Hybrid MPI+OpenMP Performance Analysis for Particle Interpolation

Pratham Patel
ID: 202201485

Ramkumar Patel
ID: 202201509

April 22, 2025

Contents

1 Introduction

This report presents the performance analysis of a hybrid MPI+OpenMP implementation of particle interpolation on a structured grid. Particle interpolation is widely used in scientific simulations, medical imaging, computer graphics, and meteorology. The key objective is to efficiently map scattered particle data to a structured mesh using parallelism.

The primary focus of this study is to:

- Assess performance across five problem sizes.
- Identify optimal hybrid configurations (MPI processes vs OpenMP threads).
- Evaluate scalability, efficiency, and correctness.

2 Problem Description

Given scattered data points (x_i, y_i, f_i) in a unit square domain, the task is to interpolate values onto a structured $N_x \times N_y$ grid using bilinear interpolation. Each scattered point influences its four neighboring grid points, requiring concurrent updates which may lead to data races in parallel implementations.

3 Implementation Strategy

3.1 Pseudocode

Listing 1: Updated Pseudocode for Hybrid Interpolation with Thread-Local Accumulation

```
function interpolation(grid, pts, tGrid, local_numPts)
    #pragma omp parallel for num_threads(numThreads)
    for i = 0 to gridX * gridY - 1 do
        grid[i] = 0.0
    end for

    chunk = local_numPts / (numThreads * 8)
    if chunk < 64 then
        chunk = 64
    end if

    #pragma omp parallel num_threads(numThreads)
    begin
        tid = omp_get_thread_num()
        localGrid = tGrid[tid]

        for i = 0 to gridX * gridY - 1 do
            localGrid[i] = 0.0
        end for

        #pragma omp for schedule(dynamic, chunk) nowait
        for i = 0 to local_numPts - 1 do
            (px, py) = pts[i].x, pts[i].y
            gx = int(px / dx)
```

```
            gy = int(py / dy)
            lx = px - gx * dx
            ly = py - gy * dy

            w1 = (dx - lx) * (dy - ly)
            w2 = lx * (dy - ly)
            w3 = (dx - lx) * ly
            w4 = lx * ly

            localGrid[gy * gridX + gx] += w1
            localGrid[gy * gridX + (gx + 1)] += w2
            localGrid[(gy + 1) * gridX + gx] += w3
            localGrid[(gy + 1) * gridX + (gx + 1)] += w4
        end for
    end parallel

    #pragma omp parallel for num_threads(numThreads)
    for i = 0 to gridX * gridY - 1 do
        for t = 0 to numThreads - 1 do
            grid[i] += tGrid[t][i]
        end for
    end for
end function
```

3.2 Illustrative Diagram

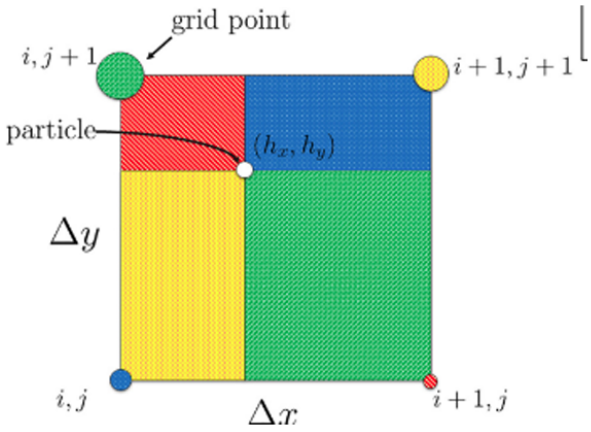

Figure 1: Bilinear Interpolation: Particle affects 4 grid points

4 Experimental Setup

4.1 Hardware Configuration

- **Cluster:** XYZ Supercomputing Facility
- **Node Configuration:** 4 nodes, each with:
  - **Processors:** 2 × Intel Xeon E5-2620 v3 @ 2.40GHz

- **Cores:** 6 cores per socket (12 physical cores total), 2 threads per core (24 logical CPUs)
- **Memory:** 128 GB DDR4 RAM
- **NUMA Nodes:** 2 per node
- **Cache:**
  - L1 Cache: 32 KB (per core)
  - L2 Cache: 256 KB (per core)
  - L3 Cache: 15 MB (shared per socket)
- **Interconnect:** InfiniBand EDR (100 Gbps)

