1. Introduction

The purpose of this report is to detail the development of a Jacobi relaxation algorithm that makes efficient use of the CPU cache and cores. The algorithm was implemented in C++. Jacobi relaxation is used here to solve Poisson's equation, which gives the electric potential from a given charge distribution. The next section of this report describes the architecture of the computer used. Sections 3, 4, 5, and 6 analyse the algorithms performance and describe methods used to increase speed. Timing information is found in Section 7. The report is concluded in Section 8.

2. Computer Architecture

The program was written and tested on the CAE lab computers, which have Intel Core i7-7700k Processors. The CPU has 4 cores, 64 GB RAM, and an 8 MB Intel Smart Cache. This is a three level cache; the first level of cache is 32 kB, the second level is 262 kB, and the third level is 8 MB. The operating system is Linux Mint 19, version 4.15.0-20.

3. Cache Analysis

Table 1. Cache statistics for ours and the naive Jacobi relaxation implementations. All tests were done with ten iterations and a single point charge at the centre of the volume.

Our Implementation								
Size	**Dr**	**D1mr**	**DLmr**	**Difference**	**Dw**	**D1mw**	**DLmw**	**Difference**
101	103,238,600	4.97%	2.42%	-23,444,595	10,711,650	12.0%	11.3%	-9,054,073
201	812,876,600	4.98%	2.54%	-342,368,374	82,822,650	12.3%	12.2%	-90,519,685
301	2,728,914,600	4.99%	2.53%	-1,567,027,247	276,333,650	12.3%	12.3%	-477,236,224
401	6,451,352,600	4.99%	2.62%	-3,699,385,772	651,244,650	12.4%	12.4%	-1,128,428,812
501	11,317,655,530	5.55%	5.55%	-6,916,369,258	1,267,555,650	12.4%	12.4%	-2,200,641,306
Naive Implementation								
Size	**Dr**	**D1mr**	**DLmr**	**Difference**	**Dw**	**D1mw**	**DLmw**	**Difference**
101	71,922,180	39.6%	3.62%	+23,444,595	10,406,047	99.0%	12.1%	+9,054,073
201	567,644,180	60.8%	10.3%	+342,368,374	81,612,047	99.5%	36.3%	+90,519,685
301	1,907,166,180	60.8%	32.2%	+1,567,027,247	273,618,047	99.7%	99.7%	+477,236,224
401	4,510,488,180	60.7%	32.2%	+3,699,385,772	646,424,047	99.8%	99.7%	+1,128,428,812
501	8,797,610,180	60.7%	32.2%	+6,916,369,258	1,260,030,047	99.8%	99.8%	+2,200,641,306

Table 1 gives cache statistics for ours and the naive Jacobi relaxation implementations. Ir is the number of instructions executed, Dr is the number of memory reads, and Dw is the number of memory writes. A 1 indicates the level-1 cache, an L indicates the last-level cache, and an m indicates a cache miss. Each test was run with a single point charge in the centre of the matrix for ten iterations.

Our program takes advantage of the cache by using a linear memory access for the flattened 3-D array *input*. This means the core for loop of the algorithm accesses array elements from 0, 1, 2, … , NxNxN - 1, whereas the naive program follows a non-linear sequence from index 0 to NxNxN - 1. Figure 1 shows how the for loop was changed to speed up memory access. The linear memory access is faster because it utilises the cache better (i.e. has fewer cache misses). The Intel i7-7700k used for this assignment has an SRAM cache and DRAM main memory. SRAM memory is faster than DRAM because it does not require a capacitor to be charged, like DRAM does, and does not need to be refreshed.

```
for (unsigned int x = 0; x < xsize; x++) {
    for (unsigned int z = 0; z < zsize; z++) {
        for (unsigned int y = 0; y < ysize; y++) {

for (z = start_index; z <  zsize; z+=increment) {
  for (y = 0; y < ysize; y++) {
    for (x = 0; x < xsize; x++) {
```

Figure 1: For loops with non-linear memory access (top) and linear memory access (bottom).

