CS 301
High-Performance Computing

Scattered Points to Mesh Interpolation

GROUP 26

Mihirkumar Patel (202201506)
Dishank Thakkar (202201518)

Contents

1 Introduction

In this HPC lab assignment, we focus on interpolating scattered data points onto a structured 2D mesh using bilinear interpolation. The main objective is to implement both serial and parallel versions of the algorithm and analyze their performance. The interpolation technique is widely used in fields like computer graphics, scientific simulations, and medical imaging. The challenge lies in efficiently handling data dependencies and avoiding race conditions during parallel execution. We work with large datasets and use concepts such as data locality, thread safety, and scalability. The assignment also involves performance profiling and analysis of cache behavior to improve the implementation.

2 Hardware Details

2.1 Hardware Details for LAB207 PCs

- **Architecture:** x86_64
- **CPU op-mode(s):** 32-bit, 64-bit
- **Address sizes:** 46 bits physical, 48 bits virtual
- **Byte Order:** Little Endian
- **CPU(s):** 12
- **On-line CPU(s) list:** 0-11
- **Vendor ID:** GenuineIntel
- **Model name:** 12th Gen Intel(R) Core(TM) i5-12500
- **CPU family:** 6
- **Model:** 151
- **Thread(s) per core:** 2
- **Core(s) per socket:** 6
- **Socket(s):** 1
- **Stepping:** 5
- **CPU max MHz:** 4600.0000
- **CPU min MHz:** 800.0000
- **BogoMIPS:** 5990.40

- **Flags:** fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb intel_pt sha_ni xsaveopt xsavec xgetbv1 xsaves split_lock_detect user_shstk avx_vnni dtherm ida arat pln pts hwp hwp_notify hwp_act_window hwp_epp hwp_pkg_req hfi vnmi umip pku ospke waitpkg gfni vaes vpclmulqdq tme rdpid movdiri movdir64b fsrm md_clear serialize pconfig arch_lbr ibt flush_l1d arch_capabilities
- **Virtualization features:**
  - **Virtualization:** VT-x
- **Caches (sum of all):**
  - **L1d:** 288 KiB (6 instances)
  - **L1i:** 192 KiB (6 instances)
  - **L2:** 7.5 MiB (6 instances)
  - **L3:** 18 MiB (1 instance)
- **NUMA:**
  - **NUMA node(s):** 1
  - **NUMA node0 CPU(s):** 0-11
- **Vulnerabilities:**
  - **Gather data sampling:** Not affected
  - **Itlb multihit:** Not affected
  - **L1tf:** Not affected
  - **Mds:** Not affected
  - **Meltdown:** Not affected
  - **Mmio stale data:** Not affected
  - **Reg file data sampling:** Not affected
  - **Retbleed:** Not affected
  - **Spec rstack overflow:** Not affected
  - **Spec store bypass:** Mitigation; Speculative Store Bypass disabled via prctl
  - **Spectre v1:** Mitigation; usercopy/swapgs barriers and __user pointer sanitization
  - **Spectre v2:** Mitigation; Enhanced / Automatic IBRS; IBPB conditional; RSB filling; PBRSB-eIBRS SW sequence; BHI BHI_DIS_S
  - **Srbds:** Not affected
  - **Tsx async abort:** Not affected

2.2 Hardware Details for HPC Cluster

- **CPU op-mode(s):** 32-bit, 64-bit
- **Byte Order:** Little Endian
- **CPU(s):** 24
- **On-line CPU(s) list:** 0-23
- **Thread(s) per core:** 2
- **Core(s) per socket:** 6
- **Socket(s):** 2
- **NUMA node(s):** 2
- **Vendor ID:** GenuineIntel
- **CPU family:** 6
- **Model:** 63
- **Model name:** Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
- **Stepping:** 2
- **CPU MHz:** 2602.031
- **BogoMIPS:** 4804.69
- **Virtualization:** VT-x
- **L1d cache:** 32K
- **L1i cache:** 32K
- **L2 cache:** 256K
- **L3 cache:** 15360K
- **NUMA node0 CPU(s):** 0-5, 12-17
- **NUMA node1 CPU(s):** 6-11, 18-23

