C2: Heat Diffusion Simulation Optimization - Coursework Assignment

Xueqing Xu (xx823)
Department of Physics, University of Cambridge
April 7, 2025

Word Count: 2990

AI Declaration

GitHub Copilot was used during development to assist with code completion, function documentation, and debugging. All AI-generated code underwent thorough review to ensure correctness, adherence to project requirements, and proper error handling. The core algorithms and methodological decisions remain my intellectual contribution, with Copilot serving only as a productivity tool for implementing standard techniques and reducing time spent on repetitive coding tasks.

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 GCUPS**
MacOS (M1 Pro)	MPI 8 cores	17.05
Docker (ARM64)	Hybrid 2p×5t	13.24
CSD3 (Icelake)	19p×4t across 2 nodes	9.58

Table 1: Comparative Performance Analysis Across Hardware Platforms

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.

Result Validation

To ensure numerical correctness across all implementations, we developed a comprehensive validation system that verifies simulation results remain consistent despite code optimizations.

The primary validation method performs a direct cell-by-cell comparison between corresponding frames from different implementations. Each value must match within a specified tolerance to account for acceptable floating-point variations. For large simulations, the validator supports an efficient checksum-only mode that computes and compares aggregate checksums for each frame rather than individual cell values.

The validation system provides several configuration options:

- Tolerance threshold: Adjustable precision for floating-point comparisons (default: $1e-10$)
- Selective validation: Ability to compare specific frames
- Maximum frame limit: Option to restrict the number of frames compared

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 using above validation tool, 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

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

Platform	Processor	Cores	Memory
macOS (M1 Pro)	Apple M1 Pro	10	16GB
Docker (ARM64)	Virtualized ARM64	10 vCPUs	8GB
CSD3 (Icelake)	2× Intel Xeon 8352Y	76 (total)	256GB

Table 2: Hardware Platforms Used for Performance Evaluation

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.

Performance Metrics

We employed two complementary approaches to evaluate the parallel performance of our heat diffusion implementations:

Our primary performance metric is billion cell updates per second (GCUPS), which measures the raw computational throughput of the simulation regardless of implementation details:

$$\text{GCUPS} = \frac{\text{Grid Width} \times \text{Grid Height} \times \text{Iterations}}{\text{Execution Time (seconds)} \times 10^9} \tag{1}$$

This metric allows direct comparison across different hardware configurations and parallelization strategies.

We ensured time 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).

Scaling Efficiency

We analyze both strong and weak scaling properties:

- **Strong Scaling**: Measures how solution time varies with the number of processing elements for a fixed total problem size (5000×5000 grid). 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:

$$S(N) = \frac{T(1)}{T(N)} \tag{2}$$

  where N is the number of processors. Parallel efficiency, which measures how effectively additional processors are utilized, is calculated as:

$$E(N) = \frac{S(N)}{N} = \frac{T(1)}{N \times T(N)} \tag{3}$$

  Efficiency values range from 0% to 100%, with 100% representing perfect utilization.

- **Weak Scaling**: 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:

$$E_w(N) = \frac{T(1)}{T(N)} \times 100\% \tag{4}$$

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

We employed various profiling tools on Docker to evaluate performance. For the gprof and callgrind analyses, we executed the base benchmark using a 100×100 grid with 100 iterations across 10 separate runs. When measuring Cachegrind performance, we tested the benchmark at multiple grid sizes (100×100, 200×200, 500×500, and 1000×1000), each running for 100 iterations in a single execution. The massif benchmark was conducted on a 500×500 grid for 100 iterations in a single run.

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

Table 3: Performance Analysis Using gprof on the Base Implementation (1000×1000 grid with 1000 iterations). We observed that the saveFrame function was called 1000 times during the benchmark. This frequency occurred because we configured the system to save a frame every 10 iterations while executing 10 complete runs.

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 4: Cachegrind Analysis of Base Implementation Using a 1000 × 1000 Grid with 100 Iterations. The displayed percentages indicate each operation's proportional contribution to the total computational load.

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 12,602,250 LL data cache read misses and 12,501,900 LL 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 L1 cache performance, where 37,880,250 read misses and 12,501,900 write misses occurred in the update function alone. Thus, frequent main-memory accesses cause significant performance penalties.

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.cpp`, 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 neglected optimized v3 for other platforms.

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 GCUPS.

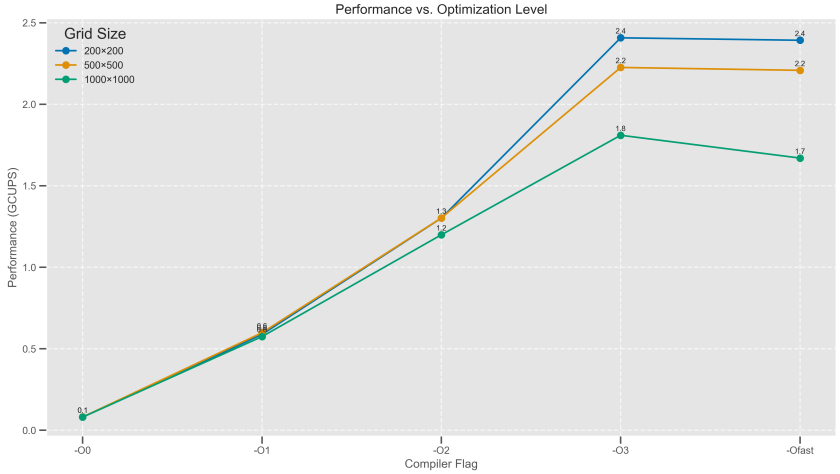

Figure 1: Performance (in GCUPS) of the optimized heat diffusion simulation across different compiler optimization levels and grid sizes. The performance increases significantly with each optimization level, with the most substantial improvement occurring between -O2 and -O3. Note that larger grid sizes (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.

