C2 Coursework: Optimisation and Parallelisation of heat diffusion simulation

jn492

April 14, 2025

Word count: 2976 words

1 Introduction

This coursework aims to double the performance of a given heat diffusion simulation by leveraging the computational resources of two Intel Ice Lake nodes. The report outlines the development process, beginning with profiling the baseline code to identify suitable optimisation (Section 3) and parallelisation (Section 4) techniques. Each technique is implemented and evaluated in isolation to measure its individual impact, before combining the most effective strategies into a final optimised version. To begin, I will first provide an overview of the testing methodology, performance metrics, the computer system, and Doxygen documentation.

1.1 Testing

While `animate.ipynb` allows for visual inspection of output consistency, it does not provide a quantitative measure. Bitwise comparison is possible, but when implementing MPI or OpenMP parallelisation, small discrepancies may arise due to the non-deterministic order of floating-point operations, leading to rounding errors.

To address this, I will use the mean squared error (MSE) as a metric. I have chosen a passing threshold of MSE $< 1 \times 10^{-6}$, as this implies the acceptable error per cell is up to 0.001°C, which represents a high degree of precision given that temperatures are typically 4–5 orders of magnitude larger. The test is implemented in the `/test` directory and is fully automated using Google Test. Continuous integration is facilitated through Git workflow practices, where each optimisation or parallelisation feature is developed in its own branch and merged into `main` only after passing automated tests. Tests can also be ran manually using standard commands such as `make clean` followed by `make` in the test folder.

1.2 Performance tracking and Profiling

To evaluate each optimisation and parallelisation technique, all code variants are run 20 times, timed, and averaged. Speedup is measured relative to the baseline average. Since system conditions can vary across development days, a fresh baseline run was performed before timing each optimized or parallelised version to ensure fair comparison. The speed and error is given by Equation 1 and 2.

$$\text{Speedup} = \frac{\overline{T}_{\text{baseline}}}{\overline{T}_{\text{optimised}}} \tag{1}$$

$$\sigma_{\text{speedup}} = \text{Speedup} \times \sqrt{\left(\frac{\sigma_{\text{baseline}}}{\overline{T}_{\text{baseline}}}\right)^2 + \left(\frac{\sigma_{\text{optimised}}}{\overline{T}_{\text{optimised}}}\right)^2} \tag{2}$$

In addition, profiling will be carried out using Valgrind to analyse performance at a finer granularity. Specifically, the following command will be used:

```
valgrind --tool=callgrind --simulate-cache=yes ./$(BUILD_DIR)/heat_diffusion 1
```

This command runs the simulation once through Callgrind, a profiling tool that collects detailed statistics such as instruction counts and function call information. This data helps identify computational bottlenecks and better understand the sources of any observed speedup. The `--simulate-cache=yes` flag enables simulation of L1 and L2 cache behavior, allowing analysis of cache usage and cache misses. This is useful for evaluating whether optimisations like cache blocking have improved memory access efficiency.

1.3 Computer system

All reported optimisation performance is based on runs conducted on my local machine (4-core MacBook Air with an Intel i5 chip). However, profiling of the optimised and parallelised code is performed on CSD3, as tools like Valgrind require a Linux environment. CSD3 is also used for evaluating parallelised performance, as it supports multi-node and multi-process execution.

1.4 Documentation

All code is documented using Doxygen, with comments structured to support automatic generation of developer documentation. The compiled HTML reports can be found in `docs/html/index.html` within each folder. In the generated documentation, comments for optimised code sections are highlighted in bold for clarity.

For example, running Doxygen on the `optimisation_2_blocking/best_block` folder produced Figure 1 as the main documentation page, and Figure 2 showing the class list, including function-level descriptions and optimisation details.

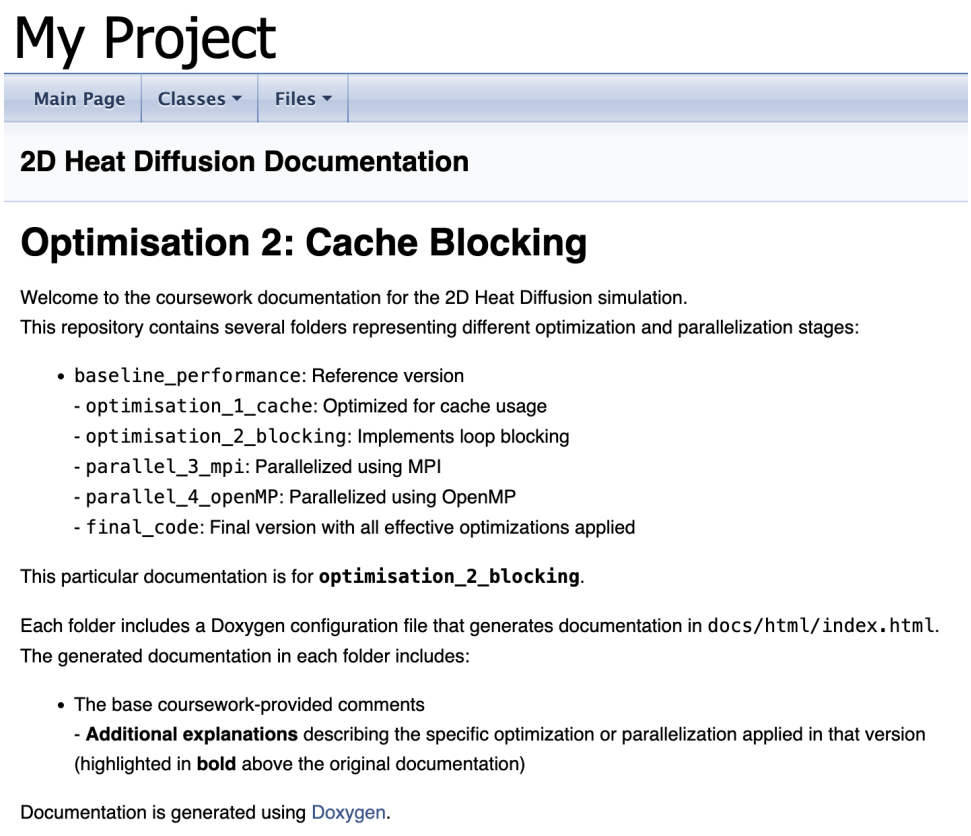

Figure 1: Doxygen-generated main documentation page.

