High-Performance Parallel Interpolation of Scattered Data on Structured Grids

Pratham Patel (202201485)
Ramkumar Patel (202201509)

April 15, 2025

Contents

1 Introduction

1.1 Problem Statement

Given a set of scattered data points in a 2D domain with known function values, the objective is to interpolate these values onto a structured mesh using bilinear interpolation. This problem has real-world applications in areas such as computer graphics, scientific visualization, FEA, medical imaging, and machine learning.

1.2 Objective of the Work

To implement an efficient and accurate bilinear interpolation algorithm and optimize it using parallel programming techniques for High Performance Computing (HPC) platforms.

1.3 Relevance in High Performance Computing

Interpolating large sets of scattered data in real-time or near real-time is computationally expensive. Parallelization and HPC can significantly reduce the execution time while maintaining accuracy.

2 Mathematical Foundation

2.1 Grid and Scattered Data Definitions

Let the scattered data points be:

$$S = \{(x_i, y_i, f_i) \mid i = 1, 2, \ldots, N\}$$

where $f_i = 1$ for all i.

The structured grid is defined as:

$$G = \{(X_i, Y_j) \mid X_i = i\Delta x, Y_j = j\Delta y,\ i, j = 0, 1, \ldots, M-1\}$$

with $\Delta x = \frac{X_{\max}}{M}$ and $\Delta y = \frac{Y_{\max}}{M}$.

2.2 Bilinear Interpolation

Bilinear Interpolation Weights

$$\begin{aligned} \mathrm{dx} &= \frac{x - X_i}{\Delta x} \\ \mathrm{dy} &= \frac{y - Y_j}{\Delta y} \\ w_{i,j} &= (1 - \mathrm{dx})(1 - \mathrm{dy}) \\ w_{i+1,j} &= \mathrm{dx}(1 - \mathrm{dy}) \\ w_{i,j+1} &= (1 - \mathrm{dx})\mathrm{dy} \\ w_{i+1,j+1} &= \mathrm{dxdy} \end{aligned}$$

3 Implementation Approach

3.1 Input File Generation and Structure

The input points are stored in binary file format using a utility ‘input fileMaker.c‘. This helps in generating files of varying size and complexity efficiently.

3.2 Serial Implementation Strategy

- Read the input binary file.
- For each scattered point, identify the grid cell.
- Compute weights and update the four surrounding grid points.

3.3 Parallelization Strategy

- Each thread handles a separate chunk of the scattered points.
- Grid updates are done using atomic operations or thread-local arrays to avoid race conditions.

3.4 Optimization Techniques Used

- Blocking and data locality improvements
- Minimizing false sharing
- Cache-friendly access patterns

3.5 Pseudocode and Architecture Diagram

Pseudocode for Parallel Bilinear Interpolation

```
Parallel for each scattered point (x, y):
    Locate the grid cell (i, j) that contains (x, y)

    Compute local distances:
        dx = (x - X_i) / delta_x
        dy = (y - Y_j) / delta_y

    Compute bilinear interpolation weights:
        w_ij      = (1 - dx) * (1 - dy)
        w_i+1_j   = dx * (1 - dy)
        w_i_j+1   = (1 - dx) * dy
        w_i+1_j+1 = dx * dy

    Atomically update the 4 surrounding grid values
```

4 Theoretical Analysis

4.1 Code Balance

Code Balance

The **code balance** is a key metric for analyzing memory-bound performance:

$$\text{Code Balance} = \frac{\text{Total Bytes Transferred}}{\text{Total Floating Point Operations (FLOPs)}}$$

4.2 Time and Space Complexity

Complexity Analysis

- **Time Complexity:** $\mathcal{O}(N)$, where N is the number of scattered data points.
- **Space Complexity:** $\mathcal{O}(M^2)$, where $M \times M$ is the size of the structured grid.

4.3 Cache Access Patterns

Cache Access Patterns

Memory access is largely linear due to structured grid layout. Using thread-local accumulation improves cache efficiency and minimizes contention, leading to reduced cache misses in parallel execution.

4.4 Parallel Efficiency Analysis

Parallel Efficiency

Parallel efficiency quantifies the effective usage of multiple threads:

$$E = \frac{T_1}{P \cdot T_P}$$

where:

- T_1: Time taken by the serial implementation.
- T_P: Time taken by the parallel implementation using P threads.

4.5 Unlimited HPC Resource Strategy

Use shared-memory + distributed hybrid MPI-OpenMP model with domain decomposition.

