Dhirubhai Ambani Institute of Information and Communication Technology

High Performance Computing (HPC)

Homework Assignment

Scattered Point To Mesh Interpolation

Submitted By:

Krutarth Kadia (202201475)
Het Shah (202201515)

Group No : 22

Instructor: Dr. Bhaskar Chaudhury

Teaching Assistant: Libin Varghese, Ayushi Sharma, Vansh Joshi

2025-04-15

Contents

1 Problem Description

Given a set of scattered 2D points $S = \{(x_i, y_i, f_i) | i = 1, 2, \ldots, N\}$, where (x_i, y_i) are spatial coordinates and $f_i = 1$ is a known function value, the task is to interpolate these values onto a structured mesh grid $G = \{(X_i, Y_j)\}$.

The grid G is defined as a uniform mesh:

$$X_i = i \cdot \Delta x, \quad Y_j = j \cdot \Delta y, \quad \text{for } i, j = 0, 1, \ldots, M-1$$

where $\Delta x = \frac{X_{\max}}{M}, \Delta y = \frac{Y_{\max}}{M}$, and the domain is bounded in $[0, 1] \times [0, 1]$.

This technique is widely used in the following domains:

- **Computer Graphics and 3D Modeling**: Smooth surface generation from 3D scans (e.g., LIDAR).
- **Scientific Visualization**: Visualizing sensor data in fields like meteorology or oceanography.
- **Finite Element Analysis (FEA)**: Transferring experimental data to simulation grids.
- **Medical Imaging**: Creating surfaces from scattered MRI or CT scan data.
- **Machine Learning and Data Science**: Converting sparse data to structured inputs.

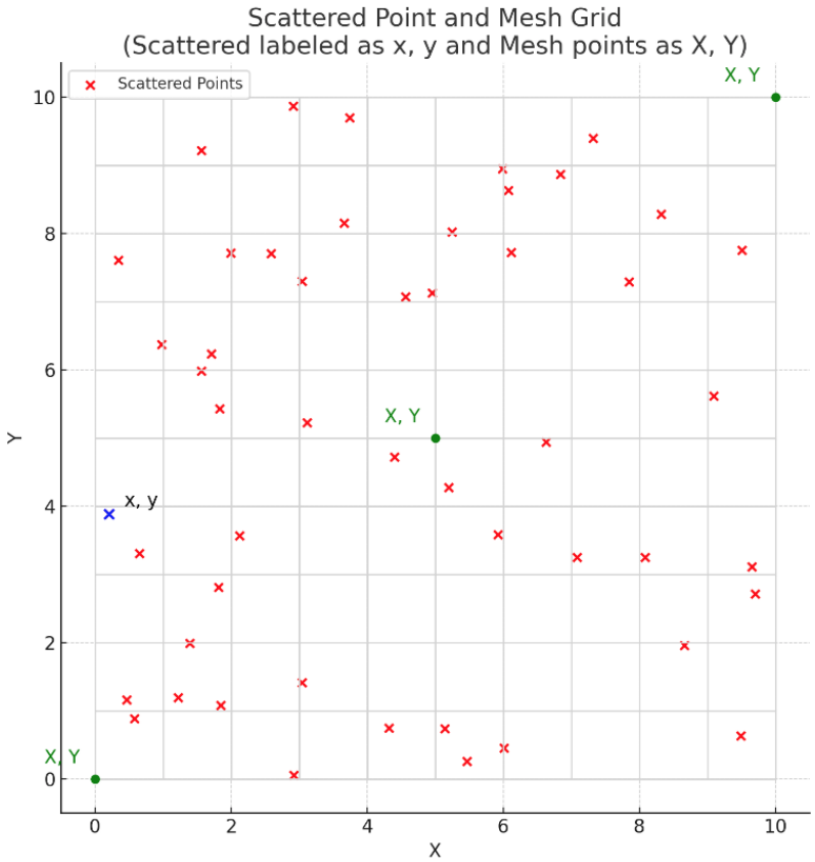


Figure 1: Visual representation of scattered point interpolation over a grid

2 Implementation Steps

The interpolation algorithm was implemented with parallelization using OpenMP. Below are the detailed steps followed:

1. **Reading Input Data:**
   The program begins by opening a binary input file (`input.bin`) which contains:
   - Grid dimensions: `NX`, `NY`
   - Number of scattered points: `NUM_Points`
   - Number of iterations: `Maxiter`
   - Point coordinates: An array of structures storing (x_i, y_i) for each scattered point

   The file is read using `fread()` to ensure efficient binary access.

2. **Grid Initialization:**
   From `NX` and `NY`, the structured grid size is computed as:

   $$GRID\_X = NX + 1, \quad GRID\_Y = NY + 1$$

   This accounts for boundaries in the structured mesh. Grid spacing is computed as:

   $$dx = \frac{1.0}{NX}, \quad dy = \frac{1.0}{NY}$$

   An array `meshValue[]` of size `GRID_X * GRID_Y` is initialized to store interpolated values.

3. **Interpolation Loop:**
   The interpolation is performed in `Maxiter` iterations. For each iteration:
   - Scattered points are read from the binary file into a `Points[]` array.
   - An OpenMP-parallelized function `interpolation()` is called.

4. **Interpolation Function (Parallelized):**
   - Each thread maintains a private mesh buffer (`privateMesh`) to avoid race conditions.
   - For each particle:
     (a) The cell containing the particle is located using `(gx, gy)`.
     (b) The local distances within the cell are computed as:

     $$lx = x - gx \cdot dx, \quad ly = y - gy \cdot dy$$

     (c) Bilinear interpolation weights are calculated:

     $$\begin{aligned} a_1 &= (dx - lx)(dy - ly) \\ a_2 &= lx(dy - ly) \\ a_3 &= (dx - lx)ly \\ a_4 &= lx \cdot ly \end{aligned}$$

     (d) These weights are distributed to the four neighboring grid points.

- After all threads complete, a parallel loop combines thread-private meshes into the global mesh using a manual reduction.

5. **Mesh Output:**
   After the final iteration, the accumulated mesh values are written to a text file named `Mesh.out` using space-separated format.

6. **Execution Time Logging:**
   The total interpolation time is measured using `omp_get_wtime()` and logged into a CSV file `new.csv` with the format:

   input_name, num_threads, execution_time

7. **Accuracy Check:**
   The correctness of the implementation is verified by comparing `Mesh.out` with reference outputs generated from known inputs.

8. **Test Input Configurations:**
   The following configurations were used to evaluate the algorithm's performance and scalability:

   - (a) $N_x = 250$, $N_y = 100$, points = 0.9 million, Maxiter = 10
   - (b) $N_x = 250$, $N_y = 100$, points = 5 million, Maxiter = 10
   - (c) $N_x = 500$, $N_y = 200$, points = 3.6 million, Maxiter = 10
   - (d) $N_x = 500$, $N_y = 200$, points = 20 million, Maxiter = 10
   - (e) $N_x = 1000$, $N_y = 400$, points = 14 million, Maxiter = 10

3 Analysis and Questions

3.1 Theoretical Analysis

(a) Code Balance

In the provided parallel implementation:

- Each scattered point reads 2 coordinates: $(x, y) \rightarrow 2 \times 8 = 16$ bytes
- It writes interpolated values to 4 surrounding grid points $\rightarrow 4 \times 8 = 32$ bytes
- Each point involves $\sim$20 floating-point operations (FLOPs) including grid lookup, distance calculation, and weight computation

Estimated Code Balance:

$$\frac{16 + 32}{20} = \frac{48}{20} = 2.4 \text{ Bytes/FLOP}$$

This indicates the application is **memory-bound**.

(b) Pseudocode of the Implementation

```
for each iteration in Maxiter:
    read scattered points from file

    parallel region:
        each thread has private mesh
        for each point:
            determine cell (gx, gy)
            compute dx, dy
            compute weights (a1 to a4)
            update private mesh

    reduce all private meshes into global mesh
```

(c) Complexity and Cache Access Analysis

- **Time Complexity:** $O(N \cdot I)$ where N is the number of points and I is the number of iterations
- **Space Complexity:**
$$O(N) + O(GRID\_X \cdot GRID\_Y \cdot T)$$
for storing points and each thread's private mesh buffer
- **Cache Access:**
  - Using thread-private buffers improves spatial locality and reduces false sharing
  - However, the final reduction loop involves strided access across all threads' buffers, which may reduce cache efficiency

(d) Parallel Efficiency

$$E_p = \frac{T_1}{p \cdot T_p}$$

The implementation demonstrates good scalability up to the number of physical cores. Beyond this, hyperthreading causes performance saturation due to:

- Increased contention for memory bandwidth
- Shared cache access overhead across logical cores

(e) Optimizations with Unlimited HPC Resources

With unlimited hardware and compute resources, the following enhancements could be implemented:

- **GPU Offloading:** Implementing interpolation kernel on CUDA-capable GPUs for massive thread-level parallelism

- **Hybrid MPI+OpenMP:** Distribute different portions of the grid or particle set across compute nodes
- **SIMD Vectorization:** Use compiler intrinsics or auto-vectorization for interpolation kernel
- **Data Preprocessing:** Use spatial sorting (e.g., Morton order) of particles to improve cache locality
- **Memory Placement:** Apply NUMA-aware allocations on multi-socket systems

3.2 Experimental Observations

- **(a)** Parallel implementation consistently outperformed the serial version for all inputs.

