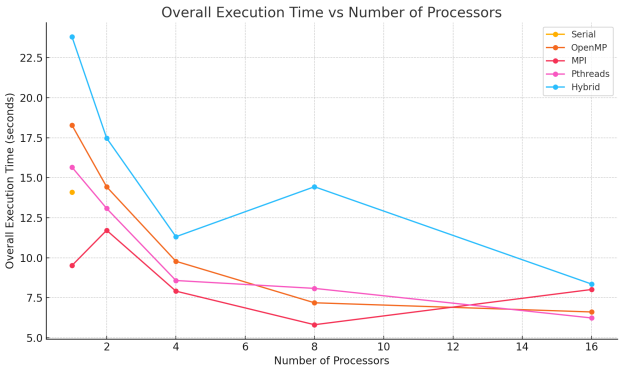


Fig. 7: All Programs' Overall Time.

Looking at the overall charts we can see that our hybrid program is actually running significantly slower overall than the rest of the programs, showing that even though we have more parallelization, we do not have a faster program. Another thing to note is that MPI seems to perform the best out of the programs overall however after 8 cores it starts to slow down.

D. Program Speedup

In this section, we examine the speedup achieved by each parallel implementation—Pthreads, OpenMP, MPI, and Hybrid—when run on a 5000×5000 matrix across varying processor counts (1, 2, 4, 8, and 16). Speedup is a fundamental performance metric in parallel computing that measures how much faster a program runs on multiple processors compared to a single processor. Ideal speedup increases linearly with the number of processors; however, in practice, factors such as communication overhead, synchronization delays, and workload imbalance can impact scalability.

By analyzing the overall speedup trends for each model, we gain insights into how efficiently each implementation utilizes available computing resources, and how well they scale as we increase parallelism. The results presented highlight both the strengths and limitations of each approach, offering a comparative perspective on their real-world performance.

Figure 8 shows the efficiency of all programs across 16 threads for a 5000x5000 matrix.

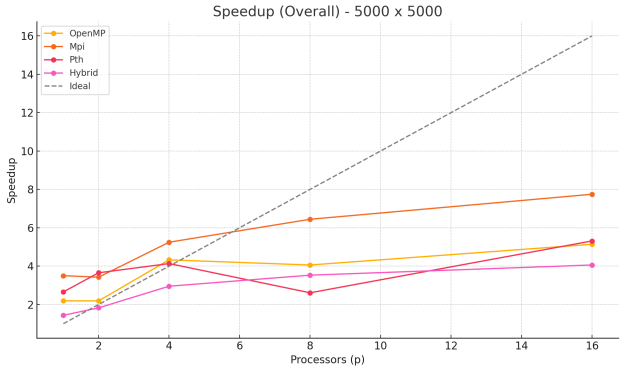


Fig. 8: Overall Speedup.

The speedup graph provides a clear view of how each implementation scales as additional processors are introduced. MPI demonstrates the strongest and most consistent speedup across the board, reaching approximately 7.7× speedup at 16 threads—well above the other models and suggesting that MPI handles workload distribution and communication with relatively low overhead. OpenMP follows with moderate but steady performance, leveling off slightly past 4 processors, which is typical in shared memory environments where contention and cache coherence start to limit scaling benefits. Pthreads show competitive speedup up to 4 threads but then drop at 8 before recovering slightly at 16, indicating that performance may have been impacted by thread management or resource contention at mid-level thread counts. The Hybrid implementation achieves the lowest speedup overall, maxing out just over 4× at 16 processors. This likely stems from compounded overhead introduced by coordinating both MPI and OpenMP layers, leading to diminishing returns as more processors are added.

Across all implementations, none reach ideal linear scaling, which is represented by the dashed line. This emphasizes the limits imposed by Amdahl's Law and the inherent overhead of parallel execution. However, the results still illustrate meaningful performance gains, particularly from MPI, which stands out as the most scalable and effective approach for this problem size.

E. Program Efficiency

To further evaluate the scalability and effectiveness of each parallel implementation, we analyze the efficiency of the PThreads, OpenMP, MPI, and Hybrid (MPI + OpenMP) versions of the stencil program. Efficiency is calculated as the speedup divided by the number of threads, providing insight into how well each approach utilizes available computational resources.