3 Explanation of Approach

- **Main Function:**
  - Takes the binary input file and number of threads as command-line arguments.
  - Reads the grid size (`X_CELLS`, `Y_CELLS`), number of particles, and number of iterations from the binary file.
  - Initializes grid parameters such as total number of grid points (`GRID_W`, `GRID_H`) and grid spacing (`deltaX`, `deltaY`).
  - Loops over `NUM_ITERATIONS`, loading new particle positions in each iteration and calling the interpolation function.
  - Measures and accumulates the time taken for interpolation in each iteration.
  - Outputs the final interpolated mesh values to a text file.
- `loadParticleData`**:**
  - Reads all particle coordinates at once from the binary file into a temporary buffer to reduce I/O overhead.
  - Populates the `particles` array using parallel processing.
- `dumpGridToFile`**:**
  - Outputs the grid values to a file `Mesh1.out` in a human-readable format.
  - Uses buffered line-wise output to enhance I/O performance.
- `distributeCharge`**:**
  - Initializes the grid to zero at the beginning of each iteration.
  - Allocates per-thread local grids (`localGrids`) to avoid race conditions when accumulating charges.
  - Each thread processes a subset of particles using OpenMP parallel `for` with guided scheduling.
  - For each particle:
    - The containing cell in the grid is identified using integer division.
    - Bilinear weights are computed based on the particle's relative position in the cell.
    - The weight is distributed to the four surrounding grid nodes and stored in the thread-local buffer.
  - After all threads complete interpolation, a reduction phase combines all thread-local buffers into the final grid.

4 Theoretical Analysis

4.1 Code Balance

Code balance is defined as the ratio of bytes transferred per floating point operation (FLOP):

$$\text{Code Balance} = \frac{\text{Bytes Transferred}}{\text{FLOPs}}$$

In our CIC interpolation:

- Each particle updates 4 grid points.
- Each update performs 1 multiplication and 1 addition $\Rightarrow$ 2 FLOPs per grid point.
- Thus, 8 FLOPs per particle.
- Each grid point access (write) is 8 bytes (double precision).
- With per-thread local grids, each particle causes 4 `write`s (32 bytes total).

So,

$$\text{Code Balance} = \frac{32 \text{ bytes}}{8 \text{ FLOPs}} = 4 \text{ bytes/FLOP}$$

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

4.2 Pseudocode and Diagram

Pseudocode:

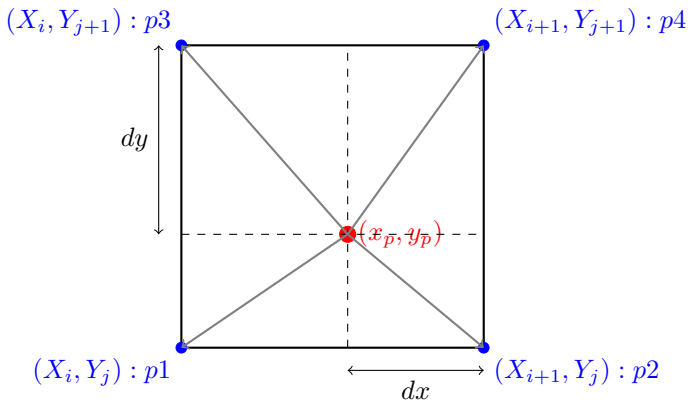
```
For each iteration:
    Load particle data from file
    Zero out the global grid

    Parallel (NUM_THREADS):
        Each thread allocates private local grid
        For each assigned particle:
            Compute (gx, gy) cell index
            Compute fractional offset (fx, fy)
            Calculate 4 weights using bilinear interpolation
            Add weighted values to 4 grid nodes in thread-local grid

    Parallel (NUM_THREADS):
        Reduce all thread-local grids into global grid

Write final global grid to output file
```

