C2 coursework

March 2025

1 Introduction

Code optimization—especially system-specific optimization—is a niche area of programming often confined to specialized communities. As a result, large language models (LLMs) tend to lack sufficient training data in this domain and typically perform poorly when tasked with such optimizations. Therefore, it is crucial for computational researchers and developers to understand the fundamentals of optimization and the hardware architectures upon which these techniques are based.

This report focuses on optimizing a diffusion model initially generated by ChatGPT. The remainder of the report is organized as follows: first, we describe the testing environment, repository structure, and test suite. We then analyze the baseline model produced by the LLM. After that, we conduct single-process optimizations to improve performance. Once satisfactory improvements are achieved, we explore further acceleration through parallelization. Finally, we summarize our findings and discuss broader implications.

2 Testing environment and the repository organization

All tests are performed on cascade lake nodes (CPU nodes) provided within Cambridge Service for Data Driven Discovery (CSD3). For the single process optimization (i.e. without parallelization), single CPU coming with 3410 MiB memory within single node was used for the both baseline and tests. This size of memory is capable of handling operations of 1,000,000 double floats, and for larger problem size, memory swapping may slow down the process significantly. As the slowdown by memory swapping is not the focus of this project, the matrix size was limited to no larger than 10,000x10,000. For multi-process optimization, upto 2 nodes with 56 cpus each, and corresponding amount of memory are exploited. Icelake nodes were not used as they were more busy during the project, but the resource used stayed within the availability of two icelake nodes. For the simplicity of comparison, only two different sets of optimization options are used: -O0 and -O3. Also, other compiler options are unified to

”g++ -O$opt_{o}ptions - I < mpi_{i}nclude_{d}ir >< mpi_{c}ompile_{f}lags > -fopenmp - oBin/program_{n}ameSource/program_{n}ame.cpp < mpi_{l}ibs >< mpi_{l}ink_{f}lags >$ ” for the simplicity of the comparison. For single process performance breakdown for each of code components, additional profiling options for using valgrind are specified.

3 Baseline diffusion model and its profile breakdown

This section explains the implementation of diffusion model, and its execution time profile for the baseline model.

3.1 Implementation of Baseline diffusion model

The heat diffusion equation is given as $dT/dt = \alpha \cdot \nabla T$, where α is a thermal diffusibity constant and T is the temperature. It can be rewritten for a two-dimensional Cartesian cordinate, as

$$dT/dt = \alpha \cdot (d^2T/dx^2 + d^2T/dy^2).$$

For the numerical integration within the standard Cartesian gridding, this equation can then be discretized, using central differentiation, as

dT(x, y)/dt

$= \alpha \cdot \big((T(x+1,y) - T(x,y)) - (T(x,y) - T(x-1,y))$

$+ (T(x,y+1) - T(x,y)) - (T(x,y) - T(x,y-1))\big)$

$= \alpha \cdot \big(T(x+1,y) + T(x-1,y) + T(x,y+1) + T(x,y-1) - 4T(x,y)\big).$

This is implemented within the baseline model as:

```
temperature = std::vector<std::vector<double>>(height, std::
    vector<double>(width, 20.0));

for (int y = 1; y < height - 1; y++) {
    for (int x = 1; x < width - 1; x++) {
        // Discrete Laplacian operator
        double laplacian =
            temperature[y+1][x] + temperature[y-1][x] +
            temperature[y][x+1] + temperature[y][x-1] -
            4 * temperature[y][x];

        nextTemperature[y][x] = temperature[y][x] +
    diffusionRate * laplacian;
    }
}
```

It simulates the time evolution of the process with an initial value of homogeneous 20.0 degree celcius with a disturbance of anomalous 100.0 degree celcius placed upon the central 6x6 grid points. There are no division or multiplication between the variables, and each time steps will only produce the values averaged out from the initial values. This characteristics make the model less sensitive to the float point errors, making the calculation numerically stable. This implies the diffusion model will still give correct results with stable execusion, even after aggressive optimizations which may change the results of calculation.

The entire program for the baseline model breaks down into as follows.

