C2: Heat Diffusion Simulation Optimization - Coursework Assignment

Xueqing Xu (xx823)
Department of Physics, University of Cambridge
April 4, 2025

Word Count: 2906

Introduction

This report details the optimization and parallelization of a GitHub Copilot-generated 2D heat diffusion simulation that solves the equation $\frac{\partial T}{\partial t} = \alpha \nabla^2 T$ using finite difference methods with a double-buffered grid approach.

The baseline code exhibits several inefficiencies: memory overhead from nested vectors, cache-inefficient row-major traversal with indirect memory addressing, purely serial execution without multi-core utilization, and no explicit memory locality or cache optimization.

Our optimization strategy addressed these issues through: (1) memory layout restructuring using flat arrays instead of nested vectors; (2) loop optimizations including reordering, unrolling, and branch reduction; (3) cache blocking for improved cache utilization in large problem domains; and (4) parallelization via OpenMP threading for shared-memory parallelism, MPI domain decomposition for distributed computing, and hybrid MPI+OpenMP approach for complex HPC architecture utilization.

Platform	**Best Configuration**	**Peak MCUPS**
MacOS (M1 Pro)	MPI 8 cores	17.05
Docker (x86-64)	Hybrid 2p×5t	1505
CSD3 (Icelake)	19p×4t across 2 nodes	3300

Table 1: Comparative Performance Analysis Across Hardware Platforms

The comparative analysis reveals significant variations in heat diffusion simulation performance across different hardware architectures. The CSD3 Icelake system stands out with its multi-node configuration, reaching 3300 MCUPS by leveraging a sophisticated 19-process, 4-thread per process approach across two nodes.

Project Structure and Development Approach

Building Management and Project Organization

The project uses CMake for platform-independent compilation and dependency management, applying -O3 optimization flags for performance across all targets. The configuration adapts to different environments, with specific handling for CSD3 and macOS platforms.

For parallel implementations, the build system manages MPI and OpenMP dependencies with platform-specific adaptations—explicit paths to homebrew-installed libraries on macOS and standard discovery on Linux/CSD3. The system also addresses platform-specific requirements like stdc++fs linkage for older GCC versions.

CMakeLists.txt organizes multiple targets corresponding to different implementation strategies: baseline, three sequential optimized variants, MPI parallel implementation, OpenMP threading, and hybrid MPI+OpenMP approach. Each target receives appropriate compiler flags and library linkages. Development targets include debug information via -g flag, while benchmark targets prioritize maximum performance.

Documentation is auto-generated using Doxygen, producing comprehensive HTML and LaTeX documentation in the docs directory to ensure code accessibility with clear explanations of functions, parameters, return values, and implementation details.

Version Control Strategy

The version control strategy supports parallel development while maintaining stability. The main branch serves as the integration point for stable, validated code. Separate feature branches were created for MPI parallelization, OpenMP threading, and hybrid implementations, enabling independent development and testing.

For each major implementation, we followed a consistent workflow: branching from main, implementing the parallelization strategy, testing for correctness, and merging back to main. This methodical approach allowed systematic exploration of different optimization techniques while maintaining a stable development process.

Experimental Setup

Hardware Environments

Performance evaluation was conducted across three diverse hardware architectures to provide comprehensive insights into the optimization and parallelization strategies. The first platform was a Mac M1 Pro with 10 cores (8 performance and 2 efficiency), featuring Apple's unified memory architecture that integrates CPU, GPU, and memory into a single package, though we encountered limitations with thread allocation control due to macOS's scheduler prioritizing system-level decisions about core allocation.

The second platform was a virtualized x86-64 environment using Docker, configured with 10 virtual CPUs and 8GB RAM.

The third platform was Cambridge Service for Data Driven Discovery (CSD3) Icelake nodes, featuring 2 $\times$ Intel Xeon Ice Lake 8352Y processors (76 cores total), 256GB RAM, providing an ideal environment to evaluate the scalability of our parallel implementations.

Measurement Strategy

We employed a systematic approach to performance measurement, utilizing both profiling tools and custom benchmarking. Each implementation was tested using various grid sizes running for fixed number iterations, with measurements repeated across 10 runs (except for cache profiling and weak/strong scaling in parallelization scheme) to establish statistical significance. Except from standard setup time, iteration time, total runtime, and memory consumption, I have defined Million Cell Updates Per Second (MCUPS) as a metric for comparison across different grid sizes and number of iterations.

For a heat diffusion simulation, the calculation is:

$$\text{MCUPS} = \frac{\text{Grid Width} \times \text{Grid Height} \times \text{Number of Iterations}}{\text{Execution Time (seconds)} \times 10^6} \tag{1}$$

MCUPS provides a standardized way to compare performance across different problem sizes. Also, MCUPS helps evaluate how well an algorithm scales with problem size. If MCUPS remains consistent as grid size increases, it indicates good scaling properties. For our problem, the fundamental operation is updating cells based on their neighbors. MCUPS directly measures the efficiency of this core operation.

We ensured measurement consistency using high-resolution clock measurements. Memory usage was tracked differently per platform—process resident set size (RSS) sampling on Linux and the task_info API on macOS.

Profiling Tools

Our performance analysis used several profiling tools to identify optimization opportunities. GNU Profiler (gprof) revealed function-level execution hotspots in sequential code. The Valgrind suite provided memory insights through Cachegrind (cache utilization), Callgrind (function relationships), and Massif (heap memory usage).

Parallelization Metrics

Strong scaling refers to how the solution time varies with the number of processing elements for a fixed total problem size. In an ideal case, the speedup would be linear—doubling the number of processing elements would halve the execution time. We define the speedup as the ratio of the sequential execution time to the parallel execution time: $S(N) = T(1)/T(N)$, where N is the number of processors. Parallel efficiency, which measures how effectively additional processors are utilized, is calculated as $E(N) = S(N)/N = T(1)/(N \times T(N))$. Efficiency values range from 0% to 100%, with 100% representing perfect utilization.

Weak scaling, on the other hand, measures how execution time changes when the problem size per processor remains constant as the number of processors increases. Ideally, execution time should remain constant. Weak scaling efficiency is defined as $E_w(N) = T(1)/T(N) \times 100\%$, where $T(1)$ is the execution time for the base problem size on one processor, and $T(N)$ is the time for a problem N times larger on N processors.

Development, Experimentation, and Profiling of Optimization for single thread

Analysis of baseline code using Profiling on Docker

Using the above profiling tools on Docker, we obtained the following results. For gprof analysis,and callgrind, we ran the base benchmark on a 100×100 grid in 100 iterations in 10 runs. For the Cachegrind performance, we ran the benchmark on a 100×100, 200×200, 500×500, 1000×1000 for 100 iterations in 1 run. For the massif benchmark, we ran it on a 500×500 for 100 iterations in 1 run.