5 Experimental Evaluation

5.1 Hardware and Environment Setup

To compare the performance of different architectures, we analyzed two machines: Lab 207 PC and HPC Cluster. The key specifications are summarized in Table 1.

Specification	**Lab 207 PC**	**HPC Cluster**
Architecture	x86_64	x86_64
CPU Model	12th Gen Intel Core i5-12500	Intel Xeon E5-2620 v3 @ 2.40GHz
Clock Speed (GHz)	0.8 - 4.6	2.4
Cores / Threads	6 / 12	12 / 24
L1 Cache (KB)	288 (6 instances)	32 (data + instruction)
L2 Cache (KB)	7.5 MiB (6 instances)	256 KiB
L3 Cache (MB)	18	15
Memory (GB)	8 (DDR4)	64 (DDR4)
Memory Channels	2	2 (NUMA: 2 nodes)
Hyper-Threading	Yes	No

Table 1: Comparison of Hardware Architectures

5.1.1 Compilation and Optimization Flags

To optimize the runtime performance of our parallel code on a PC, we compiled it using GCC with high-performance flags: `-O3 -march=core-avx2 -mtune=core-avx2 -fopenmp -ffast-math -mavx2 -flto -funroll-loops -fomit-frame-pointer -fno-stack-protector -mfpmath=both` . These flags enable advanced vectorization, loop unrolling, OpenMP parallelism, and other architecture-specific optimizations.

5.2 Execution Time vs Number of Threads

5.2.1 Problem 1 [Nx=250, Ny=100, points=0.9 million, Maxiter=10]

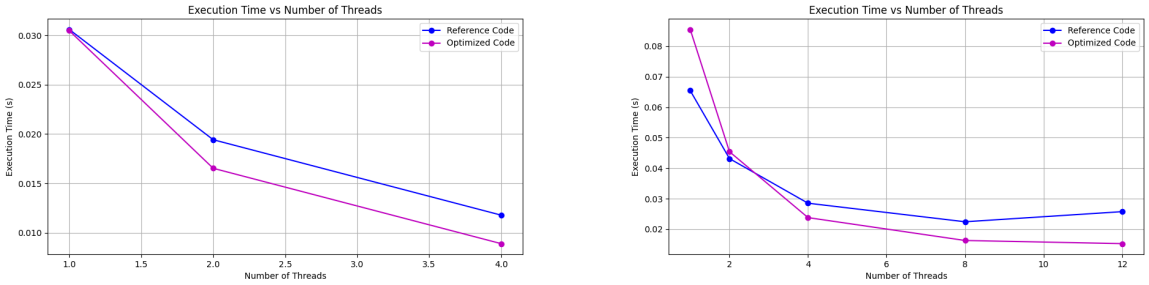

Figure 1: Execution time vs threads for Problem 1 on Lab PC (left) and HPC Cluster (right).

5.2.2 Problem 2 [Nx=250, Ny=100, points=5 million, Maxiter=10]

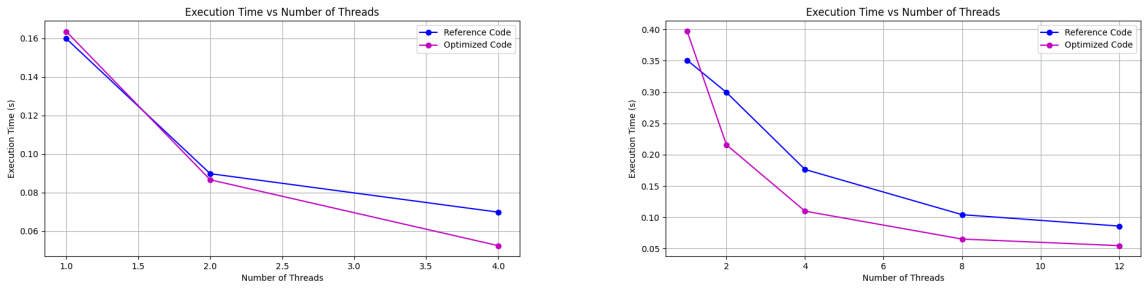

Figure 2: Execution time vs threads for Problem 2 on Lab PC (left) and HPC Cluster (right).

