CM30225 Parallel Computing
Coursework 1: Shared Memory Architecture

November 20, 2016

Contents

1 Introduction and Sequential Approach

Shared memory refers to multiprocessors accessing main memory on a shared bus. Memory access is fundamentally slow. Multiple processors accessing shared memory simultaneously across a system bus results in increased latency for memory access. In a bid to reduce traffic across the system bus, varying levels and designs of hierarchical cache are introduced. Relative to main memory access, L1 and L2 cache hits are extremely fast - approximately 0.5ns for L1 comparative to 100ns for a main memory reference [Dean, 2007] - meaning latency is greatly reduced. However, this presents the *cache coherency* problem, where processors operate on local variables within their own cache, leading to inconsistencies between shared data. Various approaches to the cache coherency problem have been proposed and

evaluated [Agarwal et al., 1988]. This includes the snoopy cache protocol, a broadcast-like mechanism for coherency across the bus, and a scalable directory-based scheme protocol [Lenoski et al., 1990]. One of the primary objectives of these cache coherence protocols is to prevent *race conditions*. In the context of this project we are therefore trying to avoid what C defines as *undefined behaviour*, where operations happening in varying orders produces differing results.

Related, but not equivalent to a race condition, is the term *data race*. A data race is any unsynchronised, concurrent access to data involving a write [Bradford, 2016]. Given the relaxation approach to be covered in this project, it is crucial that data is kept consistent between averaging adjacent cells and checking the precision of the written results. With an $n * n$ matrix, $(n-2)^2$ cells will be relaxed and new cell values computed. With a cell $c_{i,j}$, its adjacent neighbours $c_{i+1,j}$, $c_{i-1,j}$, $c_{i,j+1}$, $c_{i,j+1}$ will be read in the computation of the averaged value. Consequently it is clear that synchronisation issues may arise should one of these cells contain the value of an already complete relax operation. I therefore introduced an additional matrix, in which the relaxation function writes its output. Importantly, this avoids potential data races by only writing to the second matrix once for every cell in the first matrix and during this period no read-access occurs. Equivalent cells $c_{i,j}$ across both matrices are checked for precision, but this is purely read-access on both matrices - no write-access occurs either. In both cases the scenario for a data race as defined above is avoided. Nevertheless this solution does bring added considerations, primarily the additional space complexity involved with a second matrix. Specifically this is $\mathcal{O}(2n) = \mathcal{O}(n)$ - meaning that the input matrix dimension linearly scales with memory dynamically allocated on the heap. However, I believe that this additional space complexity is balanced with a reduction in time complexity (and indeed code complexity) of viable secondary solutions. Therefore my sequential approach to this solution is as follows in psuedocode:

```
DETERMINE environment variables (e.g array size, precision)
INIT matrix with random numbers
WHILE outside precision threshold
        relax array
        set new precision
        swap matrices
END WHILE
```

Swapping matrices in the C implementation of this psuedocode is an inexpensive operation. Rather than swapping the contents of each matrix for the next loop, pointers to each matrix are swapped after a full relaxation instead. This sequential approach, including my early thinking around avoiding data races, formed the basis of my approach towards the parallelisation task.

2 Approach to Parallelisation

2.1 Allocation of Work

A major task early on was deciding on the fairest way to split the work for different threads to compute. A square matrix with dimensions $n \cdot n$ has $(n-2)^2$ cells which require splitting between t threads. Clearly if $(n-2)^2 \bmod t = 0$, then the work can be split equally and each thread will receive the same number of cells to work on. However, if $(n-2)^2 \bmod t > 0$ then $(n-2)^2 \bmod t$ cells will need to be allocated elsewhere. Initially I allocated the remaining number of cells to a random thread. Quickly I realised that this wasn't the optimal solution. Given that the order of execution for threads is arbitrary, a thread with the additional cells to work on could be scheduled last, preventing the program from progressing while the thread works through its cells. A better solution was to distribute the

remaining cells equally over the existing threads, giving each thread an expected similar computation time.