4. Multithreading Analysis

Threading reduces program execution times by allowing multiple threads of execution simultaneously. The key advantages of multithreading are that it reduces data dependency stalls in the execution pipeline and it allows other threads to run while a thread is waiting for a structural hazard. The processor mentioned in this assignment uses hyper-threading, which is Intel's proprietary simultaneous multithreading implementation.

The disadvantages of multiple threads is the coupling they have with hardware resources such as caches. Our implementation processes different array chunks separately in the threads. The effects of increasing the number of threads is shown in Fig. 2. There is no benefit to increasing the number of threads above four in our case because the CPU resources are fully utilized there.

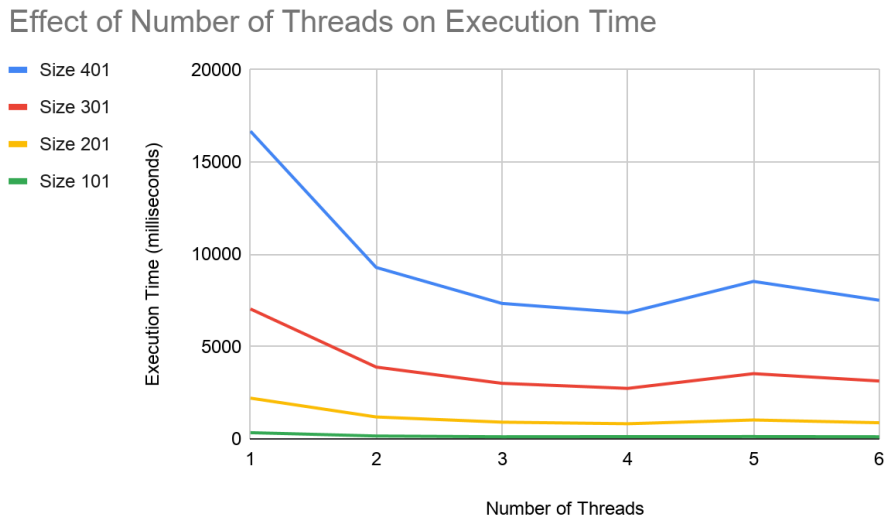

Figure 2: Effect of number of threads on execution time.

The initial implementation of threading was done by generating start and end indexes for each thread. The start and end indexes were used in the outermost loop allowing data that is close in proximity to be accessed first, therefore making better use of the cache. This method was effective, however, due to the cache being shared by multiple threads, a different indexing method was found to be more effective. With the new alternating indexing method, the threads operate on blocks of memory adjacent to each other. Since the threads run at approximately the same speed, the cached memory is spatially localised, and the same cache could be used by multiple threads. The old indexing method resulted in the cache being frequently overwritten. Figure 3 describes our indexing method visually for an array of size 16. Table 2 gives performance results for the new indexing method.

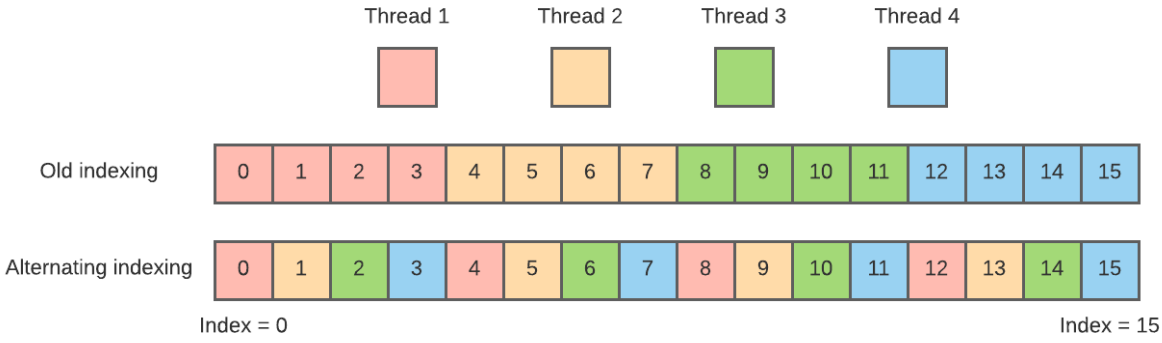

Figure 3: Indexing methods used to maximize cache performance.

