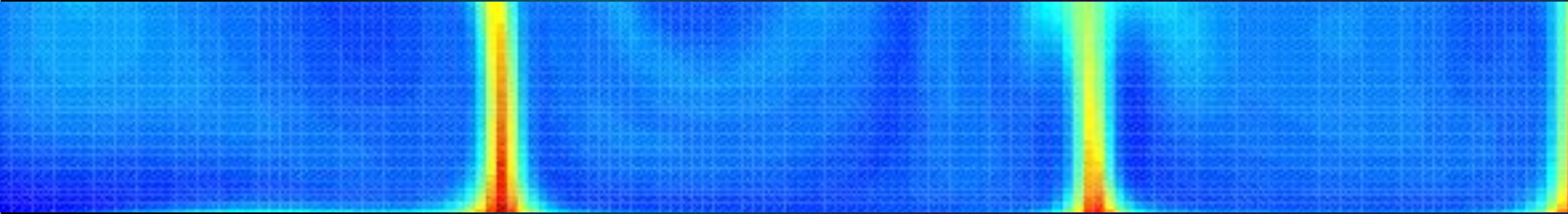
2D Parallel Heat Transfer Simulation



04.30.25
CSCI_473: Parallel Systems
Coastal Carolina University

PRESENTATORS: Justin LaForge, Kyle Wallace

Professor: Megan Hickman Fulp

Introduction

This project simulates heat transfer across a 2D square surface using a nine-point stencil method, where each grid point updates based on its neighbors. The left and right edges are kept hot, while the top and bottom are cold, causing heat to flow over time.

To experiment with parallelization, we implemented the simulation using both serial and parallel approaches—Pthreads, OpenMP, MPI, and a hybrid. Each version was tested on large grids (5k x 5k, 10k x 10k, 20k x 20k, 40k x 40k) to compare speed, scalability, and efficiency. Performance plots help show how well each method uses computing resources.

Background

Why Simulate Heat Transfer?

- Common in science and engineering experiments
- Helps model real-world heat flow, like:
    - Heat in materials (e.g., thermal composites)
    - Weather systems (e.g., atmospheric temps)
    - Electronics (e.g., microchip cooling)

How We Visualize It:

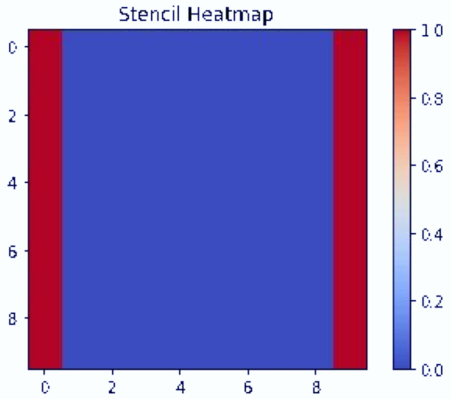
- Series of heat maps show how temperature changes over time:
  - Initial: Hot and cold edges
  - Midway: Heat begins to spread
  - Final: Reaches thermal equilibrium

1.0	0.0	0.0	0.0	1.0
1.0	0.0	0.0	0.0	1.0
1.0	0.0	0.0	0.0	1.0
1.0	0.0	0.0	0.0	1.0
1.0	0.0	0.0	0.0	1.0

Iteration 0

1.0	0.0	0.0	0.0	1.0
1.0	0.6	0.4	0.6	1.0
1.0	0.7	0.6	0.7	1.0
1.0	0.6	0.4	0.6	1.0
1.0	0.0	0.0	0.0	1.0

Iteration 14



The Stencil Method

What is a Stencil?

- A pattern used to update each cell based on its neighbors
- Nine-point stencil uses all 8 neighbors
- More accurate than the simpler five-point stencil

Five-point:
[0 1 0]
[1 **X** 1]
[0 1 0]

Nine-point:
[1 1 1]
[1 **X** 1]
[1 1 1]

Parallel Models

Why Use Parallel Computing?

- Heat transfer simulations are computationally intensive
- Parallelism speeds up processing of large grids

Parallel Approaches Explored:

- **Pthreads** – Manages threads manually for concurrency
- **OpenMP** – Simplifies shared-memory parallelism
- **MPI** – Efficient for distributed-memory systems
- **Hybrid** – Combines all three to use all available resources

Goal:

- Reduce runtime
- Evaluate resource usage
- Understand trade-offs in performance

Design

We began by creating tools to generate and print a 2D stencil matrix. The main simulation was developed in five versions:

Serial, Pthreads, OpenMP, MPI, and a hybrid of all three.