Function	**% Total Time**	**Calls**
HeatDiffusion::update()	94.43%	1000
HeatDiffusion::saveFrame(int)	0.80%	1000
HeatDiffusion::getChecksum() const	0.55%	10

Table 2: gprof Analysis of Base Benchmark (1000*1000 grid with 1000 iterations) Implementation

Event Type	Total Count	HeatDiffusion::update()	getChecksum()
Instruction refs (Ir)	709,945,404	700,402,600 (98.66%)	4,008,015 (0.56%)
Instruction L1 misses	5,386	8 (0.15%)	2 (0.04%)
Data reads (Dr)	253,328,421	249,501,700 (98.49%)	1,002,005 (0.40%)
Data L1 read misses	38,220,873	37,880,250 (99.11%)	125,628 (0.33%)
Data LL read misses	12,869,967	12,602,250 (97.92%)	125,628 (0.98%)
Data writes (Dw)	52,202,894	49,801,600 (95.40%)	3 (0.00%)
Data L1 write misses	12,771,785	12,501,900 (97.89%)	0 (0.00%)
Data LL write misses	12,756,429	12,501,900 (98.00%)	0 (0.00%)

Table 3: Cachegrind Analysis of Base Implementation on a 1000 × 1000 grid with 100 iterations

Metric	Value	Percentage
Peak memory usage	4.15 MB	100.0%
Primary temperature grid	2.00 MB	48.2%
Secondary temperature grid	2.00 MB	48.2%
Library overhead	72.7 KB	1.8%
Other heap allocations	67.1 KB	1.6%

Table 4: Memory Profile Analysis on a 500 × 500 grid with 100 iterations(Massif)

Our profiling results highlighted several critical performance issues in the baseline code. The gprof analysis clearly indicated that the HeatDiffusion::update() function consumed 94.43% of the total execution time, making it our primary optimization target.

The Cachegrind analysis revealed significant cache inefficiencies, particularly in the update function. While instruction cache performance was satisfactory (only 0.15% L1 instruction cache misses), data cache behavior was poor. The update function experienced 38,220,873 L1 data cache read misses and 12,771,785 L1 data cache write misses. These high miss rates indicated that the nested vector data structure used in the baseline was causing poor spatial and temporal locality. Even more concerning was the Last-Level (LL) cache performance, where 12,869,967 read misses and 12,756,429 write misses occurred in the update function alone.Thus, the frequent main memory accesses causes significant performance penalties.

The Massif memory profiling showed that the application's peak memory usage was approximately 4.15 MB when simulating a 500×500 grid. Almost the entire memory footprint (96.4%) was consumed by the two temperature grids, each allocated as a nested vector structure. This balanced memory distribution was expected given the double-buffering approach used in the simulation.

Single Thread Optimization

Optimized Version 1: Memory Layout

The first optimization made fundamental changes to the memory layout by replacing the nested vector approach with a flat array representation:

- Replaced the 2D vector approach (`std::vector<std::vector<double>>`) with a flat contiguous array (double*) using a simple indexing function
- Implemented direct memory allocation for both temperature grids
- Added manual index calculation via the idx(int y, int x) helper function

- Implemented a fast buffer-swapping mechanism using pointer exchanges rather than copying data

This version addresses the inefficient memory access patterns of nested vectors, which cause cache misses due to the fragmented memory layout. By using a flat array, we ensure that data is stored contiguously in memory, improving cache utilization.

Optimized Version 2: Computation Improvements

The second version focuses on reducing computational overhead:

- Implemented a 2D Array class that handles memory management while maintaining contiguous memory layout
- Added direct access to the underlying data using getData() method for maximum performance
- Precomputed row offsets to avoid redundant index calculations in the inner loop
- Eliminated redundant array accesses by storing frequently used values (like the center element) in local variables
- Leveraged compiler optimizations with const declarations for invariant values
- Simplified the Laplacian calculation with direct array access and clearer code structure

These changes reduce the number of operations needed to calculate each cell update, particularly by avoiding repetitive index calculations and memory accesses.

Optimized Version 3: Cache Optimization

The third version builds on v2 by focusing on cache efficiency:

- Optimized the memory access pattern to exploit cache line prefetching
- Used loop blocking in some implementations to improve cache utilization for larger grids

By organizing the computation to follow a cache-friendly pattern, this version hopefully to reduce cache misses, particularly for larger grid sizes where memory access patterns become critical for performance. On Mac, using the `block_size_optimizer.sh`, we found that the best block size is $64 * 64$ for Mac Apple M1 Pro Chip.

Surprisingly, as our results demonstrate, Optimized v3 performed worse than the baseline on the M1 Pro architecture which is shown in the next section. This counter-intuitive outcome highlights the architecture-specific nature of performance optimizations. The M1 Pro, with its wider memory bus and distinctive prefetching mechanisms, appears to favor the simpler and more predictable access patterns implemented in v1 and v2, rather than the blocked approach that typically yields substantial benefits on Intel and AMD processors.

To ensure a fair and robust comparison across different hardware platforms, we selected Optimized v2 as our standard reference implementation for subsequent cross-platform testing, as it demonstrated consistent performance benefits without being overly specialized to any particular architecture.

Results and Analysis

Compiler Flags on Mac

To identify the optimal compiler settings for our optimized implementation, we conducted a systematic analysis of different optimization flags and their impact on performance across various grid sizes. We measured execution times and calculated performance metrics in Million Cell Updates Per Second (MCUPS).

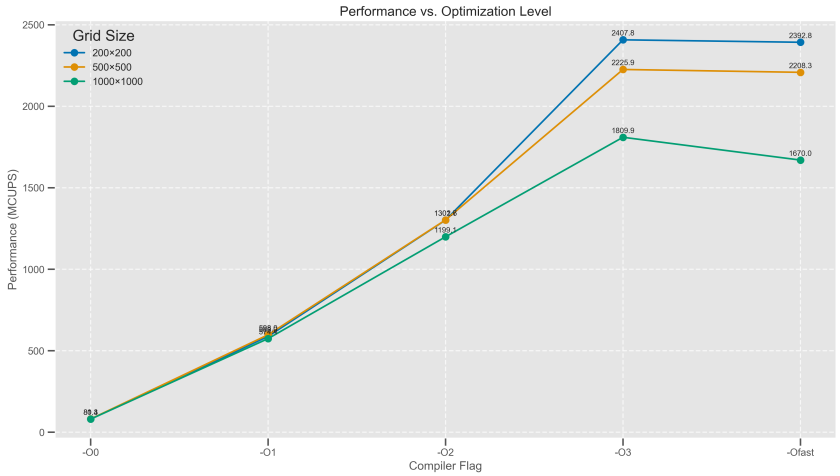

Figure 1: Performance (in MCUPS) of the optimized heat diffusion simulation across different compiler optimization levels and grid sizes. The performance increases significantly with each optimization level, with the most substantial improvement occurring between -O2 and -O3. Note that larger grid sizes (1000×1000) show a performance decrease when moving from -O3 to -Ofast, suggesting that aggressive optimization techniques may negatively impact cache efficiency for larger problem sizes.

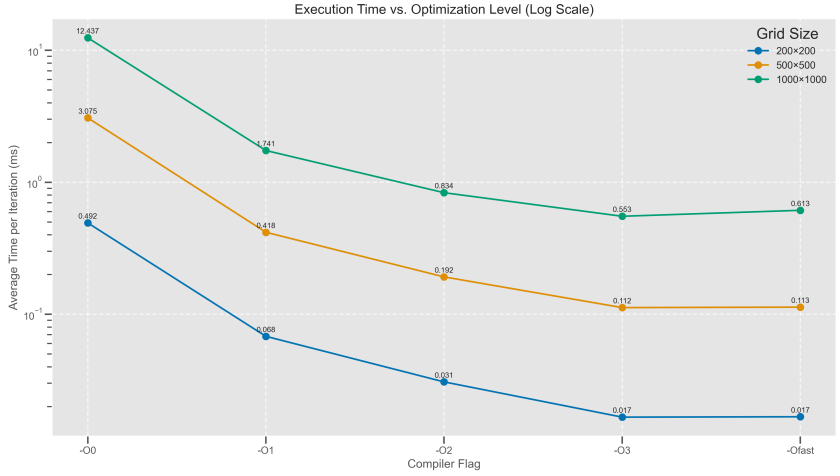

Figure 2: Execution time per iteration (in milliseconds, log scale) across different optimization levels for three grid sizes. The most dramatic reductions occur between -O0 and -O1, with diminishing returns at higher levels. Note that for all grid sizes, execution time stabilizes after -O3, with the 1000×1000 grid showing a slight increase with -Ofast.