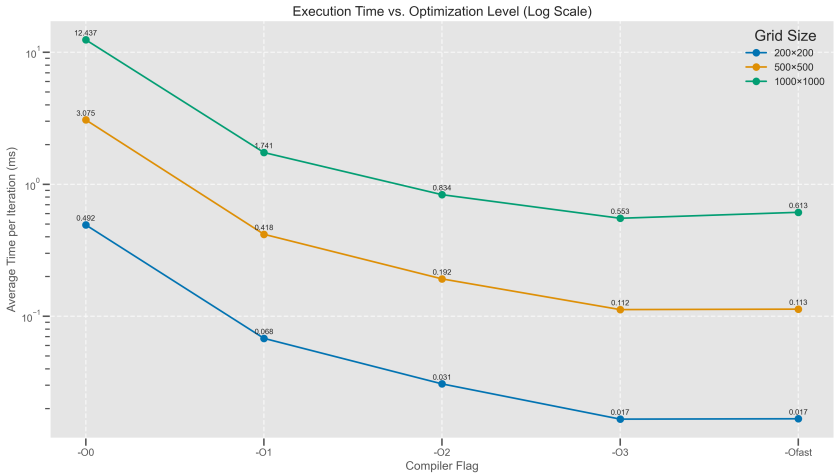

Figure 2: Execution time per iteration (in milliseconds, log scale) across different optimization levels for three grid sizes. The most dramatic reductions occur between -O0 and -O1, with diminishing returns at higher levels. Note that for all grid sizes, execution time stabilizes after -O3, with the 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 24× speedup for small grids (200×200) and 18× 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` (≈6×) 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 worse than the baseline for 1000×1000 grids. 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.

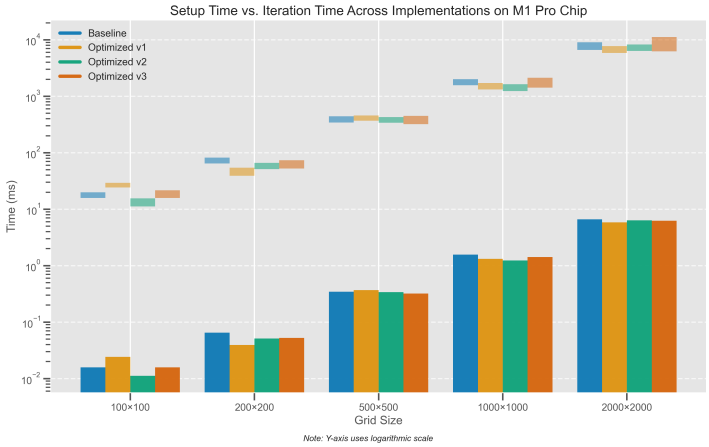

Figure 4: Performance Comparison: Setup versus Iteration Time Across Grid Sizes. This logarithmic-scale visualization compares computational efficiency between baseline and three optimized implementations (v1-v3) across five grid resolutions. The chart reveals that while setup time (solid color) increases proportionally with grid dimensions, iteration time (lighter shade) demonstrates improved scaling in optimized versions, particularly at larger grid sizes.

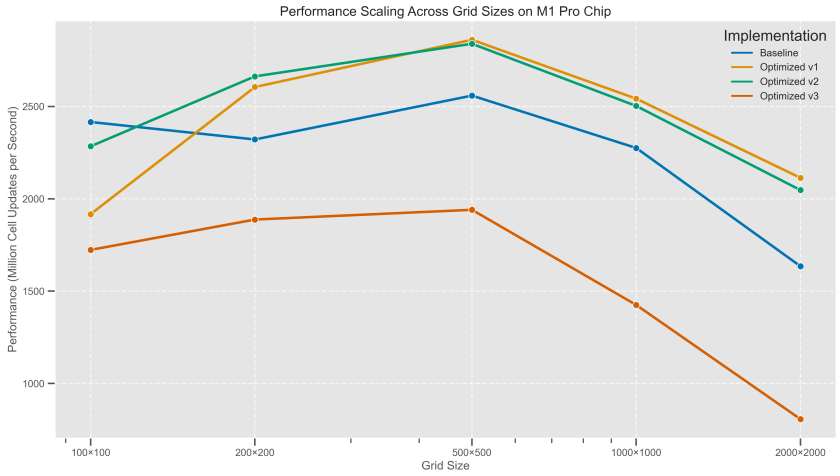

Figure 5: Performance scaling (MCUPS) across grid sizes for all implementations on the M1 Pro chip. All implementations achieve peak performance at 500×500 grid size, with Optimized v1 and v2 showing the best performance. Optimized v3 shows significant performance degradation for larger grids.

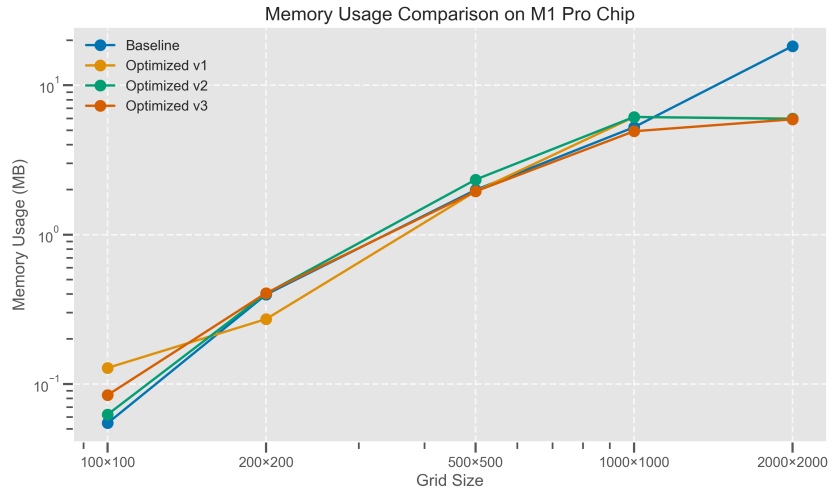

Figure 6: Memory usage (MB) across different grid sizes for all implementations. The baseline shows exponential growth for the largest grid size, while optimized versions maintain more moderate growth rates, demonstrating better memory management.