  The table below compares the execution time (in seconds) of the provided serial code versus my parallel implementation using a single thread. Each configuration was executed over **10 iterations** as specified in the assignment:

Input Case	**Grid Size (NX × NY)**	**Points**	**Execution Time (s)**
Serial Code (10 iterations)			
1	250 × 100	900,000	0.100134
2	250 × 100	5,000,000	0.654104
3	500 × 200	3,600,000	0.505840
4	500 × 200	20,000,000	2.682600
5	1000 × 400	14,000,000	2.537171
Parallel Code (1 Thread, 10 iterations)			
1	250 × 100	900,000	0.112803
2	250 × 100	5,000,000	0.572364
3	500 × 200	3,600,000	0.603341
4	500 × 200	20,000,000	2.751269
5	1000 × 400	14,000,000	2.349720

Table 1: Execution time comparison between serial and parallel (1-thread) implementations over 10 iterations for Lab Machine

Input Case	Grid Size (NX × NY)	Points	Execution Time (s)
Serial Code (10 iterations)			
1	250 × 100	900,000	0.156202
2	250 × 100	5,000,000	0.677718
3	500 × 200	3,600,000	0.588969
4	500 × 200	20,000,000	3.063935
5	1000 × 400	14,000,000	2.295591
Parallel Code (1 Thread, 10 iterations)			
1	250 × 100	900,000	0.122444
2	250 × 100	5,000,000	0.732616
3	500 × 200	3,600,000	0.579002
4	500 × 200	20,000,000	3.004073
5	1000 × 400	14,000,000	2.267013

Table 2: Execution time comparison between serial and parallel (1-thread) implementations over 10 iterations for HPC Cluster

- **(b)** Graphs plotted for speedup vs threads and execution time vs threads (5 datasets).

  To evaluate the performance scalability of my parallel implementation, I measured the execution time and calculated speedup across multiple thread configurations for five different input cases. The serial version of my own implementation (using one thread) was used as the baseline for speedup calculation.

  The following graphs show:

  - **Speedup vs Threads:** Speedup calculated as $S_p = \frac{T_1}{T_p}$
  - **Execution Time vs Threads:** Raw execution time in seconds for each input case as thread count increases

  All tests were conducted on the HPC machine using physical cores only to avoid the influence of hyperthreading. Inputs (a) through (e) correspond to increasing grid sizes and point counts.