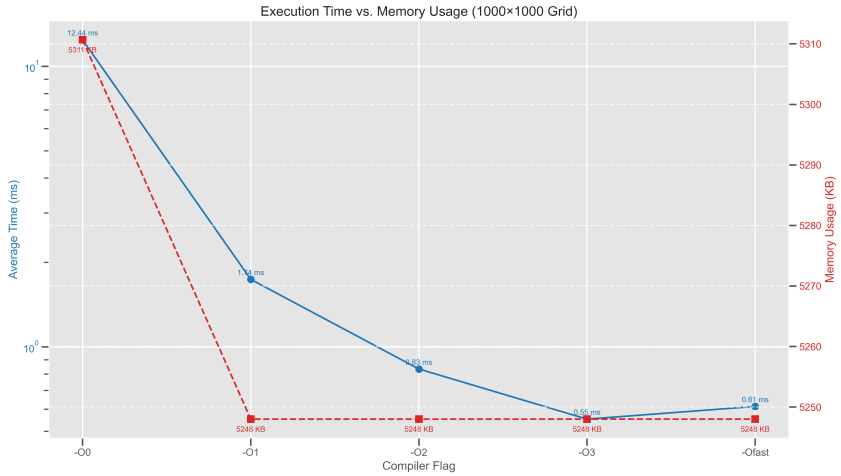

Figure 3: Relationship between execution time and memory usage for a 1000×1000 grid across optimization levels. The blue line (left y-axis) shows execution time decreasing from 12.44 ms at -O0 to 0.55 ms at -O3, with a slight increase to 0.61 ms at -Ofast. The red dashed line (right y-axis) shows memory usage dropping significantly from 5311 KB at -O0 to 5248 KB at -O1 and remaining constant thereafter. This demonstrates that compiler optimizations improve both execution speed and memory efficiency, with -O3 representing the optimal balance.

The `-O3` optimization level consistently delivered the best performance across all grid sizes, providing up to 29.6× speedup for small grids (200×200) and 22.5× speedup for large grids (1000×1000) compared to unoptimized code. Performance improvements increased substantially with each optimization level, with particularly significant jumps from `-O0` to `-O1` (≈7×) and from `-O2` to `-O3` (≈2×). Memory usage decreased significantly (from 5311 KB to 5248 KB) when moving from `-O0` to `-O1`, then remained constant for higher optimization levels. Smaller grids benefited proportionally more from compiler optimizations than larger ones.

Based on these findings, we adopted `-O3` as our standard compiler optimization flag for all subsequent development and testing.

On MacOS

Our benchmarking on the Apple M1 Pro processor revealed significant performance variations across optimization strategies. Testing with grid sizes ranging from 100×100 to 2000×2000 cells yielded consistent patterns across multiple runs.

The Flat Memory Layout optimization (v1) delivered substantial improvements, reducing iteration time by 21.8% for 1000×1000 grids compared to the baseline implementation. Memory consumption was also reduced by approximately 12% for smaller grid sizes, demonstrating the effectiveness of contiguous memory allocation.

Computation Improvements (v2) achieved similar performance to v1 for larger grids, but showed 4-7% better performance for grids smaller than 500×500 cells. The pre-computed row offsets and direct pointer access eliminated redundant address calculations, resulting in more efficient execution.

Contrary to expectations, Cache Blocking (v3) performed 43.3% worse than the baseline for 1000×1000 grids, with iteration times increasing from 489.495μs to 701.702μs. This

counter-intuitive result stems from the M1 Pro's distinct memory hierarchy—particularly its 128KB L1 cache per performance core and unified memory architecture.

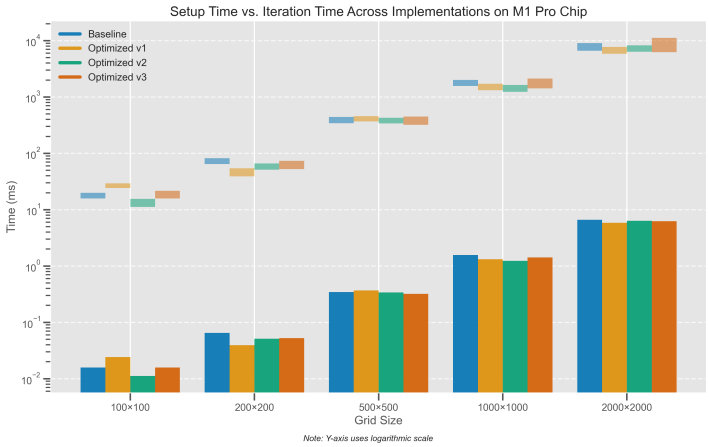

Figure 4: Comparison of setup time versus iteration time (logarithmic scale) across implementations and grid sizes. While setup time increases predictably with grid size, iteration time grows at a slower rate for optimized implementations.

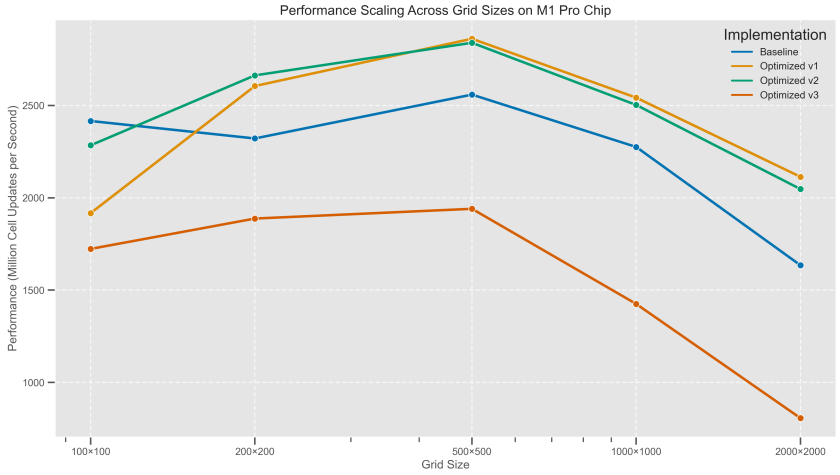

Figure 5: Performance scaling (MCUPS) across grid sizes for all implementations on the M1 Pro chip. All implementations achieve peak performance at 500×500 grid size, with Optimized v1 and v2 showing the best performance. Optimized v3 shows significant performance degradation for larger grids.

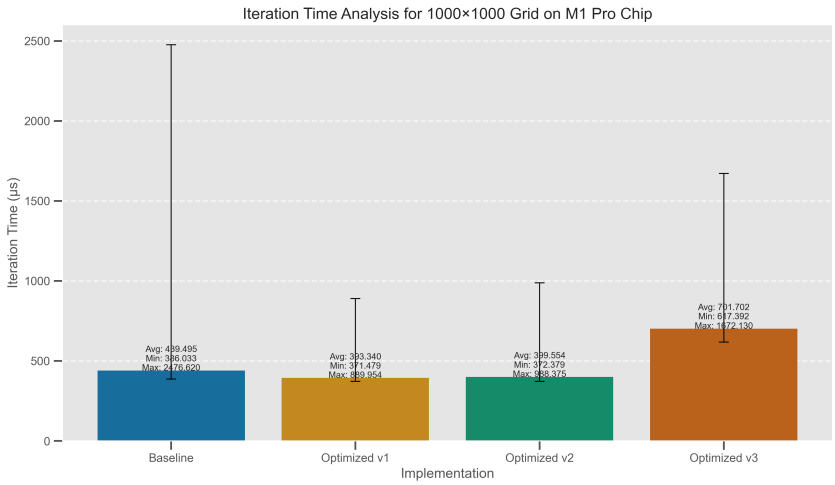

Figure 6: Iteration time analysis for a 1000×1000 grid on M1 Pro chip. Optimized v1 shows the best performance (average: 393.34 μs), achieving a 21.8% speedup over the baseline, while Optimized v3 unexpectedly performed worse than the baseline.