1. Define initial temperature field as the 2 dimensional vector in vector array of the specified size.

2. Time-integrate the temperature field, using the diffusion equation implemented as described above.

3. Each 10 steps, output the current temperature field in text format.

4. Repeat 2 and 3 until time step reaches the user defined maximum.

The original size of the temperature field was 100x100, and maximum integration steps were 100. The O-scaling of this problem is linear such that $O(size_of_matrix)$ and $O(numberoftimesteps)$.

3.2 Execution profile of the baseline model

The profiling of execution time was performed using chrono. To measure the scalability, matrix sizes are varied from 100x100 to 10,000x10,000, incrementing the number of columns and rows in 10th fold. The profiling result for major components of the program is listed as Table 1 and the ratio to the total execution time as Table 2. For every tested matrix sizes, the largest bottleneck was found to be text output, taking up more than 80% of the total execution time for higher optimization option, indicating the large output streaming is causing the slowdown. Second largest bottle neck was diffusion loop, which is taking up the most of execution time left after the text output process. Higher optimization of -O3 resulted in a significant improvement of calculation for diffusion loop.

Opt Level	Matrix Size	Avg Diffusion Loop Time (s)	Avg Text Output Time (s)	Total Exec Time (s)
-O0	100	2.10E-03	0.14	1.73
-O0	1000	0.19	0.39	22.59
-O0	10000	17.81	36.62	2127.35
-O3	100	6.45E-05	0.09	0.98
-O3	1000	0.01	0.27	3.42
-O3	10000	0.67	27.34	341.51

Table 1: Execution time for the baseline model (seconds) for each of -O0 and -O3 Optimization levels and Matrix sizes ranging from 100 to 10,000 (both width and height). Average for 10 attempts are taken.

4 Single process optimization

Previous section identified the bottlenecks to be the main text output and the diffusion loops. Therefore, the optimization effort will be focused on those.

Opt Level	Matrix Size	Total Diffusion (%)	Total Binary Output (%)
-O0	100	12.17	82.67
-O0	1000	84.30	17.27
-O0	10000	83.72	17.21
-O3	100	0.66	94.16
-O3	1000	18.96	79.45
-O3	10000	19.68	80.07

Table 2: Ratio of execution time to total execution time in percentage. Each of columns are the same with Table 1, except diffusion loop and text output, where total ellapsed time was used instead of average for each loops.

4.1 Optimizing the file output

The in-efficiency of text output mainly comes from the type conversion to double to text, and the streaming of large amount of data to the output file. Therefore, a change of output format from text to binary was conducted. This change not just reduce the time consumed for type convesion, it will reduce the size of data to be streamed. As a note, a supplmental update to support visualization was adopted to animate.ipynb.

```
void saveFrame(int frameNumber) {
    system("mkdir -p output");
    std::string filename = "output/frame_" + std::to_string(
    frameNumber) + ".bin";
    std::ofstream outFile(filename, std::ios::binary);
    outFile.write(reinterpret_cast<const char*>(temperature.
    getPointer()), temperature.getSize() * sizeof(double));
    outFile.close();
}
```

4.2 Optimizing the diffusion loop

4.2.1 Change the 2d array from vector in vector to flat C-style array

The inefficient memory access caused by 2d array allocation using vector-in-vector was first corrected by using C++ flat array. This provided an improvement of nearly two-fold for larger matrix sizes. Additionally, double to float conversion was adopted. This is tested and found to be a safe change and does not cause the numerical insterbility.

4.2.2 Change the data type from double to single float

As was mentioned in subsection 3.1, the diffusion calculation is relatively numerically stable, and less likely to trigger fatal errors from float point errors. Therefore, the data type was changed to double to float for faster calculation.