This setup allows us to test and compare different parallelization strategies for heat transfer simulation.

Design Summary

Matrix generation: make-2d.c, print-2d.c

Simulation programs:

- Serial: stencil-2d.c
- Pthreads: stencil-2d-pth.c
- OpenMP: stencil-2d-omp.c
- MPI: stencil-2d-mpi.c
- Hybrid: stencil-2d-hybrid.c

Overall Process

Create Stencil Matrix

Create a program that will make stencil matrix of sizes NxN.

In this implementation we will have matrices of NxN where N is 5000 10000 20000 40000

Serial Heat Transfer

Create a serial program to perform 9-point stencil process on the given matrices to produce a final stencil.

The Golden Standard for Heat Transfer, the output from other programs will be compared to the output from this.

Parallel Multiplication

Create four parallel programs to perform 9-point stencil process, one for each PThreads, OpenMP, MPI and Hybrid

Thread count for each program will be 1 2 4 8 16

Compare Results

Compare matrix results from all parallel versions with that of the serial program.

Taking the runtime, speedup, and efficiency from all different implementations, compare the results.

Serial Heat Transfer

- Provides the baseline for the speedup and efficiency analysis.
- Copy Matrix to prevent overwriting current values.
- Applies the 9-point stencil in a triple-nested loop (rows × cols × time steps)
- Pseudo code is provided --- >

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

Pthread Implementation

- Each thread runs a worker function on a chunk of matrix rows
- Work is divided using BLOCK_LOW and BLOCK_HIGH macros
- Threads update their section of the matrix using the 9-point stencil
- Barriers ensure synchronization before and after each iteration
- Thread 0 merges results and prepares for the next time step
- Maintains accuracy and consistency in shared memory

Algorithm 2 Pthread Stencil Worker Thread

$local\_start \leftarrow \mathrm{BLOCK\_LOW}(id, num\_threads, rows - 2) + 1$
$local\_end \leftarrow \mathrm{BLOCK\_HIGH}(id, num\_threads, rows - 2) + 1$
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

OpenMP Implementation

- Uses #pragma omp parallel to create threads
- #pragma omp for collapse(2) parallelizes the nested loops over the matrix
- Threads compute different regions of the matrix simultaneously
- #pragma omp single ensures only one thread handles matrix swapping
- Avoids race conditions while maintaining correct results

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

MPI Implementation

- Uses separate processes instead of threads for parallelization
- MPI_Init and MPI_COMM_WORLD establish communication between processes
- Matrix is divided and distributed using MPI_Scatter
- Each process computes its part of the stencil
- MPI_Gather reassembles the results
- MPI_Bcast shares configuration; MPI_Barrier ensures synchronization
- Enables distributed memory parallelism across cores

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
```

Hybrid Implementation

The Hybrid process is a complex process which divides the matrix into multiple sections to send out to the cores and then splits these sections up to have OpenMP and Pthread threads run operations on them.

An visual example of this is represented 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
 │          └── (optional) pthreads split columns
 │
 ├── MPI process 1 handles rows 200-399
 │      ├── OpenMP thread 0
 │      ├── OpenMP thread 1
 │          └── (optional) pthreads
 │
 ...
```

Hybrid Process

- Combines all three models for maximum parallelism
- **MPI** distributes matrix chunks across processes
- Each **MPI process** uses **OpenMP** to parallelize row sections
- Each **OpenMP thread** uses **Pthreads** to divide column work
- All partial results are reassembled before the next iteration
- Rare in practice due to complexity, but useful for testing scalability