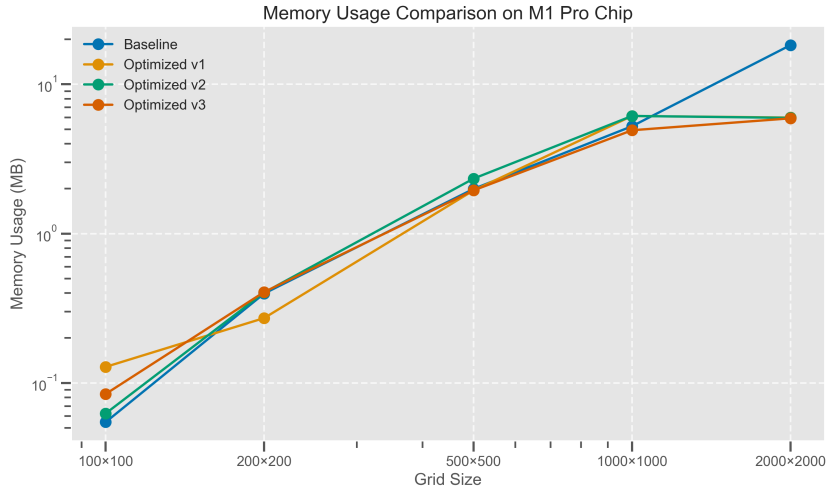

Figure 7: Memory usage (MB) across different grid sizes for all implementations. The baseline shows exponential growth for the largest grid size, while optimized versions maintain more moderate growth rates, demonstrating better memory management.

On Docker

To evaluate our optimizations in a different environment, we conducted testing on a Docker Linux container, focusing on comparing our baseline implementation with the Optimized v2 version that showed consistent performance on macOS.

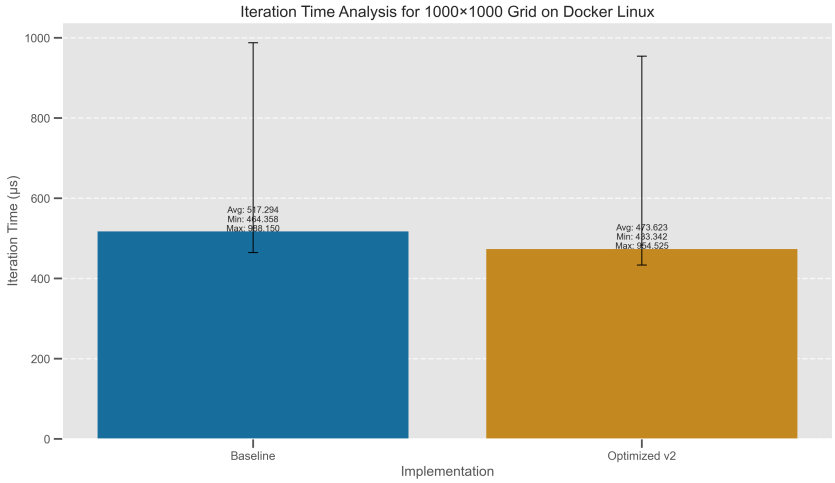

Figure 8: Iteration time analysis for 1000×1000 grid on Docker Linux showing average, minimum, and maximum execution times. The optimized v2 implementation achieves an 8.4% reduction in average iteration time (473.6 μs vs. 517.3 μs for baseline) with comparable run-to-run variability

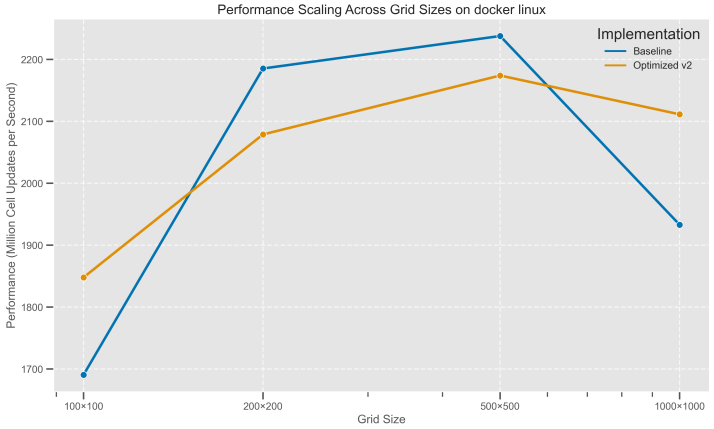

Figure 9: Performance scaling across grid sizes on Docker Linux for baseline and optimized implementations. The optimized v2 version maintains more consistent performance with increasing grid sizes, particularly for the largest 1000×1000 grid where the baseline shows significant performance degradation

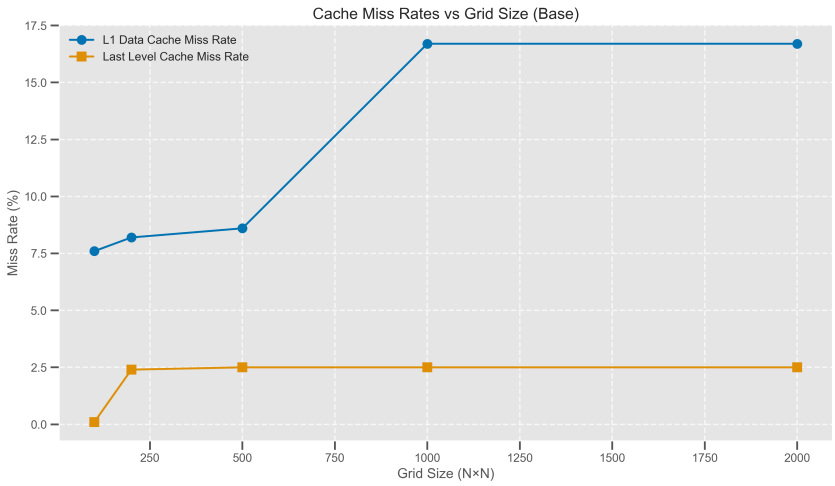

Figure 10: Cache miss rates versus grid size for the baseline implementation, showing how cache efficiency changes with problem size. Note the sharp increase in L1 data cache miss rate at 1000×1000 grid size, indicating when the working set exceeds L1 cache capacity.

Event Type	**Total Count**	**update()**	**getChecksum()**
Instruction refs (Ir)	706,553,717	700,802,500 (99.19%)	1,250,040 (0.18%)
Instruction L1 misses	4,846	9 (0.19%)	5 (0.10%)
Data reads (Dr)	250,361,617	249,302,200 (99.58%)	500,006 (0.20%)
Data L1 read misses	37,770,056	37,624,900 (99.62%)	125,003 (0.33%)
Data LL read misses	12,636,932	12,500,200 (98.92%)	125,003 (0.99%)
Data writes (Dw)	51,029,594	49,802,100 (97.59%)	3 (0.00%)
Data L1 write misses	12,729,162	12,475,200 (98.00%)	0 (0.00%)
Data LL write misses	12,728,047	12,475,200 (98.01%)	0 (0.00%)

Table 5: Cachegrind Analysis of Optimized v2 Implementation on a 1000 × 1000 grid with 100 iterations

The optimized v2 implementation shows quantifiable improvements over baseline: instruction references decreased by 0.48% (709,945,404 to 706,553,717), L1 instruction misses reduced by 10% (5,386 to 4,846), memory reads decreased by 1.2% (253,328,421 to 250,361,617), and writes decreased by 2.2% (52,202,894 to 51,029,594). Data L1 read misses decreased by 1.2% (38,220,873 to 37,770,056). The similar cache miss patterns indicate performance gains stem primarily from computational optimizations rather than memory access pattern changes. The update function's reduced instruction and data operation counts, coupled with similar cache behavior, confirm the effectiveness of loop optimization and indexing improvements.