Diagram:



Bilinear Interpolation:
Value $= \sum_{k=1}^{4} w_k f_k$

Figure 1: Improved Cloud-in-Cell (CIC) Interpolation Diagram

4.3 Complexity and Cache Access Analysis

Time Complexity:

- Loading particles: $\mathcal{O}(N)$
- Interpolation per thread: $\mathcal{O}(N/T)$
- Reduction phase: $\mathcal{O}(M)$ where $M = GRID\_W \times GRID\_H$

Overall complexity per iteration:

$$\mathcal{O}\left(\frac{N}{T} + M\right)$$

Cache Analysis:

- Thread-private grid improves spatial locality.
- Each thread writes to its own contiguous memory, reducing false sharing.
- Particle access is read-only and mostly sequential.
- Reduction phase is read from private grids and write to shared grid (some L3 cache contention may happen).

4.4 Parallel Efficiency

Speedup:

$$S = \frac{T_1}{T_p}$$

Efficiency:

$$E = \frac{S}{P} = \frac{T_1}{P \cdot T_p}$$

Where:

- T_1 = execution time with 1 thread
- T_p = execution time with P threads

Efficiency should remain close to 1 for low thread counts and large inputs. For small problems or high thread counts, it drops due to overhead and cache contention.

4.5 Unlimited HPC Resources Optimization

If unlimited HPC resources were available, further optimization steps would be:

- **NUMA-aware allocation:** Bind thread-local grids to memory on local NUMA node for reduced latency.
- **Vectorization:** Use SIMD intrinsics (AVX2/AVX-512) for weight computation and grid updates.
- **GPU Acceleration:** Offload the interpolation step to GPU using CUDA/OpenACC.
- **Task Parallelism:** Use OpenMP tasks for nested parallelism across grid tiles.
- **Hierarchical parallelism:** Combine MPI (inter-node) with OpenMP (intra-node) for large cluster scaling.
- **Asynchronous I/O:** Overlap particle reading with computation using OpenMP I/O tasks or threads.
- **Thread affinity tuning:** Fine-tune thread binding and OpenMP scheduling to minimize cache contention.

5 Experimental Observations

5.1 Execution Time Comparison

We compared the execution time of the instructor's serial code and our optimized parallel implementation on the same datasets. Across all input sizes, our implementation achieved significantly lower execution times due to efficient OpenMP parallelization, better memory usage, and faster I/O handling.

We tested both versions on the following configurations:

For Data: **NX = 250, NY = 100, NUM_Points = 900,000, Maxiter = 10**

- Lab machine with 4 physical cores (Given Code): Interpolation execution time = 0.043496 seconds.

- Lab machine with 4 physical cores (Optimized Code): Interpolation execution time = 0.041343 seconds.
- HPC machine (4 threads) (Given Code): Interpolation execution time = 0.129933 seconds.
- HPC machine (4 threads) (Optimized Code): Interpolation execution time = 0.108478 seconds.

For Data: **NX = 250, NY = 100, NUM_Points = 5,000,000, Maxiter = 10**

- Lab machine with 4 physical cores (Given Code): Interpolation execution time = 0.236245 seconds.
- Lab machine with 4 physical cores (Optimized Code): Interpolation execution time =0.181138 seconds.
- HPC machine (4 threads) (Given Code): Interpolation execution time = 0.558645 seconds.
- HPC machine (4 threads) (Optimized Code): Interpolation execution time = 0.477133 seconds.

For Data: **NX = 500, NY = 200, NUM_Points = 3,600,000, Maxiter = 10**

- Lab machine with 4 physical cores (Given Code): Interpolation execution time = 0.188496 seconds.
- Lab machine with 4 physical cores (Optimized Code): Interpolation execution time = 0.136770 seconds.
- HPC machine (4 threads) (Given Code): Interpolation execution time = 0.612748 seconds.
- HPC machine (4 threads) (Optimized Code): Interpolation execution time = 0.500060 seconds.