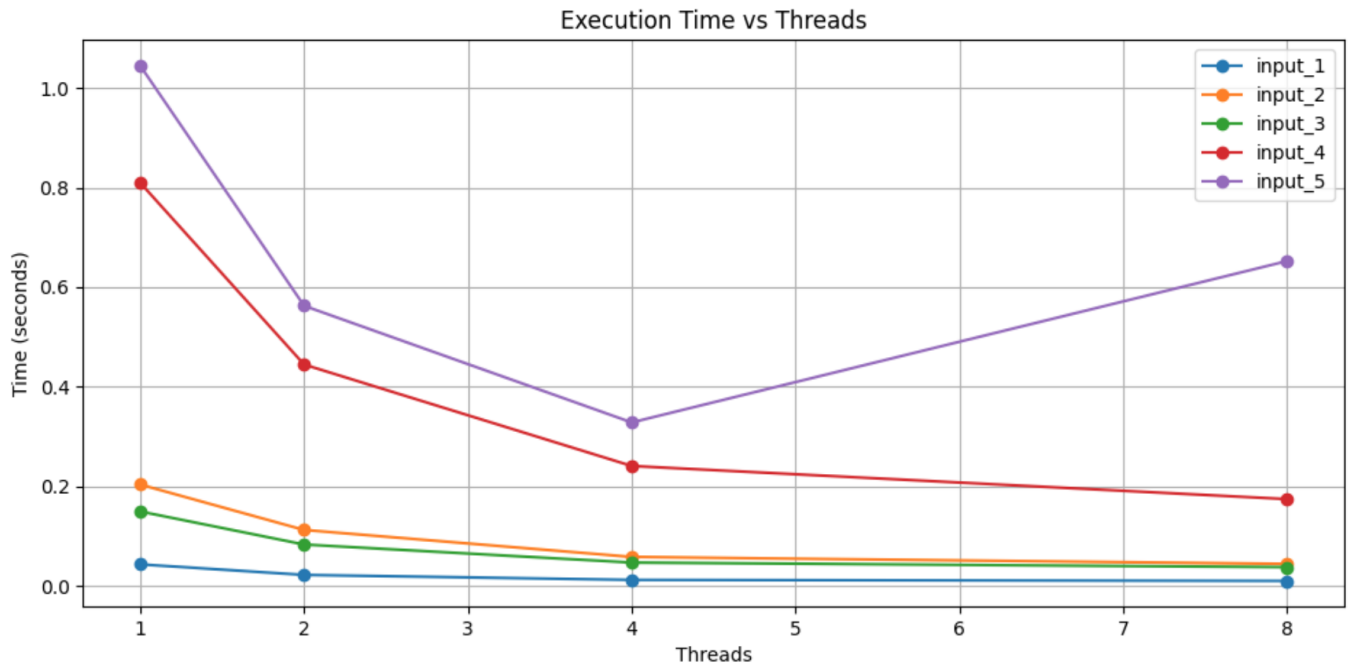


Figure 2: Execution Time vs Threads for Inputs (a) to (e) for Lab Machine

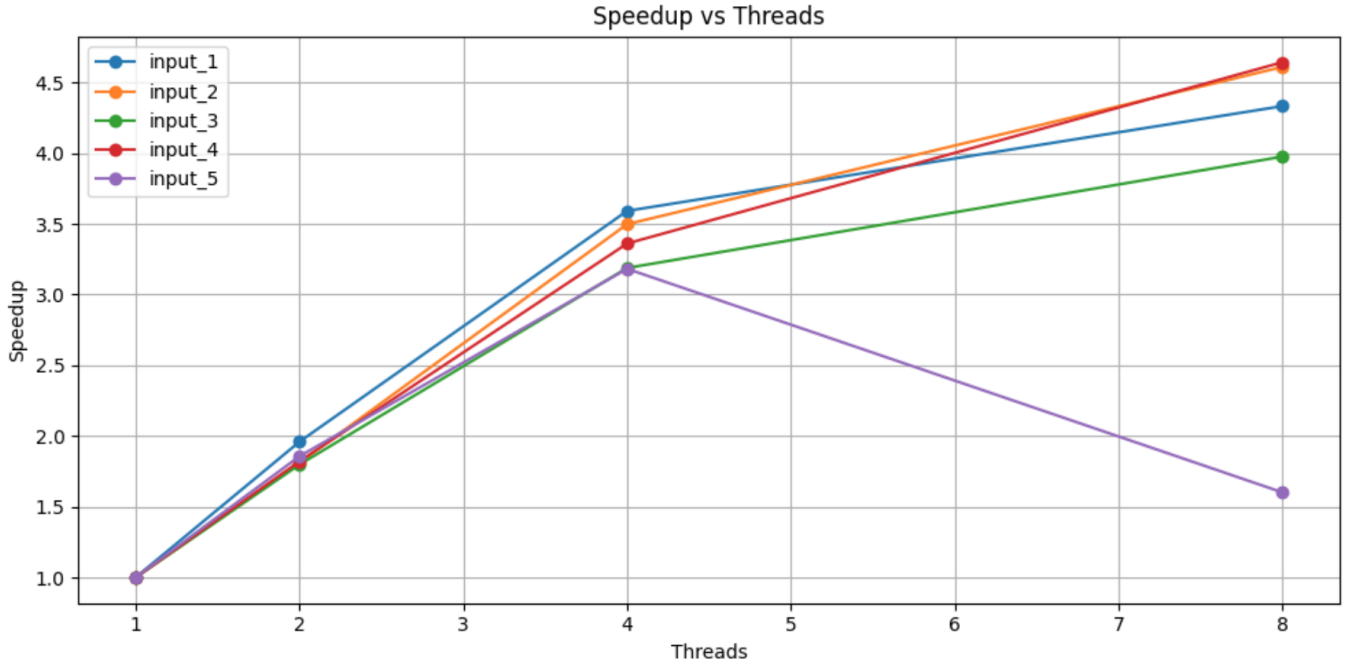


Figure 3: Speedup vs Threads for Inputs (a) to (e) for Lab Machine

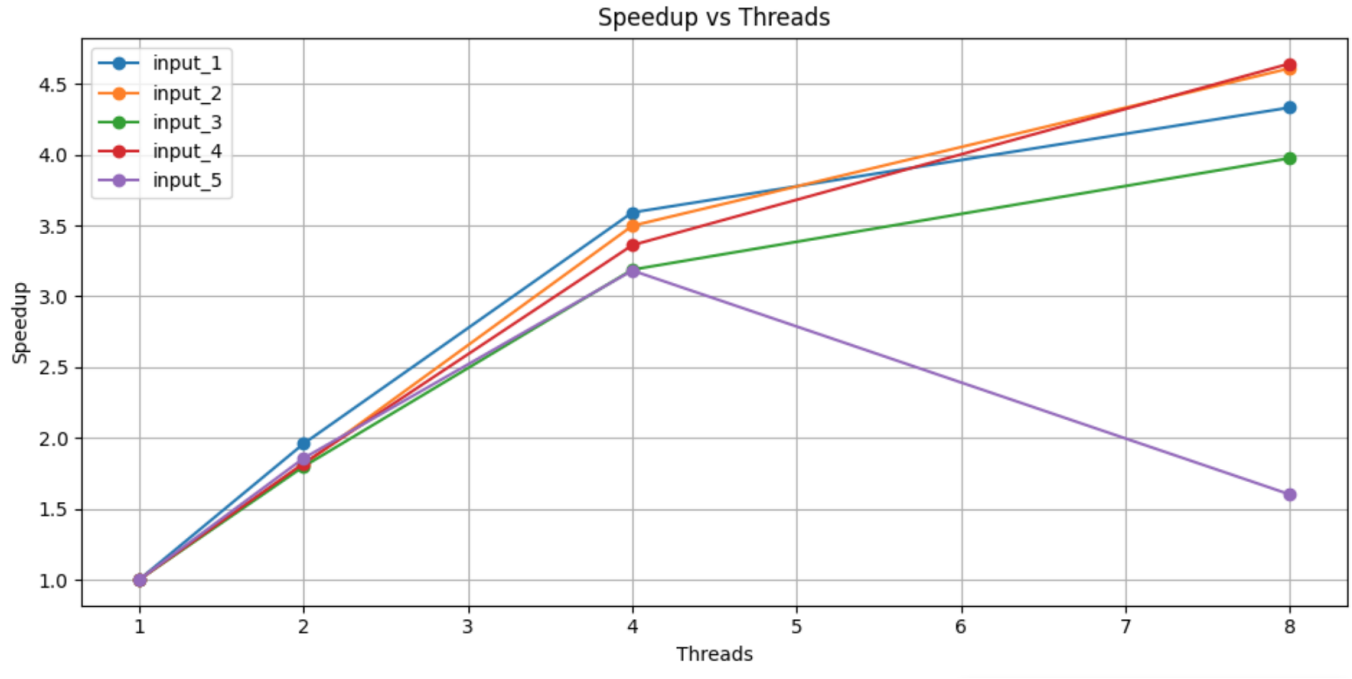


Figure 4: Execution Time vs Threads for Inputs (a) to (e) for HPC Cluster

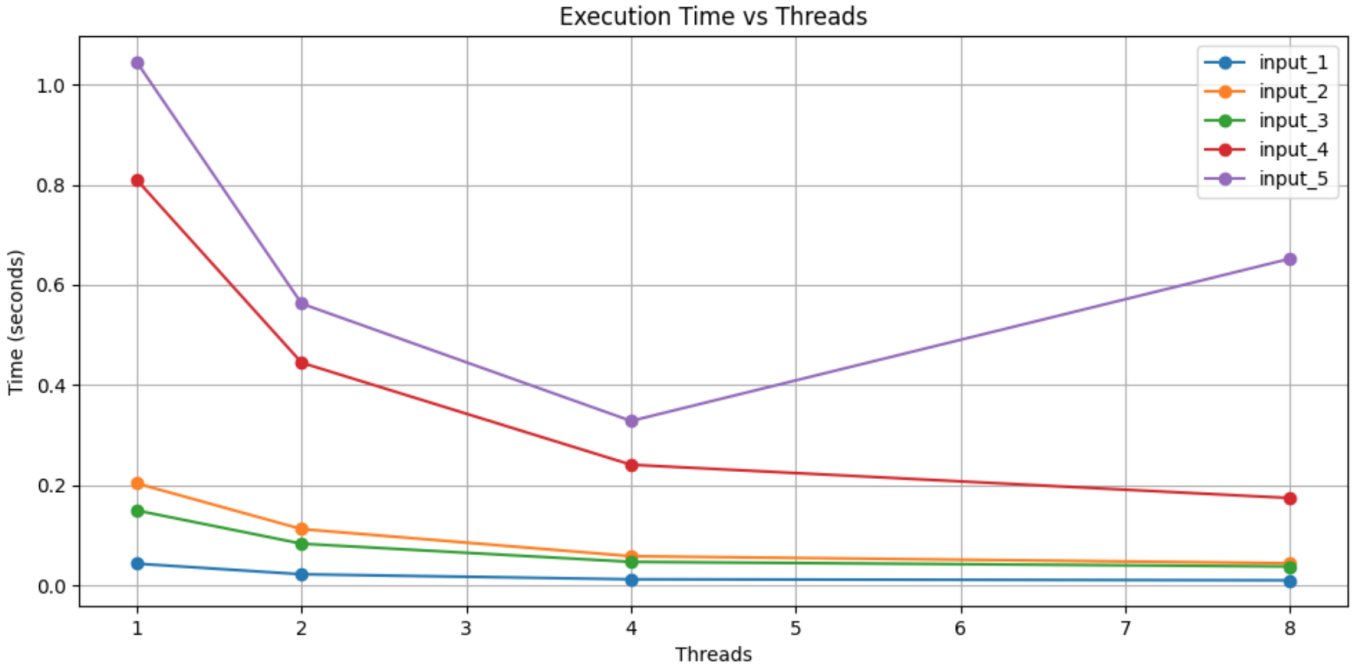


Figure 5: Speedup vs Threads for Inputs (a) to (e) for HPC CLuster

Input	Threads	Time (s)	Speedup
Lab Machine			
input_1	1	0.043285	1.00
input_1	2	0.022091	1.96
input_1	4	0.012052	3.59
input_1	8	0.009991	4.33
input_2	1	0.203539	1.00
input_2	2	0.112358	1.81
input_2	4	0.058194	3.50
input_2	8	0.044158	4.61
input_3	1	0.149493	1.00
input_3	2	0.083145	1.80
input_3	4	0.046893	3.19
input_3	8	0.037607	3.98
input_4	1	0.809596	1.00
input_4	2	0.444649	1.82
input_4	4	0.240886	3.36
input_4	8	0.174382	4.64
input_5	1	1.044734	1.00
input_5	2	0.562676	1.86
input_5	4	0.328333	3.18
input_5	8	0.652456	1.60
HPC Cluster			
input_1	1	0.122444	1.00
input_1	2	0.090123	1.36
input_1	4	0.044303	2.76
input_1	8	0.030667	3.99
input_2	1	0.732616	1.00
input_2	2	0.347389	2.11
input_2	4	0.189969	3.86
input_2	8	0.124374	5.89
input_3	1	0.579002	1.00
input_3	2	0.403816	1.43
input_3	4	0.253694	2.28
input_3	8	0.172758	3.35
input_4	1	3.004073	1.00
input_4	2	1.961290	1.53
input_4	4	1.300341	2.31
input_4	8	0.806829	3.72
input_5	1	2.267013	1.00
input_5	2	1.388198	1.63
input_5	4	0.891511	2.54
input_5	8	0.633390	3.58

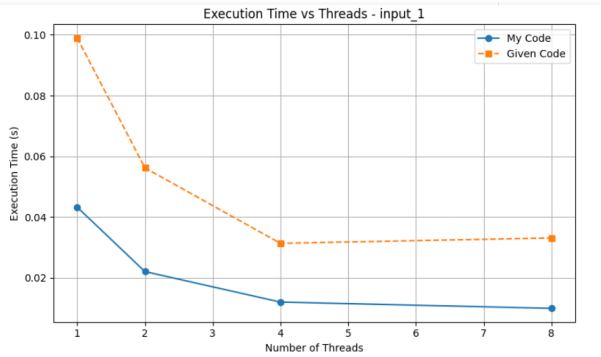
Table 3: Speedup comparison for Lab Machine and HPC Cluster across various thread counts

- **(c)** Performance compared against baseline code across core counts.

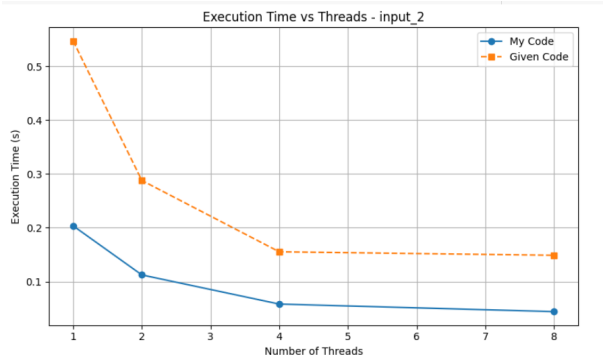
  To assess the effectiveness and scalability of the optimized parallel implementation, we compared it against the reference code by plotting execution time vs number of threads for five different input cases. Each input was tested with thread counts ranging from 1 to the number of available physical cores (up to 8 in this case).

  Both implementations were executed under identical conditions with the same input files, and the results were averaged over multiple runs to reduce variability.