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
```

Experimental Evaluation - Overview

- This section evaluates and compares all implementations based on runtime, speedup Metrics Collected:
    - Total runtime (T_overall)
    - Computation time (T_computation)
    - Overhead time (T_other = T_overall - T_computation)
    - Speedup (S = T_serial / T_parallel)
    - Efficiency (E = S / p)

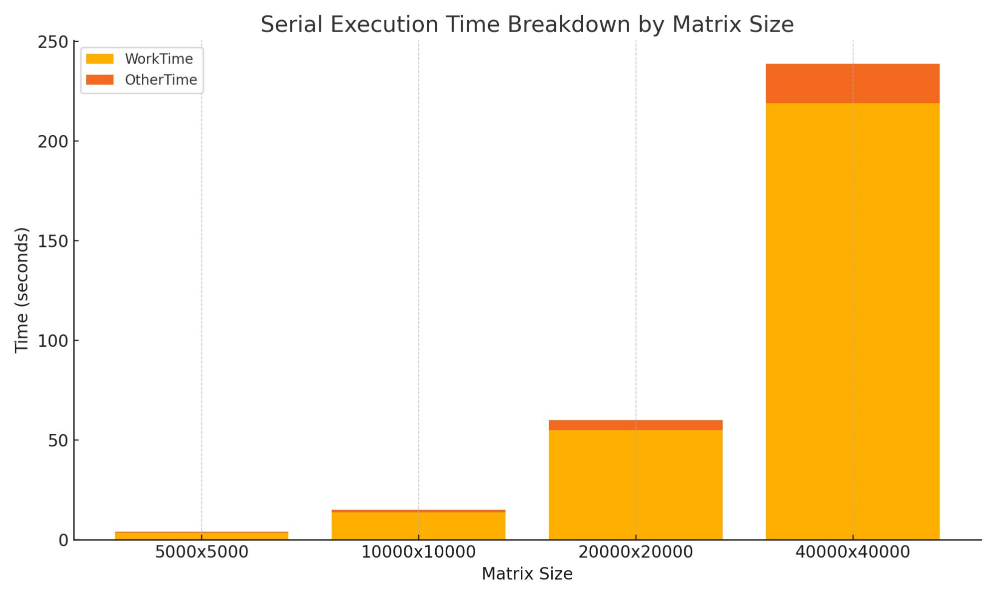
- Test Conditions:
    - Matrix sizes: 5000x5000
    - Threads: 1, 2, 4, 8, 16
    - Standard iteration count: 14

- Purpose:
    - Identify which model offers the best performance and scalability.
    - Understand trade-offs in complexity vs speedup, and efficiency.

Experimental Evaluation - Serial

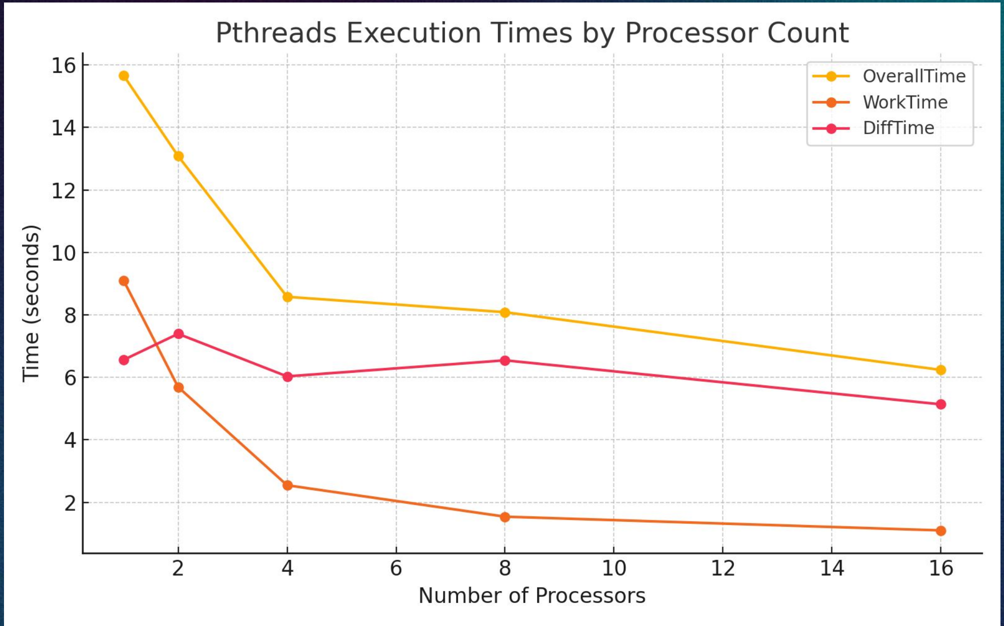