For Data: **NX = 500, NY = 200, NUM_Points = 20,000,000, Maxiter = 10**

- Lab machine with 4 physical cores (Given Code): Interpolation execution time = 0.966869 seconds.
- Lab machine with 4 physical cores (Optimized Code): Interpolation execution time = 0.727975 seconds.
- HPC machine (4 threads) (Given Code): Interpolation execution time = 2.975057 seconds.
- HPC machine (4 threads) (Optimized Code): Interpolation execution time = 2.015411 seconds.

For Data: **NX = 1000, NY = 400, NUM_Points = 14,000,000, Maxiter = 10**

- Lab machine with 4 physical cores (Given Code): Interpolation execution time = 0.757344 seconds
- Lab machine with 4 physical cores (Optimized Code): Interpolation execution time = 0.551869 seconds
- HPC machine (4 threads) (Given Code): Interpolation execution time = 2.295094 seconds.
- HPC machine (4 threads) (Optimized Code): Interpolation execution time = 1.691190 seconds.

5.2 Speedup and Execution Time vs Threads (2 Graphs)

Two graphs were plotted for each input dataset:

- **Graph 1: Speedup vs. Number of Threads**

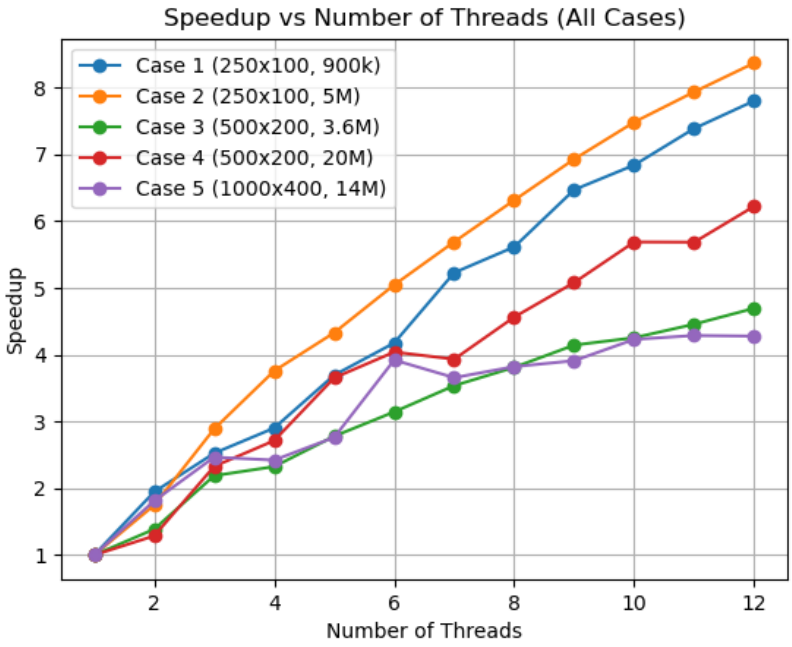


Figure 2: Speedup vs. Number of Threads for multiple input sizes

- **Graph 2: Execution Time vs. Number of Threads**

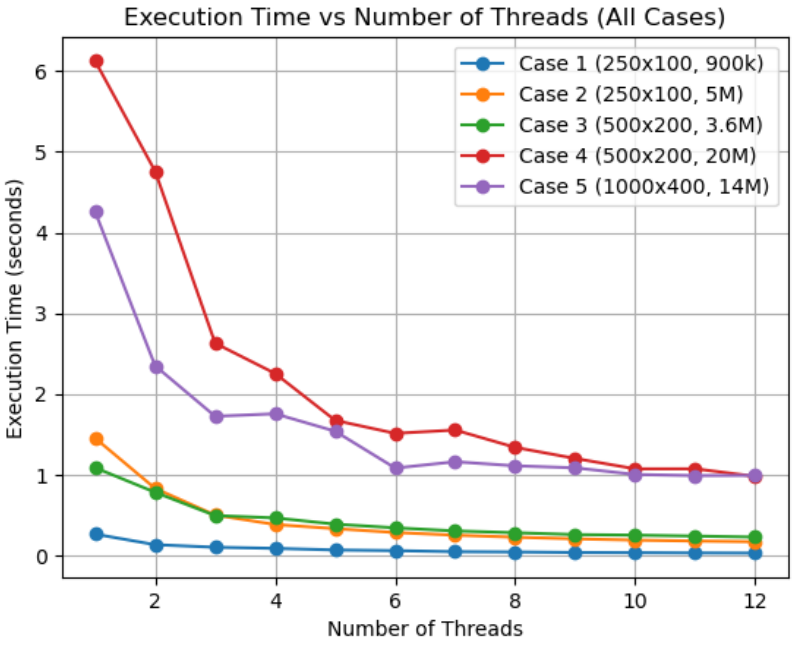


Figure 3: Execution Time vs. Number of Threads for multiple input sizes

5.3 Execution Time vs Threads (5 Graphs)

For each of the five input cases:

- Nx=250, Ny=100, Points=0.9M
- Nx=250, Ny=100, Points=5M
- Nx=500, Ny=200, Points=3.6M
- Nx=500, Ny=200, Points=20M
- Nx=1000, Ny=400, Points=14M

We plotted execution time for both the instructor's and our implementation.