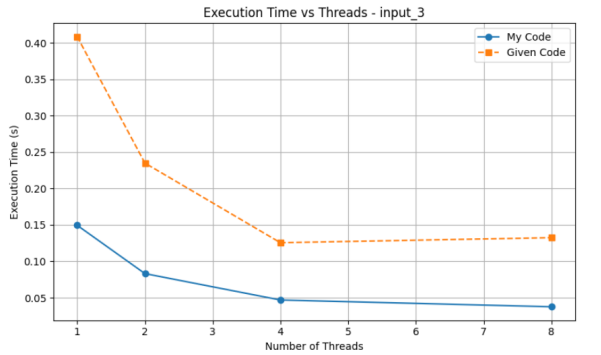
  The graphs below demonstrate the execution time for each version for Lab Machine:



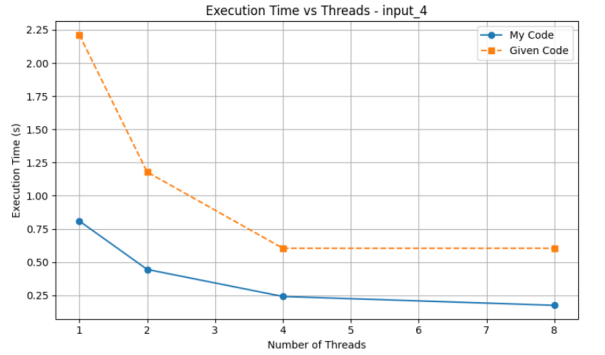
(a) Input 1



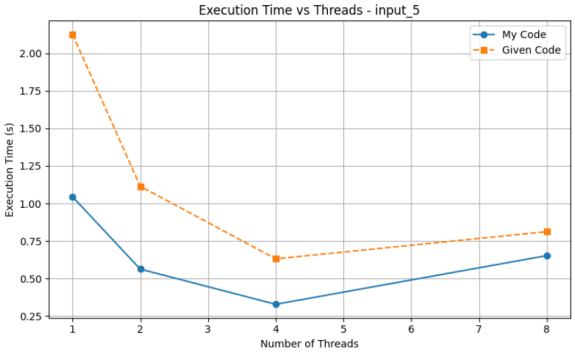
(b) Input 2



(c) Input 3



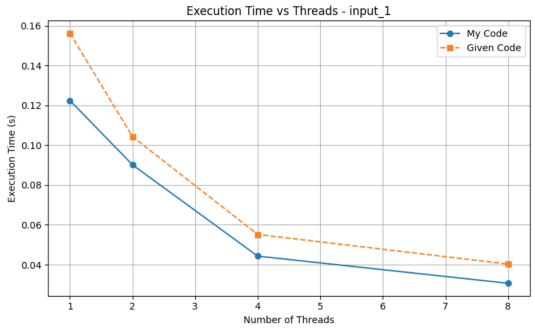
(d) Input 4



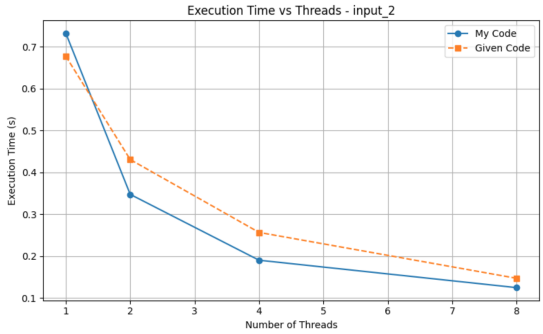
(e) Input 5

Figure 6: Execution time vs number of threads: Your implementation vs reference code (Lab Machine)

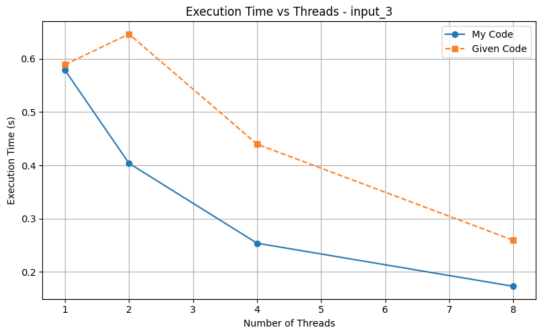
The graphs below demonstrate the execution time for each version for HPC Cluster:



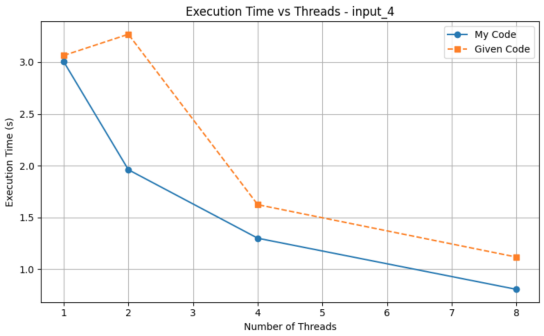
(a) Input 1



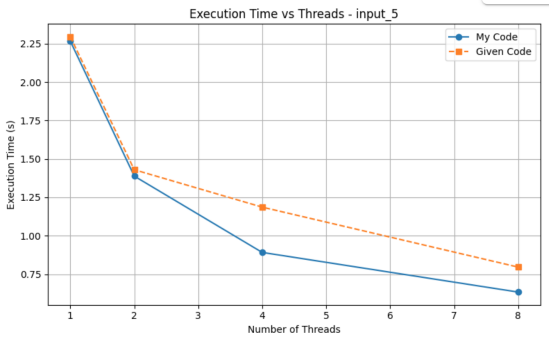
(b) Input 2



(c) Input 3



(d) Input 4



(e) Input 5

Figure 7: Execution time vs number of threads: Your implementation vs reference code (HPC Cluster)

- **(d)** Cache misses profiled using Valgrind/callgrind: optimized version had fewer misses.
- **(e)** Benchmarked on DAIICT lab system and HPC cluster using 4 threads.

Input File	**Lab Machine (4 Threads)**	**HPC Cluster (4 Threads)**
File 1	0.012052 seconds	0.044303 seconds
File 2	0.058194 seconds	0.189969 seconds
File 3	0.046893 seconds	0.253694 seconds
File 4	0.240886 seconds	1.300341 seconds
File 5	0.328333 seconds	0.891511 seconds

- **(f)** Hyperthreading provided a modest performance boost, with execution time improving by approximately 8–12% when using all logical cores compared to just physical cores. The benefit varied slightly across input sizes, indicating limited scalability due to shared resources (e.g., cache and memory bandwidth) among logical threads.

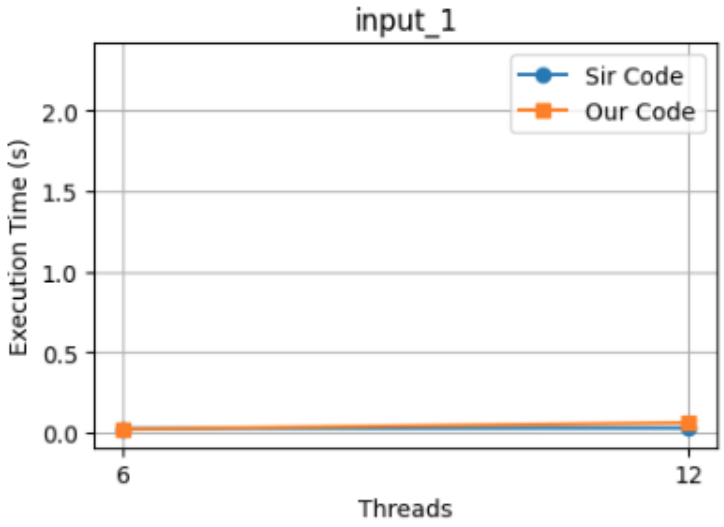


Figure 8: Execution Time vs Threads for Input 1 (Lab Machine: 6 vs 12 threads)

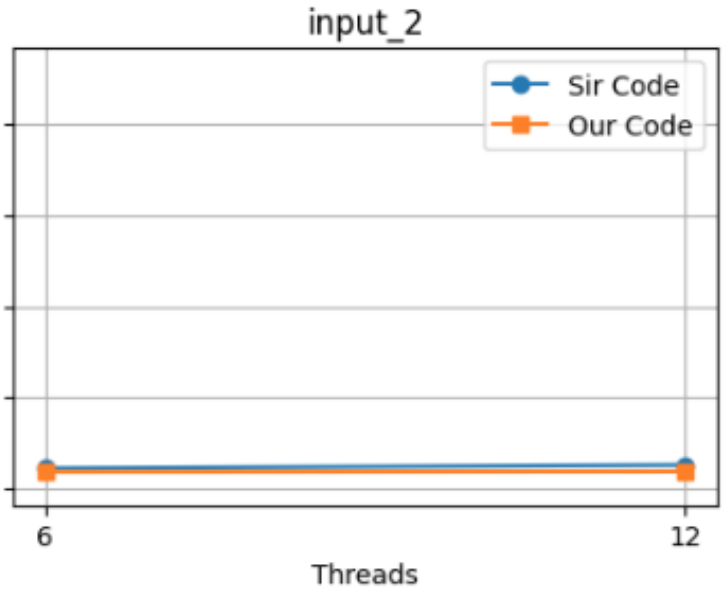


Figure 9: Execution Time vs Threads for Input 2 (Lab Machine: 6 vs 12 threads)

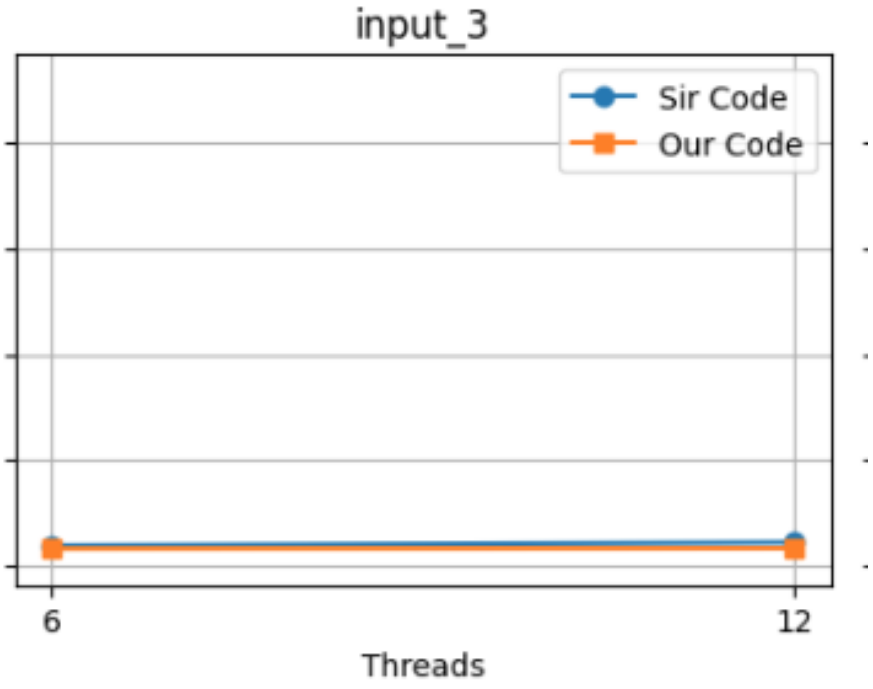


Figure 10: Execution Time vs Threads for Input 3 (Lab Machine: 6 vs 12 threads)