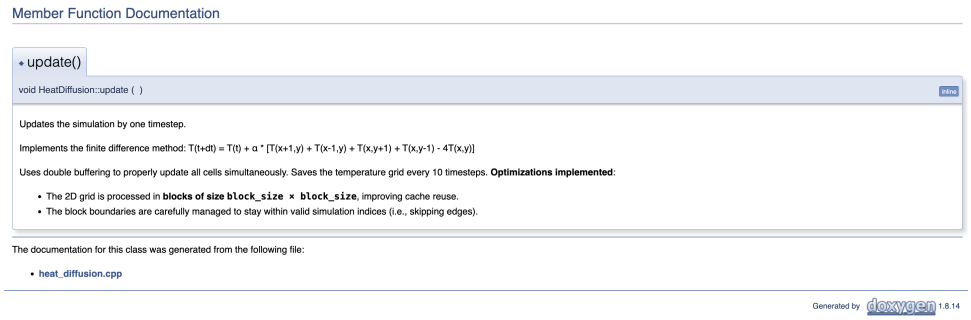

Figure 2: Doxygen class list showing function-level documentation and optimisations.

2 Baseline performance

The execution time of the original code averaged to 203 ms ± 28.08ms. The profiling summary shown in Table 1 shows very low cache misses relative to the total number of instructions, implying that the baseline code generally fits well into the L1 cache. This is reasonable, as each frame update uses a 100×100 grid of doubles (8 bytes) for the current temperature, with an additional grid for the next temperature values, totaling to 160KB of data per update. An inspection of the CSD3 cache structure shows 80KB of L1 cache (per core), which confirms that a single update is roughly the right size to partially fit in L1 cache.

Table 1: Valgrind Summary Statistics

Metric	Count
Instructions Executed (Ir)	577,928,983
Data Reads (Dr)	179,372,601
Data Writes (Dw)	110,262,922
L1 Instruction Cache Misses (I1mr)	8,048
L1 Data Cache Misses (D1mr)	169,486
L1 Data Write Misses (D1mw)	132,502
Last-Level Instruction Cache Misses (ILmr)	2,563
Last-Level Data Cache Misses (DLmr)	9,087
LL Data Write Misses (DLmw)	4,514

Table 2 provides a more detailed breakdown of the code profile. Functions such as `__printf_fp_l` and `__mpn_divrem` are excluded, as they belong to external libraries and are not directly related to the optimisation of the heat diffusion code.

Table 2: Top Functions by Instruction Count and Cache Misses

Function	Ir	D1mr	D1mw
`HeatDiffusion::update()`	119,221,320	127,630	125,000
`std::vector<std::vector<double>> ::operator[]`	94,119,886	10	0
`std::vector<double>::operator[]`	73,951,339	7,603	
`HeatDiffusion::saveFrame(int)`	2,537,838	12,802	20

Table 2 reveals that the `update()` function dominates both instruction count and L1 cache misses, likely due to its repeated access to the 2D temperature grid. In Section 3.1, I will explore how flattening the grid into a 1D vector may improve spatial locality and reduce overhead. Given that `update()` also accounts for the most L1 cache misses, Section 3.2 will investigate what block sizes can improve cache performance.

The second largest contributor to instructions is `std::vector<std::vector<double>>::operator[]`, further suggesting that significant computational resources are spent accessing the 2D grid. The third most significant contributor involves calls to std :: vector<double>::operator[], which cannot be modified without changing the precision of calculations, which I will avoid doing to preserve the original code's output. Finally, as the `saveFrame()` function has negligible performance impact, it will not be considered for optimisation.

3 Optimisation

3.1 Flattened 1D vector

Flattening the vector was implemented through indexing the flattened grid using the formula `y * width + x`, where `y` is the row index, `width` is the number of columns in the grid (100), and `x` is the column index.

The Valgrind profile in Table 3 shows a clear reduction in memory overhead, as the instruction count, data reads, and data writes have all decreased. This is due to the flattened grid's one-dimensional, contiguous memory layout, which allows for faster row-wise access. Cache misses also improved due to better spatial locality. When the CPU loads a value, adjacent elements are fetched into the cache and reused during updates. Furthermore, since all rows are stored in a single memory block, accessing neighbouring rows is more efficient. In contrast, a `vector<vector<double>>` stores each row in a separate memory allocation, which can lead to scattered memory access and more frequent cache misses when accessing elements in different rows.