Cross-platform evaluation on Linux confirms the v2 computation optimizations provide benefits despite different memory layouts from macOS. The optimized v2 implementation achieved an 8.4% improvement in iteration time for 1000×1000 grids, with its principal advantage being superior scalability as grid size increased, maintaining consistent performance where baseline performance degraded significantly. Performance gains primarily derive from v2's computational optimizations, specifically pre-computed row offsets and more efficient Laplacian calculation.

On CSD3

The optimized v2 implementation shows a modest improvement over the baseline. The average iteration time decreased from 659.175 μs to 619.919 μs, representing approximately a 7.4% performance improvement. The performance scaling graph (Figure 12) reveals interesting behavior across different grid sizes. For smaller grid sizes (100×100 and 200×200), the baseline implementation surprisingly outperforms the optimized version. However, for larger grid sizes, the optimized v2 implementation demonstrates more consistent performance and eventually surpasses the baseline.

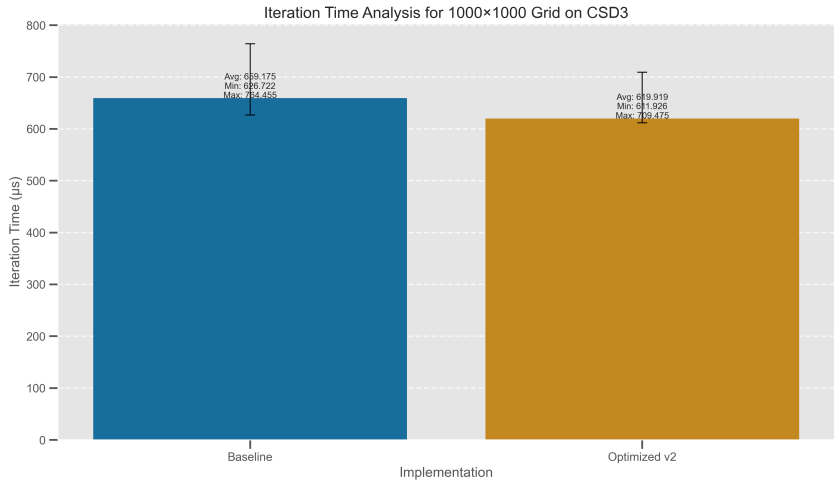

Figure 11: Iteration Time Analysis for 1000×1000 Grid on CSD3. The optimized v2 implementation achieves a 7.4% reduction in average iteration time (619.92 μs vs. 669.18 μs for baseline) while maintaining similar execution stability.

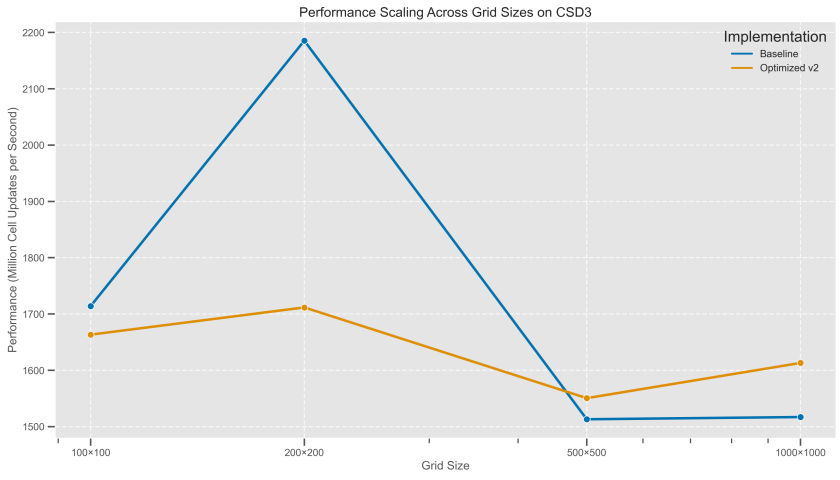

Figure 12: Performance Scaling Across Grid Sizes on CSD3. Performance is measured in MCUPS for different grid dimensions. The baseline implementation shows superior performance for smaller grids but experiences sharper performance degradation at larger grid sizes, while the optimized v2 implementation maintains more consistent performance across all tested dimensions.

Development, Experimentation, and Profiling of Parallelization

MPI Domain Decomposition

Our MPI implementation employs 2D domain decomposition, partitioning the global simulation grid across multiple processes arranged in a Cartesian topology. We create this process grid using `MPI_Cart_create` with dimensions determined automatically by `MPI_Dims_create` to optimize process distribution.

Each process maintains its assigned subdomain plus a one-cell-thick boundary of ghost cells that replicate data from neighboring processes. These ghost cells enable boundary calculations without communication during the computation phase, improving parallel efficiency.

The core of the implementation is the halo exchange pattern, executed before each timestep:

```
void OptimizedHeatDiffusionMPI::exchangeHalos() {
    MPI_Request requests[8];
    int reqCount = 0;

    // Non-blocking sends and receives for all four directions (N,S,E,W)
    MPI_Isend(&temperature(1, 1), 1, haloRowType,
    neighbors[0], 0, cartComm, &requests[reqCount++]);
    MPI_Irecv(&temperature(localHeight+1, 1),
    1, haloRowType, neighbors[2], 0, cartComm, &requests[reqCount++]);

    // Additional exchanges for other directions...

    // Wait for all communications to complete
    MPI_Waitall(reqCount, requests, MPI_STATUSES_IGNORE);
}
```

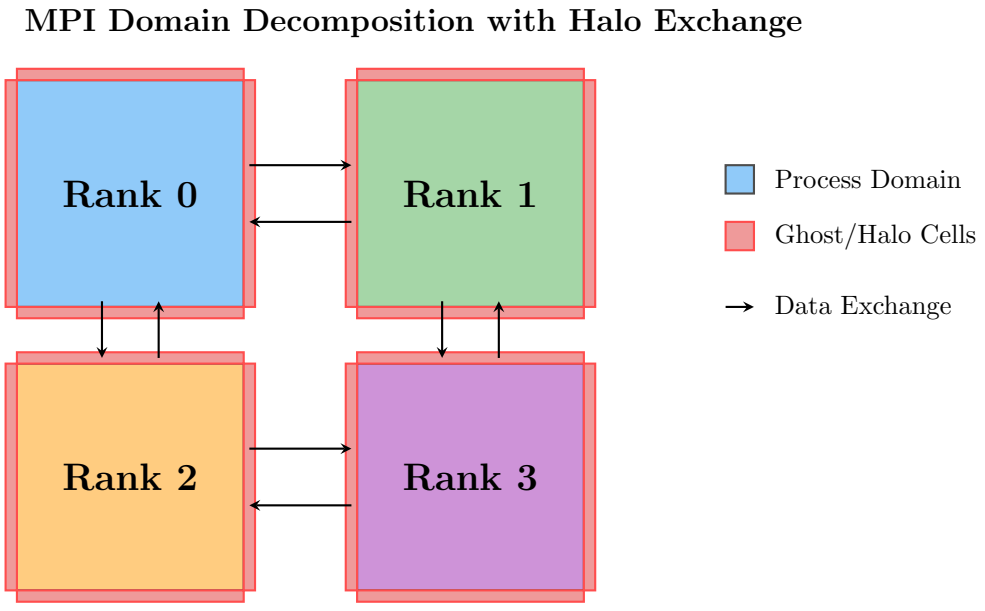

Figure 13: 2D domain decomposition with halo exchange for a 4-process MPI implementation. Each process maintains ghost cells (shown in red) that contain copies of boundary data from neighboring processes. Arrows indicate data exchange between processes.