5.2.3 Problem 3 [Nx=500, Ny=200, points=3.6 million, Maxiter=10]

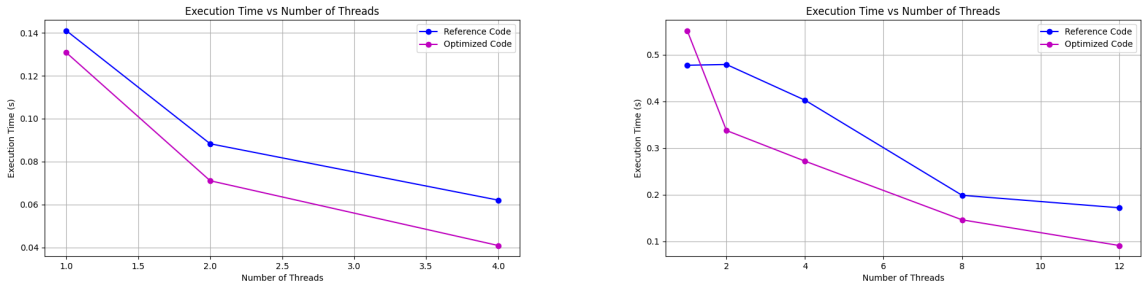

Figure 3: Execution time vs threads for Problem 3 on Lab PC (left) and HPC Cluster (right).

5.2.4 Problem 4 [Nx=500, Ny=200, points=20 million, Maxiter=10]

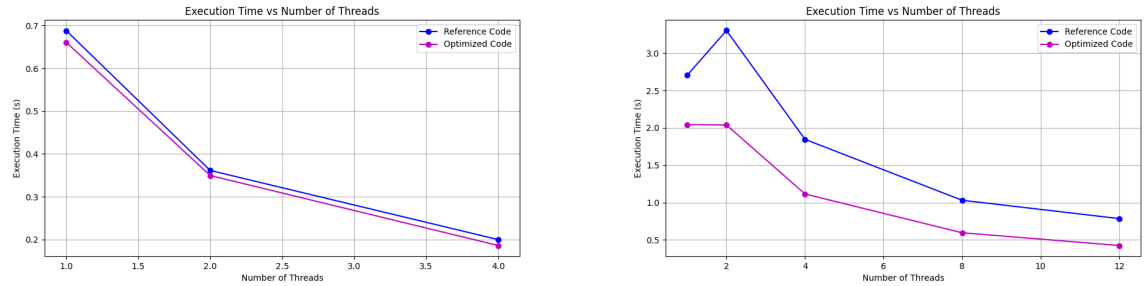

Figure 4: Execution time vs threads for Problem 4 on Lab PC (left) and HPC Cluster (right).

5.2.5 Problem 5 [Nx=1000, Ny=400, points=14 million, Maxiter=10]

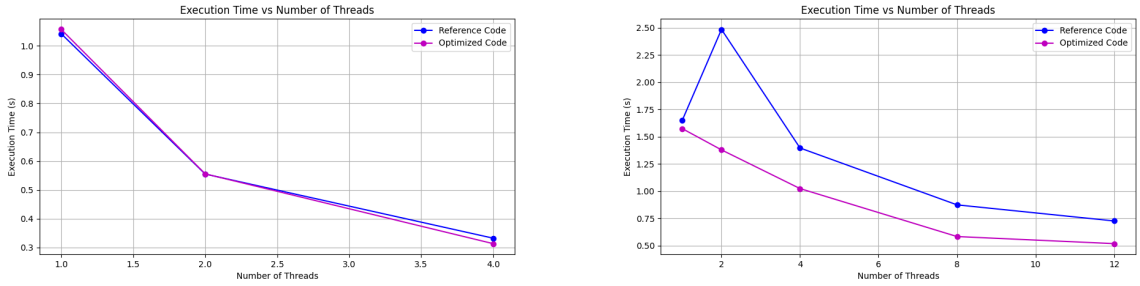

Figure 5: Execution time vs threads for Problem 5 on Lab PC (left) and HPC Cluster (right).

5.3 Speedup Analysis

5.3.1 Problem 1 [Nx=250, Ny=100, points=0.9 million, Maxiter=10]

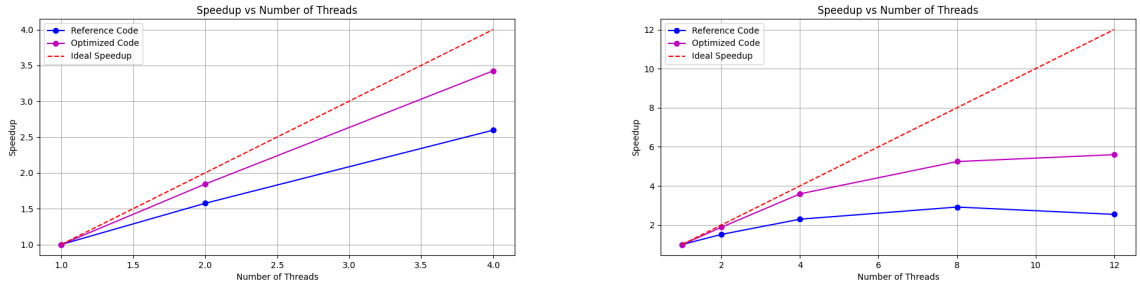

Figure 6: Speedup vs threads for Problem 1 on Lab PC (above) and HPC Cluster (right).