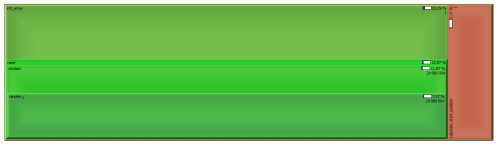


Figure 1: main() cycle estimation before

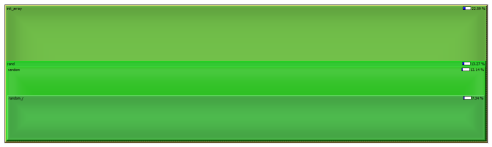


Figure 2: main() cycle estimation after

The next challenge was to determine each thread's starting position in the matrix. My first solution started at the first inner cell and one-by-one went to the next non-edge cell in the matrix, decrementing the thread's number of cells to work on until the value reached 0. At this point the next thread's starting position is known. Although decrementing variables and checking for non-edge cells is a relatively inexpensive operation, this process does not scale well as the dimension of the matrix increases. Profiling my program with Valgrind and Callgrind showed a small but notable amount of CPU cycles expensed in the *calculate_start_position* function. This is visually represented by KCachegrind[1] displaying a red rectangle as a proportion of *main()*'s cycle estimation in Figure 1. The green proportion of *main()* was taken up by standard C libraries such as *rand()* used to populate the array - not related with relaxation logic.

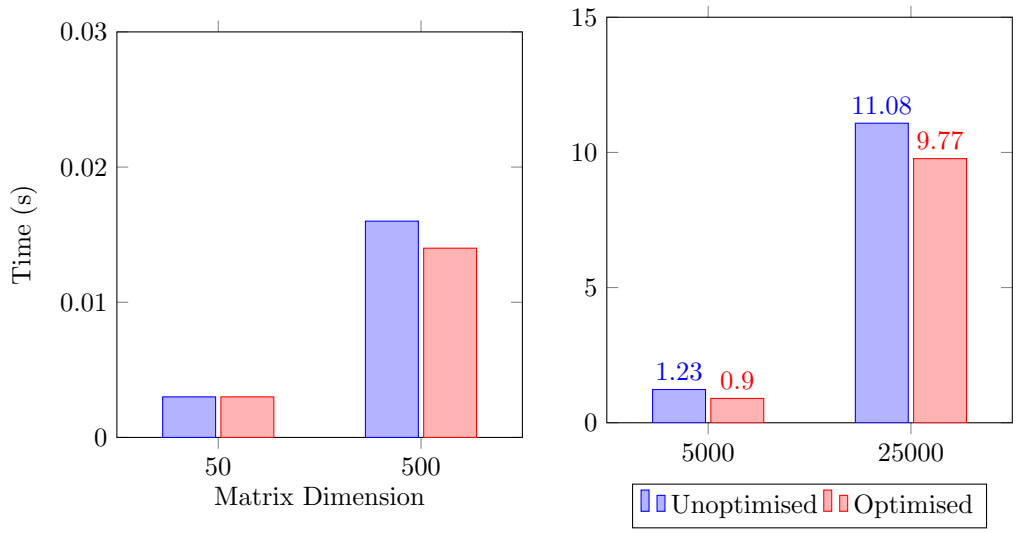


Figure 3: Averaged results of 25 runs (Fixed Precision: 0.1, Fixed Thread Number: 10, Matrix Dimensions: Varied as above)

The solution for calculating the thread's starting position more efficiently was to calculate row and column positions from an index using division and modular arithmetic. Most significantly, this removed the need for the repeated decrement operation and non-edge cell check, reducing estimated CPU cycles for *main()* as shown in Figure 2. A small test program with a fixed number of threads was run on *linux.bath.ac.uk* with the program running up until the starting positions had been allocated. The results of this test are displayed