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 versions that showed consistent performance on macOS.

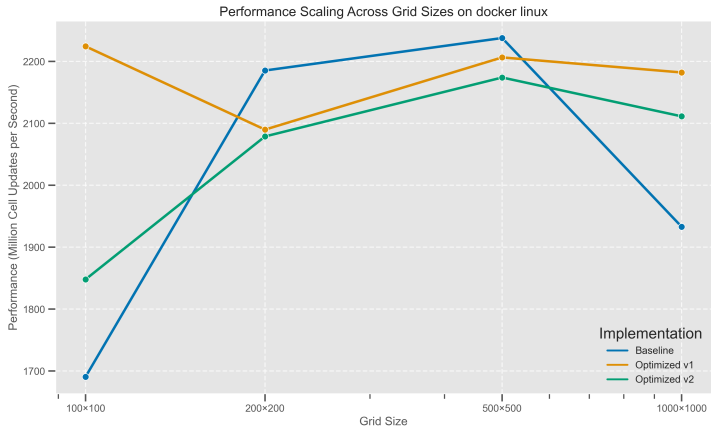

Figure 7: Performance scaling (in MCUPS) across grid sizes on Docker Linux for baseline and optimized implementations. The baseline implementation (blue) shows excellent performance for mid-range grids but drops significantly (by 13%) at 1000×1000 scale. Optimized v2 (green) gradually excels as grid dimensions increase. In contrast, Optimized v1 (orange) demonstrates the most consistent scaling behavior, starting with the highest performance for small grids but maintaining stable efficiency as problem size grows.

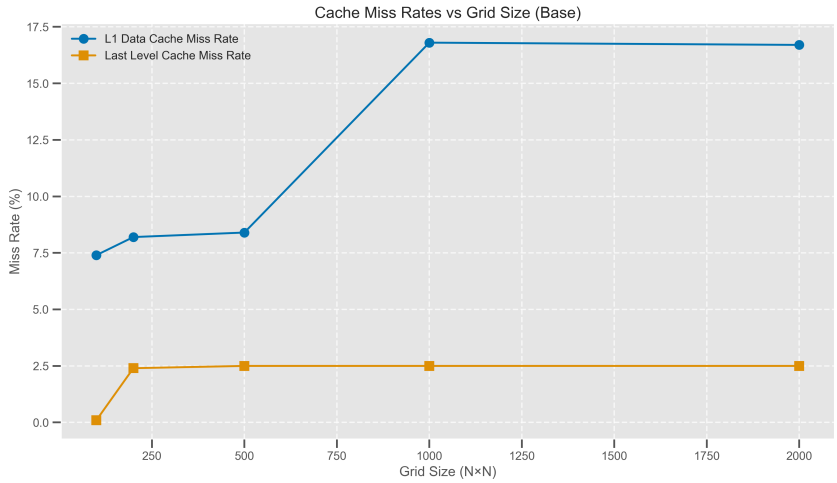

Figure 8: Cache miss rates versus grid size for the baseline implementation, showing how cache efficiency changes with problem size. Note the sharp increase in L1 data cache miss rate at 1000×1000 grid size, indicating when the working set exceeds L1 cache capacity.

Event Type	Total Count	update()
Instruction refs (Ir)	706,569,177	700,802,500 (**+0.06%**)
Instruction L1 misses	4,981	9 (**+12.5%**)
Data reads (Dr)	250,366,698	249,302,200 (**-0.08%**)
Data L1 read misses	37,770,473	37,624,900 (**-0.67%**)
Data LL read misses	12,637,168	12,500,200 (**-0.81%**)
Data writes (Dw)	51,030,876	49,802,100 (**+0.001%**)
Data L1 write misses	12,729,189	12,475,200 (**-0.21%**)
Data LL write misses	12,728,049	12,475,200 (**-0.21%**)

Table 5: The Cachegrind analysis of the Optimized v1 implementation, running on a 1000×1000 grid through 100 iterations, reveals notable improvements in memory access efficiency. The percentages highlighted in bold quantify the proportional reductions compared to the baseline implementation.

The optimized v1 implementation shows quantifiable improvements over the baseline: the instruction references of the update function increased by 0. 06%, the reads decreased by 0. 08%, and the writes increased slightly by 0.001%. Data L1 read misses decreased by 0.67%. The similar instruction reference patterns indicate performance gains stem primarily from memory access pattern changes rather than computational optimizations.

The optimized v1 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.

When comparing Docker to native macOS performance, the optimized implementations showed notably lower absolute performance values due to the virtualization overhead. While the M1 Pro achieved up to 2.5 GCUPS with the optimized v1 implementation on a 1000×1000 grid, the Docker environment peaked at around 2.2 GCUPS for the same configuration. This substantial performance gap can be attributed to several virtualization-related factors. The Docker container runs within a virtualized ARM64 environment that doesn't have direct access to the M1 Pro's specialized memory architecture and cache hierarchy.

Additionally, the Docker environment suffered from more erratic performance at larger grid sizes, with more pronounced performance drops as problem size increased. This indicates that memory-bound operations in virtualized environments face additional overhead from the hypervisor layer mediating memory accesses. CPU scheduling within the container also introduces unpredictability, as the virtualized environment competes with the host system for resources.

