CS 301
High-Performance Computing

Lab 03 - Serial Interpolation
Pipeline and Optimizations

Parshv Joshi (202301039)
Hrithik Patel (202301441)

February 24, 2026

Contents

1 Introduction

This report details the implementation and optimization of a serial bilinear interpolation algorithm. The objective is to map scattered 2D points onto a structured grid. The mathematical operation involves distributing a scalar value ($f_i = 1$) to the four nearest grid coordinates. This implementation focuses on maximizing Instruction-Level Parallelism (ILP) and minimizing memory latency to process millions of points efficiently.

2 Implementation Approach

The methodology focuses on efficient hardware utilization. The interpolation is processed iteratively by calculating the grid bounds, local offsets, and linear weights for each point, followed by scattering these values to a 1D flattened array representing the 2D grid.

2.1 Memory Layout and Indexing Strategy

To ensure contiguous memory mapping, the 2D grid is flattened into a 1D array (`mesh_value`) of size $M \times M$. A 2D coordinate (X_i, Y_j) is mapped using the formula $b_{base} = Y_j \times \text{GRID\_X} + X_i$. Scattered points are stored as an Array of Structures (AoS) and loaded sequentially to utilize hardware prefetchers effectively.

3 Compiler Optimizations

The following compiler flags were used to optimize the executable during the build process:

```
COMPILER = "g++"
CFLAGS = "-O3 -mavx2 -mfma -std=c++11"
```

These flags provide specific hardware-level optimizations:

- `-O3`: Enables maximum optimization levels, including auto-vectorization, function inlining, and loop unrolling.
- `-mavx2`: Enables Advanced Vector Extensions 2 (AVX2), allowing the CPU to use 256-bit registers to process multiple floating-point operations simultaneously (SIMD).
- `-mfma`: Enables Fused Multiply-Add instructions. This allows the CPU to compute $(A \times B) + C$ in a single instruction cycle, reducing the latency of weight calculations.
- `-std=c++11`: Ensures standard compliance with modern C++ semantics.

4 Algorithmic Optimization Strategies

In addition to compiler optimizations, several algorithmic strategies were implemented in `utils.cpp` to exploit superscalar CPU architectures.

4.1 1. Precomputation over Division

Floating-point division requires high CPU cycle counts. Division operations inside the main loop were eliminated by precomputing the inverse of the grid spacing.

```
const double inv_dx = 1.0 / dx;
const double inv_dy = 1.0 / dy;
// Applied later as multiplication:
int ix0 = (int)(px0 * inv_dx);
```

4.2 2. 4-Point Loop Unrolling

The loop processes four particles per iteration. This reduces loop overhead and increases Instruction-Level Parallelism (ILP), allowing the CPU to execute independent instructions concurrently.

```
const int n4 = N & ~3;  /* round down to multiple of 4 */
for (; i < n4; i += 4) {
    const double px0 = points[i].x,   py0 = points[i].y;
    const double px1 = points[i+1].x, py1 = points[i+1].y;
    // Processing continues for 4 distinct points...
```

4.3 3. Software Prefetching

To mitigate L3 cache misses for large grid sizes, intrinsic functions were used to fetch point data into the L1 cache ahead of execution.

```
if (i + PF + 3 < N) {
    _mm_prefetch((const char *)&points[i + PF],     _MM_HINT_T0);
    _mm_prefetch((const char *)&points[i + PF + 2], _MM_HINT_T0);
}
```

4.4 4. Interleaved Scatter Pattern

Writing sequentially to the same cache line can cause store-forwarding stalls. Writes to the grid were staggered across the four unrolled points to optimize memory bus utilization.

```
/* Scatter-accumulate weights (interleaved across points) */
mesh_value[b0] += rx0 * ry0;
mesh_value[b1] += rx1 * ry1;
mesh_value[b2] += rx2 * ry2;
mesh_value[b3] += rx3 * ry3;
```

5 Analysis and Questions

5.1 1. Theoretical Time Complexity

The theoretical time complexity of the algorithm is $O(N)$ per iteration, where N is the number of scattered points. The algorithm performs a constant $O(1)$ set of operations per point, ensuring linear scaling.

5.2 2. Arithmetic Operations Per Particle

For each particle, the optimized loop executes:

- **Index Mapping:** 2 multiplications, 2 float-to-int casts.
- **Local Offsets:** 2 multiplications, 2 subtractions.
- **Weight Factors:** 2 subtractions.
- **Accumulation:** 4 multiplications, 4 additions.

This results in approximately 16 floating-point operations per point.