5.3.2 Problem 2 [Nx=250, Ny=100, points=5 million, Maxiter=10]

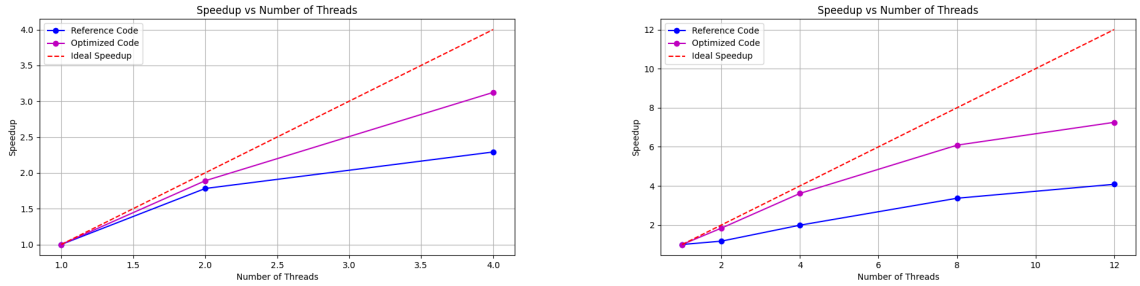

Figure 7: Speedup vs threads for Problem 2 on Lab PC (left) and HPC Cluster (right).

5.3.3 Problem 3 [Nx=500, Ny=200, points=3.6 million, Maxiter=10]

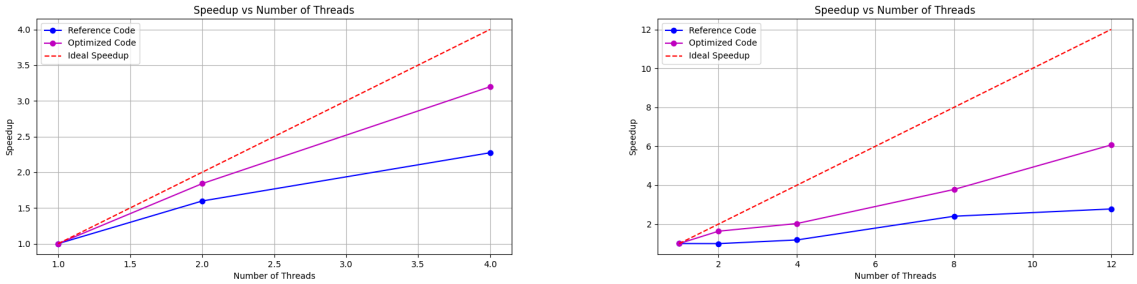

Figure 8: Speedup vs threads for Problem 3 on Lab PC (left) and HPC Cluster (right).

5.3.4 Problem 4 [Nx=500, Ny=200, points=20 million, Maxiter=10]

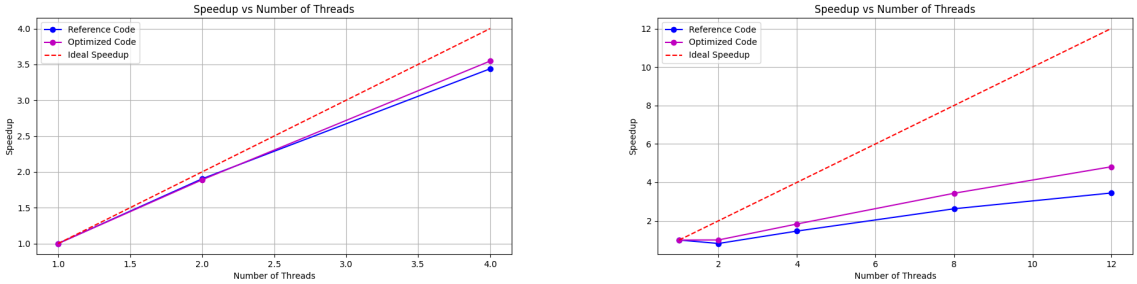

Figure 9: Speedup vs threads for Problem 4 on Lab PC (left) and HPC Cluster (right).