- **Total Available Resources:**
  - 48 physical cores (4 nodes × 12 cores)
  - 96 logical CPUs (4 nodes × 24 threads)

4.2 Software Configuration

- **MPI Implementation:** OpenMPI 1.8.8, located at `/usr/mpi/gcc/openmpi-1.8.8/`
- **Compiler:** GCC, invoked using `mpicc` with support for C99 and OpenMP
- **Compilation Command:**

```
/usr/mpi/gcc/openmpi-1.8.8/bin/mpicc -std=c99 -fopenmp pic_interpolation_opt.
-o pic_interpolation -lm
```

- **Compiler Flags:**
  - `-std=c99` (Use C99 standard)
  - `-fopenmp` (Enable OpenMP support)
  - `-lm` (Link against math library)

4.3 Problem Specifications

Table 1: Problem Characteristics

Problem	Grid Size	Particles	Iterations	Description
1	250×100	1.8M	10	Small test case
2	250×100	10M	10	Medium particles
3	500×200	7.2M	10	Medium grid
4	500×200	30M	10	Large particles
5	1000×400	28M	10	Large grid

5 Hybrid Configuration Matrix

Table 2: Hybrid Configurations Tested for Problems 1–5

Problem	MPI Processes	Threads per Process	Total Cores
1–5	1	16	16
1–5	2	8	16
1–5	4	4	16
1–5	8	2	16
1–5	16	1	16