Table 2. Comparison of the old indexing method (block indexing) vs alternating indexing.

Block Size	301	501	701	901
Block Indexing Average Time (s)	3.6	18.3	50.2	105.7
Alternating Indexing Average Time (s)	2.8	14.5	41.0	94.8
Improvement	+22.2%	+20.8%	+22.4%	+11.5%

5. Profiling Analysis

Profiling the program using gprof indicates *poisson_thread_function* accounts for almost all of the program execution time; this is the function that executes the Jacobi relaxation algorithm. This result is obvious, so the benefits of profiling are limited in this assignment. Further program optimisations should focus on the inner for loop of this function. One improvement that could be made is removing the if and else clauses from the inner loop and handling the boundary conditions elsewhere; this is called loop unswitching. A test implementation of this found the performance improvement was small (about 3%) for $N \geq 301$, so we did not pursue this implementation further. It is possible the compiler already unswitched the loops, because our fastest implementation compiles with the -O3 flag, which enables -funswitch-loops.

The naive program used memcpy, which was slow, so it was replaced with pointer swapping. This gave a 3.8% improvement in execution time. Profiling also showed that the: *potential[((z * ysize) + y) * xsize + x] = res;* operation was the slowest part of the inner loop. However, no alternative was identified that could replace this operation.

6. Compiler Optimisation Analysis

Table 3. Compiler optimisation statistics for the Jacobi relaxation program. All tests done with matrix dimensions of 401 x 401 x 401 for twenty iterations and four threads. Percentage changes are relative to the no compiler optimisation case.

Compiler Optimisations	**Time to Execute (seconds)**	**Time to Execute Change**
None	5.669	-
-Os	2.050	-63.8%
-O1	2.013	-64.5%
-O2	2.007	-64.6%
-O3	2.006	-64.6%
-funroll-loops	5.685	+0.282%

-ffast-math	5.647	-0.388%
-O3 -funroll-loops -ffast-math	1.993	-64.8%

In terms of speed, the most effective compiler optimisation that can be used is -O3, though -O1, -O2, and -Os show very similar results. These flags enable compiler optimisations, with -O1 enabling the fewest and -O3 enabling the most. The program size may be increased if -O3 is enabled.

Although for loops make up almost all of the program execution time, invoking -funroll-loops does not speed it up. There are a few reasons for this:

- Loop unrolling has diminishing gains as the number of loop iterations increases, since much of the gain comes from avoiding flushing the pipeline. Flushing occurs when the processor incorrectly assumes which was a branch instruction will go, which happens less frequently in large loops.
- Unrolling a loop results in more I-cache being needed to store the instructions. For large loops, the instructions may not fit in a single cache line, so the program will stall as instructions are read from main memory.
- Loop unrolling may not increase speed if the compiler can't predict how many iterations are done at compile time. For our program, the loop sizes are dependent on arguments passed at run-time.

7. Timing Results

Table 3. Timing results for the final program. The program was run 5 times at each cube size to get accurate results.

Cube Size	101	201	301	401	501	601	701	801	901
Average Time (s)	0.01	0.85	2.83	6.84	13.72	24.44	40.33	57.97	87.10
Std Dev (s)	0.010	0.0020	0.021	0.021	0.28	0.62	2.1	1.2	1.7

8. Conclusion

A Jacobi relaxation algorithm for solving Poisson's equations was optimised using the cache, multithreading, profiling, and compiler optimisation options. Compared to the naive implementation, the largest performance gains came from changing how the for loops index the array and implementing threading. Four threads is optimal for the i7-7700k processor used. The -O1, -O2, -O3, and -Os compiler optimisations more than double the program speed, though -O3 was already used in the naive implementation. Timing results have been summarized in Table 3.