5.3.5 Problem 5 [Nx=1000, Ny=400, points=14 million, Maxiter=10]

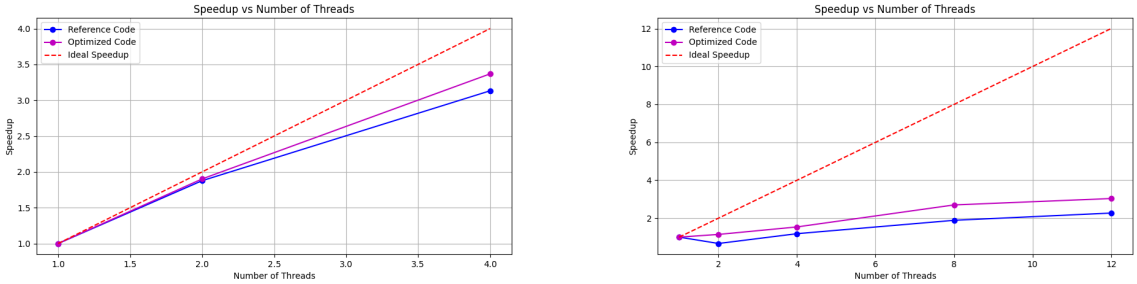

Figure 10: Speedup vs threads for Problem 5 on Lab PC (left) and HPC Cluster (right).

5.4 Hyperthreading Analysis on HPC Cluster

5.4.1 Problem 1 [Nx=250, Ny=100, points=0.9 million, Maxiter=10]

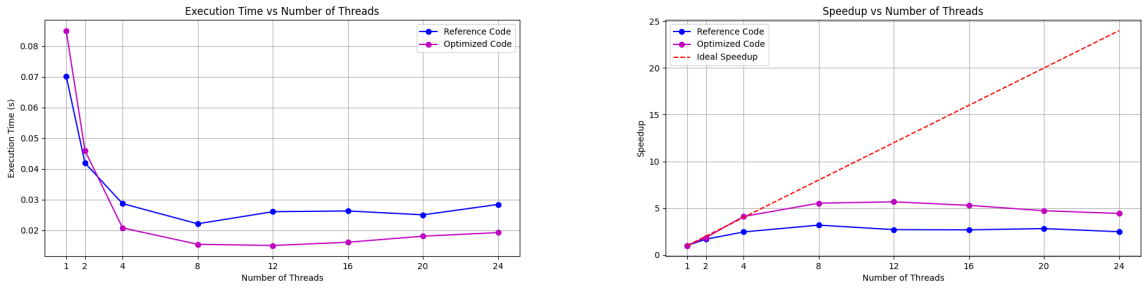

Figure 11: Hyperthreading effect on Cluster for Problem 1: Execution Time (left), Speedup (right).

5.4.2 Problem 2 [Nx=250, Ny=100, points=5 million, Maxiter=10]

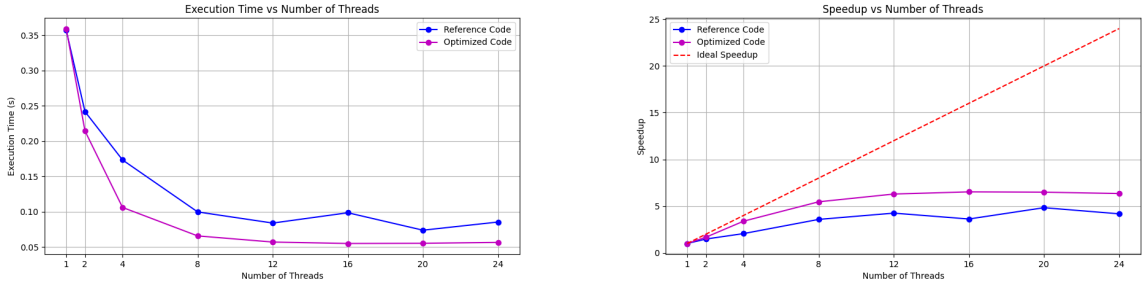

Figure 12: Hyperthreading effect on HPC Cluster for Problem 2: Execution Time (left), Speedup (right).