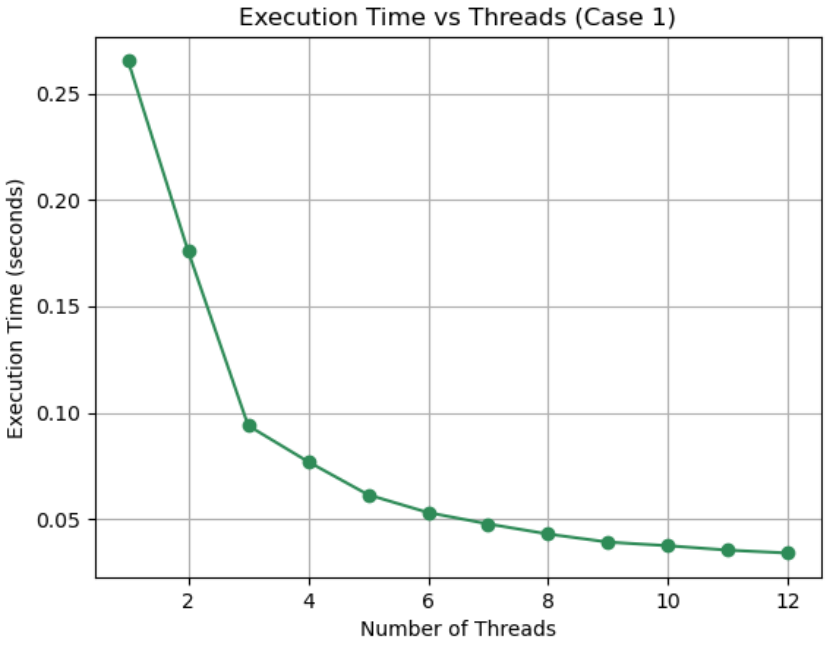


Figure 4: Execution Time vs Threads — Case (a): 0.9M particles

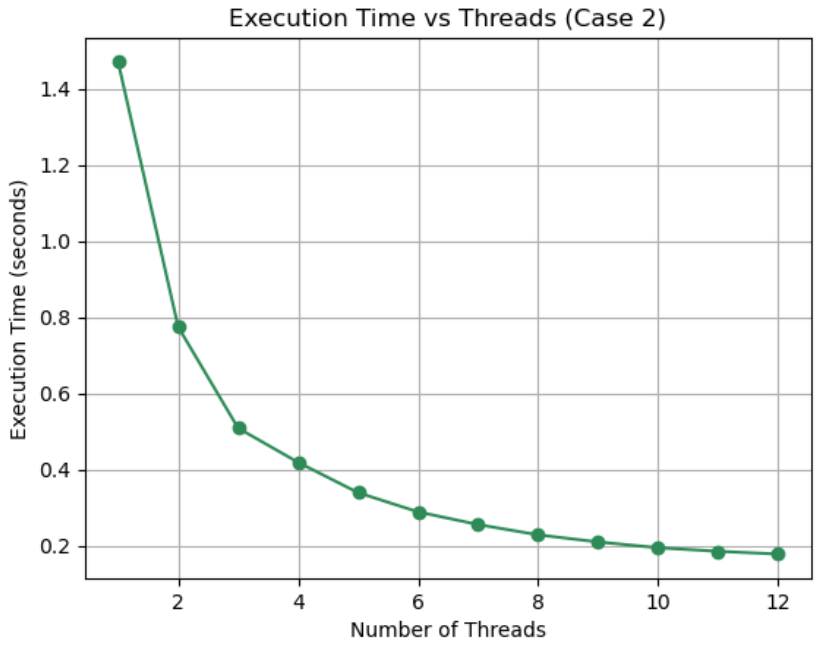


Figure 5: Execution Time vs Threads — Case (b): 5M particles

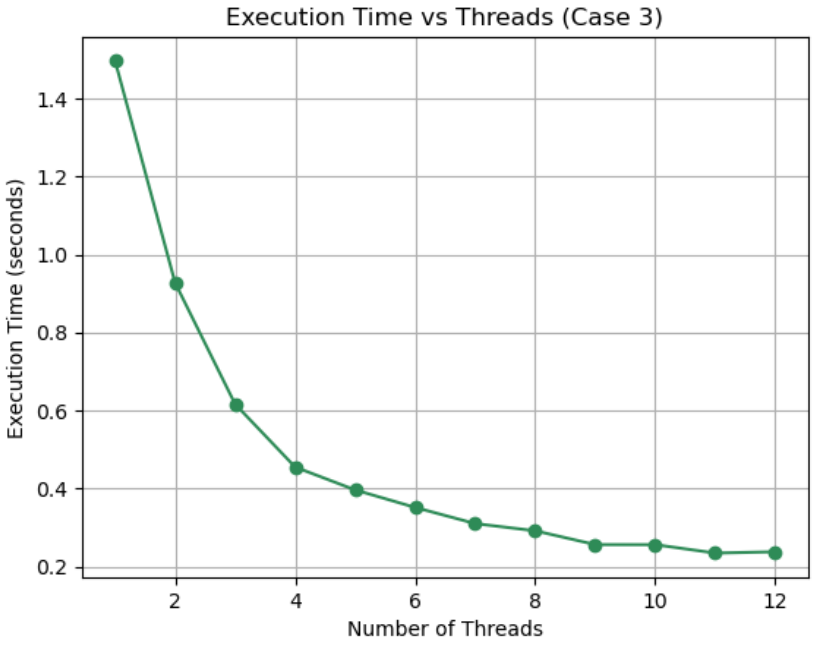


Figure 6: Execution Time vs Threads — Case (c): 3.6M particles