On CSD3

The optimized implementations show a significant improvement over the baseline. The performance scaling graph (Figure 13) reveals interesting behavior across different grid sizes. For smaller grid sizes (size less than 500), the optimized v1 implementation surprisingly outperforms the rest. However, for larger grid sizes, the optimized v2 implementation demonstrates more consistent performance and eventually surpasses the baseline.

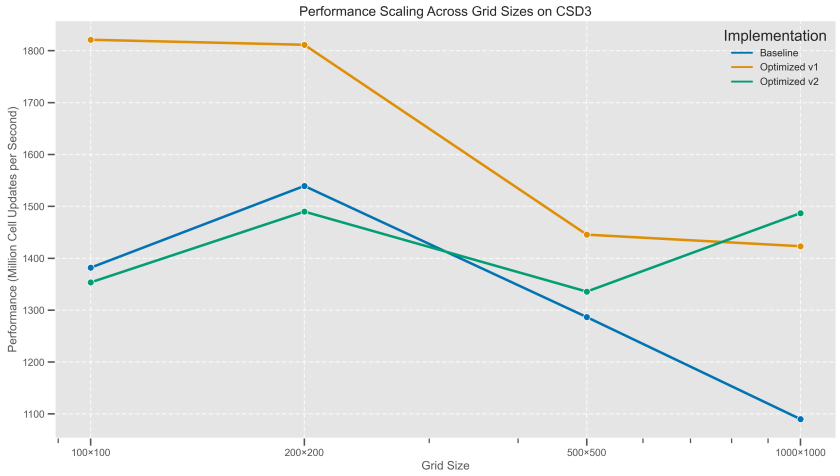

Figure 9: Performance Scaling Across Grid Sizes on CSD3. Performance is measured in MCUPS for different grid dimensions. The optimized v1 implementation shows superior performance for smaller grids but experiences sharper performance degradation at larger grid sizes, while the optimized v2 implementation excels at larger grid size.