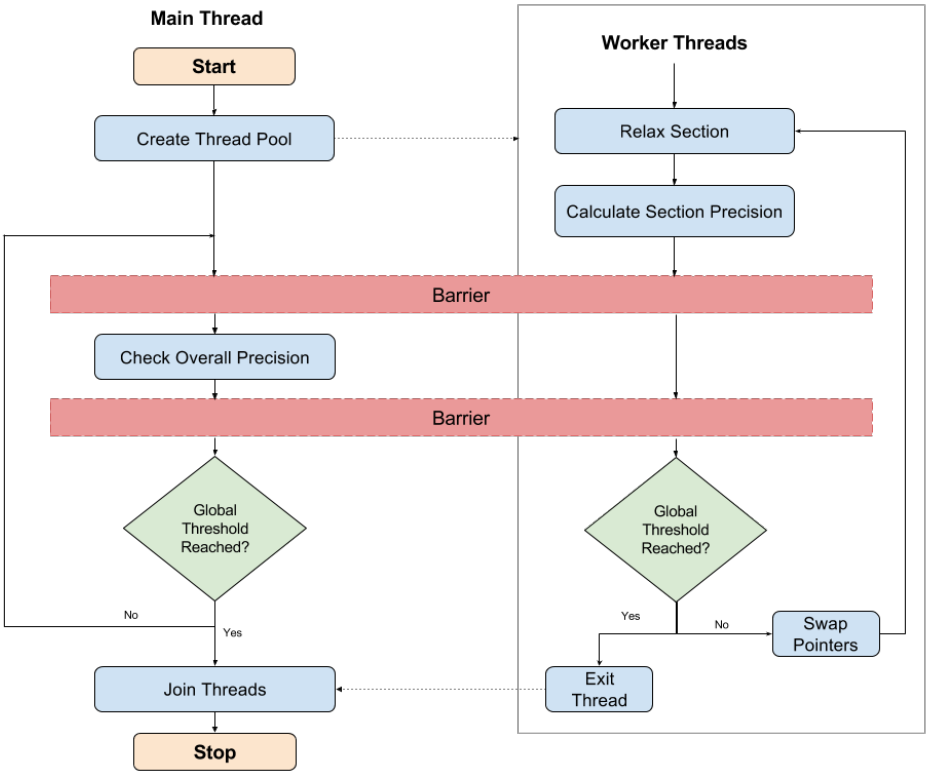
[1] https://kcachegrind.github.io/html/Home.html

in Figure 3. As expected, the overhead between unoptimised and optimised solutions was negligible on smaller matrix dimensions, but an improvement in performance emerged as the matrix scaled upwards.

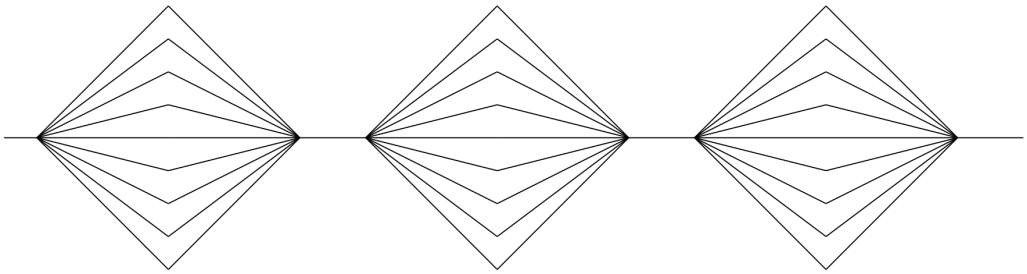
2.2 Thread Work

The nature of this parallel task is that threads will work on a small section of a matrix, before hitting some sort of synchronisation mechanism. A precision check will then be completed and depending on the outcome, the thread's work will be repeated.

Creating and destroying threads consumes CPU instructions, particularly on certain operating systems. With a small input precision (resulting in an increased number of relaxations), if threads aren't managed carefully the program could potentially be creating and destroying thousands of threads over the lifespan of the program. To reduce the overhead of potentially expensive thread creation and destruction, I utilised the simple *thread pool* design pattern to reuse a considerably smaller set of threads during the program.



A flow chart representing the structure of my program is included above. Worker threads are initialised with a starting position and a number of cells to work on, before hitting two barriers. Between barriers, the main thread checks the overall precision of the matrix and sets a global variable, and then all threads proceed past the barrier. If a global threshold hasn't been reached, worker threads swap their pointers to both matrices and this process loops around. If the threshold has been reached, the main thread blocks execution until all worker threads have terminated successfully.