5.4.3 Problem 3 [Nx=500, Ny=200, points=3.6 million, Maxiter=10]

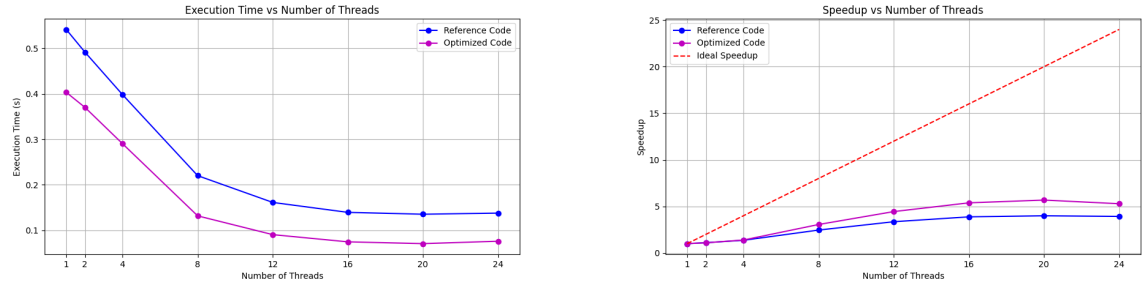

Figure 13: Hyperthreading effect on HPC Cluster for Problem 3: Execution Time (left), Speedup (right).

5.4.4 Problem 4 [Nx=500, Ny=200, points=20 million, Maxiter=10]

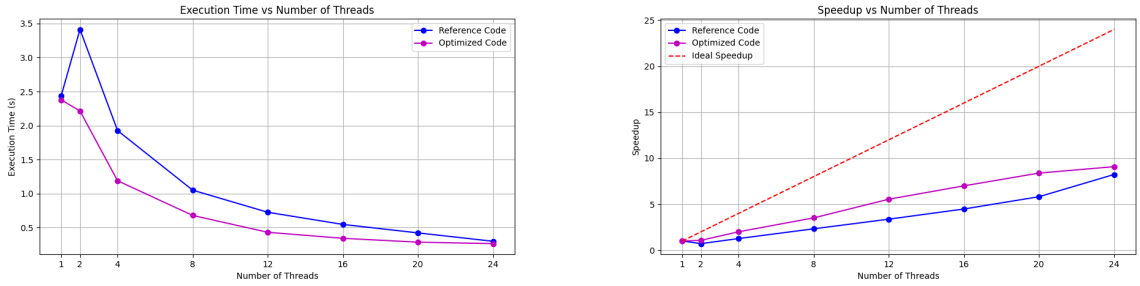

Figure 14: Hyperthreading effect on HPC Cluster for Problem 4: Execution Time (left), Speedup (right).

5.4.5 Problem 5 [Nx=1000, Ny=400, points=14 million, Maxiter=10]

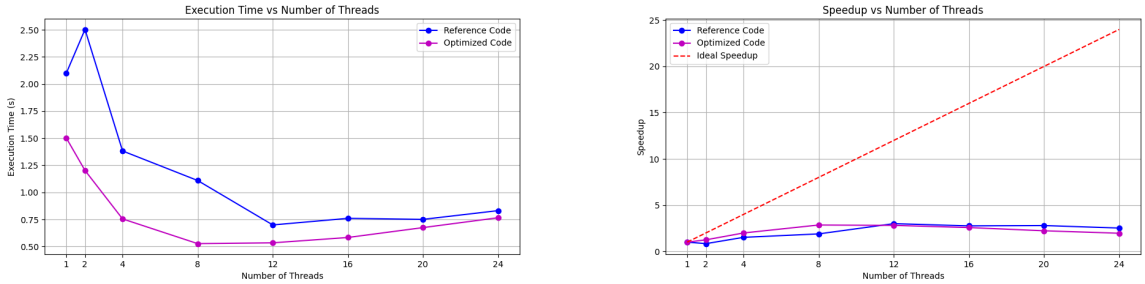

Figure 15: Hyperthreading effect on HPC Cluster for Problem 5: Execution Time (left), Speedup (right).

6 Cache Miss Profiling

Cache miss profiling was performed using `Instrument Software(Macbook)` tool to analyze how efficiently the cache is being utilized in our parallel implementation. High cache miss rates can significantly degrade performance, especially in memory-bound operations. The following figures represent snapshots from different phases of code execution and reveal hotspots in memory access patterns that may benefit from optimization.