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);
}
```

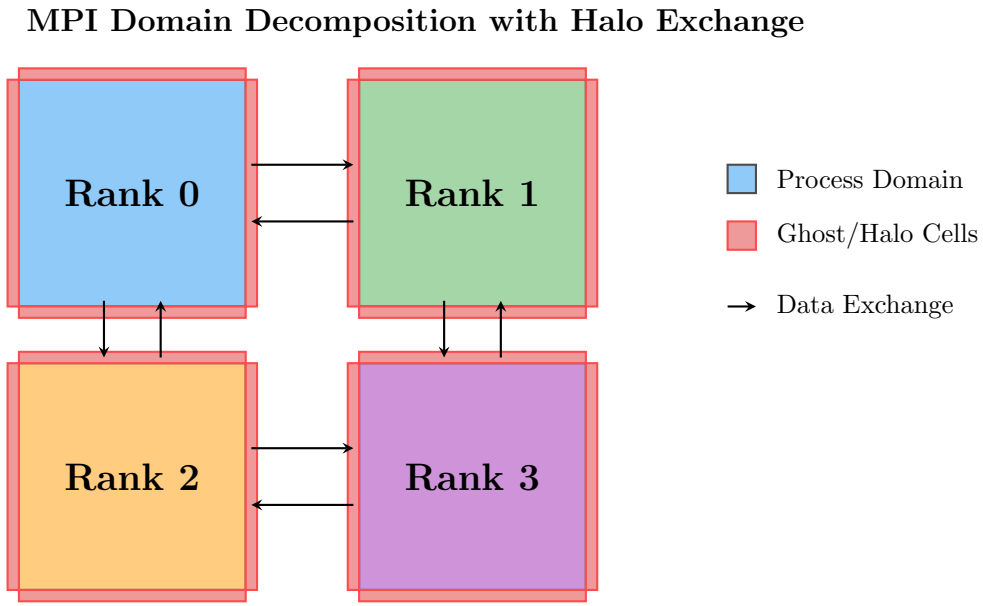

Figure 10: 2D domain decomposition with halo exchange for a 4-process MPI implementation. Each process maintains ghost cells (shown in red) that contain copies of boundary data from neighboring processes. Arrows indicate data exchange between processes.

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

Common Scaling Patterns

Across all platforms, we observed several consistent performance patterns:

- **Super-linear scaling** at low parallelism levels (2-4 units) due to improved cache utilization when problem domains are divided.
- **MPI vs. OpenMP tradeoffs**: MPI implementations consistently achieved higher absolute performance (GCUPS), while OpenMP demonstrated superior efficiency, particularly at moderate thread counts.
- **Diminishing returns** with increasing parallelism, with efficiency typically decreasing sharply beyond 8-16 processing units.
- **Hybrid approaches** performed best when aligned with hardware topology (typically 2-4 threads per process).

on MacOS

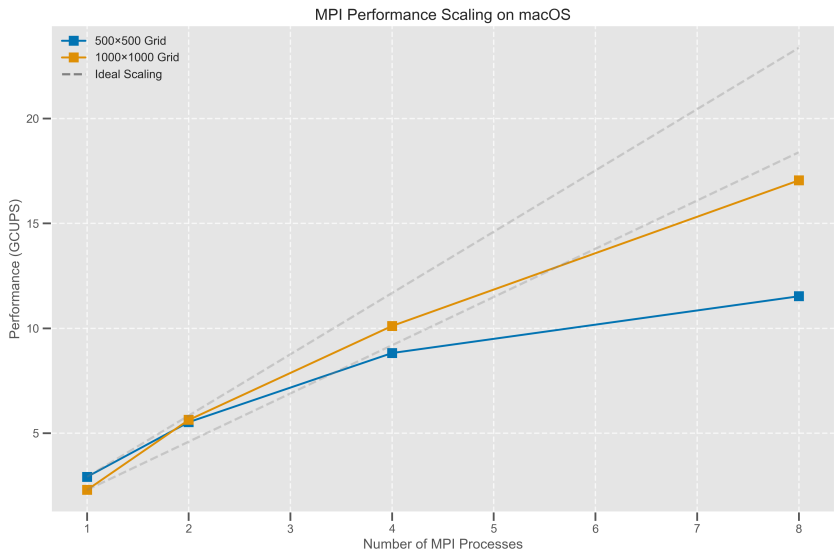

Figure 11: MPI performance scaling on macOS measured in GCUPS. The 1000×1000 grid (orange) achieves near-linear scaling up to 4 processes and reaches approximately 17 GCUPS with 8 processes. The 500×500 grid (blue) shows lower absolute performance but follows a similar scaling pattern, reaching approximately 11.8 GCUPS with 8 processes. Dashed gray lines indicate theoretical ideal scaling.

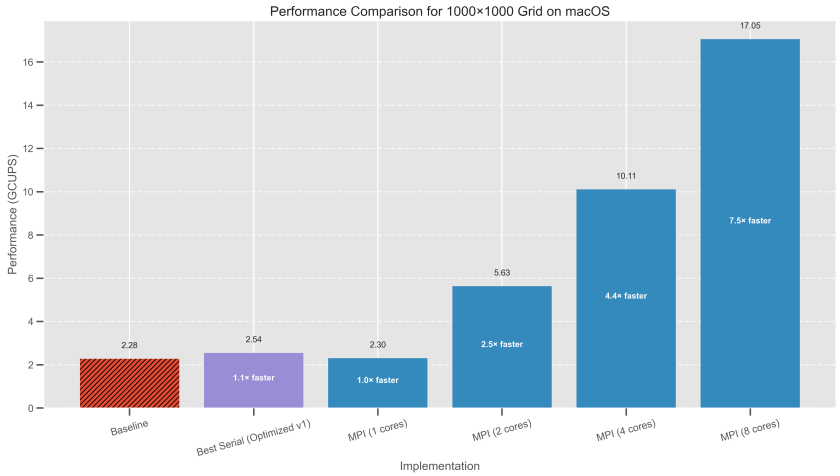

Figure 12: Performance Scaling of Heat Diffusion Simulation on macOS M1 Pro for a 1000×1000 Grid. The results demonstrate near-linear speedup and the effectiveness of both serial optimizations and parallel computing strategies on the M1 Pro architecture.

The M1 Pro demonstrated exceptional parallelism efficiency with MPI, achieving near-linear scaling up to 4 processes and super-linear efficiency (122%) with 2 processes for 1000×1000 grids. This architecture's unified memory system provided substantial advantages for our memory-bound application, resulting in the highest overall performance (17.05 GCUPS) despite having the fewest cores among tested platforms.

on Docker

Our containerized testing revealed the impact of virtualization overhead on parallelization strategies:

- OpenMP achieved extraordinary efficiency (264.7% with 2 threads, 259.1% with 4 threads) but lower maximum performance (9.83 GCUPS).
- MPI provided higher peak performance (13.82 GCUPS with 8 processes) despite lower efficiency.
- Both implementations showed performance peaks at 4 processing units in weak scaling tests.