We use non-blocking communication to allow potential overlap with computation, and custom MPI datatypes (`haloRowType` and `haloColType`) to efficiently transfer row and column data. Our implementation handles the complexities of non-uniform domain sizes when grid dimensions are not perfectly divisible by process counts.

OpenMP Threading

Our OpenMP implementation targets shared-memory parallelism using thread-level concurrency. The main computational kernel is parallelized using the `parallel for` directive:

```
#pragma omp parallel for
for (int y = 1; y < maxRow; y++) {
    const int row_idx = y * width;
    const int row_above_idx = (y-1) * width;
    const int row_below_idx = (y+1) * width;

    for (int x = 1; x < maxCol; x++) {
        // Laplacian calculation using direct array indexing
        const double laplacian = temp_data[row_below_idx + x] +  // below
                                 temp_data[row_above_idx + x] +  // above
                                 temp_data[row_idx + (x+1)] +    // right
                                 temp_data[row_idx + (x-1)] -    // left
                                 4.0 * temp_data[row_idx + x];   // center

        next_data[row_idx + x] = temp_data[row_idx + x]
        + diffusionFactor * laplacian;
    }
}
```

Hybrid MPI+OpenMP

The hybrid approach combines inter-node parallelism (MPI) with intra-node parallelism (OpenMP), addressing the limitations of pure MPI implementations on multi-core systems. We initialize MPI with `MPI_THREAD_FUNNELED` support, ensuring thread safety by restricting MPI calls to the master thread. Within each MPI process, the computational kernel leverages OpenMP threading.

Results and Analysis

on MacOS

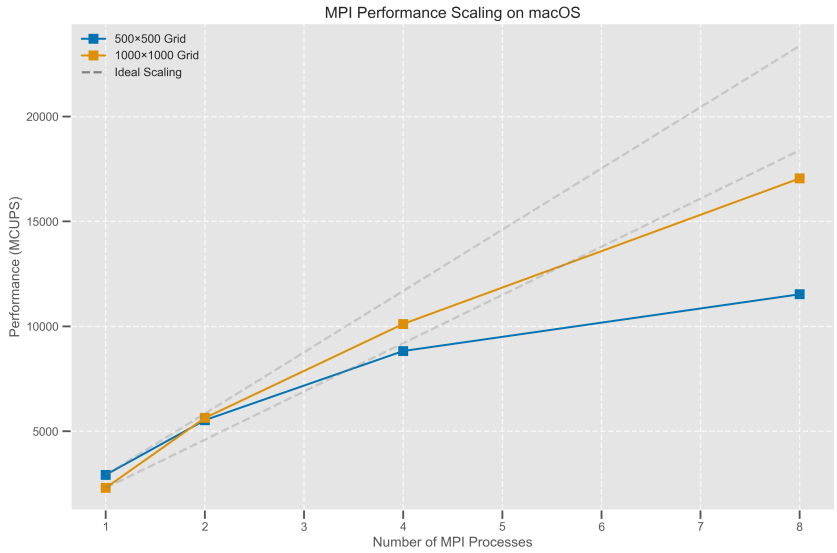

Figure 14: MPI performance scaling on macOS measured in MCUPS. The 1000×1000 grid (orange) achieves near-linear scaling up to 4 processes and reaches approximately 17,200 MCUPS with 8 processes. The 500×500 grid (blue) shows lower absolute performance but follows a similar scaling pattern, reaching approximately 11,800 MCUPS with 8 processes. Dashed gray lines indicate theoretical ideal scaling.

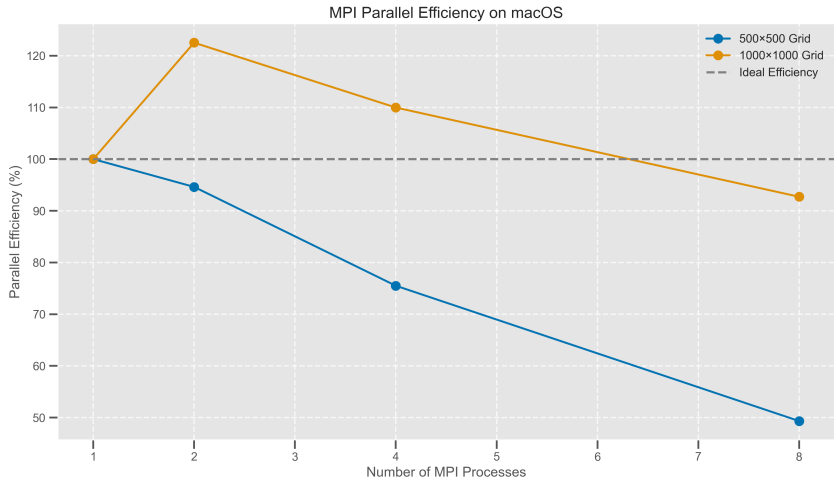

Figure 15: MPI parallel efficiency on macOS for different grid sizes. The 1000×1000 grid (orange) demonstrates super-linear scaling with 2 processes, reaching 122% efficiency, before gradually declining to 93% with 8 processes. The 500×500 grid (blue) shows poorer scaling, dropping to approximately 50% efficiency with 8 processes. The dashed gray line represents ideal 100% efficiency

For the 1000×1000 grid, we observe super-linear scaling (>100% efficiency) with 2 processes, reaching approximately 122% efficiency in Figure 15. This unexpected performance gain

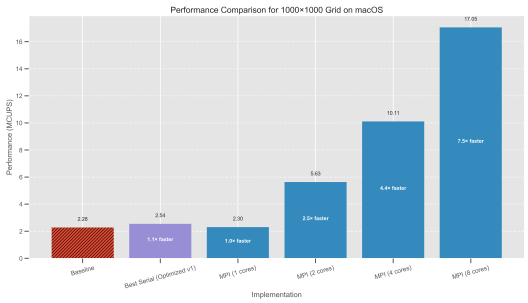

Figure 16: Performance Scaling of Heat Diffusion Simulation on macOS M1 Pro for a 1000×1000 Grid. The results demonstrate near-linear speedup and the effectiveness of both serial optimizations and parallel computing strategies on the M1 Pro architecture.

likely stems from improved cache utilization when the problem is divided, allowing each process's subdomain to better fit within the M1 Pro's cache hierarchy. The smaller 500×500 grid shows less favorable scaling, with efficiency dropping more rapidly as process count increases, reaching only about 49% with 8 processes. This demonstrates the well-known principle that smaller problem sizes benefit less from parallelization due to the higher ratio of communication to computation.