The inclusion of barriers form a fundamental characteristic of the overall program structure and can best be described through the superstep diagram above. Minimising the sequential section between the synchronisation of multiple threads starting and finishing is vital for achieving the benefits of parallelisation. Rather than comparing $(n-2)^2$ cells to their previous values on the main thread, each worker thread computes the precision for their section only and reports this to a global array. Between barriers, the main thread has very little work to do. The main thread simply checks an char array of size t (equal to the number of threads) which determines whether the global threshold has been reached. This avoids the main thread holding up the program for longer than necessary and maximises the use of multiple threads.

Alongside a barrier object, there are two global variables used in this program to accomplish parallelisation. The first is *precision*, a char array which each thread writes a 1 or a 0 to, depending on if all cells in their section are within the precision threshold. Given that multiple threads are accessing this shared resource, it might be expected that a lock is used to prevent concurrent access. However, each thread only writes a single byte to an array index associated with their thread number - e.g no two threads will modify the same array element/memory location concurrently. Therefore the use of a lock can be avoided (although this potentially creates a scalability problem not explored in this project in cacheline bouncing through false sharing [Bolosky and Scott, 1993]). The first barrier then synchronises threads, ensuring that the the *precision* array contains up-to-date precision values from each thread. The outcome of the check across all values in the *precision* array is reflected into the second global variable *outside_threshold*. This variable is updated before the second barrier is reached, guaranteeing that all threads will read the latest value of this variable on proceeding. *outside_threshold* is used to determine whether worker threads should perform another relaxation, or simply terminate because the final precision for all threads has been reached.

3 Correctness Testing

3.1 Manual Validation

The relaxation of a matrix has been completed by hand as transcribed below. (Note: The averaging operation of adjacent cells has not been detailed.) A simple $5 \cdot 5$ matrix is used for illustration purposes and we take three full iterations to arrive at the final matrix. The precision has been set to 0.15 and 3 threads are used.
Threads t_1, t_2, t_3 will be allocated equally to 9 inner cells as follows.

- t_1: $c_{1,1}$, $c_{1,2}$, $c_{1,3}$
- t_2: $c_{2,1}$, $c_{2,2}$, $c_{2,3}$
- t_3: $c_{3,1}$, $c_{3,2}$, $c_{3,3}$

Starting

1	1	1	1	1
1	0.840188	0.394383	0.783099	1
1	0.798440	0.911647	0.197551	1
1	0.335223	0.768230	0.277775	1
1	1	1	1	1

The matrix has been randomly seeded with doubles from 0 to 1.

First Iteration

1	1	1	1	1
1	0.798206	0.883734	0.647984	1
1	0.771764	0.539651	0.743130	1
1	0.891667	0.631161	0.741445	1
1	1	1	1	1

Each thread's section has been relaxed. A precision value is generated for each thread's section - 1 for all cells in precision, otherwise 0. Cells outside the precision threshold for each thread are listed.
t_1: **0** - $c_{1,2}$
t_2: **0** - $c_{2,2}$, $c_{2,3}$
t_3: **0** - $c_{3,1}$, $c_{3,3}$

Second Iteration

1	1	1	1	1
1	0.913875	0.746460	0.906716	1
1	0.807381	0.757447	0.732270	1
1	0.850731	0.793191	0.843573	1
1	1	1	1	1

The matrix has been relaxed again and the following precision values for each thread's section are:
t_1: **0** - $c_{1,3}$
t_2: **0** - $c_{2,2}$
t_3: **0** - $c_{3,2}$

Third Iteration

1	1	1	1	1
1	0.888460	0.894509	0.869683	1
1	0.880513	0.769826	0.876934	1
1	0.900143	0.862938	0.881365	1
1	1	1	1	1

The matrix has been relaxed again and the following precision values for each thread's section are:
t_1: **1**
t_2: **1**
t_3: **1**
The precision has been reached for each section, and therefore for each cell, so the program terminates.