Table 3: Comparison of Callgrind Metrics: Baseline vs. Flattened Optimisation

Metric	**Baseline**	**Flattened**	**% Change**
Instructions (Ir)	577,928,983	434,759,724	−24.77%
Data Reads (Dr)	179,372,601	136,931,576	−23.66%
Data Writes (Dw)	110,262,922	79,043,406	−28.31%
L1 I-Cache Misses (I1mr)	8,048	7,716	−4.13%
L1 D-Cache Read Misses (D1mr)	169,486	157,128	−7.29%
L1 D-Cache Write Misses (D1mw)	132,502	129,991	−1.90%
LL I-Cache Misses (ILmr)	2,563	2,463	−3.90%
LL D-Cache Read Misses (DLmr)	9,087	9,085	−0.02%
LL D-Cache Write Misses (DLmw)	4,514	4,389	−2.77%

Flattening the grid into a one-dimensional array yielded a 1.14 ± 0.03 speed-up on my local machine.

3.2 Cache Blocking

3.2.1 Tuning

Cache blocking was also explored as it might benefit the heat diffusion simulation. Since each cell update depends on its top, bottom, left, and right neighbours, blocking enables loading and processing small regions of the grid (e.g., 8×8) at a time instead of loading the grid row-wise. This may improves spatial locality and increases cache reuse, as neighbouring values are more likely to remain in cache during computation.

To implement this, the update function was modified to use a nested loop that iterates over the grid in blocks, with bounds clamped to avoid exceeding grid edges. Two key considerations influence the effectiveness of cache blocking. Firstly, the optimal block size varies with hardware, depending on CPU cache sizes and architecture. Thus, this subsection investigates block size tuning on my local machine to identify the best-performing configuration.

Cache blocking is more effective on larger grids, where memory access cost outweighs block management overhead. To highlight performance differences, grid sizes of 256×256 and 512×512 were chosen, as they are large enough to exceed L1 and L2 cache limits. They also align to powers of two, which helps

reduce alignment overhead, making performance increase from cache blocking more apparant. These sizes also remain feasible on a personal machine. Block sizes from 1 to 128 were tested, with five runs per configuration.

Table 4 shows that the optimal block sizes were 69, 13, and 64 for the respective grids. These relatively small values likely reflect the small cache size of the local machine.

Table 4: Cache block tuning results showing optimal block sizes and timing statistics

Grid Size	**Best Block Size**	**Avg. Time (ms)**	**Std. Dev (ms)**
100×100	69	195	13.2
256×256	13	618	35.9
512×512	64	2380	303

Figures 3, 4, and 5 show the full cache block tuning results for each grid size. The standard deviation in timing remains relatively high, which is a result of variable overhead overshadowing the benefits of blocking. This is an incentive to scale the code up even larger for the final optimised code in Section 5 to observe the scaled advantages of optimisation and parallelization.

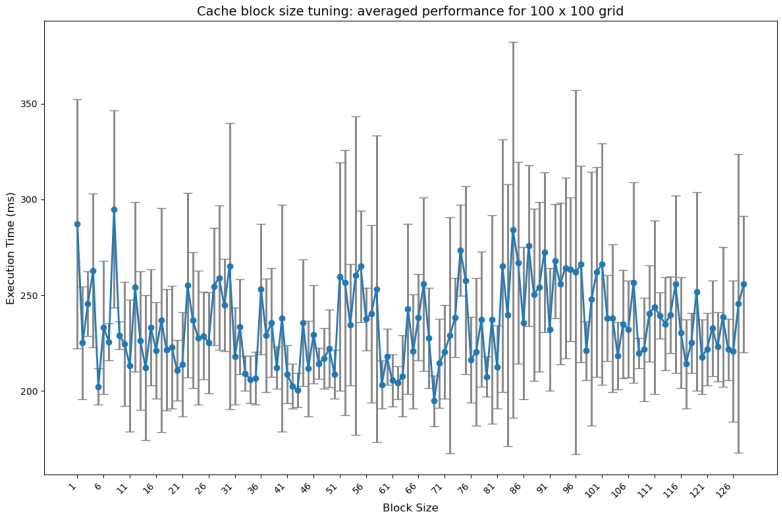

Figure 3: Performance impact of cache block sizes ranging from 1 to 128 on a 100×100 grid

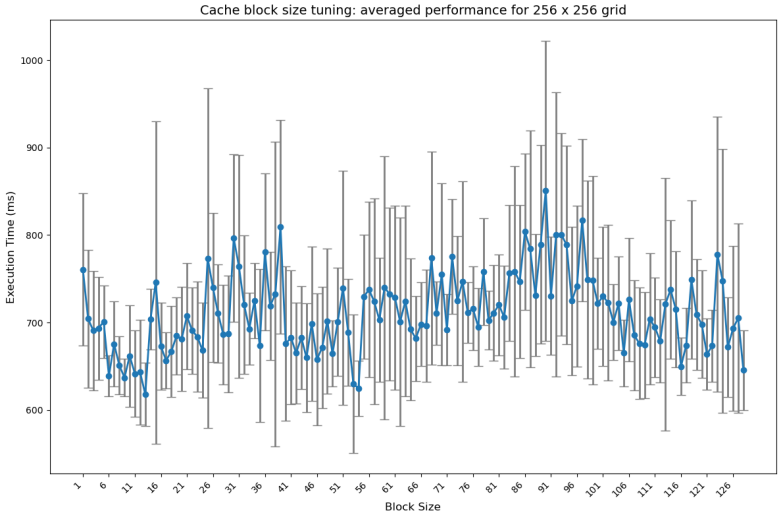

Figure 4: Performance impact of cache block sizes ranging from 1 to 128 on a 256×256 grid