4.2.3 Cache blocking

A cache blocking was also performed and tested. As cclake cpus have 32Kb size L1d cache, the block size was chosen to be 32x32 for double floats, 64x64 for single floats.

```
int block_size = 64;
for (int y = 1; y < height - 1; y+=block_size) {
    for (int x = 1; x < width - 1; x+=block_size) {
        int lim_y = std::min(y+block_size, height - 1);
        int lim_x = std::min(x+block_size, width - 1);
        for (int yy = y; yy < lim_y; yy++){
            for (int xx = x; xx < lim_x; xx++){
                float laplacian =
                    temperature(yy + 1, xx) + temperature(yy
    - 1, xx) +
                    temperature(yy, xx + 1) + temperature(yy
   , xx - 1) -
                    4.0 * temperature(yy, xx);

                nextTemperature(yy, xx) = temperature(yy, xx
   ) + diffusionRate * laplacian;
            }
        }
    }
}
```

4.3 Single process optimization result

Table 3 and 4 show the optimization results of two changes – conversion from text to binary output, and the vector-in-vector array to C-style flat array. The other single process optimizations (conversion from double to single float and cache blocking) will be discussed later in this subsection. Comparing Table 3 with Table 1, optimization has improved the performance drastically, such that the total execution time has shrunk less than the half for most matrix size and optimization levels. The larger improvement comes from the conversion from the text output to binary output, mostly because of the elimination of the step of type conversion from native double to text, and also because it reduced the data amount streamed to the output file. The improvement was more prominent for the larger matrix, as the impact of repetitive bottleneck process is larger for them. For the smallest file size of 100x100, the time consumed on other processes such as opening and closing file is dwarfing the impact of optimization. Table 4 indicates that for the matrix sizes of 1,000x1,000 and 10,000x10,000, diffusion loop is now taking up more than 80% of the execution time, making it the new largest bottle neck. For the smallest file size of 100x100, the output stays the largest bottle neck, again because of the file input-output time overhead.

Further optimizations of type conversion and cache blocking were attempted on the diffusion loop. Hereafter, as it became obvious that the main bottleneck is the diffusion loop, the profiling option was removed for faster and efficient computation. Figure 1 compares the execution time for either, both, or neither of the type-conversion and cache blocking optimization along with that of the original base line code. For the -O0 compiling option, the execution time increased with the matrix size, and between sizes of 1,000x1,000 to 10,000x10,000 the increase was linear to the matrix size. The best performance was observed with conversion to float for both of the optimization levels. Applying cache blocking was less effective than the type conversion, and somehow

Opt Level	Matrix Size	Avg Diffusion Loop Time (s)	Avg Binary Output Time (s)	Total Exec Time (s)
-O0	100	8.81E-04	0.04	0.57
-O0	1000	0.11	0.01	11.26
-O0	10000	10.90	0.71	1092.09
-O3	100	8.30E-05	0.18	1.86
-O3	1000	0.01	0.01	0.90
-O3	10000	0.86	0.80	97.08

Table 3: Same with Table 1 but for after single-process optimization

Opt Level	Matrix Size	Total Diffusion (%)	Total Binary Output (%)
-O0	100	15.55	75.37
-O0	1000	97.57	0.76
-O0	10000	99.83	0.65
-O3	100	0.45	96.00
-O3	1000	84.77	6.32
-O3	10000	88.40	8.19

Table 4: Same with Table 2 but for after single-process optimization

worked against when conbined with type conversion. This may be because the CPUs used are designed expecting more regular for loops, and using non-conventional looping made the execusion less effective. Therefore, further optimization using parallel processing was performed without cache blocking and with type conversion.

4.4 Multi process optimization result

Once the single process optimizations are performed, multiprocess optimizations using both MPI and OpenMP are attempted. Hereafter, the baseline of the comparison is the best performing single process program with type conversion. Parallelization was done for the main diffusion loop. MPI parallelization was preformed by assigning the row blocks to each of the rank to calculate. Each of those blocks were given buffer to calculate diffusion on the marginal region. OpenMP was used for threading the results.

```
        if (my_rank < num_procs - 1) {
            MPI_Sendrecv(
                local_temp.getPointer() + width *
    rows_per_proc,
                width, MPI_FLOAT, my_rank + 1, 0,
                local_temp.getPointer() + width * (
    rows_per_proc + 1),
                width, MPI_FLOAT, my_rank + 1, 1,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        if (my_rank > 0) {
            MPI_Sendrecv(
```