Choosing iterations count

- Target: ~4-minute runtime for serial on 40k matrix
- tested 8, 10, 12, 14 iterations
- 14 iterations was closest to 4-minute runtime



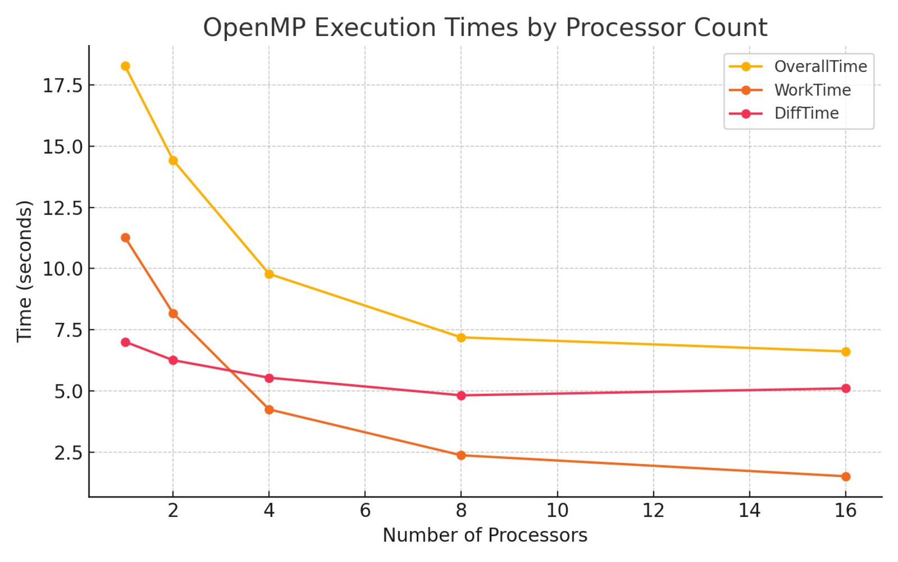
Experimental Evaluation - PThreads

- Great performance improvement up to 4 processors
- It starts evening out after 4 processors
- The overhead (DiffTime) remains relatively stable throughout
- Overall time decreases as worktime decreases which is expected.



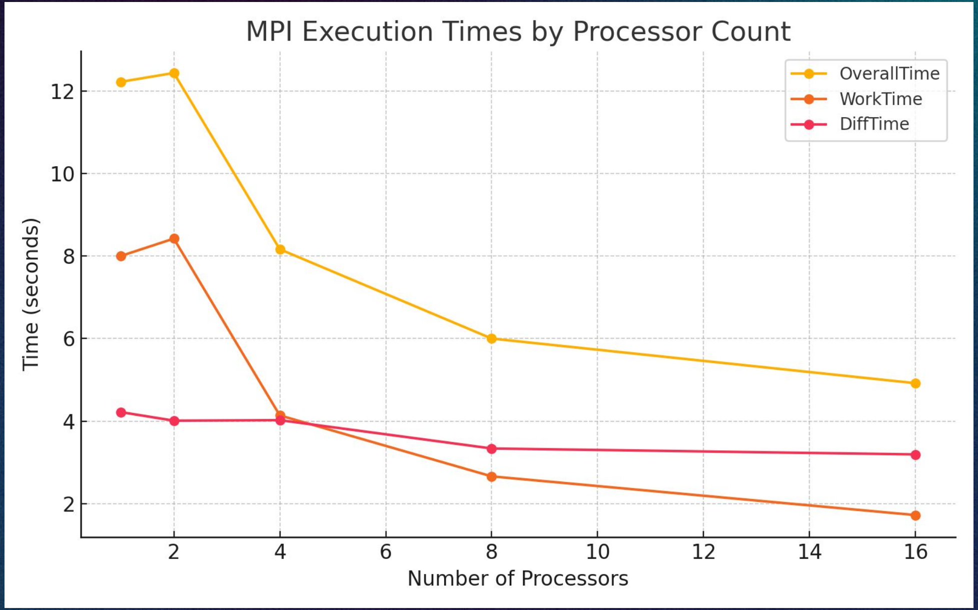
Experimental Evaluation - OpenMP

- Similar outcome to PThreads Except:
  - Increasing performance up to 8 processors instead of 4
  - Overhead time is more stable with a range of 2 seconds.



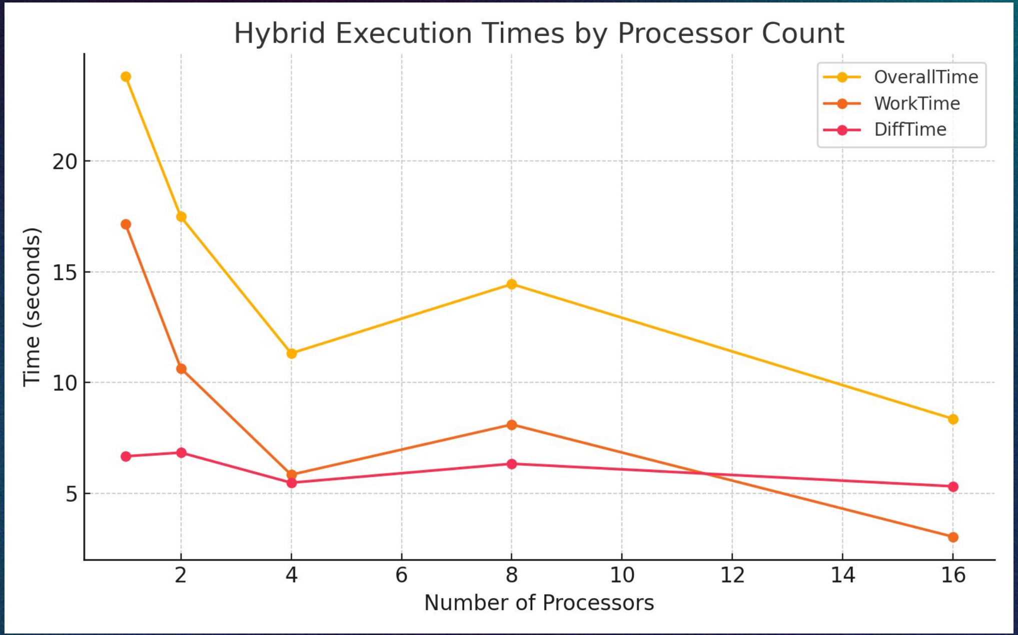
Experimental Evaluation - MPI