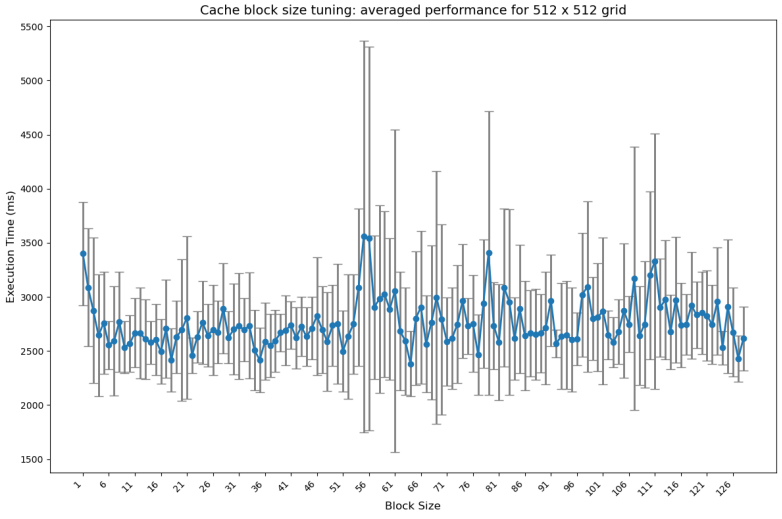

Figure 5: Performance impact of cache block sizes ranging from 1 to 128 on a 512×512 grid

While block sizes such as 69 and 13 yielded the best average timings for the 100×100 and 256×256 grids, these results are likely due to measurement noise and the limitations of small-scale tests, rather than reflecting true hardware-level optimisation. As shown in Figures 3 and 4, the large and overlapping error bars suggest significant measurement noise, making it difficult to confidently conclude that these are truly optimal for my system.

In contrast, testing on a larger 512×512 grid revealed that a block size of 64 yielded the best performance. This aligns well with the underlying hardware: on my CPU, a cache line is 64 bytes, and since each `double` is 8 bytes, a block of 64 elements fits exactly into 8 cache lines (512 bytes). This layout allows the CPU to fetch a full block with minimal memory overhead, effectively exploiting spatial locality and reducing cache misses during each update.

3.2.2 Testing best block

A block size of 69 was tested on the 100×100 grid, yielding only a 1.03 ± 0.04 speedup over the baseline when averaged over 20 runs. Given the size of the uncertainty, this result is not statistically significant. The minimal gain is likely offset by the overhead of additional loop logic and boundary checks, which dominate at small grid sizes. Additionally, the baseline already benefits from good spatial locality, as the 160 KB grid fits entirely within L2 cache. The choice of 69 also leads to inefficient tiling, since 100 is not divisible by 69, increasing branching near the grid boundaries. Valgrind profiling further confirms no improvement in cache performance.

4 Parallelisation

4.1 MPI

I chose a 2D Cartesian topology for domain decomposition, as it aligns with the heat diffusion model's need for communication with all four neighbors. Unlike row-wise, column-wise, or scattered strategies, it ensures efficient updates by preserving spatial locality in both dimensions. As previous optimisations show a clear benefit of using flattened arrays, I will incoporate this technique that into my MPI implementation. Figure 6 depicts a flowchart of the code suited for a 2D cartesian domain decomposition.

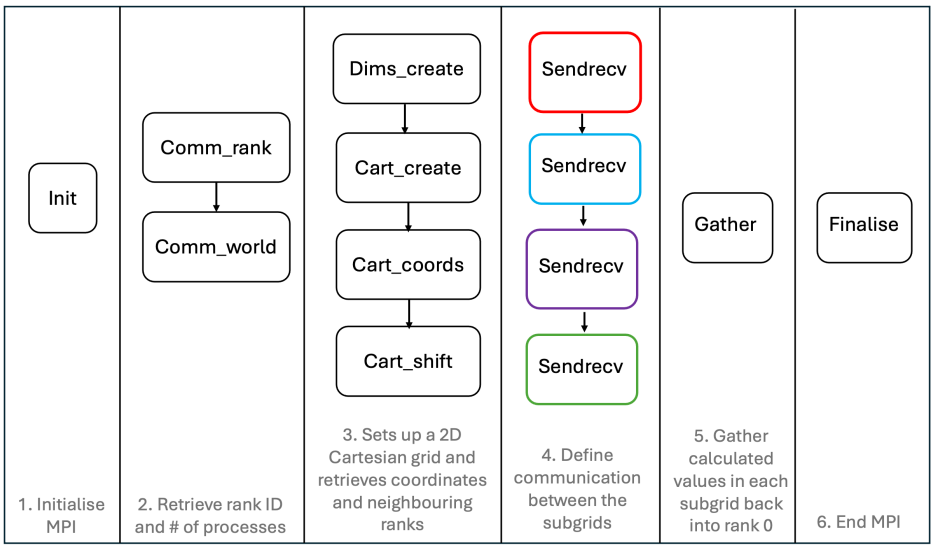

Figure 6: A flowchart showing 2D cartesian decomposition MPI code structure

`MPI_Sendrecv` is a blocking communication method chosen over `MPI_Send` and `MPI_Recv` to prevent deadlock. This is particularly important in my case, where a two-sided halo exchange is performed on a 2D Cartesian grid. If each process were to call `MPI_Send` without a matching receive, all subdomains would

be sending simultaneously with no process ready to receive, resulting in a deadlock. Nonblocking methods like `MPI_Isend` and `MPI_Irecv` are not used here, as they would require additional buffer management and synchronization. More importantly, my simulation waits for all halo data to be exchanged before proceeding with computation, leaving no opportunity to overlap communication with useful work. In this context, non-blocking communication would only increase complexity without offering any performance advantage.