The results of relaxing this matrix by hand were compared with the results of the parallel program. With the same input matrix and configuration, the output matrix and number of iterations taken to reach the final solution match between program and manual version.

```
Matrix B
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.888460 0.894509 0.869683 1.000000
1.000000 0.880513 0.769825 0.876934 1.000000
1.000000 0.900143 0.862938 0.881365 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
---------------------------------------------
Completed in 3 iterations
Matrix: 5x5. Threads 3. Precision: 0.150000
```

3.2 Further Test Cases

A small test framework was set up to collect the output of repeated tests across varying program configurations. As matrix values are randomly generated, I ensured that the same seed was used in all runs of the program for consistent results. An example output matrix M_1 is included in the appendix.

Size	**Precision**	**Threads**	**Expected Output**	**Number of Runs**	**Pass**
10	0.1	1	M_1	20	Yes
10	0.1	2	M_1	20	Yes
10	0.1	5	M_1	20	Yes
25	0.1	1	M_2	20	Yes
25	0.1	2	M_2	20	Yes
25	0.1	5	M_2	20	Yes
25	0.1	10	M_2	20	Yes
25	0.1	15	M_2	20	Yes
100	0.1	1	M_3	20	Yes
100	0.1	2	M_3	20	Yes
100	0.1	5	M_3	20	Yes
100	0.1	10	M_3	20	Yes
100	0.1	15	M_3	20	Yes
500	0.1	1	M_4	20	Yes
500	0.1	2	M_4	20	Yes
500	0.1	5	M_4	20	Yes
500	0.1	10	M_4	20	Yes
500	0.1	15	M_4	20	Yes

While these test cases certainly don't prove program correctness, the repeated outcomes suggest that the program is reliable at producing consistent outputs, regardless of how many threads used or the number of runs completed. The results from the parallel program were also compared to those from the sequential program. The results matched, using the same random seed and configuration in terms of matrix size and precision. Of course this in itself isn't an indicator of correctness - both parallel and sequential outputs could be incorrect. However, coupled with the manual relaxation testing by hand, I was comfortable that the program performed to a degree of acceptability suitable for the scope of this project.

4 Scalability

The parallel program was tested using Balena, with a variety of different program configurations. Thread numbers, precision values and dimension sizes were varied in order to target the effects on speedup and efficiency. The time taken for tasks to complete was measured using the unix command *time*, and the resulting *real* output is used as the measure of time.

Gathering measurements for speedup (and hence efficiency) requires an evaluation of the best possible sequential algorithm. As my program evolved between sequential and parallel versions, I felt that the two versions had diverged such that it would be best if sequential measurements were recorded on the parallel version with a single thread. This of course brings its own issues and arguably this isn't a true comparison due to the small but relevant cost of creating and managing a single pthread. While main and worker threads threads don't complete any substantial work while the other is active, both threads will still be running simultaneously. It is worth reflecting on the restrictions this may have had in the context of the results. However, I felt that this limitation was preferable in comparison to alternative solutions.

The first test case takes a fixed problem size and successively adds more threads to view the outcome in terms of time reduction. The configuration used for this test is a matrix of size $1000x1000$, a precision of 0.1 and the number of threads 1-16.