6.0.1 Problem 1 [Nx=250, Ny=100, points=0.9 million, Maxiter=10]

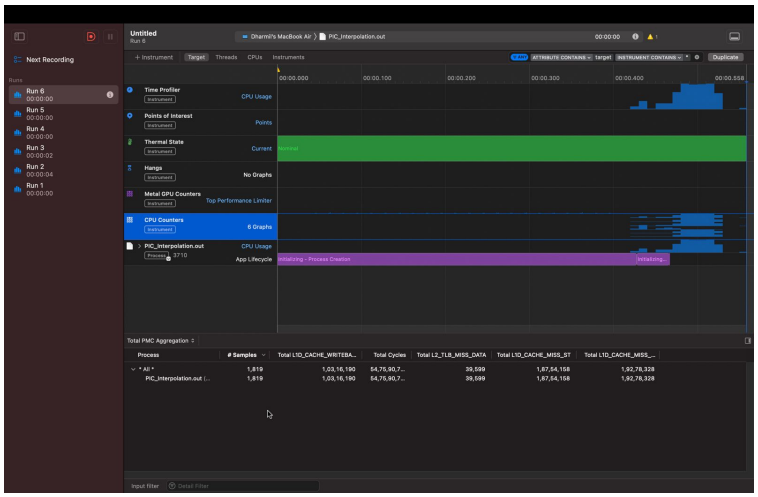

Figure 16: Cache miss profiling using `Instrument Software(Macbook)` for Problem 1.

6.0.2 Problem 2 [Nx=250, Ny=100, points=5 million, Maxiter=10]

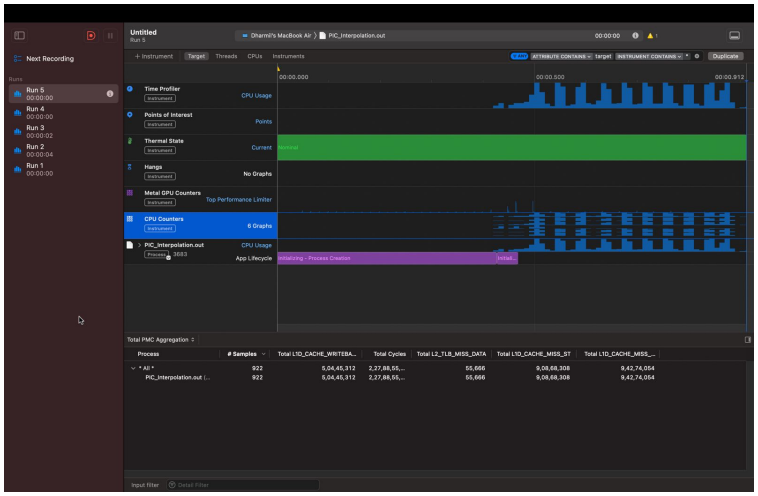

Figure 17: Cache miss profiling using `Instrument Software(Macbook)` for Problem 2.

6.0.3 Problem 3 [Nx=500, Ny=200, points=3.6 million, Maxiter=10]

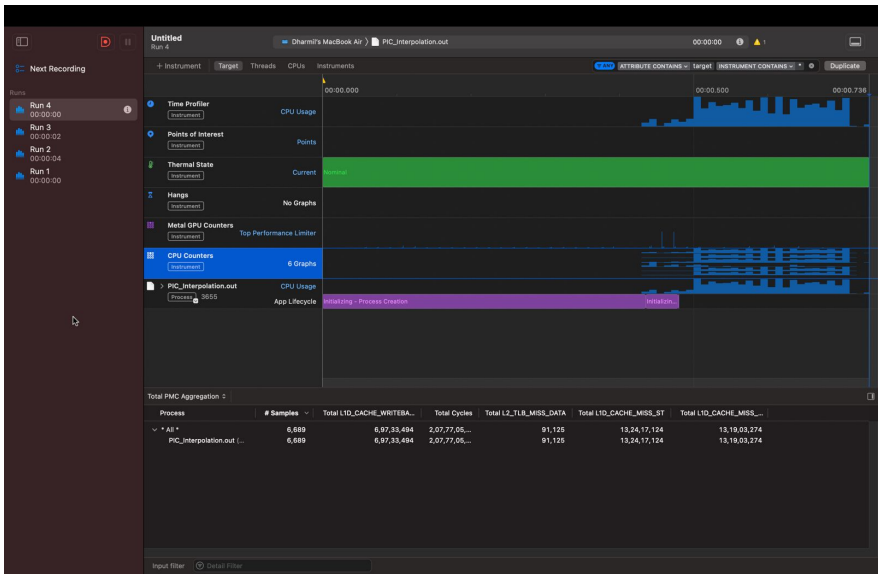

Figure 18: Cache miss profiling using `Instrument Software(Macbook)` for Problem 3.