To ensure flexibility and robustness, I have applied `MPI_Dims_create`, which automatically calculates the number of processes used per subgrid given the number of processes assigned. I have also ensured with `cartshift` that if the shifted (neighbour) returns `MPI_PROC_NULL` (beyond the grid), there will be no receiving or sending occurring. Padding is added to each subgrid to accommodate for halo cells. The send and receive and halo interactions are further depicted in Figure 7.

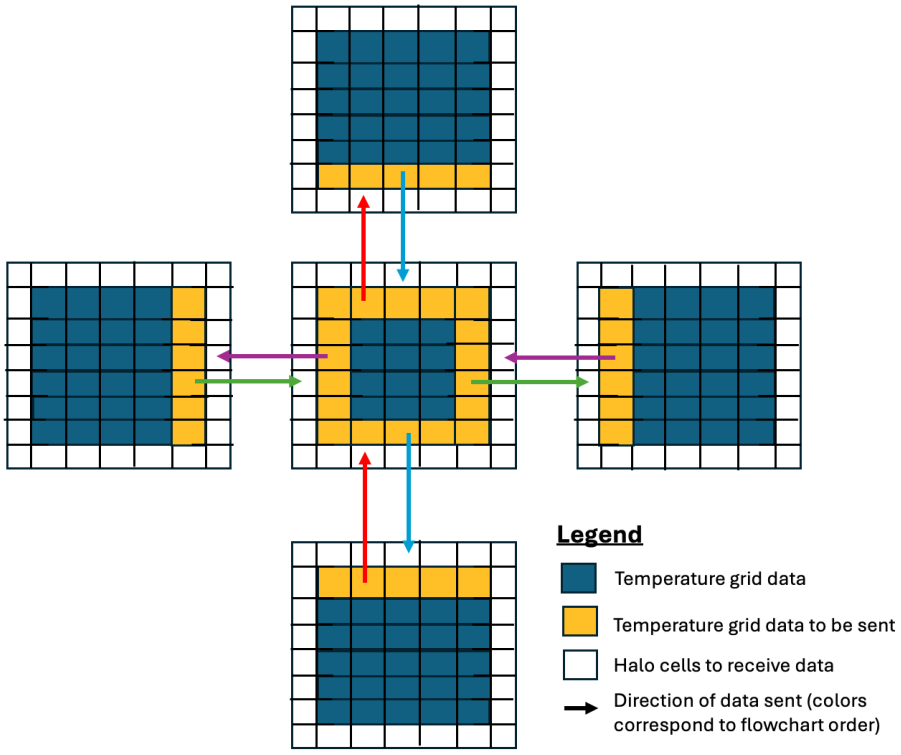

Figure 7: Diagrammatic represntation of how 2D cartesian decomposition send recieve works with halo cells

I have chosen not to support uneven grid splitting in the implementation of the MPI code, such that if the number of processes is set to 7, the program will raise an error since a 100×100 grid cannot be evenly divided among them. This decision is based on two main reasons. Firstly, uneven subdomain sizes causes load imbalance. Larger sections require more computation time, which delays the overall update because all results must be gathered at rank 0 before proceeding. Second, supporting uneven splits introduces additional overhead to the splitting logic. Since my primary goal is to optimize the baseline 100×100 grid, which is already sensitive to performance noise and overhead, I will not implement uneven grid splitting.

Firstly, I will use the flattened data representation with MPI on a single node. The performance was tested using 2, 4, 8, and 16 processes. The same tests were also conducted on two nodes. The results are shown in Figure 8.

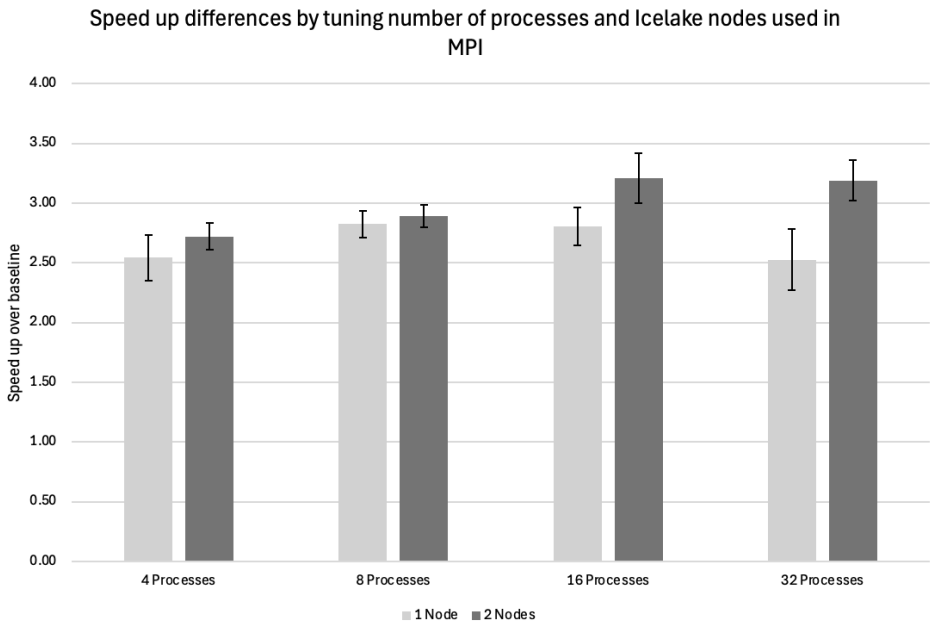

Figure 8: MPI tuning: Speed up over baseline for 4, 8, 16, 32 processes over 1 and 2 Icelake nodes

A 3.21 ± 0.21 times speedup was achieved using 16 processes distributed across 2 IceLake nodes, likely due to effective workload distribution. In contrast, performance degraded when using 32 processes on a single node, suggesting that the overhead introduced by additional processes outweighed the benefits of parallelization. The optimal configuration appears to be 16 processes across 2 nodes, providing a good balance between resource utilization and performance gain.