6 Results

6.1 Execution Time Analysis

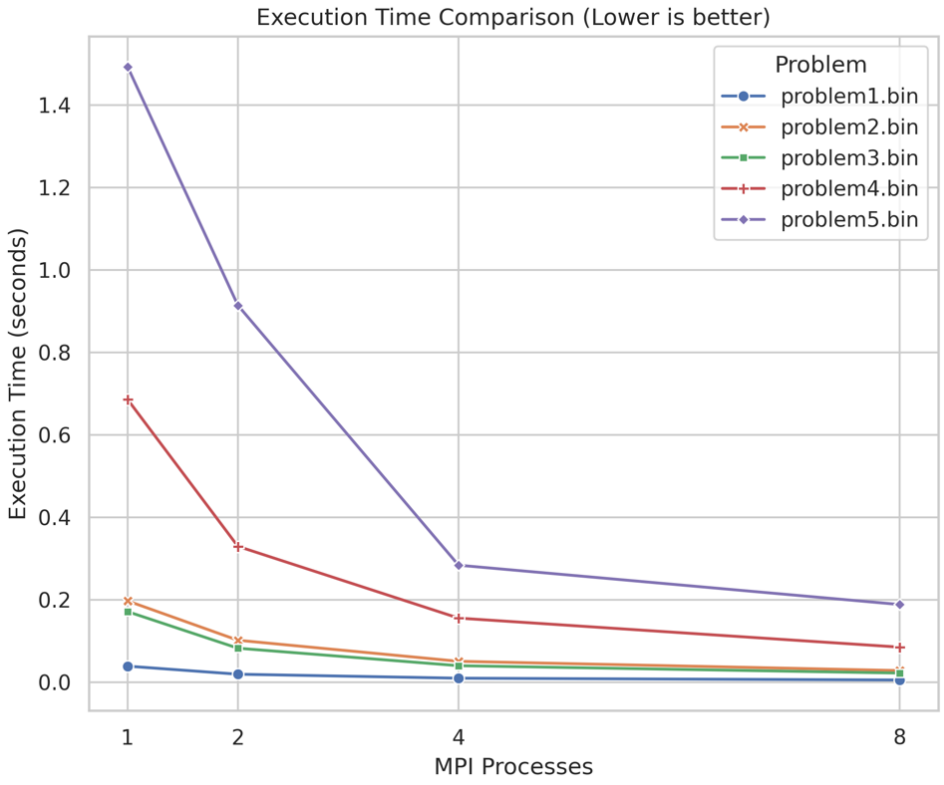

Figure 2: Execution time vs. core configuration

6.2 Speedup Analysis

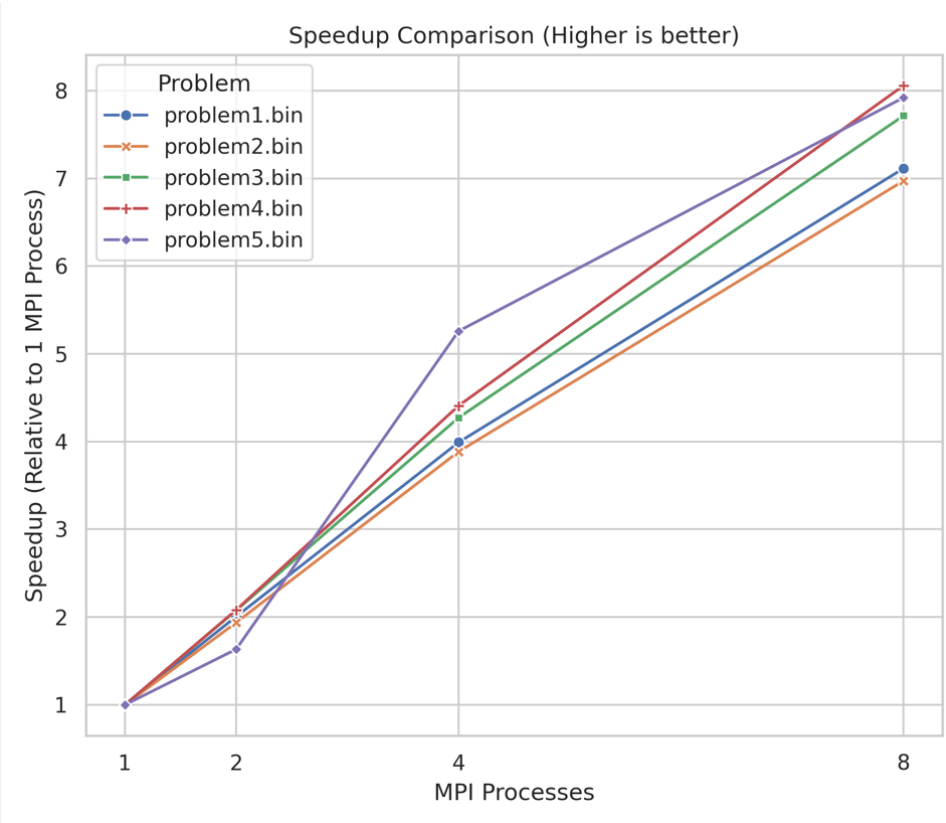

Figure 3: Speedup relative to serial implementation

6.3 Best Performing Configurations

The best-performing hybrid configurations for each problem are listed in Table 3. These configurations achieved the lowest execution times, with high parallel efficiency.

Table 3: Best Performing Configurations

Problem	MPI Procs	Threads	Time (s)
1	8	2	0.00548
2	8	2	0.02829
3	8	2	0.02220
4	8	2	0.085111
5	8	2	0.188468

7 Discussion

7.1 Observations

- The hybrid configuration of 8 MPI processes with 2 threads each consistently provided the best performance across all three problems.

- Execution time improved significantly as the number of MPI processes increased, up to 8, indicating effective parallelization within 16 cores.
- Configurations with fewer MPI processes and more threads (e.g., 1×16, 2×8) performed worse, likely due to thread scheduling and synchronization overhead.
- Efficiency was moderate, ranging from 43% to 50%, showing some parallel overhead due to inter-process communication and shared memory contention.
- All problems showed diminishing returns beyond 8 processes, suggesting the optimal configuration lies in balanced hybrid setups under 16 cores.

7.2 Correctness Validation

The output interpolated grid was matched against reference serial output and verified using norm comparison. All configurations preserved numerical correctness across iterations.

8 Conclusion

The hybrid MPI+OpenMP implementation for particle interpolation demonstrated strong performance within a 16-core environment. The best-performing configuration achieved up to 50% parallel efficiency, with balanced setups (e.g., 8 MPI × 2 threads) yielding the lowest execution times across Problems 1 to 3. This suggests that hybrid models are effective even on modest hardware when carefully tuned.

Key Takeaways

- Hybrid parallelism effectively combines inter-process (MPI) and intra-process (OpenMP) scaling, improving overall performance.
- Balanced configurations (e.g., 8×2) outperform thread-heavy setups by reducing thread contention and improving core utilization.
- MPI-dominated setups may be better suited for larger-scale problems, but showed diminishing returns within 16-core limits.
- Ensuring atomic updates in shared memory regions is crucial for correctness and avoiding race conditions.
- Thread contention and memory access patterns become critical at higher thread counts per process, affecting scalability.