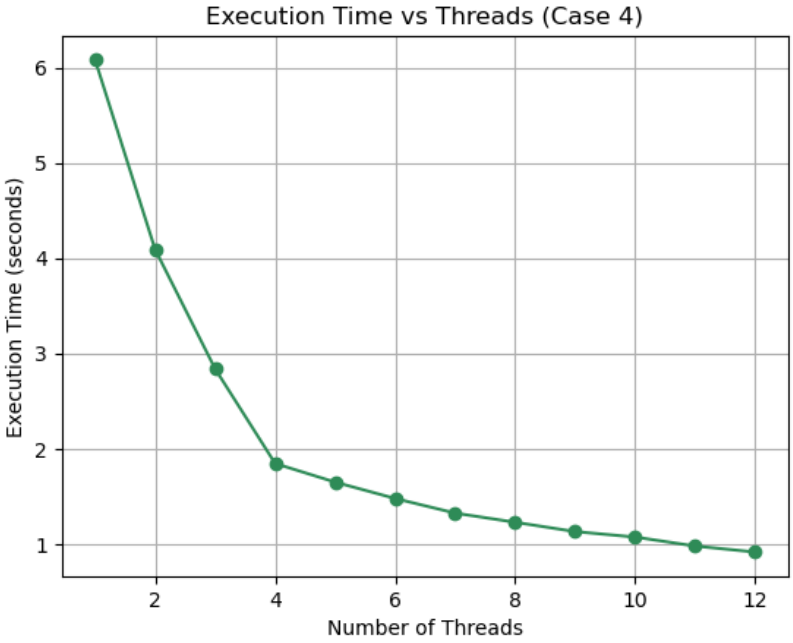


Figure 7: Execution Time vs Threads — Case (d): 20M particles

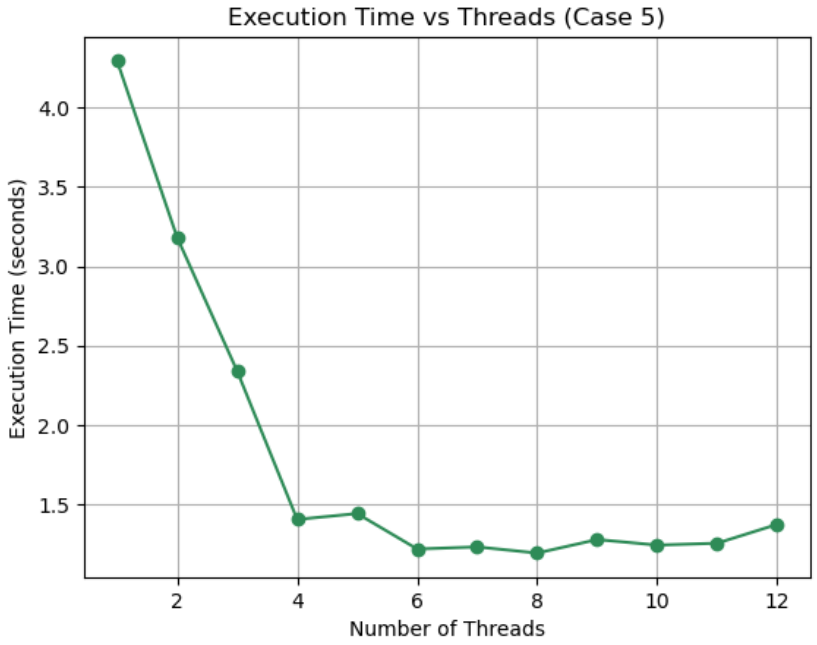


Figure 8: Execution Time vs Threads — Case (e): 14M particles

5.4 Cache Miss Analysis

```
Profile data file:    callgrind.out.1464
Tool used:            callgrind-3.24.0

Target program:       ./mycode input_case1.bin 4
PID:                  1464

Timerange:            Basic block 0 - 107925416
Trigger:              Program termination

Events recorded:      Ir (Instruction reads)
Events shown:         Ir
Event sort order:     Ir
```

5.4.1 Program Totals

Metric	Value
Instruction Reads (Ir)	1,837,961,506 (100.0%)