I also investigated MPI reordering, which theoretically allows MPI to optimize communication by reassigning rank numbers. However, when applied to the best-performing configuration (16 processes across 2 nodes), reordering actually reduced performance, as shown in Figure 9.

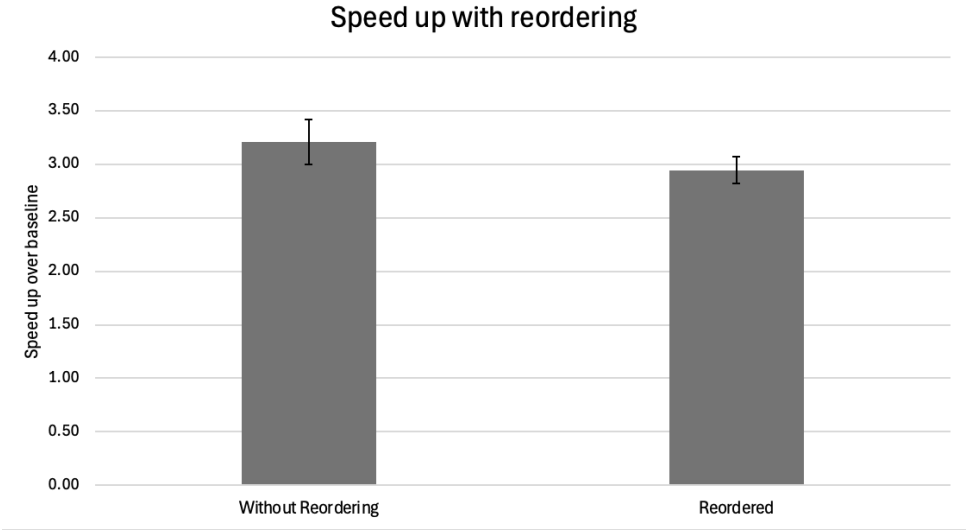

Figure 9: Reordering using the best performing mpi configuration of 16 processes over 2 nodes

I further investigated this issue by examining the workload distribution across cores. As shown in Table 4.1, runs with reordering enabled resulted in all processes being allocated to a single node, rather than being evenly distributed across both nodes. This is not typically an outcome of rank reordering

alone, hence further investigation is required to determine whether this behavior is due to reordering or other processes. To avoid such issues, future implementaions should consider explicitly enforcing balanced rank distribution using `--map-by ppr:<processes_per_node>:node`.

Rank	Ordered	Reordered
0	cpu-q-239	cpu-q-7
1	cpu-q-239	cpu-q-246
2	cpu-q-239	cpu-q-246
3	cpu-q-239	cpu-q-246
4	cpu-q-239	cpu-q-246
5	cpu-q-239	cpu-q-246
6	cpu-q-239	cpu-q-246
7	cpu-q-240	cpu-q-246
8	cpu-q-240	cpu-q-246
9	cpu-q-240	cpu-q-246
10	cpu-q-240	cpu-q-246
11	cpu-q-240	cpu-q-246
12	cpu-q-240	cpu-q-246
13	cpu-q-240	cpu-q-246
14	cpu-q-240	cpu-q-246
15	cpu-q-240	cpu-q-246

Table 5: MPI Rank-to-Node Mapping Comparison between `reorder=0` and `reorder=1`

Valgrind is unsuitable for profiling MPI applications because it cannot effectively handle multiple parallel processes. Although I attempted to use other MPI profiling tools such as TAU and mpirun-based profiling, I was unable to complete this on CSD3 due to missing system libraries. Despite several attempts, the profiling runs consistently resulted in errors, preventing me from collecting meaningful performance data using these tools.

4.2 OpenMP

OpenMP is used for multithreading within a node, which unlike MPI can allow threads to share memory without explicit communication. I chose to use `static` scheduling, as all threads perform equal work (heat diffusion using neighboring cells), and dynamic scheduling would add unnecessary overhead. In this section, I will focus on tuning the number of threads and testing whether collapsing nested loops leads to better performance.

Figure 10 shows that for OpenMP parallelisation with 4, 8, 16, 32, and 76 threads, the uncollapsed version performs better with fewer threads, while the collapsed version achieves better performance as the thread count increases. Generally, 32 threads and uncollapsed loop lead to the best performance, leading to a 1.61 ± 0.03 times speed up.