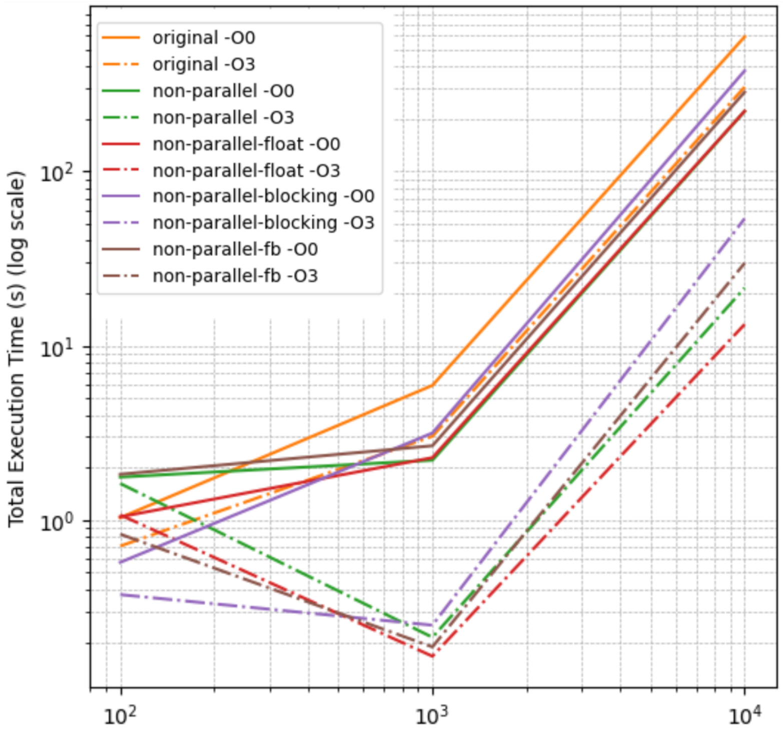


Figure 1: the plot of execution times for each of (red) type conversion from binary to single float, (purple) cache blocking, (brown) both of them combined, (green) neither of them applied but with other single-process optimizations, along with the (orange) baseline original code. Solid and broken lines show results for -O0 and -O3 optimization levels each. The vertical and horizontal axes denotes the total execution time and the matrix size (both number of rows and columns).

```
                local_temp.getPointer() + width,
                width, MPI_FLOAT, my_rank - 1, 1,
                local_temp.getPointer(),
                width, MPI_FLOAT, my_rank - 1, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }

        #pragma omp parallel for collapse(2) schedule(static
    )
        for (int y = 1; y <= rows_per_proc; ++y) {
            for (int x = 1; x < width - 1; ++x) {
                float laplacian =
                    local_temp(y + 1, x) + local_temp(y - 1,
     x) +
                    local_temp(y, x + 1) + local_temp(y, x -
     1) -
                    4.0f * local_temp(y, x);

                local_nextTemp(y, x) = local_temp(y, x) +
    diffusionRate * laplacian;
            }
        }

        local_temp.swap(local_nextTemp);

        MPI_Gather(
            local_temp.getPointer() + width,
            rows_per_proc * width, MPI_FLOAT,
            temperature.getPointer(),
            rows_per_proc * width, MPI_FLOAT,
            0, MPI_COMM_WORLD);
```

Figure 2 presents the optimization result for number of ranks ranging between 1 to 16, and each of -O0 and -O3 options for varying matrix sizes. For the original matrix size of 100x100, the change was almost negligible, and using no parallelization performed better. This is expected considering the overhead of execution time for mpi initialization, as well as exchanging data between CPU. For the matrix size of 10,000x10,000 and for -O0, the higher parallelization performed better with improvement with the number of the CPU cores used. For the -O3 option, the change was not as prominent.

Multi threading using OpenMP show more linear and clear improvement with the number of threads used (Figure 3). For the matrix size of 10,000x10,000, for both optimization options, the consistent decrease of execution time was observed for larger number of threads. Also, for the same number of parallelization, OpenMP performed generally better than MPI. This is because increasing number of thread does not introduce a communication overhead, unlike that for MPI parallelization.