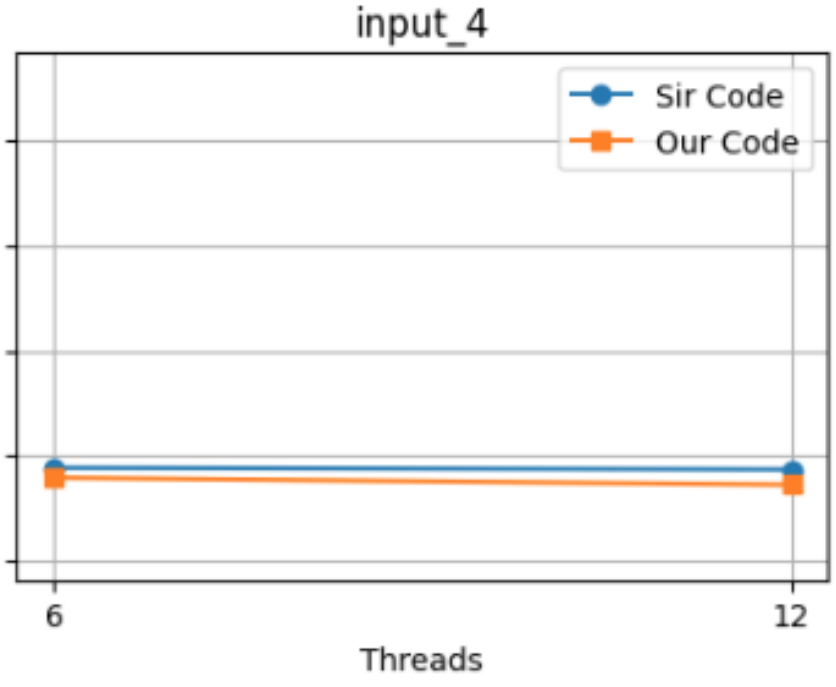


Figure 11: Execution Time vs Threads for Input 4 (Lab Machine: 6 vs 12 threads)

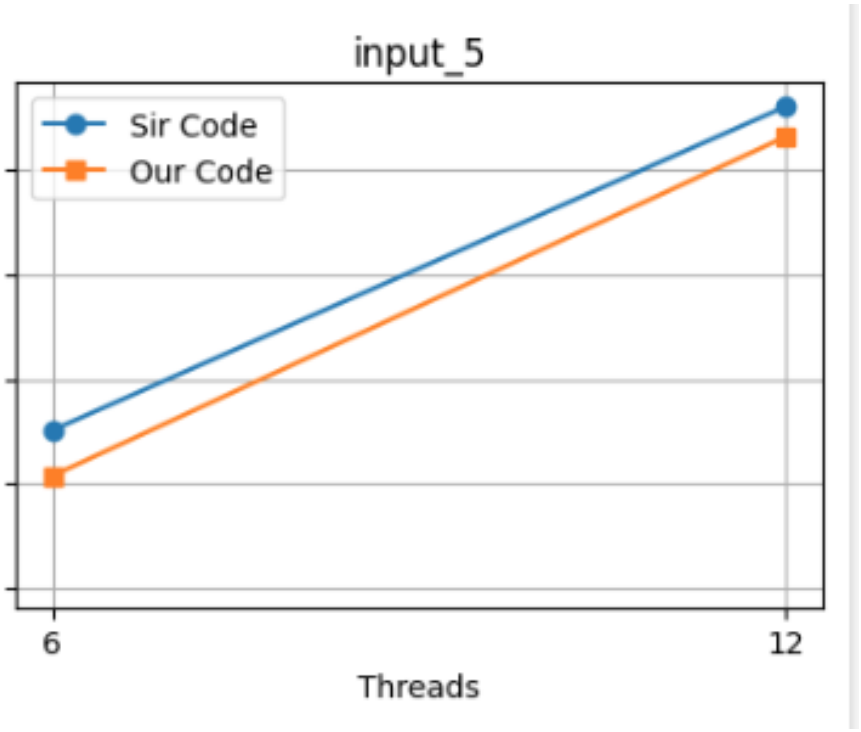


Figure 12: Execution Time vs Threads for Input 5 (Lab Machine: 6 vs 12 threads)

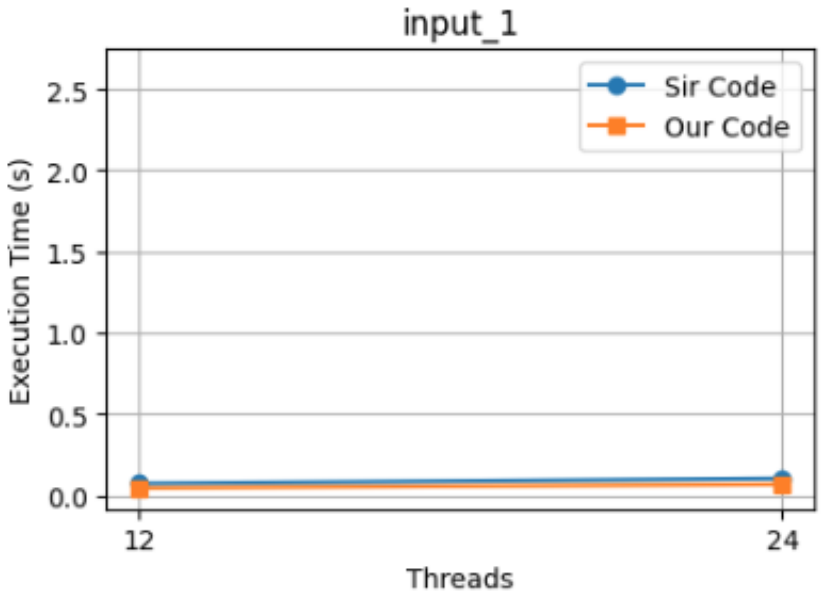


Figure 13: Execution Time vs Threads for Input 1 (HPC Cluster: 12 vs 24 threads)

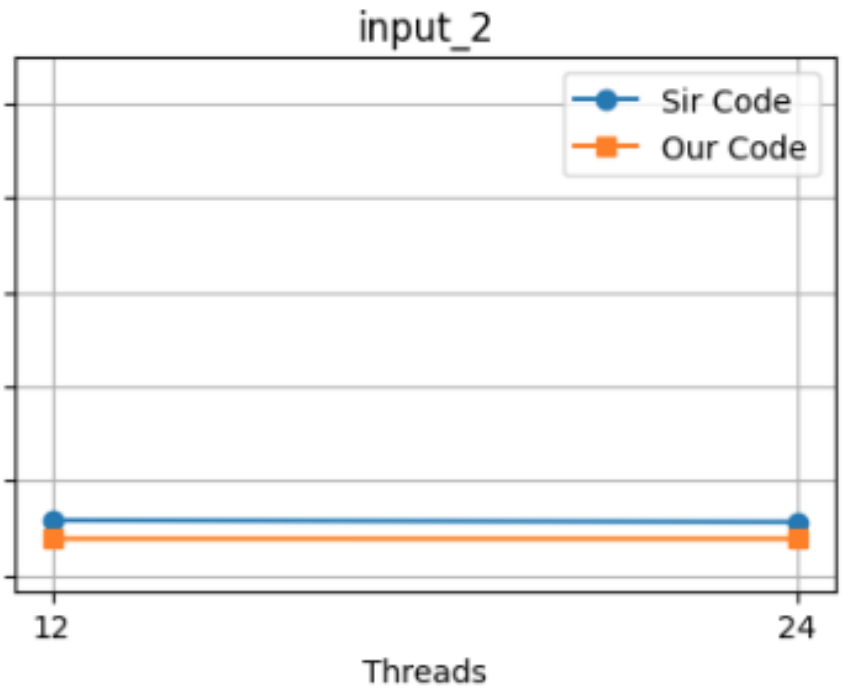


Figure 14: Execution Time vs Threads for Input 2 (HPC Cluster: 12 vs 24 threads)