4.1 Time for 1-16 threads, $S = 1000$, P=0.1

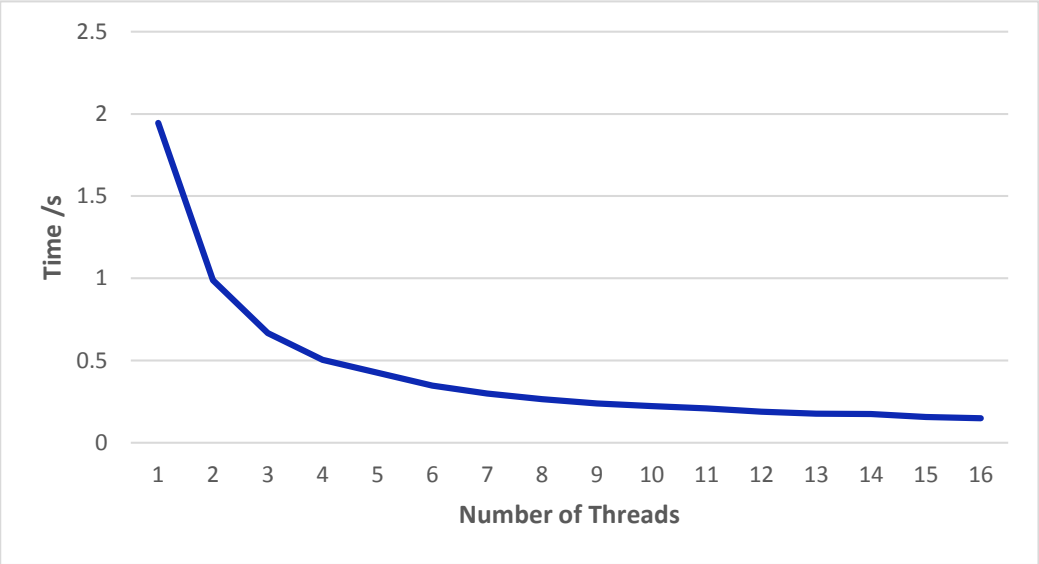


Figure 4: Time taken for $S = 1000, P = 0.1$ and $T = 1 - 16$

Each additional thread added leads to successively smaller returns in time reduction - e.g using 10 threads does not result in a 10-fold time reduction (the law of diminishing returns). This is consistent with Amdahl's Law as explored in Figure 6. Relative speedup between 8 threads and 16 threads is largely consistent with a scaling problem size, as evidenced in the Appendix. For instance, 16 threads will reach the final solution faster than 8 threads will as the problem size grows between matrix sizes.

4.2 Time for 8-32 threads, $S = 1000$, $P = 0.1$

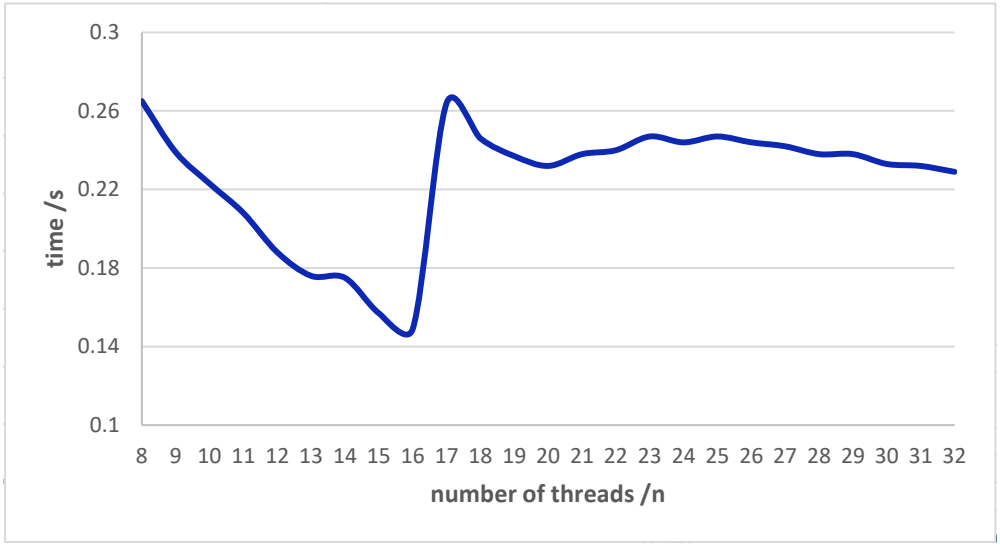


Figure 5: Time taken for $S = 1000, P = 0.1$ and $T = 8 - 32$

Each node on Balena has 16 cores which support 16 threads overall. Specifying an additional node enables more cores to take on extra threads. However, the overhead of sharing the program between nodes is clearly present in Figure 5. As soon as the program is shared between nodes, there is a spike in time taken to relax the matrix in full. The cost of communication between nodes and threads idling becomes a limiting factor, and this will be reflected in efficiency measures for more than 16 threads.