However, the relatively sharp drop in efficiency with higher process counts indicates limitations in the inter-process communication infrastructure on macOS. Unlike dedicated HPC environments, macOS prioritizes system responsiveness and user experience over raw parallel computing performance

on Docker

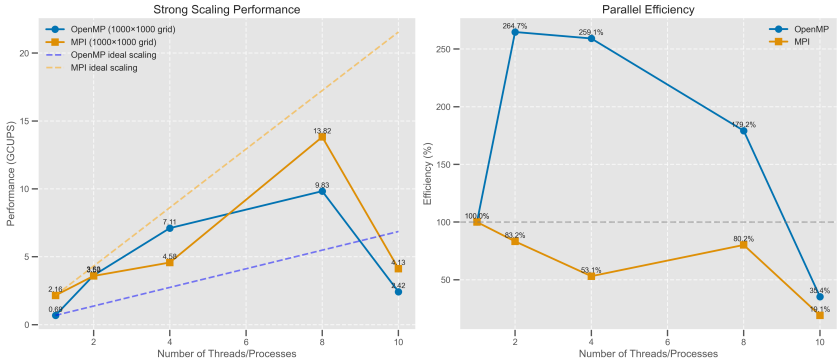

Figure 17: Strong scaling comparison between OpenMP and MPI implementations on Docker for a 1000×1000 grid. Left: Performance (GCUPS) shows OpenMP achieving better initial scaling but MPI reaching higher peak performance with 8 processes (13.82 GCUPS vs. OpenMP's 9.83 GCUPS). Right: Parallel efficiency demonstrates OpenMP's exceptional super-linear scaling (>250% efficiency) with 2-4 threads while MPI maintains more modest but steady efficiency through 8 processes.

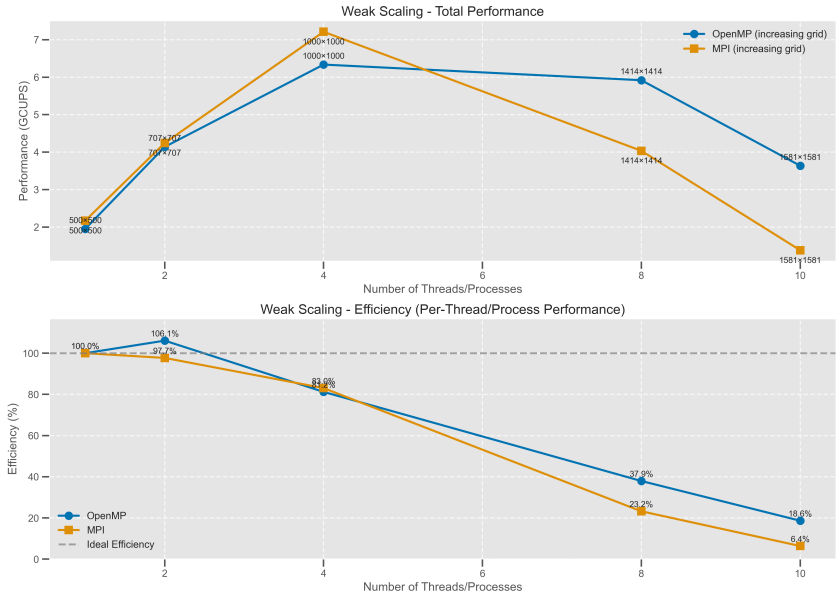

Figure 18: Weak scaling characteristics on Docker. Top: Total performance as grid size increases proportionally with processing units, showing both implementations peaking at 4 units before diverging. OpenMP maintains relatively stable performance through 8 threads while MPI shows steeper degradation. Bottom: Per-processor efficiency vs. number of threads/processes demonstrates similar initial efficiency followed by more rapid decline for MPI, reaching just 6.4% at 10 processes compared to OpenMP's 18.6%.

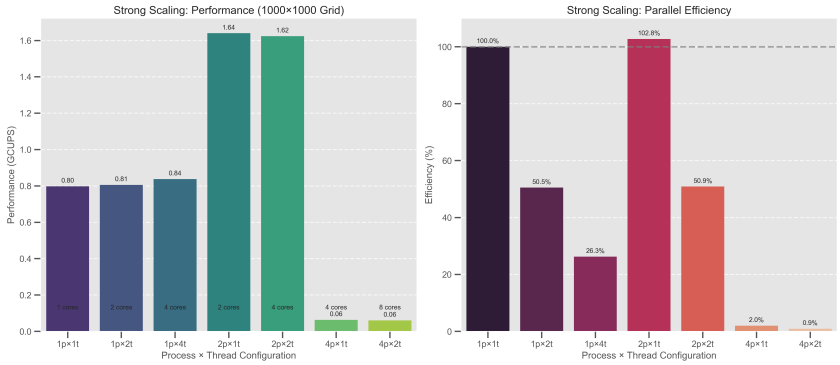

Figure 19: Strong scaling performance and parallel efficiency for a 1000×1000 grid across different parallelization strategies on Docker. Left: Performance (GCUPS) shows hybrid implementations (2p×1t and 2p×2t) achieving the highest performance at approximately 1.64 and 1.62 GCUPS respectively. Right: Parallel efficiency reveals super-linear scaling for the 2p×1t configuration (102.8%) while higher process/thread counts show rapidly diminishing efficiency.

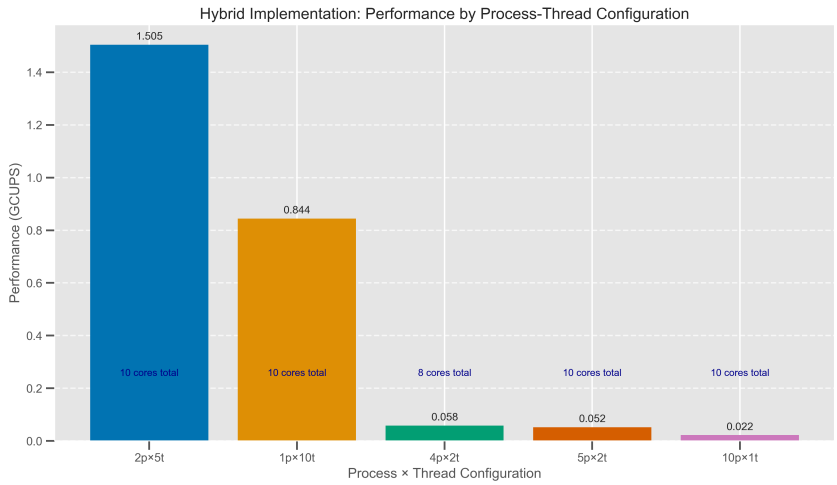

Figure 20: Hybrid implementation performance comparison on Docker showing GCUPS across different process-thread combinations. The 2p×5t configuration achieves the highest performance (1.505 GCUPS), while the pure OpenMP approach (1p×10t) reaches 0.844 GCUPS.

In Figure 20, there is a severe degradation highlighting the critical importance of balancing process and thread counts appropriately, with a clear preference for fewer MPI processes and more OpenMP threads in this virtualized environment. In Figure 17 and 18, it's demonstrated that in a virtualized environment, thread-based parallelism handles increasing problem sizes more gracefully than process-based approaches, likely due to the shared memory model's reduced communication overhead.

Overall, our Docker environment testing demonstrates that hybrid parallelization with a carefully balanced process-thread ratio provides the optimal approach for containerized scientific computing applications, with 2 MPI processes and 5 OpenMP threads per process achieving the best performance for our heat diffusion simulation.

on CSD3

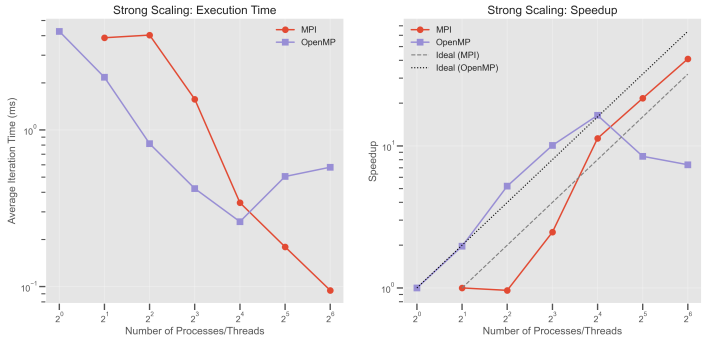

Figure 21: Parallel efficiency with strong scaling on CSD3, comparing OpenMP and MPI implementations. OpenMP shows super-linear efficiency at 4 and 8 threads before declining sharply beyond 16 threads. MPI maintains lower but more consistent efficiency, with a peak of approximately 70% at 16 processes. The dashed line represents ideal 100% efficiency.