6.0.4 Problem 4 [Nx=500, Ny=200, points=20 million, Maxiter=10]

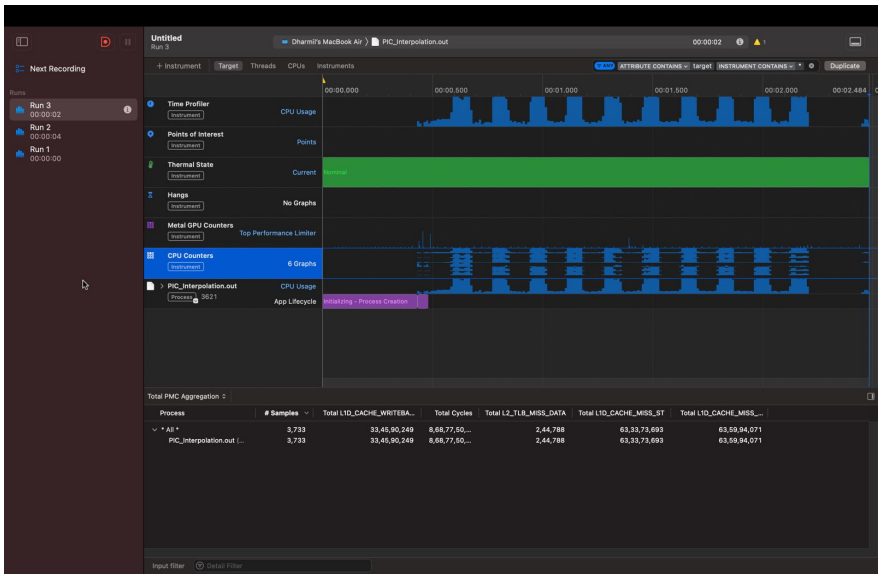

Figure 19: Cache miss profiling using `Instrument Software(Macbook)` for Problem 4.

6.0.5 Problem 5 [Nx=1000, Ny=400, points=14 million, Maxiter=10]

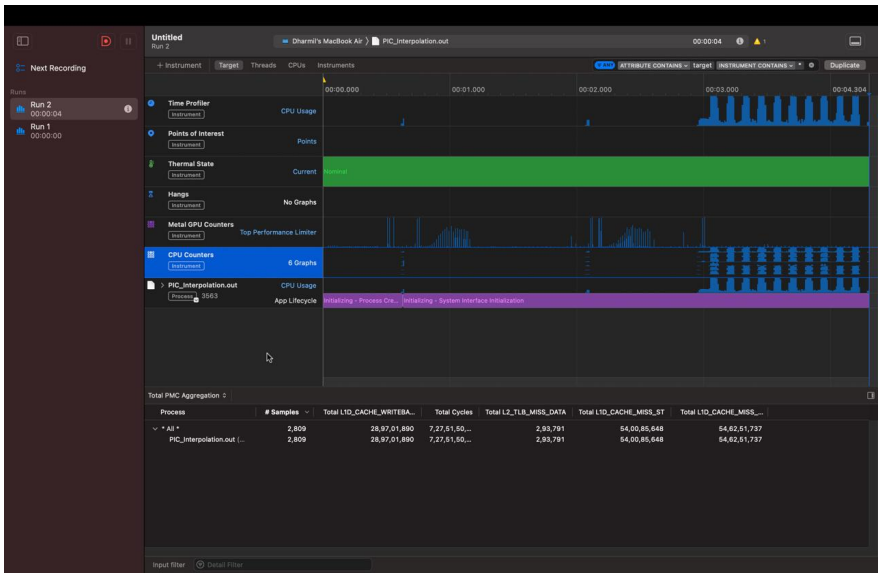

Figure 20: Cache miss profiling using `Instrument Software(Macbook)` for Problem 5.

7 Result Interpretation and Discussion

- Speedup trends and saturation analysis
- Effectiveness of parallelization strategy
- Observed vs theoretical performance

8 Conclusion and Future Work

8.1 Summary of Work

Efficient interpolation of scattered data onto structured grids using HPC.

8.2 Scope for Further Optimization

- GPU acceleration
- MPI for larger-scale distributed runs
- Load balancing improvements

8.3 Applications Beyond This Assignment

- Surface modeling from LIDAR data
- Geospatial visualizations
- Real-time scientific simulations