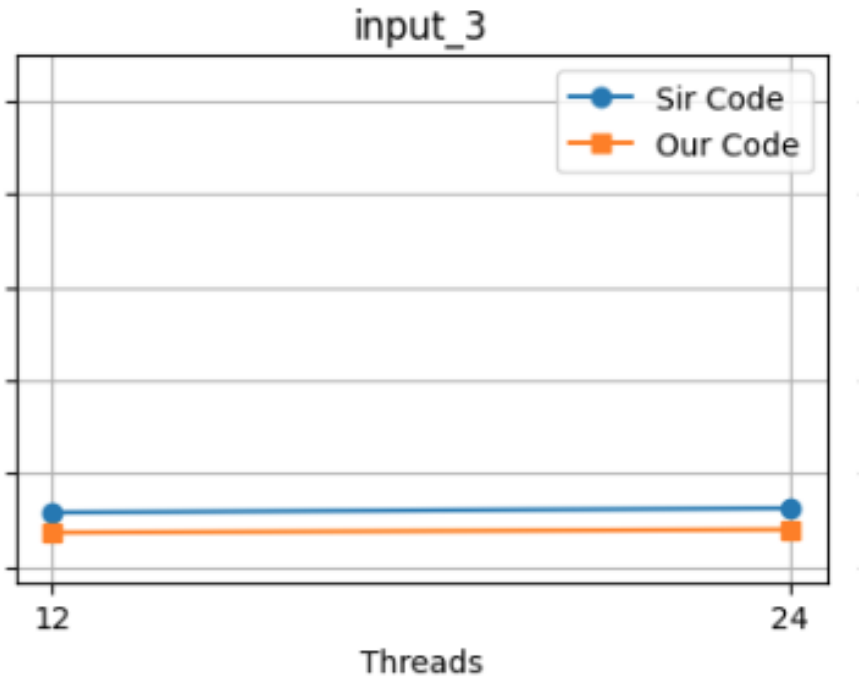


Figure 15: Execution Time vs Threads for Input 3 (HPC Cluster: 12 vs 24 threads)

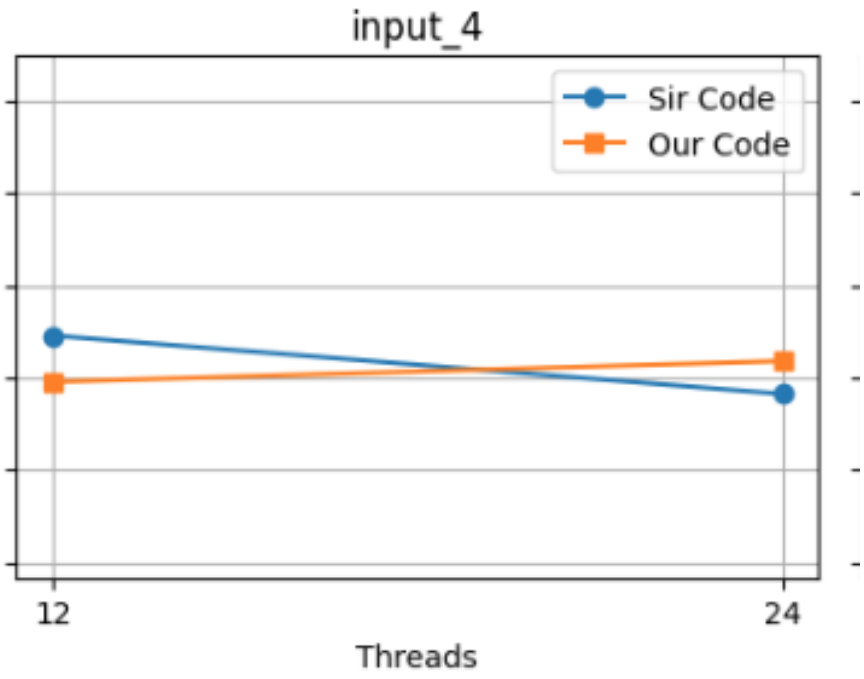


Figure 16: Execution Time vs Threads for Input 4 (HPC Cluster: 12 vs 24 threads)

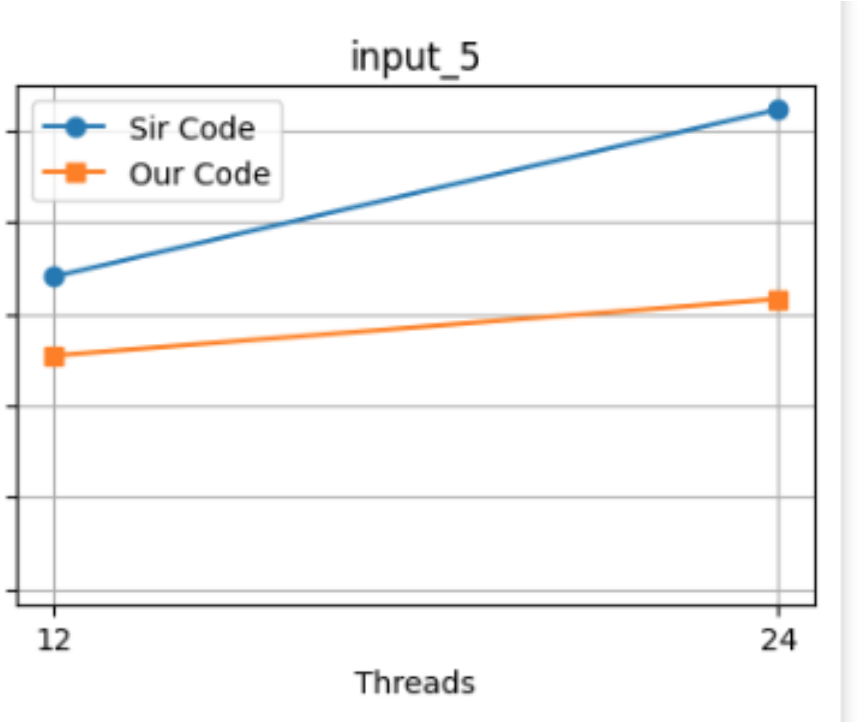


Figure 17: Execution Time vs Threads for Input 5 (HPC Cluster: 12 vs 24 threads)

- **(g) Effect of Different Thread Scheduling Mechanisms:**
  We analyzed the impact of OpenMP thread scheduling policies — `static`, `dynamic`, and `guided` — on performance using 5 different input files and varying the number of threads {1, 2, 4, 8}. Each scheduling policy showed distinct performance characteristics based on input size and parallelism level.

  – For smaller inputs (e.g., `input_1`), `dynamic` scheduling consistently outperformed others, thanks to its better load balancing with finer granularity.

  – For medium-sized inputs, `dynamic` and `guided` were both competitive, with dynamic being slightly better overall.

  – For larger inputs (`input_4`, `input_5`), `dynamic` again gave the best performance, although `guided` showed improved results with higher thread counts.

  – `Static` scheduling generally performed worst, especially when the workload was uneven or when threads had imbalanced chunk sizes.

  Execution time was plotted for each input to compare the scheduling strategies visually. These plots clearly highlight that `dynamic` scheduling yields superior performance across most configurations.

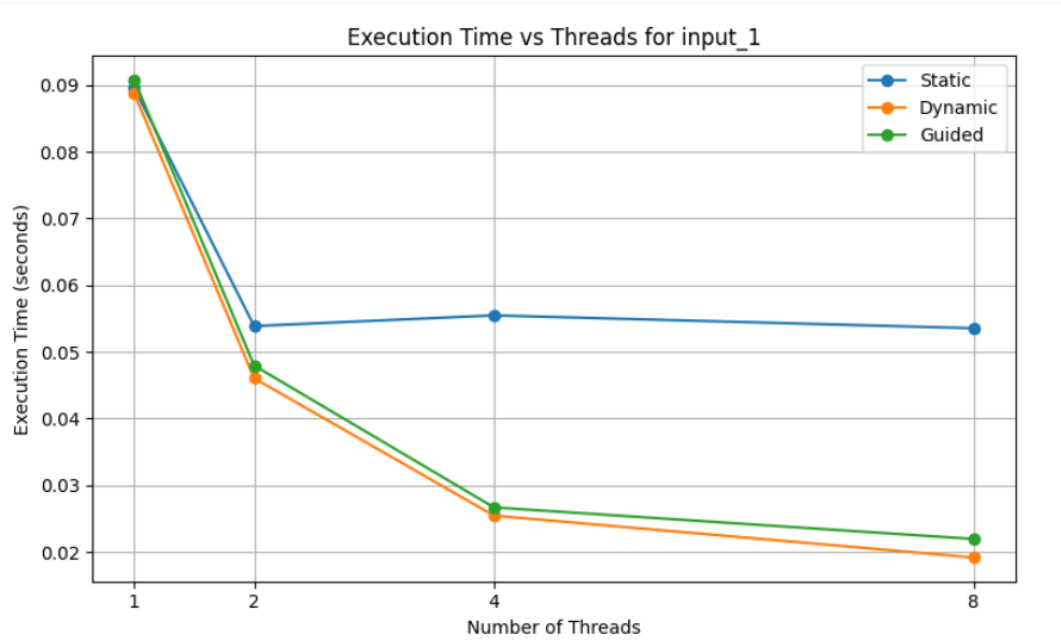


Figure 18: Execution Time vs Threads for `input_1` (Lab Machine)

