# High Performance Computing – Lab Report 03

Serial Interpolation on Scattered Data

**Tirth Kaushal Gandhi** – 202301413
**Dhyey Patel** – 202301415

*Course: CS301 / High Performance Computing*

February 24, 2026

**Abstract**

In this lab, we implemented a serial algorithm to interpolate up to 20 million scattered points onto a structured 2D grid. Because the points are random, updating the grid causes a lot of cache misses. We initially tried to fix this by sorting the points by their grid cells (to get better cache hits), but we found out that moving millions of points around in memory actually made the code slower. Instead, we got our best execution times by keeping things simple: we changed our data layout from an Array of Structures (AoS) to a Structure of Arrays (SoA), and we swapped out slow division operations for faster multiplications.

## 1 Introduction

The goal of this assignment is to take scattered data points and map them onto a structured mesh grid. For every point, we have to find the four nearest grid corners, calculate their weights based on distance, and add the values to the grid.

The main problem with doing this quickly is how the CPU handles memory. Because the input points are totally random, the code constantly jumps around different parts of the `mesh_value` array. This confuses the CPU cache and causes it to constantly wait for data from the slow RAM. Our goal was to find ways to make the memory access faster and reduce the total execution time.

## 2 Hardware Details

We used the `lscpu` command to check the hardware specs. The HPC Cluster has a 16-core Xeon processor, and the Lab PC features a modern 6-core 12th Gen Intel i5 processor.

Table 1: System Architecture Specifications

| **Component** | **Cluster** | **Lab PC** |
|---|---|---|
| CPU Model | Intel Xeon E5-2640 v3 @ 2.60GHz | Intel Core i5-12500 @ 4.60GHz |
| Cores per Socket | 8 (2 Sockets) | 6 (1 Socket) |
| L1 Data Cache | 32 KB (per core) | 48 KB (per core) |
| L2 Cache | 256 KB (per core) | 1.25 MB (per core) |
| L3 Cache | 20 MB (per socket) | 18 MB (shared) |

# 3 Implementation Approach

We tried a few different ways to make the code run faster. Here is what worked and what didn't.

## 3.1 The Failed Idea: Sorting the Points

At first, we thought about using a Counting Sort to group all the points by which grid cell they belong to. The idea was that if we process all the points for cell $(0, 0)$ at the same time, the CPU cache would get a lot of hits. However, when we coded it, it ran slower than the naive version. Sorting 20 million points meant we had to read and write huge arrays multiple times. The time we lost just moving data around in RAM was much worse than the time we saved from getting cache hits. So, we dropped the sorting idea entirely.

## 3.2 Optimization 1: Data Layout (AoS to SoA)

Instead of sorting, we just changed how the points are stored. Originally, we used an Array of Structures (AoS) where the $X$ and $Y$ coordinates were stored right next to each other in memory. We switched this to a Structure of Arrays (SoA), meaning we have one big array just for $X$'s and one big array just for $Y$'s. This is better for the CPU because when it fetches the $X$ coordinates, the cache line is filled entirely with useful data, making read operations much faster.

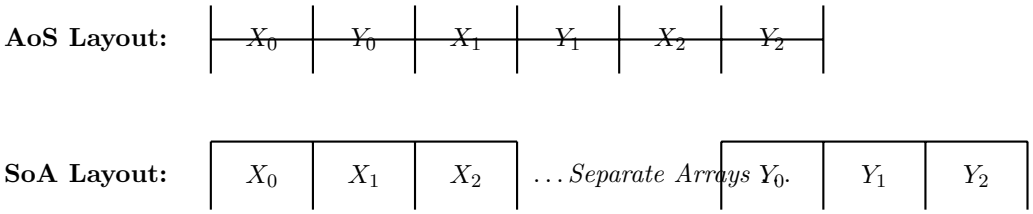



Figure 1: Visualizing the shift from AoS to SoA. Splitting the coordinates isolates memory streams, allowing the CPU to load pure X-coordinates into the cache without wasting space on Y-coordinates.

## 3.3 Optimization 2: Avoiding Division

Inside the main loop, calculating the grid index requires dividing by $dx$ and $dy$. Division is a very slow operation for a CPU. Since $dx$ is just $\frac{1.0}{NX}$, we pre-calculated $NX$ and $NY$ as doubles and just used multiplication instead. This simple math trick saved millions of slow clock cycles inside the loop.