Combination of MPI and OpenMP was also attempted (Figure 4). The result was consistent and regular with non-hybrid parallelization–such that for smaller matrix size, parallelization was not particulary efficient, and for larger matrix, parallelization lead to constant improvement with the number of process and threads. It was also ob-

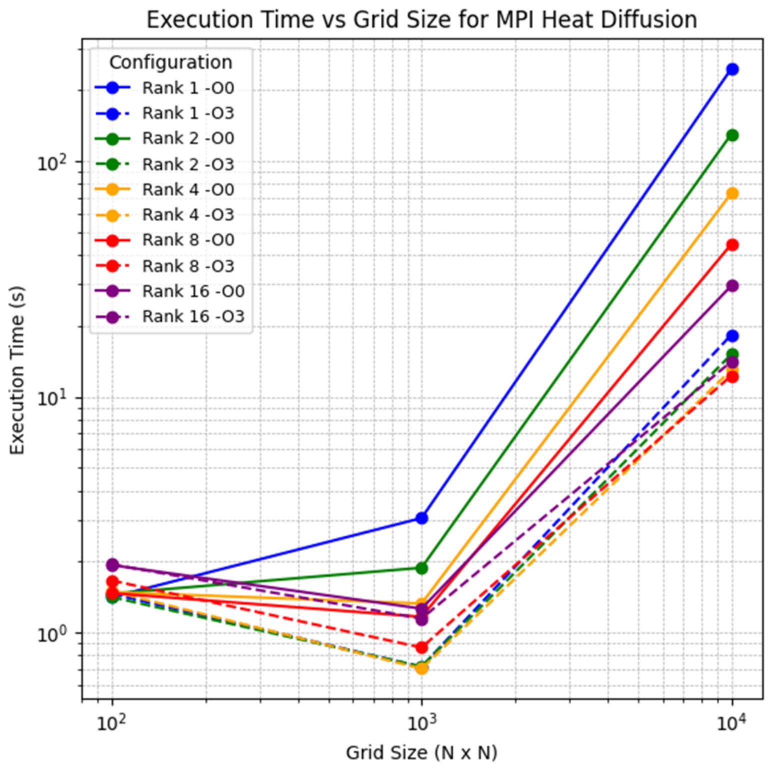


Figure 2: the plot of execution times for each of MPI parallelization levels ranging from 1 to 16. Solid and broken lines show results for -O0 and -O3 optimization levels each. The vertical and horizontal axes denotes the total execution time and the matrix size (both number of rows and columns).

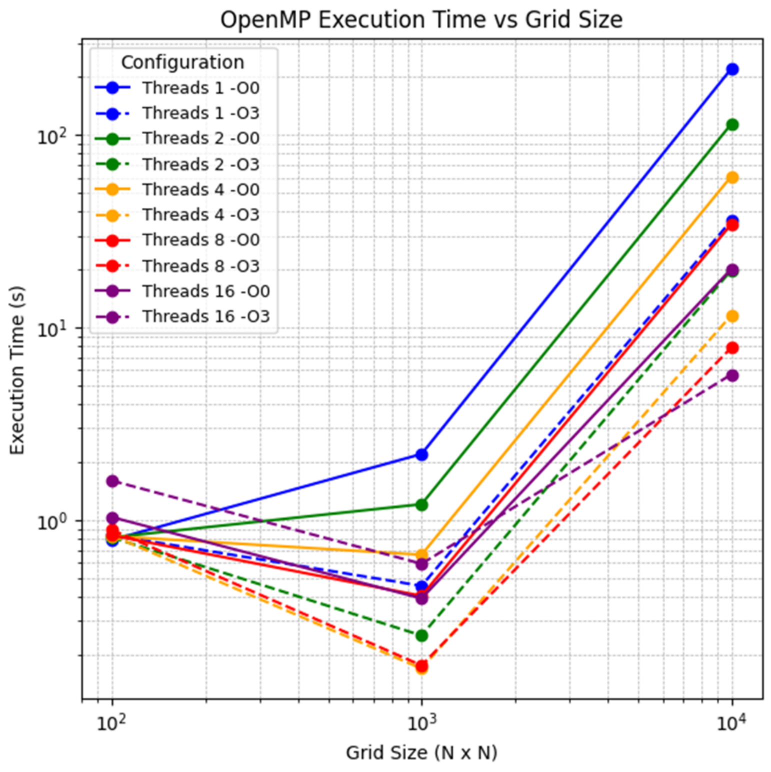


Figure 3: same with Figure 2 but for OpenMP thread parallelization.

served that the number of total threads was the dominant factor; number of process=1 with 8 threads performed as good as number of process=2 with 4 threads each.