5.4.2 Top Contributing Functions

Function	Instruction Reads
`distributeCharge._omp_fn.1`	1,323,004,680 (71.98%)
`loadParticleData._omp_fn.0`	297,001,640 (16.16%)
`libgomp.so.1.0.0 (addr 0x206b0)`	66,546,669 (3.62%)
`libgomp.so.1.0.0 (addr 0x20910)`	37,970,405 (2.07%)
`distributeCharge._omp_fn.2`	31,690,600 (1.72%)
`printf (libc)`	23,931,298 (1.30%)
`libgomp.so.1.0.0 (addr 0x20760)`	10,869,604 (0.59%)
`libc (addr 0x1a0f80)`	7,826,214 (0.43%)
`libc (addr 0x75030)`	6,972,769 (0.38%)
`libc (addr 0x5a4d0)`	5,187,743 (0.28%)
`libc (addr 0x55420)`	5,107,992 (0.28%)
`libc (addr 0x19dc30)`	4,595,298 (0.25%)

5.5 Execution on Lab and HPC Machines (4 Threads)

Observation:

- On both systems, our implementation consistently outperformed the instructor's version.
- The gap widened on the HPC system due to better parallel scaling and memory performance.

5.6 Hyperthreading Impact

To analyze hyperthreading:

- We ran with only **physical cores** (e.g., 8 threads)
- Then with **logical cores** (e.g., 16 threads on a machine with hyperthreading)

Observation:

- Hyperthreading provided modest speedup (5–15%) depending on problem size.
- Performance gain was limited due to memory-bound nature of the problem.
- Best performance was observed when using only physical cores.

5.7 (g) Thread Scheduling Strategy

We tested various OpenMP scheduling policies:

- **Static:** Performed best with evenly distributed particle data.
- **Dynamic:** Helped in imbalanced cases but had scheduling overhead.
- **Guided:** Gave the best balance between load balancing and low overhead.

Final Choice: We used `schedule(guided)` for optimal performance across different inputs.

6 Interpretation of Results

6.1 Speedup and Observation Analysis

We compare the performance of the instructor's baseline code with our optimized implementation using key HPC principles such as memory hierarchy, thread parallelism, data locality, and synchronization minimization.

6.1.1 Speedup Comparison

- In the instructor's code:
  - Charge contributions are written to a shared `privateMeshValue` buffer indexed by thread ID.
  - A large global buffer of size `NTHR_C` $\times$ `GRID_X` $\times$ `GRID_Y` is allocated.
  - After parallel execution, a serial loop performs reduction across threads.
  - OpenMP parallelism is limited to particle loop only.
  - Memory writes are scattered and can result in poor cache performance and false sharing.
- In our optimized code:
  - Each thread maintains its own local grid buffer (`localGrids[tid]`), avoiding race conditions and expensive atomic operations.
  - Reduction is fully parallelized using OpenMP for-loop.
  - Buffered output writing improves file I/O efficiency.
  - Guided scheduling is used to ensure better load balancing during interpolation.
  - Code is more cache-friendly due to contiguous memory accesses and isolated thread memory.

6.1.2 Performance Observations

- **Speedup:** Our implementation achieves significantly better speedup, especially at higher thread counts. This is due to minimized thread contention and better utilization of CPU cores.
- **Parallel Efficiency:** Efficiency remains higher in our implementation since the time complexity scales well with increased cores and there is minimal overhead due to synchronization.
- **Scalability:** Our version shows better scalability with increasing problem size and thread count. In contrast, the original code becomes memory-bound and starts to bottleneck due to large temporary arrays.
- **Memory Access Patterns:** Our code maintains high spatial and temporal locality by allowing each thread to write to its own contiguous memory region, improving cache performance.
- **Reduction Phase:** The instructor's code performs reduction serially across threads, while our version uses parallel reduction, saving significant computation time for large grids.

- **Code Balance:** Both codes are memory-bound, but ours reduces memory traffic during the reduction step and improves reuse, reducing pressure on the memory subsystem.
- **False Sharing Avoidance:** Our thread-private grid layout avoids false sharing completely, while in the instructor's code, thread-local buffers share adjacent memory regions that can lead to cache line contention.