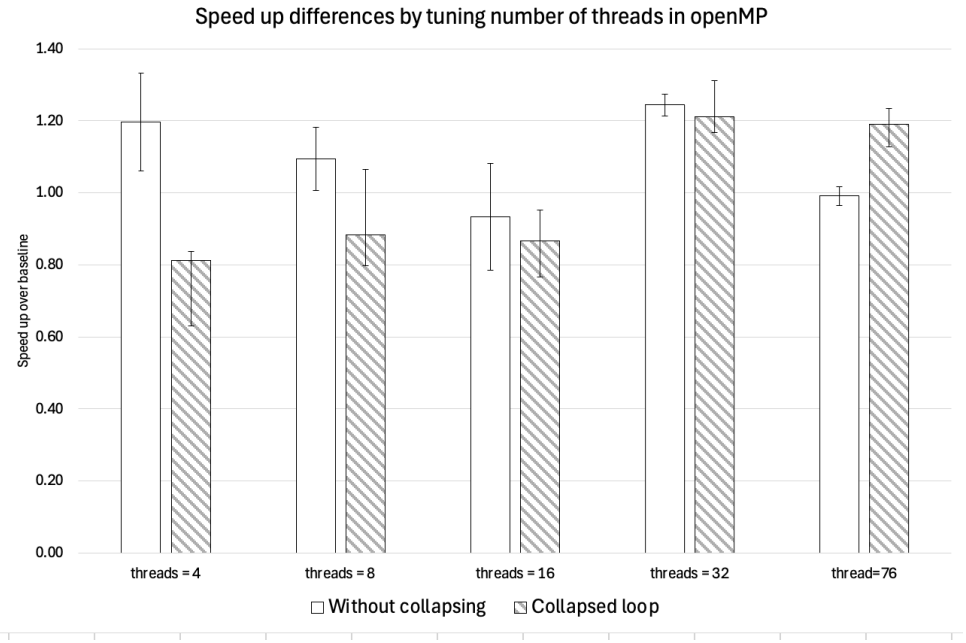

Figure 10: OpenMP performance with differing number of threads and collapsing nested looops

Figure 10 shows good performance for two combinations: 1) no collapsing with few threads or 2) collapsing with more threads. In the first case, OpenMP only parallelises the outer loop, distributing 100 rows across 4 threads. This is divisble by the number of loops, hence performs well. But with 76 threads, this becomes a problem: OpenMP can't assign 1.3 rows per thread (since $100 \div 76 \approx 1.3$, so it gives some threads 1 row (100 elements) and others 2 rows (200 elements). This causes a clear load imbalance where some threads finish early and sit idle while others continue working, possibly leading to poorer performance with more threads.

In the second case, the issue of load imbalance with more threads is solved when collapsing, as the 100 x 100 loop is flattened into 10,000 iterations, to be distributed across 76 threads. Now each thread gets approximately 132 elements ($10{,}000 \div 76 \approx 132$, which distributes the work much more evenly. Generally, `collapse(2)` reduces load imbalance and becomes more useful as number of threads increases.

To profile this, I compared Valgrind outputs for threads = 4 and threads = 76 for both collapsed and uncollapsed cases, as shown in Table 6. The differences are small and mostly insignificant, except for slightly lower L1 data read misses with 76 threads when uncollapsed and slightly lower write misses with 4 threads when uncollapsed.

Table 6: Valgrind Cache Performance Comparison (Collapsed vs. Uncollapsed)

Metric	**Collapsed (4)**	**Uncollapsed (4)**	**% Diff (4)**	**Collapsed (76)**	**Uncollapsed (76)**	**% Diff (76)**
Ir (Instr. read)	524,452,168	523,937,153	–0.10%	524,452,168	523,937,153	–0.10%
Dr (Data read)	148,768,842	150,507,108	+1.15%	148,768,842	150,507,108	+1.15%
Dw (Data write)	78,172,964	78,159,148	–0.02%	78,172,964	78,159,148	–0.02%
I1mr (Instr. L1 misses)	9,534	9,518	–0.17%	9,534	9,518	–0.17%
D1mr (Data L1 read misses)	123,665	122,978	–0.56%	306,946	303,761	–1.05%
D1mw (Data L1 write misses)	117,444	116,466	–0.84%	188,961	188,090	–0.46%
ILmr (Instr. LL misses)	2,991	2,990	–0.03%	2,991	2,990	–0.03%
DLmr (Data LL read misses)	9,242	9,241	–0.01%	9,242	9,241	–0.01%
DLmw (Data LL write misses)	7,154	7,155	+0.01%	7,154	7,155	+0.01%

I further profiled the code to compare the baseline with the best OpenMP run using 32 threads without loop collapsing, as shown in Table 7. Cache misses increased, which is likely due to disrupted memory locality at higher thread counts. However, the overall number of instructions, data reads, and writes decreased. This suggests that despite more frequent cache misses, the use of shared memory and reduced duplication of work across threads led to improved execution efficiency.

Table 7: Callgrind Metrics Comparison: Baseline vs. OpenMP (32 Threads, No Collapse)

Metric	**Baseline**	**OpenMP 32 (No Collapse)**	**% Difference**
Instructions (Ir)	577,928,983	546,618,149	–5.42%
Data Reads (Dr)	179,372,601	152,103,887	–15.20%
Data Writes (Dw)	110,262,922	77,919,217	–29.33%
L1 I-Cache Misses	8,048	9,699	+20.51%
L1 D-Cache Read Misses	169,486	217,377	+28.26%
L1 D-Cache Write Misses	132,502	151,959	+14.67%
LL I-Cache Misses	2,563	3,076	+20.02%
LL D-Cache Read Misses	9,087	9,253	+1.83%
LL D-Cache Write Misses	4,514	5,810	+28.72%

5 Finalised code

The final optimised implementation integrates MPI, OpenMP, and data flattening, which individually offered the most substantial runtime improvements. Performance is evaluated against the baseline for grid sizes 100, 1000, and 10000 using a configuration of 16 MPI processes across 2 Icelake nodes and 4 OpenMP threads (as there were node allocation issues with CSD3 when requesting for 32 threads) with uncollapsed loops.

Figure 11 shows raw average runtimes for the baseline and optimised code. Figure 12 presents the resulting speedups. The final 100×100 grid configuration achieved a speedup of 2.98 ± 0.42 compared to the baseline. While parallelisation and flattening improved performance for all grid sizes, the relative speedup decreased at larger scales.