- Hybrid configurations demonstrated that balancing processes and threads (2p×5t) could optimize performance within virtualization constraints.

Overall, our Docker environment testing demonstrates that MPI parallelization with 8 ranks provide the optimal approach for containerized scientific computing applications, achieving performance characteristics comparable to those observed on the Apple M1 Pro chip.

This finding highlights the scalability of process-based parallelism even within virtualized environments. Despite the inherent overhead of containerization, the MPI implementation reached 13.24 GCUPS (representing a 6.89× speedup over baseline), which aligns remarkably well with the performance metrics observed on native hardware.

This performance profile indicates that for Docker-based scientific computing, the optimal approach depends on the specific use case: MPI provides superior absolute performance despite lower efficiency, making it preferable for production runs where throughput is the

primary concern, while OpenMP might be more suitable for resource-constrained environments where efficiency is prioritized over raw performance.

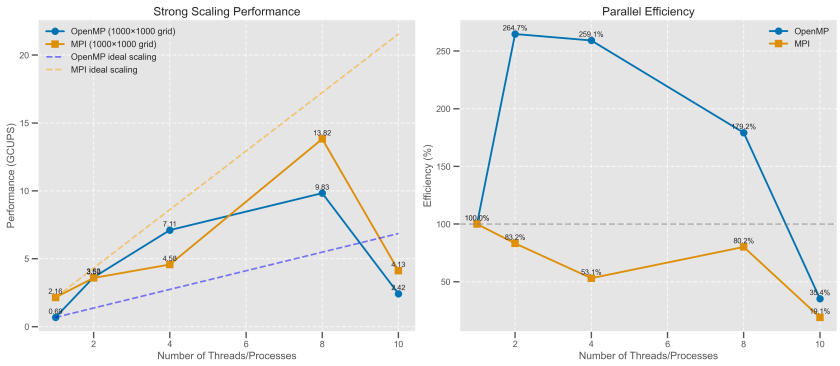

Figure 13: Strong scaling comparison between OpenMP and MPI implementations on Docker for a 1000×1000 grid. Left: Performance (GCUPS) shows OpenMP achieving better initial scaling but MPI reaching higher peak performance with 8 processes (13.82 GCUPS vs. OpenMP's 9.83 GCUPS). Right: Parallel efficiency demonstrates OpenMP's exceptional super-linear scaling (>250% efficiency) with 2-4 threads while MPI maintains more modest but steady efficiency through 8 processes.

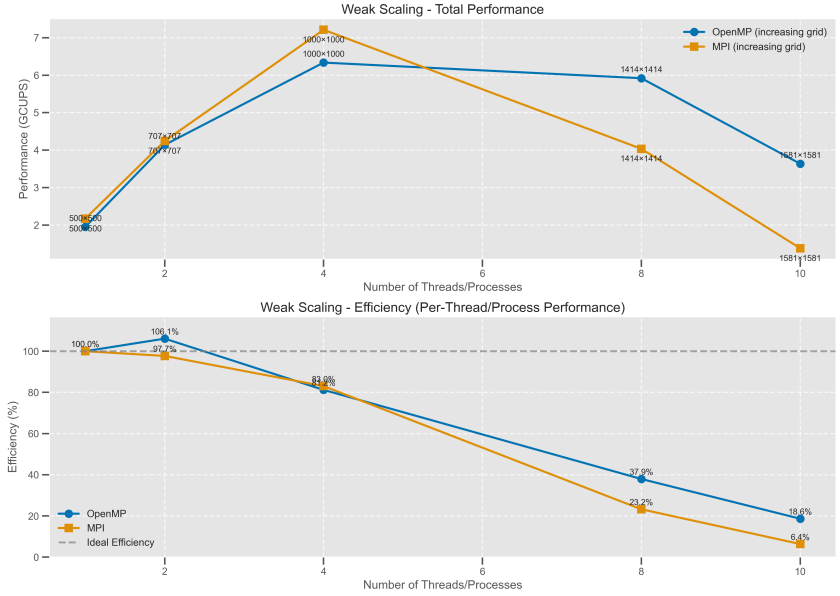

Figure 14: Weak scaling characteristics on Docker. Top: Total performance as grid size increases proportionally with processing units, showing both implementations peaking at 4 units before diverging. OpenMP maintains relatively stable performance through 8 threads while MPI shows steeper degradation. 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%.

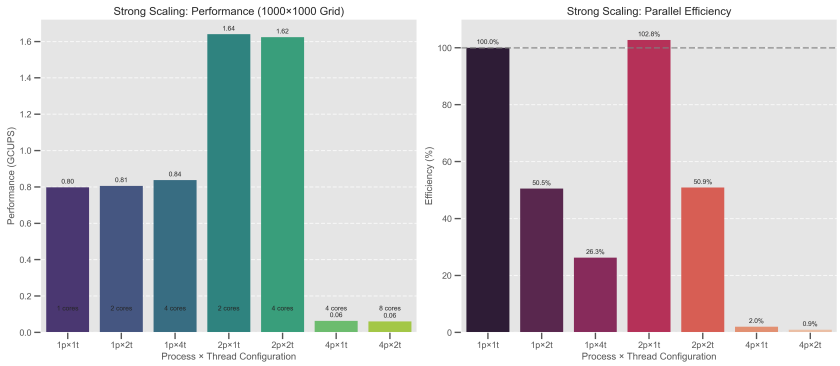

Figure 15: Strong scaling performance and parallel efficiency for a 1000×1000 grid across hybrid approach on Docker. Left: Performance (GCUPS) shows hybrid implementations (2p×1t and 2p×2t) achieving the highest performance at approximately 1.64 and 1.62 GCUPS respectively. Right: Parallel efficiency reveals super-linear scaling for the 2p×1t configuration (102.8%) while higher process/thread counts show rapidly diminishing efficiency.

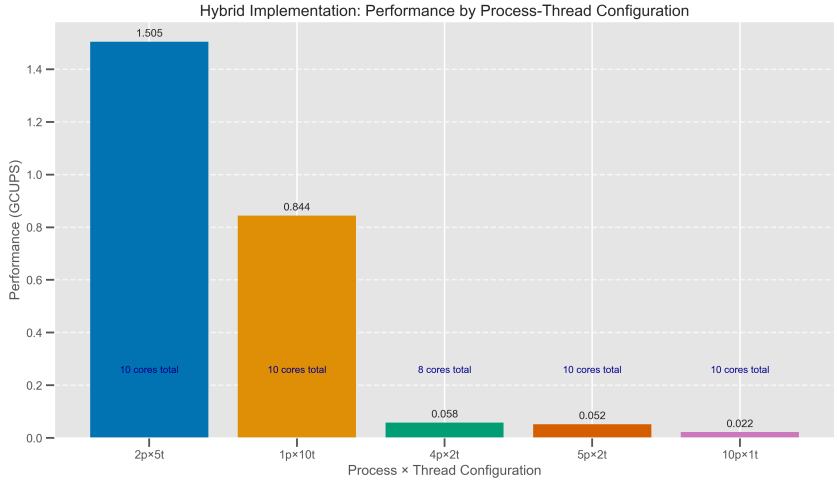

Figure 16: Hybrid implementation performance comparison on Docker showing GCUPS across different process-thread combinations. The 2p×5t configuration achieves the highest performance (1.505 GCUPS), while the pure OpenMP approach (1p×10t) reaches 0.844 GCUPS.