## 3.4 Grid Indexing and Interpolation Workflow

To map scattered points to the grid, we multiply the coordinates by $NX$ and $NY$ to find the bottom-left anchor $(i, j)$ of the grid cell. From there, we use the formula `base = j * GRID_X + i` to update the 1D mesh array.

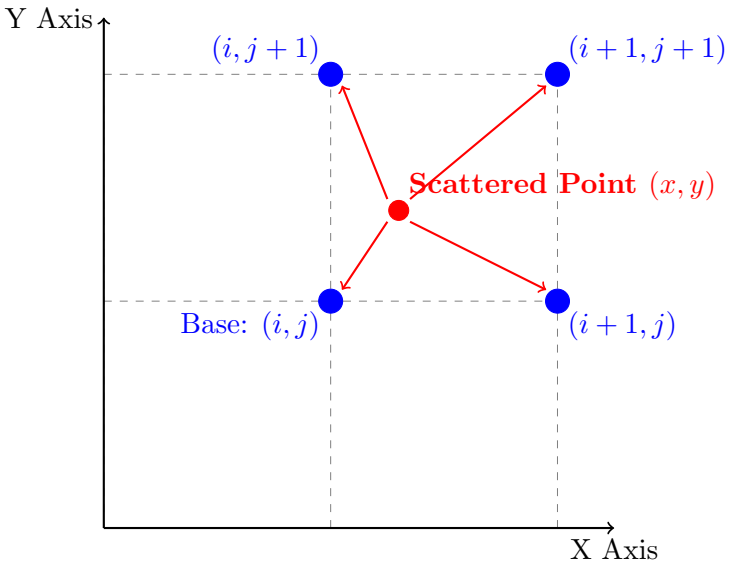



Figure 2: Grid mapping strategy. The algorithm finds the base corner and computes weights for the four surrounding structured grid cells.

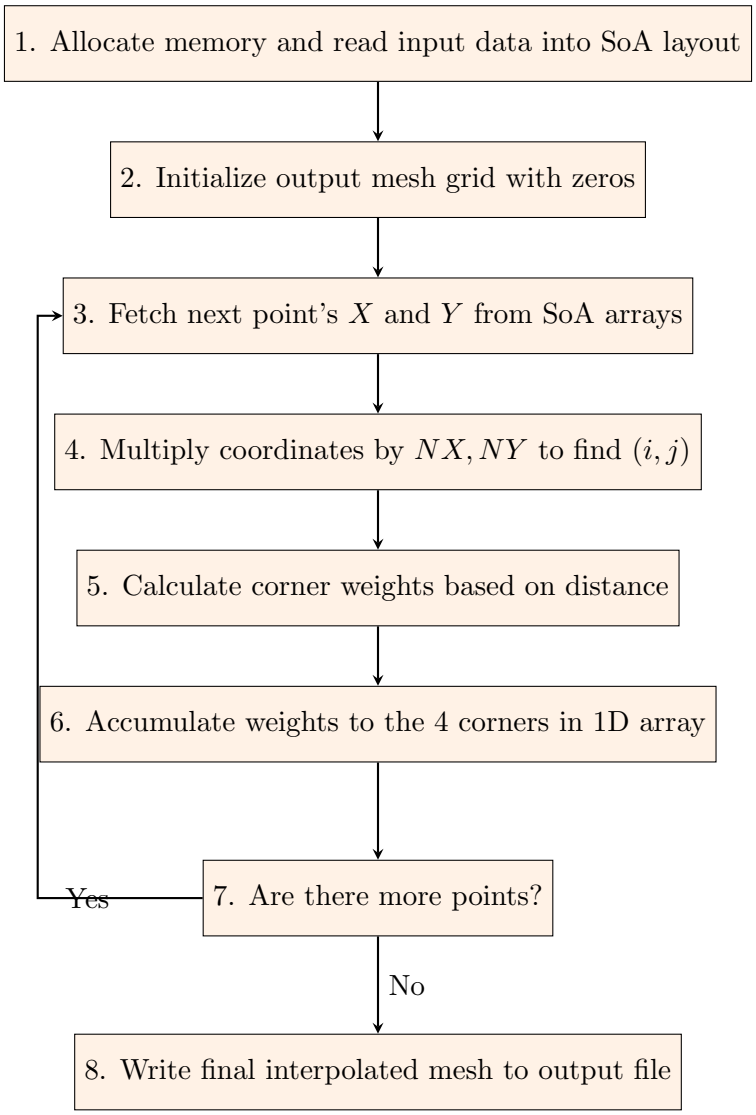



Figure 3: Logical flowchart of the final serial interpolation algorithm.