- On some runs the time for 2 processors was higher than with one
  - This could be a implementation mistake
- Stable overhead time.
- Clear speedup until 8 processors
  - Similar to OpenMP



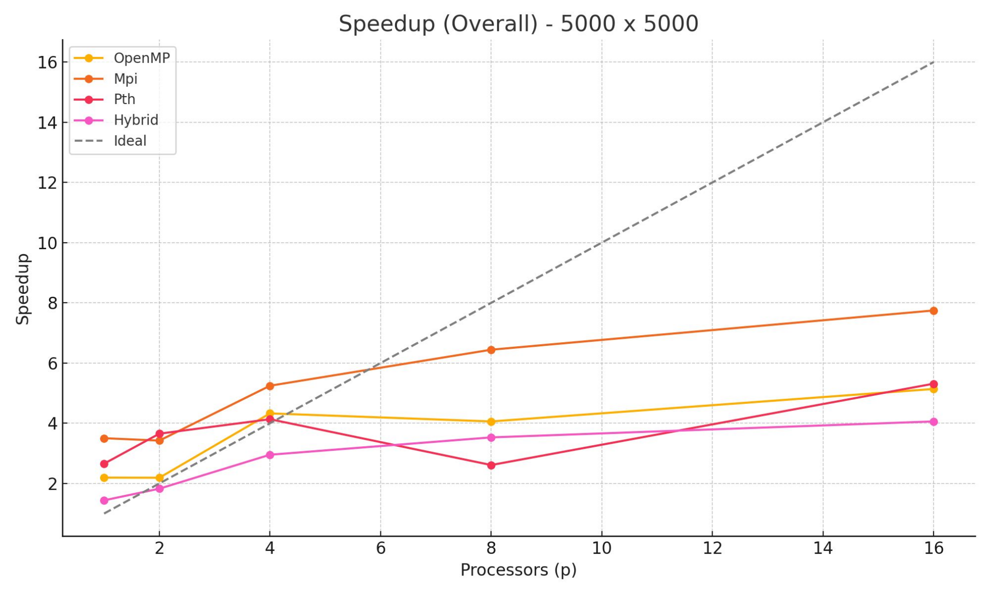
Experimental Evaluation - Hybrid

- Shows the greatest performance gain over the from 1 to 4 processors compared to the other methods.
- Longer time in general
  - Taking twice as long with small number of processors



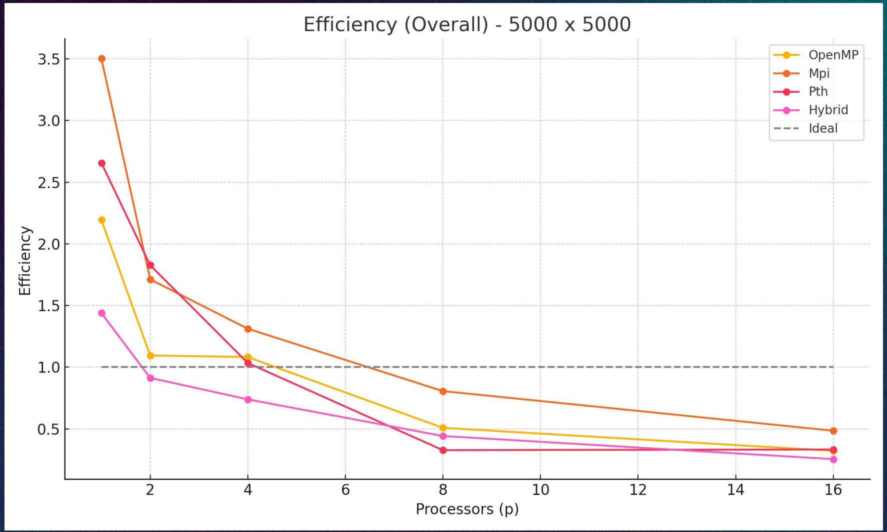
Speedup

- MPI had the best speedup
  - 7.7x at 16 threads
- OpenMP and Pthreads moderate
- Hybrid Underperformed



Efficiency

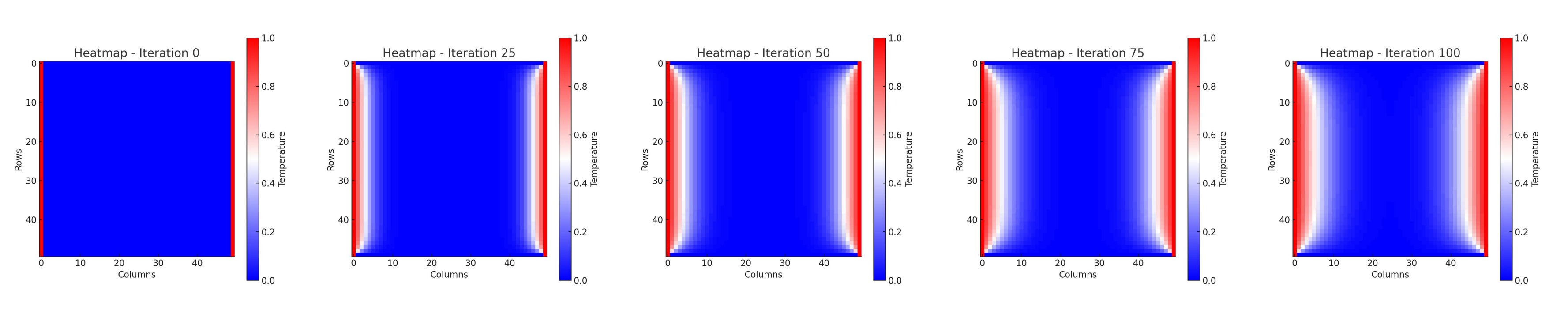
- MPI maintained over 50% efficiency up to 16 threads
- All Methods drop in efficiency less from 2-4 threads
- Hybrid Maintains the lowest efficiency throughout