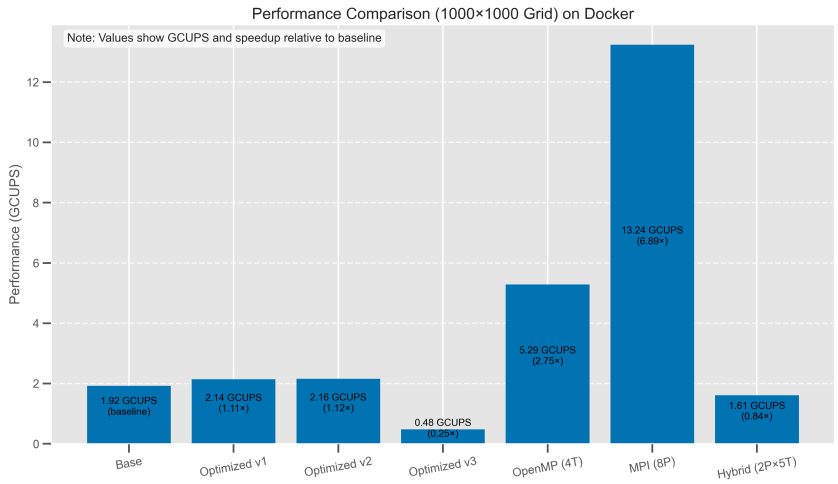

Figure 17: Performance Comparison (1000×1000 Grid) on Docker: Values show GCUPS and speedup relative to baseline for different implementation strategies. The MPI (8P) implementation achieves the highest performance at 13.24 GCUPS (6.89× speedup), while the Optimized v3 shows the lowest at 0.48 GCUPS (0.25× speedup). The OpenMP (4T) implementation reaches 5.29 GCUPS (2.75× speedup).

on CSD3

Our HPC cluster experiments revealed critical insights for large-scale deployments:

- OpenMP achieved super-linear efficiency ($> 130\%$) at 4-8 threads, but experienced dramatic performance variability and sharp efficiency drops beyond 16 threads.
- MPI maintained more consistent but lower efficiency ($\approx 70\%$ at 16 processes).
- Single-node hybrid configurations showed that matching the hardware topology (2p×38t) provided optimal performance.
- Multi-node scaling required careful process distribution, with 19p×4t (split 9/10 across nodes) achieving the best performance.

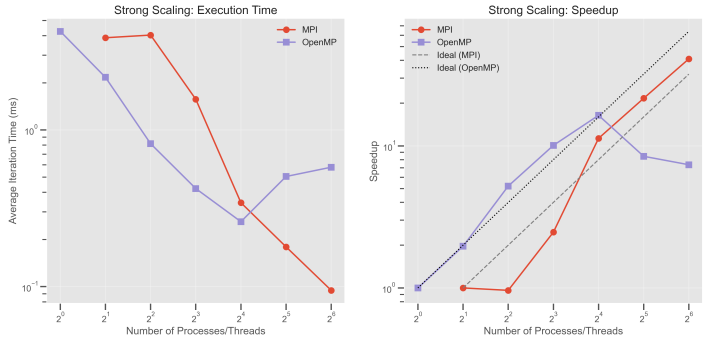

Figure 18: Parallel efficiency with strong scaling on CSD3, comparing OpenMP and MPI implementations. OpenMP shows super-linear efficiency at 4 and 8 threads before declining sharply beyond 16 threads. MPI maintains lower but more consistent efficiency, with a peak of approximately 70% at 16 processes. The dashed line represents ideal 100% efficiency.

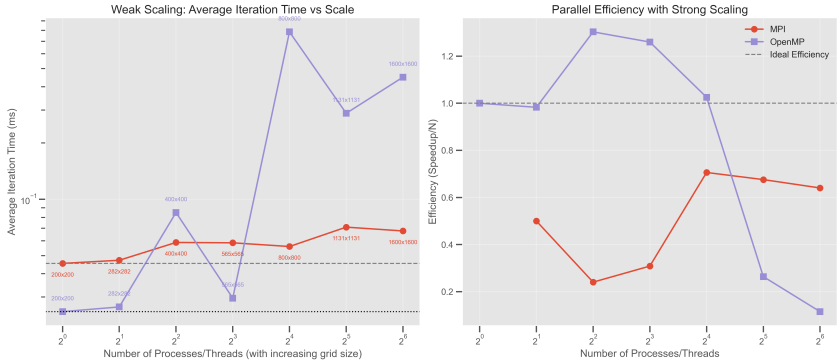

Figure 19: Weak scaling analysis on CSD3. Left: Average iteration time as grid size increases proportionally with thread/process count, showing OpenMP experiences highly variable times with dramatic spikes at certain configurations (particularly at 16 threads with 800×800 grid), while MPI maintains more consistent performance. Right: Parallel efficiency confirms OpenMP achieves super-linear efficiency at moderate thread counts but drops below 20% at higher thread counts.

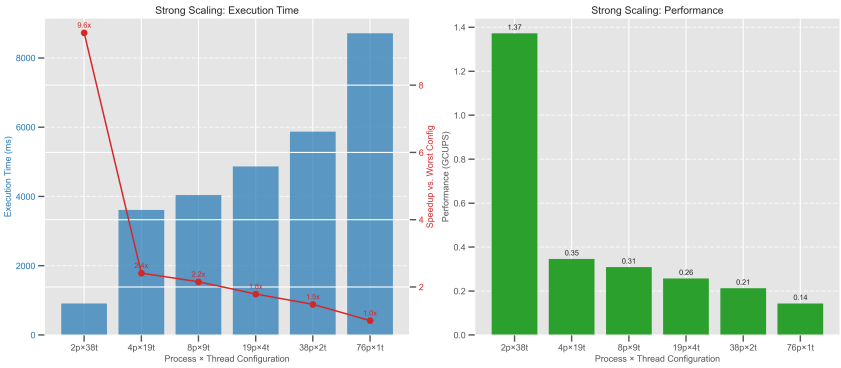

Figure 20: Strong scaling performance metrics on one CSD3 node for different process-thread configurations. Left: Execution time (ms) decreases from 885 ms with 2p×38t to 8,700 ms with 76p×1t, with a speedup factor of 9.6× between extreme configurations, though with rapidly diminishing returns as shown by the red curve. Right: Performance in GCUPS showing the 2p×38t configuration dramatically outperforms all others at 1.37 GCUPS

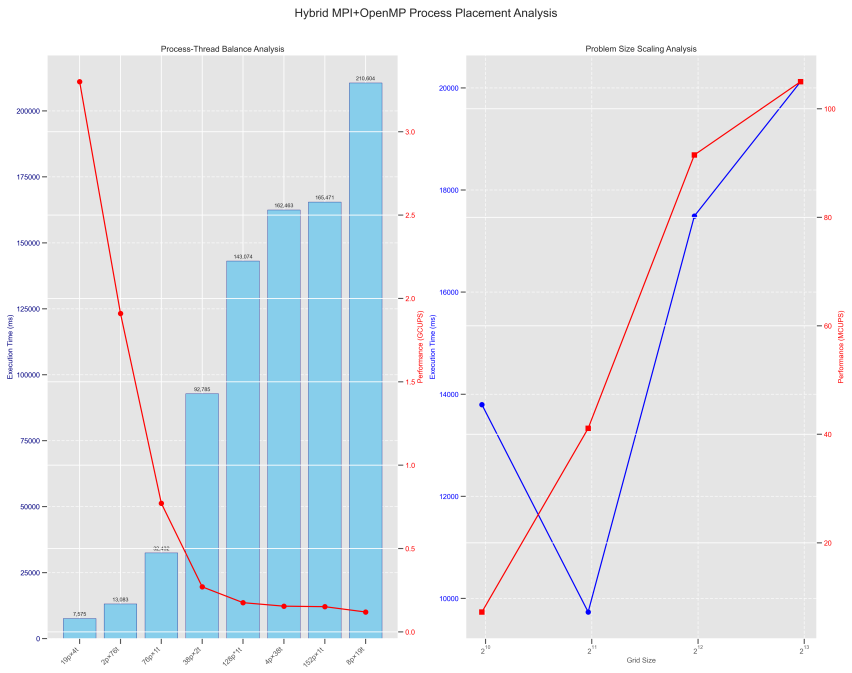

Figure 21: Hybrid MPI+OpenMP configurations across two CSD3 nodes. Left: Comparison of various process-thread combinations, where each process (p) runs multiple threads (t) per processor, with the 19p×4t arrangement (9 processes on one node, 10 on another) yielding peak performance (3.30 GCUPS). Right: Performance assessment using fixed parallelism (8 processes, 8 threads) across increasing problem dimensions, reaching 41.10 MCUPS with 2000×2000 grid size.