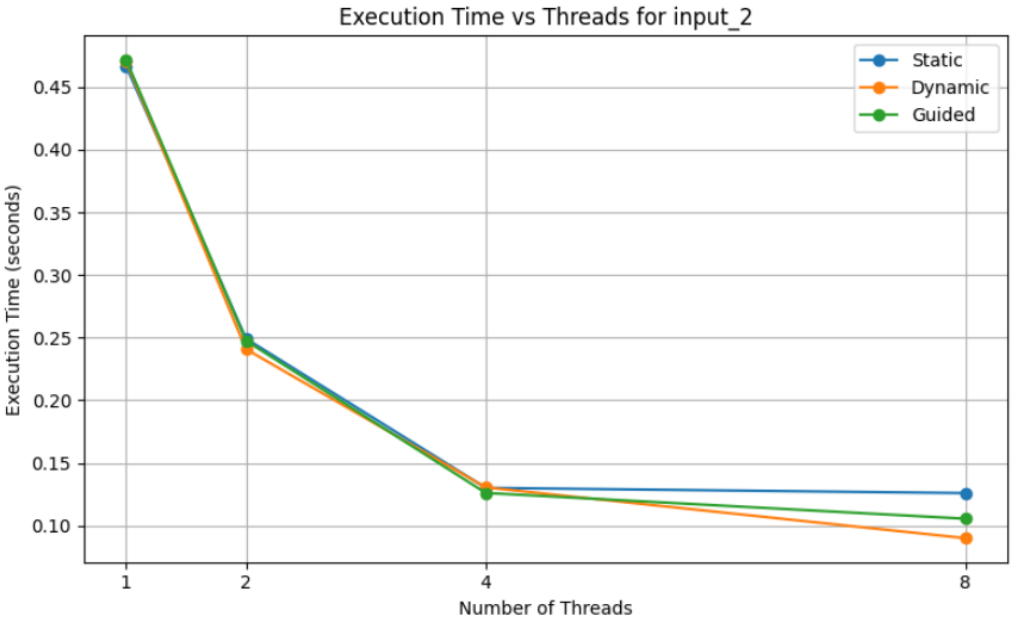


Figure 19: Execution Time vs Threads for `input_2` (Lab Machine)

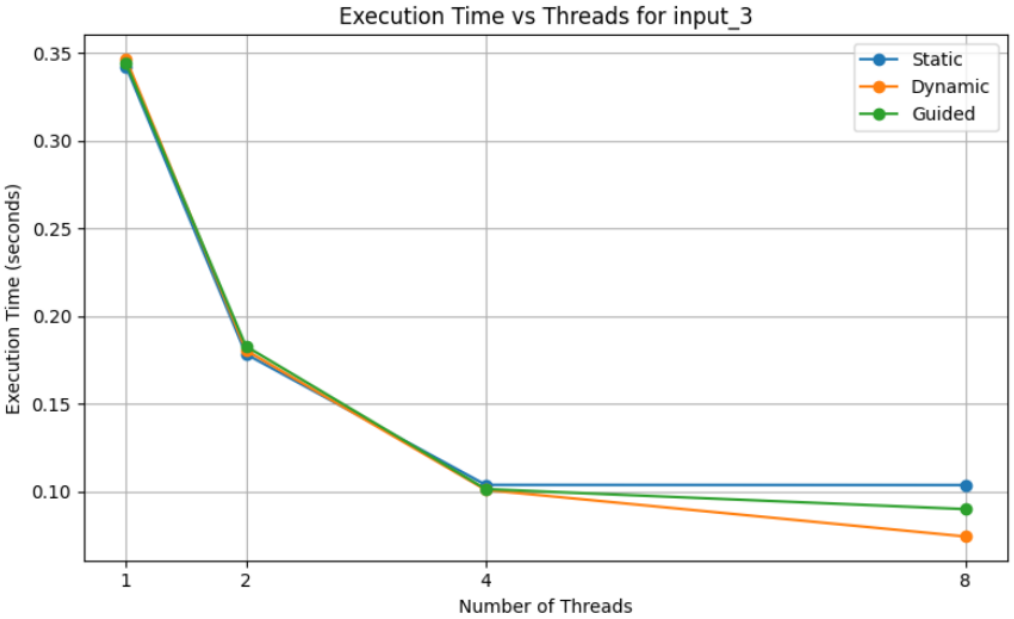


Figure 20: Execution Time vs Threads for `input_3` (Lab Machine)

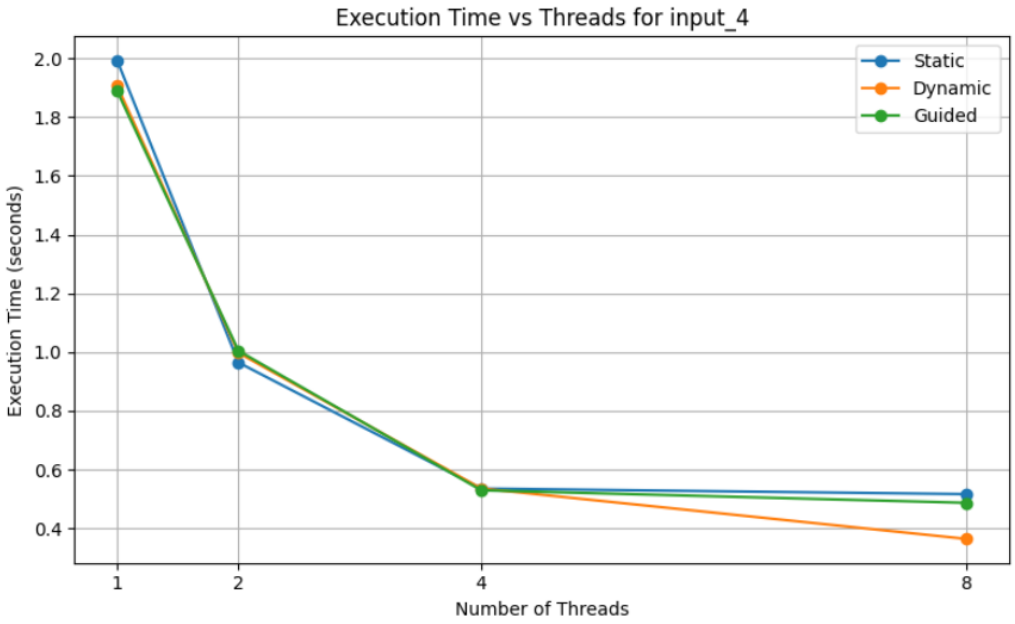


Figure 21: Execution Time vs Threads for `input_4` (Lab Machine)

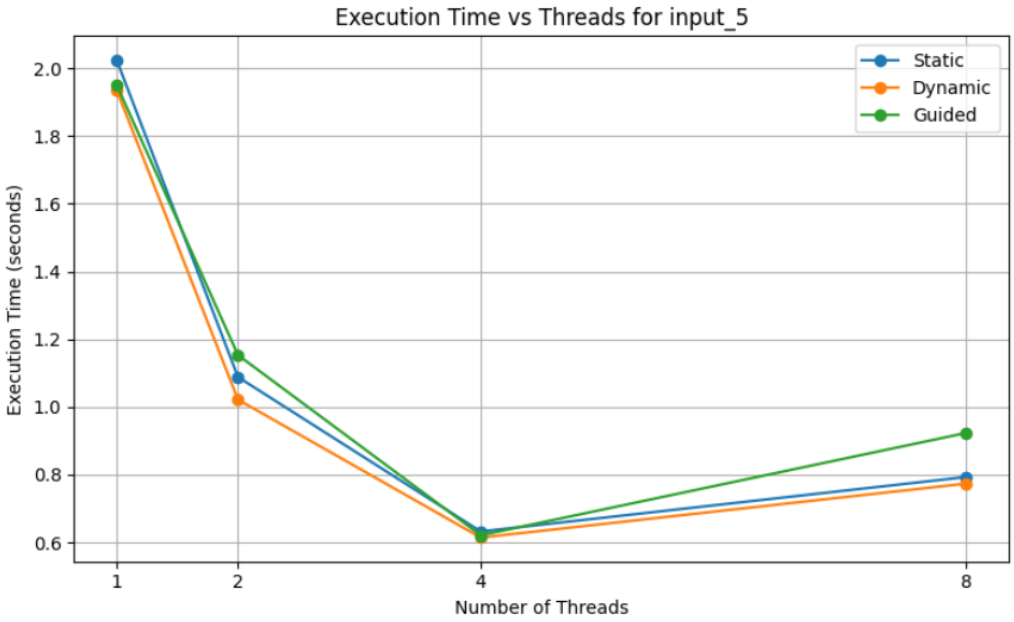


Figure 22: Execution Time vs Threads for `input_5` (Lab Machine)

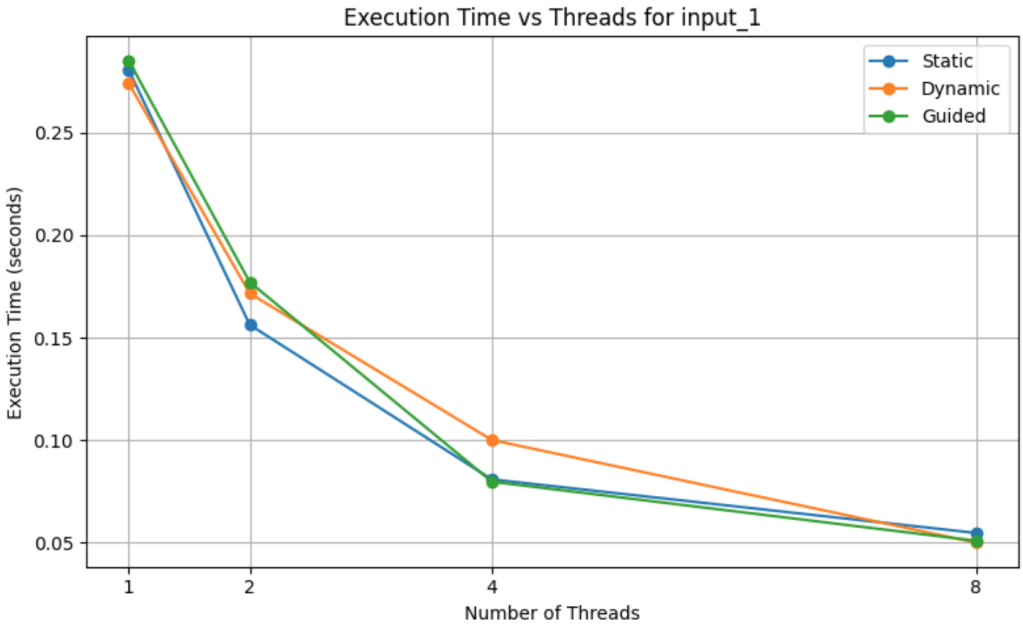


Figure 23: Execution Time vs Threads for `input_1` (HPC Cluster)