4.3 Speedup/Efficiency for 1-16 threads, $S = 1000$, $P = 0.1$

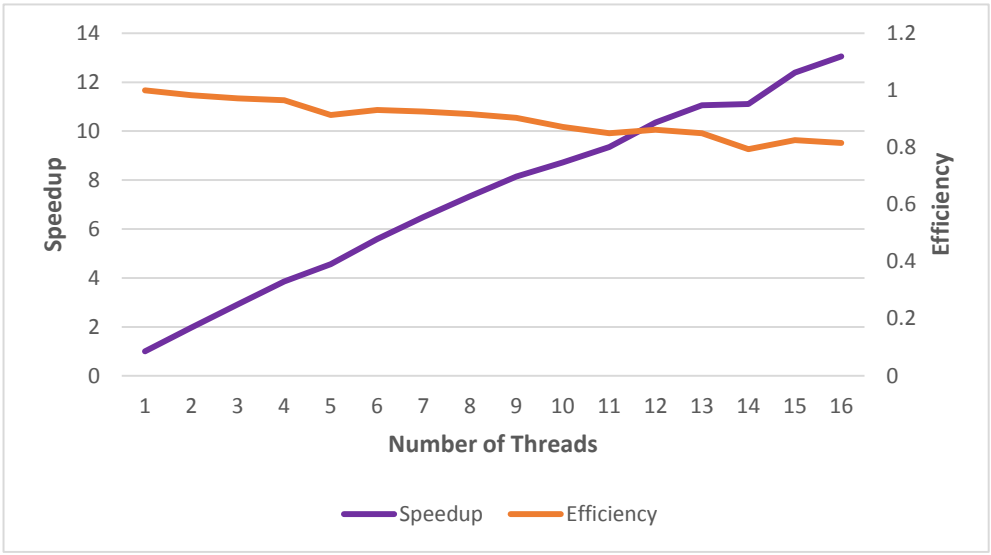


Figure 6: Speedup and Efficiency for $S = 1000, P = 0.1$ and $T = 1 - 16$

The efficiency for this fixed problem size is close to 1, meaning that the speedup achieved for this test is sub-linear but ultimately close to linear. This means a respectable level of scalability up until 16 threads, where overheads begin to dominate. Using the maximum speedup for this fixed problem size of 13.05, the implied sequential share of my program is $1/13.05 \approx 0.076$ - roughly 7.6% of this particular fixed computation problem. This prevents further parallelisation and is both the defining and limiting factor of speedup. The overhead of the sequential share describes this upper limit on parallelisation, and is therefore consistent with Amdahl's Law.