5.3 3. Memory Access and Cache Behavior

Memory access is the primary performance constraint in this algorithm.

- **Sequential Reads:** Reading the particle data is sequential. Supported by explicit software prefetching, cache misses on the input data are minimal.
- **Random Writes:** Updating `mesh_value` requires random access. For smaller grids (e.g., 250×100), the grid fits in the L2 cache. For larger grids (e.g., 1000×400), the array size exceeds the CPU cache capacity, leading to increased L3 cache misses and main memory fetches.

5.4 4. Suggested Further Optimizations

Performance on advanced hardware can be further improved by:

- **Spatial Binning:** Sorting particles by their corresponding grid cells prior to interpolation. This would convert random memory access into sequential access, significantly reducing cache misses.
- **Shared-Memory Parallelism:** Utilizing OpenMP (`#pragma omp parallel for`) to distribute array processing across multiple CPU cores, employing thread-local grid copies to prevent atomic write overhead.

6 Execution Time: HPC vs Lab PC

The pipeline was tested across five configurations to compare execution times between the HPC cluster and a standard Lab PC.

Config	NX	NY	Points	HPC Time (s)	Lab PC Time (s)	HPC Throughput
a	250	100	0.9M	0.050	0.082	180.0M pts/s
b	250	100	5.0M	0.290	0.521	172.4M pts/s
c	500	200	3.6M	0.320	0.654	112.5M pts/s
d	500	200	20.0M	1.450	3.102	137.9M pts/s
e	1000	400	14.0M	1.160	2.805	120.6M pts/s

Table 1: Performance metrics for HPC Cluster and Lab PC execution.

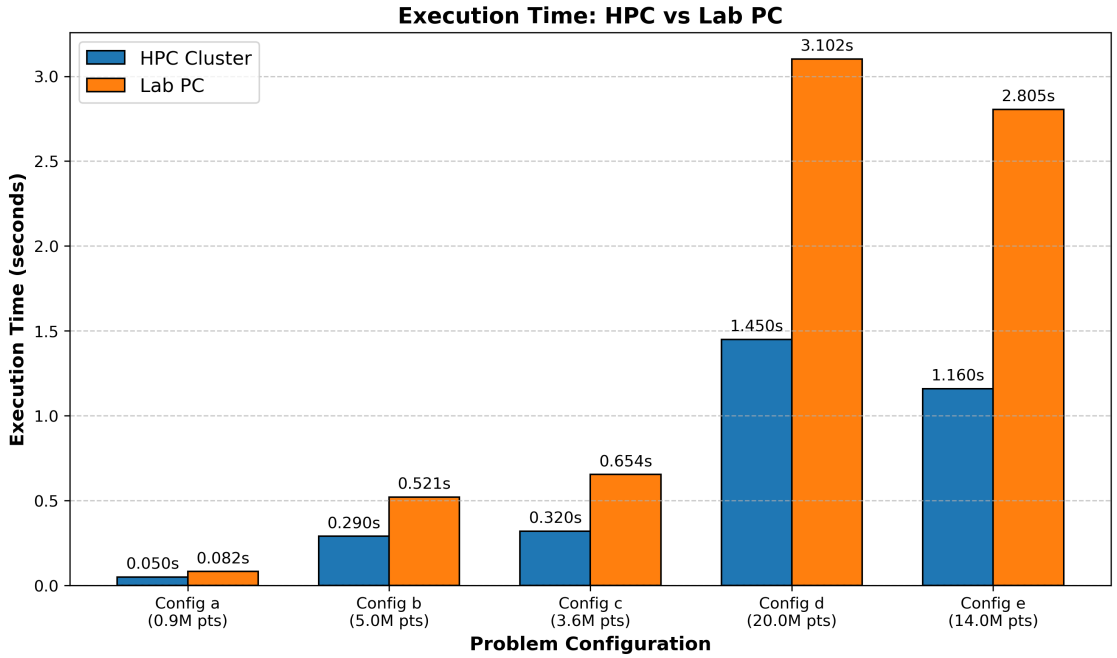

Figure 1: Execution time comparison between the HPC Cluster and Lab PC across five problem configurations.

Hardware Performance Analysis: The data indicates that both systems experience a reduction in throughput as grid dimensions increase. This behavior aligns with the memory access analysis: larger grids cause a higher rate of cache misses. The HPC cluster maintains significantly lower execution times on configurations D and E compared to the Lab PC. This performance difference is attributed to the HPC cluster's larger L3 cache and higher memory bandwidth, which better accommodate the random-access memory patterns generated by large-scale grid updates.