    - Except at 8 threads

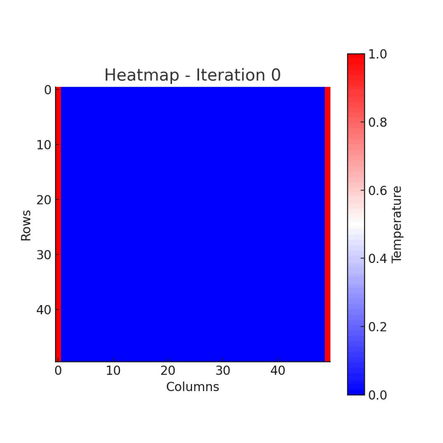


Conclusion

- Simulated 2D heat transfer using a 9-point stencil
- Compared Serial, Pthreads, OpenMP, MPI, and Hybrid implementations
- Verified correctness across all methods with small output matrices
- MPI showed best scalability and performance on large matrices
- OpenMP and Pthreads were efficient but plateaued with more threads
- Hybrid approach underperformed—added complexity didn’t yield better speed
- Future tests with larger datasets may better showcase hybrid potential
- Demonstrated trade-offs in parallelization strategies for scientific computing

THANK YOU





04.30.25
CSCI_473: Parallel Systems

PRESENTATORS: Justin LaForge, Kyle Wallace

Professor: Megan Hickman Fulp