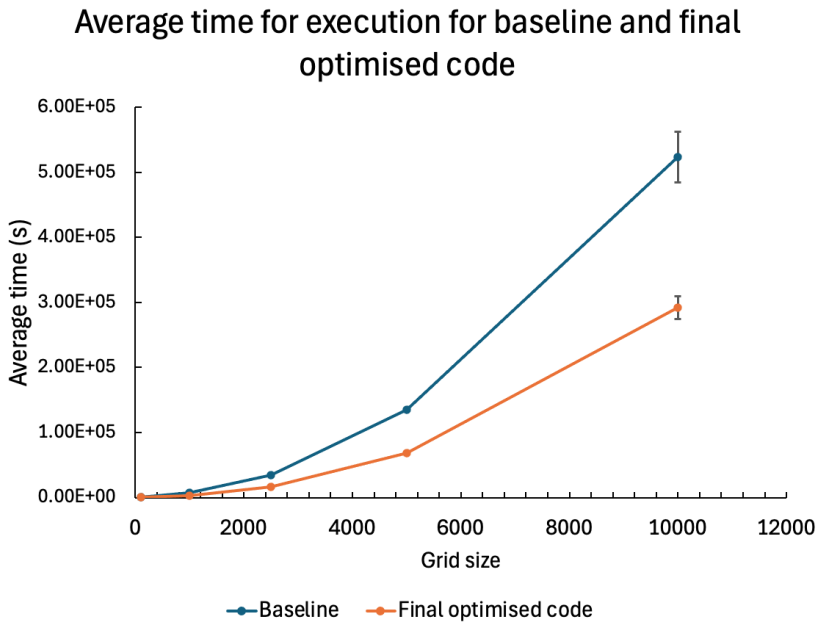

Figure 11: Average timed final code for increasing grid sizes

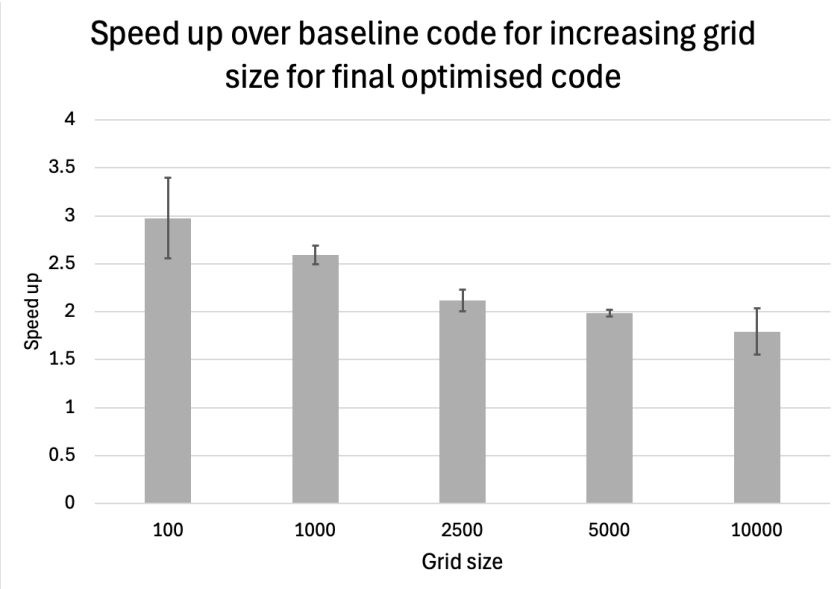

Figure 12: Average speedup of final code for increasing grid sizes

The reduced performance for larger grid sizes in my hybrid MPI+OpenMP implementation likely stems from resource underutilization. 16 MPI processes across 2 nodes with 4 threads per process implies 8 MPI processes per node $\times$ 4 threads = 32 threads were used per node, out of 76 available cores. Therefore, 44 cores per node remain idle. While this configuration works well for small problems where overhead dominates, as grid size increases, memory access patterns become a critical bottleneck that my current configuration is not suited for. Future work should focus on increasing OpenMP threads per MPI process and tuning cache blocking parameters specifically for larger problem sizes to better leverage the available hardware and memory bandwidth.

The speedup and error for each optimisation and parallelisation technique are summarised in Table 8. The highest speedup was achieved using MPI alone, without hybridising with OpenMP. This is likely because the MPI implementation fully utilises distributed memory across nodes and avoids the shared-memory overheads.

Table 8: Summary of speedups for each optimisation step (100×100 grid)

Technique	**Speedup**	**Error**
Flattened 1D vector	1.14	± 0.03
Cache blocking (block = 69)	1.03	± 0.04
OpenMP (32 threads, uncollapsed)	1.61	± 0.03
MPI (16 procs, 2 nodes)	3.21	± 0.21
Final (MPI + OMP + Flattened)	2.98	± 0.42

6 Conclusion

This project improved the performance of a 2D heat diffusion simulation by combining code optimisations with parallelisation using MPI and OpenMP. Flattening the grid helped reduce memory overhead, and cache blocking was explored for larger problem sizes. The final version, running with 16 MPI processes and 4 OpenMP threads across two nodes, achieved approximately 3$\times$ speedup. In the future, performance could be further improved by finetuning the hybrid MPI, and implementing cache blocking especially for larger grid sizes.

7 Appendix

Generation tools such as ChatGPT and Claude.ai are used to debug and refine the code associated to this report. Such tools were used to generate doxygen configuration, help develop the google test, debug MPI implementation, help formatting annotations in the code, and facilitate LaTeX formatting for figures and tables used in the report.