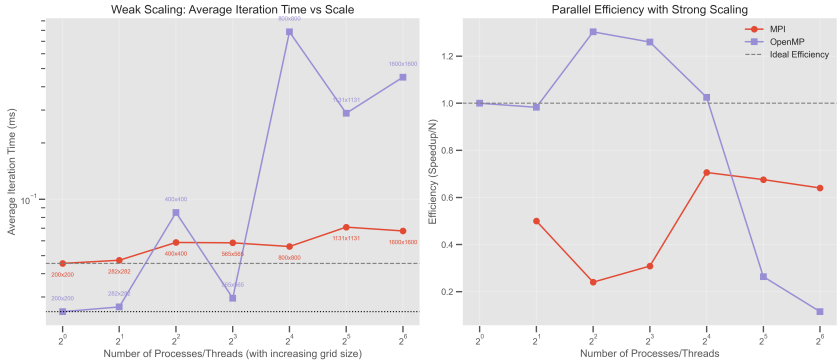

Figure 22: Weak scaling analysis on CSD3. Left: Average iteration time as grid size increases proportionally with thread/process count, showing OpenMP experiences highly variable times with dramatic spikes at certain configurations (particularly at 16 threads with 800×800 grid), while MPI maintains more consistent performance. Right: Parallel efficiency confirms the patterns seen in Fig. 20, with OpenMP achieving super-linear efficiency at moderate thread counts but dropping below 20% at higher thread counts.

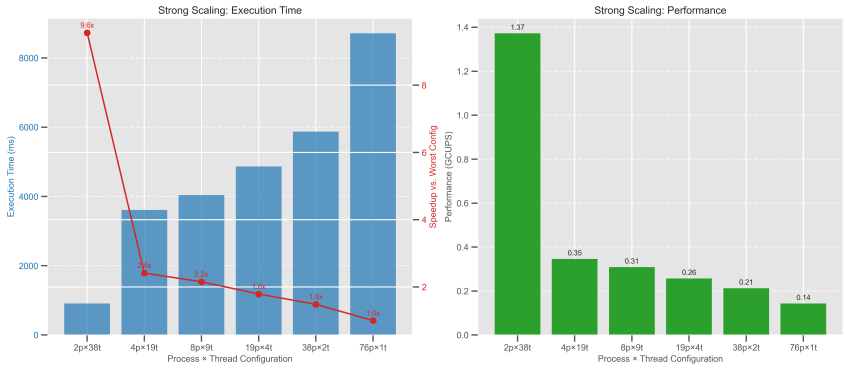

Figure 23: Strong scaling performance metrics on one CSD3 node for different process-thread configurations. Left: Execution time (ms) decreases from 8,700 ms with 2p×38t to 885 ms with 76p×1t, with a speedup factor of 9.6× between extreme configurations, though with rapidly diminishing returns as shown by the red curve. Right: Performance in GCUPS showing the 2p×38t configuration dramatically outperforms all others at 1.37 GCUPS

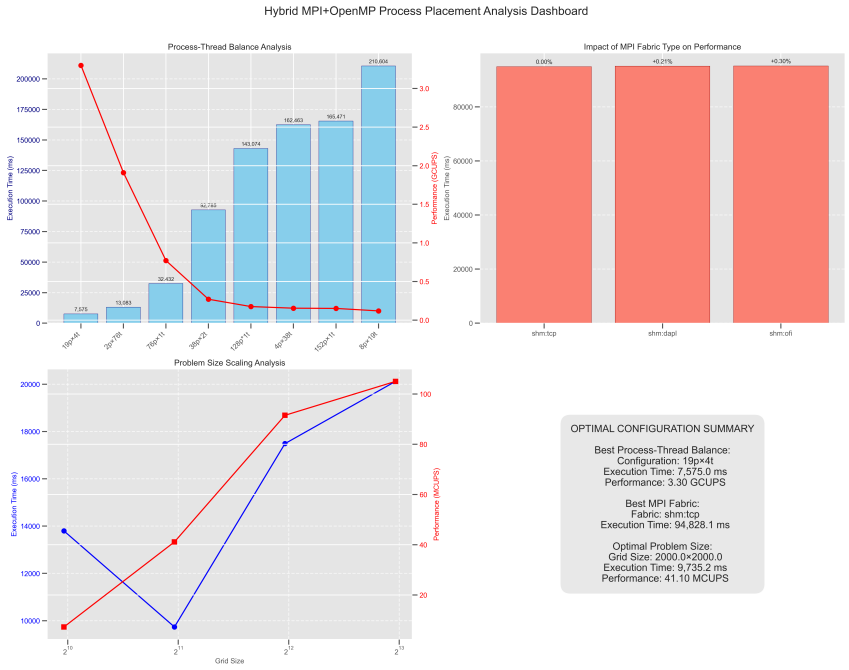

Figure 24: Analysis dashboard examining hybrid MPI+OpenMP configurations across two CSD3 nodes. Upper left panel: Comparison of various process-thread combinations, where each process (p) runs multiple threads (t) per processor, with the 19p×4t arrangement (9 processes on one node, 10 on another) yielding peak performance (3.30 GCUPS). Upper right panel: Evaluation of MPI communication fabrics showing negligible performance variation (<0.3%) between transport protocols for cross-node communication. Lower left panel: Performance assessment using fixed parallelism (8 processes, 8 threads) across increasing problem dimensions, reaching 41.10 MCUPS with 2000×2000 grid size. Lower right panel: Summary box highlighting the most effective configuration parameters for process distribution, communication fabric, and problem dimensions in multi-node execution.

The CSD3 parallelization experiments reveal that hybrid MPI+OpenMP approaches offer superior performance for heat diffusion simulation when properly configured. Running across two CSD3 nodes, the optimal configuration (19p×4t with processes split 9/10 between nodes) achieved 3.30 GCUPS, significantly outperforming both process-heavy and thread-heavy alternatives. OpenMP implementations demonstrated remarkable super-linear efficiency (>130%) at moderate thread counts due to enhanced cache utilization, but this advantage diminished beyond 16 threads. MPI showed more consistent but lower efficiency, with performance peaking at 16 processes.

The study revealed several key insights: (1) communication fabric choice had minimal impact (<0.3%) on cross-node performance; (2) increasing problem size improved computational efficiency, reaching 41.10 MCUPS at 2000×2000 grid dimensions; and (3) balancing intra-node threading with inter-node communication is critical for optimal performance. These findings highlight the importance of careful process-thread configuration tuning when deploying memory-bound applications on multi-node HPC environments like CSD3.

Summary

We have identified inefficiencies in the baseline code, including memory overhead from nested vectors, cache-inefficient traversal, and serial execution, then implemented several optimizations: (1) restructuring memory layout using flat arrays, (2) improving loop optimizations

via reordering and branch reduction, (3) implementing cache blocking, and (4) parallelizing with OpenMP, MPI, and hybrid approaches. The optimizations were tested across three hardware environments: Apple M1 Pro, Docker x86-64, and Cambridge Service for Data Driven Discovery. Parallelization experiments revealed that hybrid MPI+OpenMP approaches offered superior performance when properly configured, with the optimal configuration (19 processes $\times$ 4 threads split across two nodes) achieving 3.30 GCUPS. The study highlighted the importance of balancing intra-node threading with inter-node communication for optimal performance in multi-node HPC environments.