Like before, all programs were executed using a fixed matrix size of 5000×5000 across varying thread counts: 1, 2, 4, 8, and 16. This section compares the resulting efficiency values to identify diminishing returns, overhead costs, and which parallelization models maintain better performance as concurrency increases. Figure 9 shows the efficiency of all programs across 16 threads for a 5000x5000 matrix.

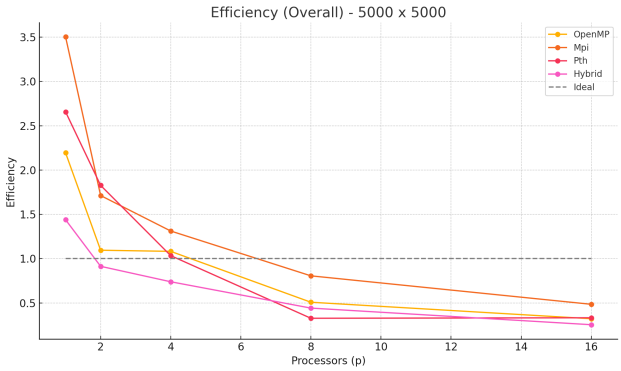


Fig. 9: Overall Efficiency.

The overall efficiency results for the 5000×5000 matrix reveal how each parallel implementation scales as the number of threads increases from 1 to 16. Notably, MPI and Pthreads both exhibit superlinear efficiency at 1 thread, which suggests their single-threaded performance outpaced the serial baseline. As the processor count increases, all implementations show a decline in efficiency, which is expected due to overhead from thread management, synchronization, and the non-parallelizable portions of the code, as described by Amdahl's Law.

Among all implementations, MPI consistently maintains the highest efficiency across all thread counts, maintaining over 50% efficiency even at 16 threads. This suggests good scalability and low overhead in the distributed communication model used by MPI. OpenMP and Pthreads follow similar patterns, with moderate efficiency at lower thread counts but noticeable drops beyond 4 or 8 threads, potentially due to contention in shared memory environments. The Hybrid model however, performs the worst in terms of efficiency, starting below other methods and dropping to around 30% by 16 threads. This indicates that the added complexity and synchronization between all layers introduces substantial overhead, outweighing the potential benefits of combining all approaches.

V. Conclusion

In this project, we utilized a stencil-based heat transfer simulation with four approaches: Pthreads, OpenMP, MPI, and a Hybrid combining all three methods. We used a serial baseline to compare and test for correctness. We did this by comparing smaller output matrices to verify that all stencil programs produced the same result.

Through our analysis we saw that MPI consistently demonstrated the best scalability and highest efficiency, especially for larger matrices. This is likely because of its ability to split the computation among multiple nodes. OpenMP and Pthreads also did well in efficiency they experienced higher

diminishing returns with larger processor counts. While the hybrid implementation preformed worse than expected, as one would assume the hybrid being the most parallel, would produce better runtime. This wasn't the case with the hybrid version, producing slower runtime and lower efficiency. Maybe if we tested it with a far larger dataset, we could see an increase in performance, as it would be able to divide up the work more than the other methods.

overall, this project showed the advantages and disadvantages when using parallelization to compute a nine-point stencil in a 2D heat transfer simulation.

VI. References

Source 1: wiki-stencil, author = Wikipedia contributors, title = Nine-point stencil — Wikipedia, The Free Encyclopedia, year = 2024, howpublished = https://en.wikipedia.org/wiki/Nine-point_stencil, note = Accessed: 2025-04-30

Source 2: ghost-cell, author = Fredrik Kjolstad, title = The Ghost Cell Pattern, year = 2014, howpublished = https://fredrikbk.com/publications/ghost_cell_pattern.pdf, note = Accessed: 2025-04-30

Source 3: barrier-sync, author = Jay Desai, title = Barrier Synchronization in Threads, year = 2018, howpublished = https://medium.com/@jaydesai36/barrier-synchronization-in-threads-3c56f947047, note = Accessed: 2025-04-30