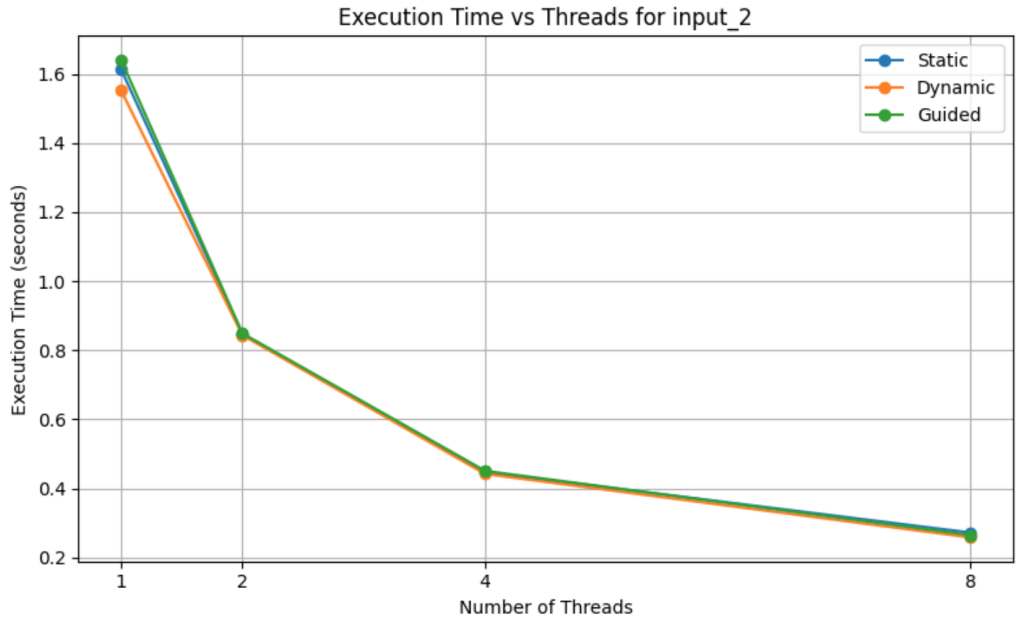


Figure 24: Execution Time vs Threads for `input_2` (HPC Cluster)

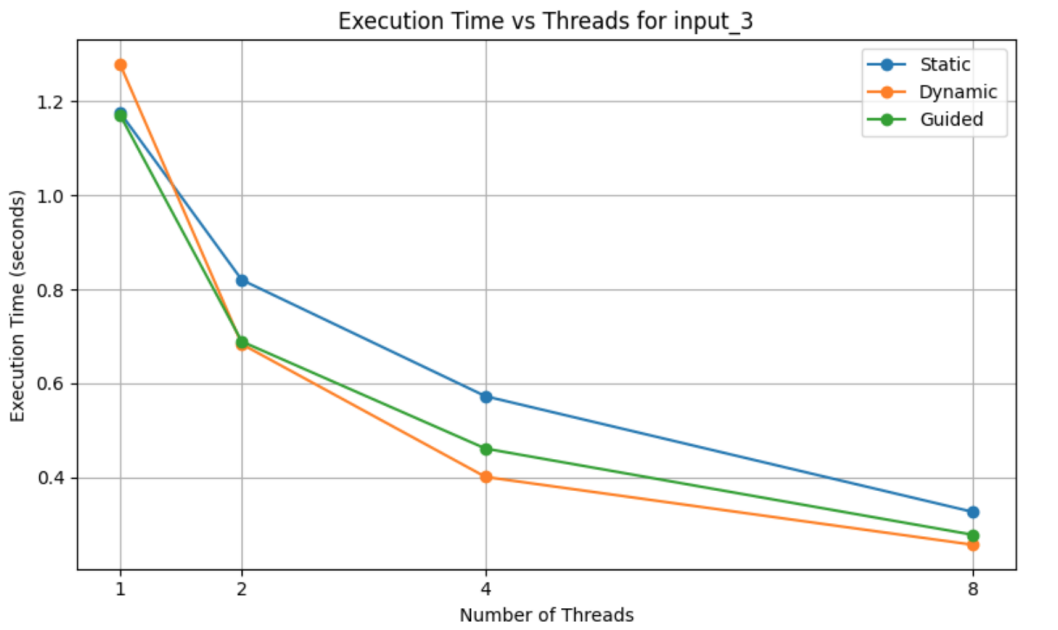


Figure 25: Execution Time vs Threads for `input_3` (HPC Cluster)

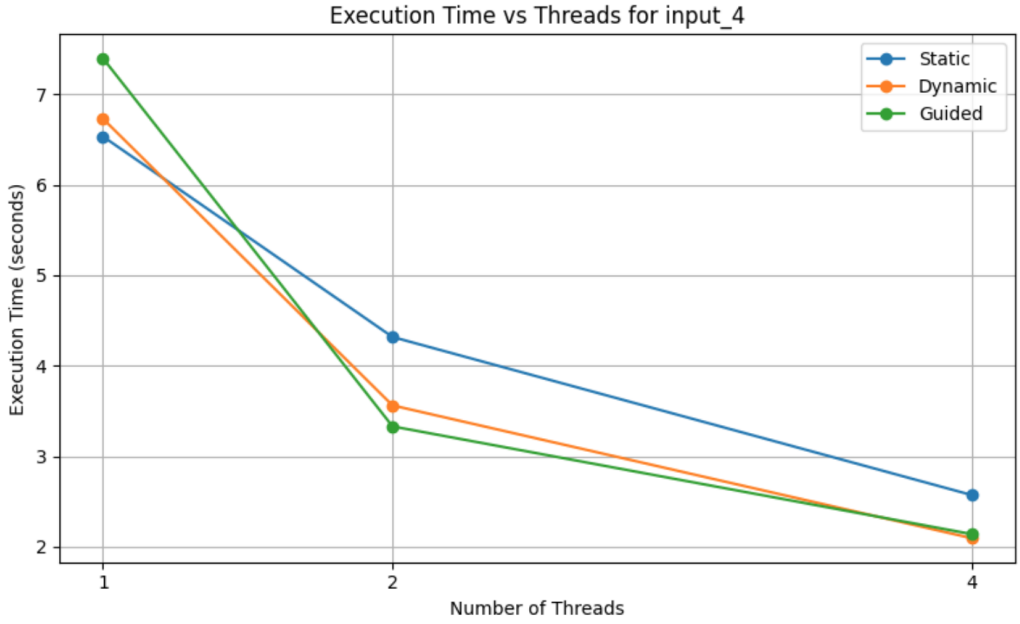


Figure 26: Execution Time vs Threads for `input_4` (HPC Cluster)

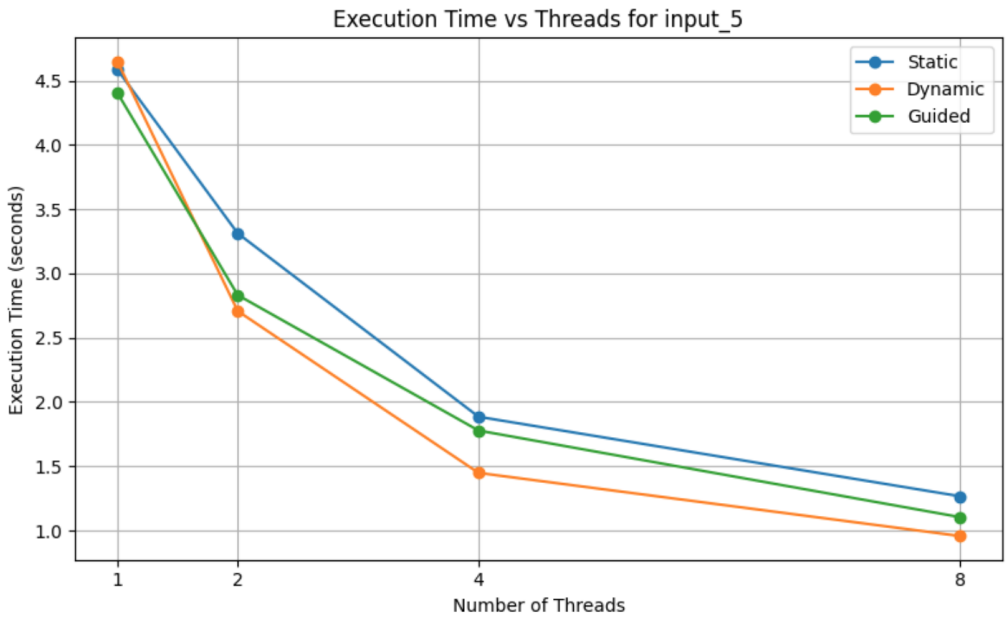


Figure 27: Execution Time vs Threads for `input_5` (HPC Cluster)

3.3 Interpretation of Results

- Speedup scales with core count until memory bandwidth becomes limiting.
- Thread-private arrays eliminate contention and improve write performance.
- Dynamic scheduling helps when input distribution is non-uniform.
- Logical cores offer minimal improvements beyond physical core saturation.