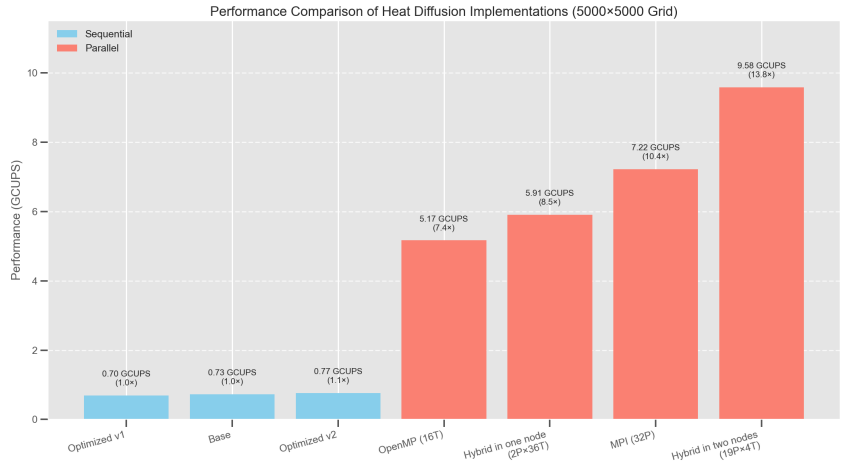

Figure 22: Computational Performance of Sequential vs. Parallel Heat Diffusion Solvers on a 5000×5000 Grid. Parallel implementations achieve up to 13.8× speedup over sequential code, with the two-node hybrid solution delivering the highest throughput at 9.58 GCUPS.

Our cross-node experiments (Figure 21) demonstrated that carefully balanced hybrid configurations deliver superior performance across multiple compute nodes. When scaling to two CSD3 nodes (152 cores total), the optimal configuration used 19 processes with 4 threads each (19p×4t), with processes split 9/10 between nodes. This arrangement achieved 3.30

GCUPS, significantly outperforming both process-heavy and thread-heavy alternatives. Increasing problem size improved computational efficiency, with a fixed parallelism configuration (8 processes, 8 threads) reaching 41.10 MCUPS at 2000×2000 grid dimensions.

When running our fully optimized implementations on a 5000×5000 grid (Figure 22), the parallel implementations achieved up to 13.8× speedup over sequential code. The two-node hybrid solution delivered the highest throughput at 9.58 GCUPS, demonstrating the cumulative benefits of our optimization and parallelization strategies.

Cross-Platform Comparison

Surprisingly, the Apple M1 Pro consistently outperformed the CSD3 HPC cluster, achieving 17.05 GCUPS compared to CSD3's 9.58 GCUPS despite CSD3's significantly higher core count and specialized architecture. This counter-intuitive result highlights that for memory-bound applications like heat diffusion simulation, architectural efficiency and memory bandwidth are more critical than raw computational resources.

This finding has significant implications for scientific computing workloads, suggesting that modern consumer-grade ARM-based architectures may offer competitive or superior performance for certain classes of memory-bound computations compared to traditional HPC infrastructure.

Summary

The heat diffusion simulation optimization study compared performance across Apple M1 Pro, Docker, and CSD3 HPC platforms, finding that memory layout restructuring and parallelization provided significant speedups. MPI implementations achieved the highest absolute performance while OpenMP showed better efficiency. Surprisingly, the consumer-grade M1 Pro (17.05 GCUPS) outperformed the specialized CSD3 HPC cluster (9.58 GCUPS), demonstrating that for memory-bound applications, architectural efficiency and memory bandwidth are more critical than raw computational resources.