5 Summary and discussion

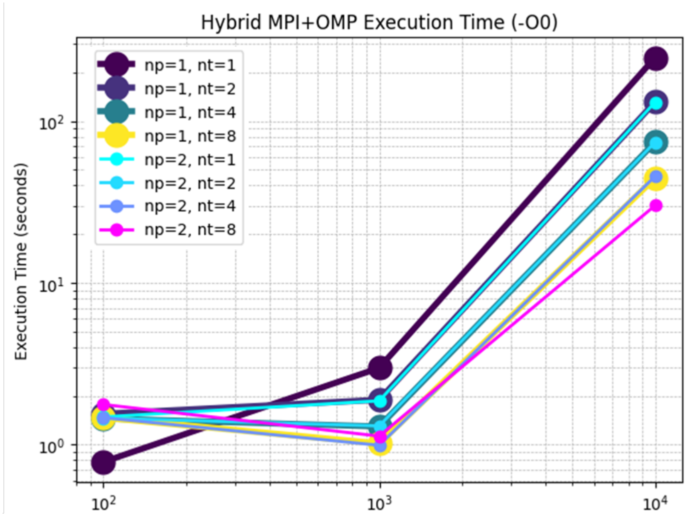
This report investigated code optimization techniques for a heat diffusion model initially generated by ChatGPT. While large language models (LLMs) can generate baseline implementations, their lack of exposure to system-specific optimization limits performance in high-performance computing contexts. To address this, profiling and optimization of the model was performed on CSD3, targeting both single-process and parallel implementations.

The initial profiling identified file I/O and the diffusion loop as major bottlenecks. Single-process optimizations, including binary output conversion, array restructuring, data type reduction from double to float, and cache blocking, resulted in substantial performance gains—cutting execution time by more than half for large matrix sizes.

Further improvements were achieved through parallelization. MPI and OpenMP were applied individually and in hybrid, demonstrating that OpenMP offered more consistent scaling due to lower communication overhead, especially on large grids. The best performance was archived with only using OpenMP with 2-16 threads, depending on the matrix size.

6 Acknowledgement

The author acknowledges the use of ChatGPT-4o and Google Colaboratory's AI-powered tools for assistance with language refinement and code optimization throughout this work. This project was conducted using computational resources provided by the Cambridge Service for Data Driven Discovery (CSD3) at the University of Cambridge. The author is supported by a government scholarship (details withheld for anonymity).



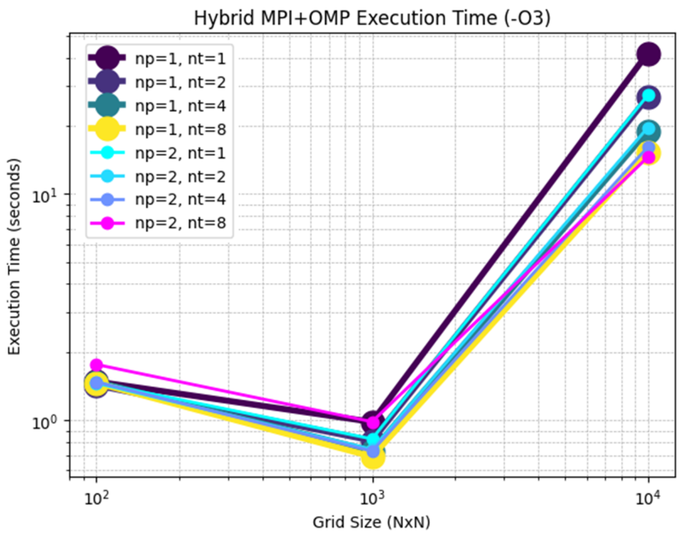


Figure 4: (Top) the plot of execution times for each of MPI and OpenMP parallelization levels ranging from 1 to 2, 1 to 8, respectively. Optimization level used was -O0. The vertical and horizontal axes denotes the total execution time and the matrix size (both number of rows and columns).) (Bottom) Same with top figure but for optimization level -O3.