# 4 Analysis and Questions

## 4.1 1. Theoretical Time Complexity

The overall time complexity is $O(N + M^2)$, where $N$ is the number of scattered points and $M$ is the grid size.

- Resetting the $M \times M$ grid at the start of each iteration takes $O(M^2)$.
- The main loop goes through each point exactly once without any nested loops, which takes $O(N)$.

## 4.2 2. Arithmetic Operations per Particle

Inside the optimized interpolation loop, each particle requires roughly:

- 4 typecasts (converting floats to integers for grid indices).
- 6 subtractions (for finding local boundaries).
- 8 multiplications (calculating weights and updating the grid).
- 4 additions (adding the final values to the mesh).

This comes out to about **22 simple operations per particle**. We made sure there are exactly zero divisions in the loop.

## 4.3 3. Memory Access Analysis and Cache Behavior

By using the SoA layout, reading the $X$ and $Y$ arrays gives us perfect spatial locality (the CPU just reads straight down the line). However, because the points are random, writing to the `mesh_value` array still causes some cache misses. We decided to just let the hardware prefetcher handle these random writes, because our tests showed that trying to manually sort the points to fix the cache misses caused too much memory overhead.

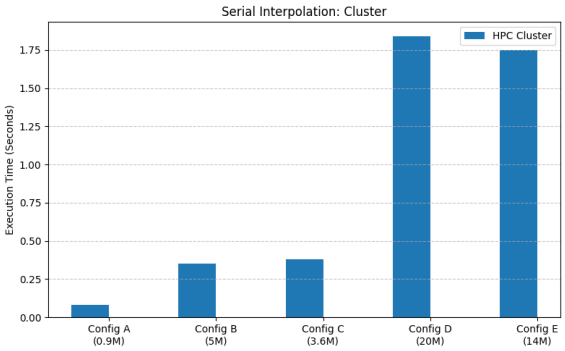
## 4.4 4. Further Optimizations

If we were running this under better hardware assumptions (multi-core or specialized instruction sets), we could:

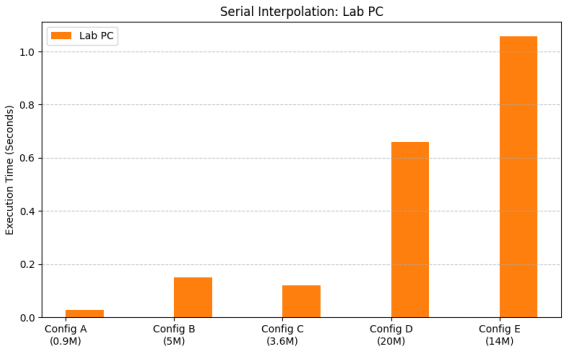
- **Use OpenMP:** We could split the iteration loop among multiple threads so the CPU cores can process different parts of the data at the same time.
- **Use Vectorization (SIMD):** We could use AVX compiler flags or intrinsics to force the CPU to compute the weights for multiple particles simultaneously in a single clock cycle.

# 5 Results: Execution Time

We compiled the code with the `-O3` and `-march=native` flags and ran it for the 5 configurations given in the lab manual.




(a) Cluster: Execution Time vs Problem Index




(b) Lab PC: Execution Time vs Problem Index

Figure 4: Performance comparison across 5 configurations.

### 5.1 Commentary on Hardware Differences

In a strictly serial task like this, single-core performance dictates the execution time. Our results show that the Lab PC performs exceptionally well across all configurations. This is because the Lab PC is equipped with a modern 12th Gen Intel i5 CPU boasting a max clock speed of 4.60 GHz.

In contrast, the HPC Cluster node relies on an older Intel Xeon E5-2640 v3. While the Xeon is great for parallel, multi-threaded workloads with its 16 cores and massive 20 MB L3 cache, its slower single-core clock speed (2.60 GHz) makes it slower for this specific serial assignment. Even though the cluster has a slightly larger L3 cache (20 MB vs 18 MB), the Lab PC's modern architecture and much faster clock speed allow it to easily outpace the cluster server on single-thread efficiency.

## 6 Conclusion

This lab was a great lesson in how memory bottlenecks work. We learned that "smart" ideas like sorting data to get perfect cache hits don't always work in reality because moving millions of items in RAM is very slow. The best way to make this serial code fast was to keep it simple: we just improved how the data was stored in memory (SoA) and avoided slow math operations like division.