4.4 Precision against time for $S = 1000$, T=[1,10]

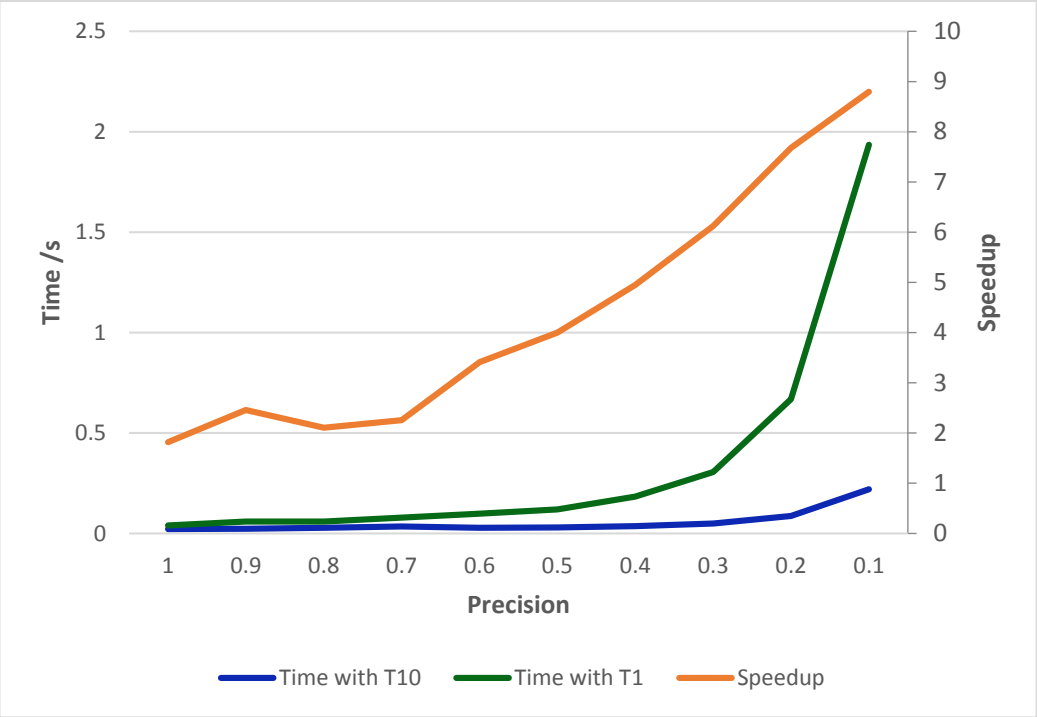


Figure 7: Time taken for $S = 1000, T = 1, 10$ and varying P values

In line with Gustafson's Law, increasing the number of threads increases efficiency given an increasing problem size. Efficiency can be calculated as *speedup*/10(threads) using Figure 7 - for example, a precision of 0.2 has an efficiency of 80%. This problem size is increased through the means of a finer precision, leading to a smaller sequential portion of the program. This results in a larger number of iterations to get to the final solution. As outlined earlier in this report, I focused on several threading practices to enable the code to be scalable given expected increases in processing power with problem size: namely thread pooling, reducing locks where possible, and allocating thread work fairly. Fundamentally this is to reduce time spent computing sequential code, and to enable efficient parallelisation of increasingly large problem sizes where possible.

This is supported by Figure 8 which increases the problem size by increasing matrix dimensions. Efficiency increases as the problem scales, consistent with Gustafson's Law.

4.5 Dimension against time for $P = 0.1$, T=4

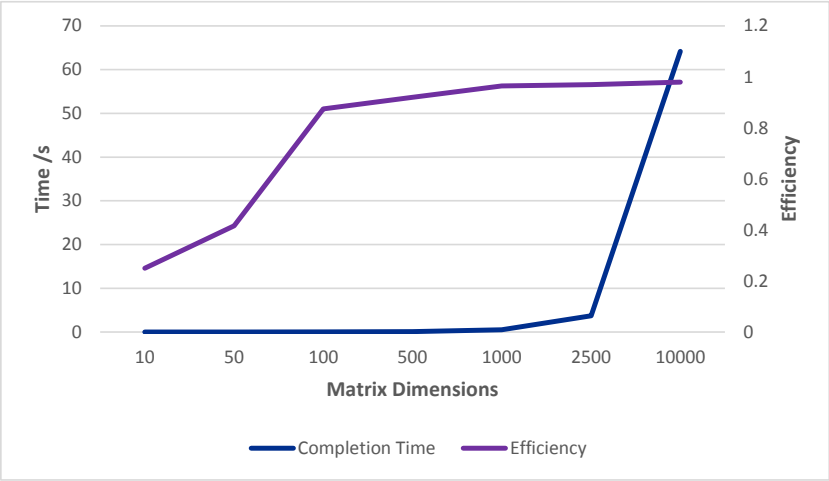


Figure 8: Time taken for varying S, $P = 0.1$ and $T = 4$

4.6 Speedup/Efficiency for all dimensions and threads

Finally visualisations have been computed for speedup and efficiency across multiple threads and multiple matrix dimensions. The visualisations have been simplified to colours representing the overall trend of each graph - final figures are available in the Appendix. Little to no speedup is shown in red-orange colours, middling speedups of around 3-6 are shown in yellow, and speedups ranging from 6 to 13 and beyond are represented with increasingly darker shades of green.

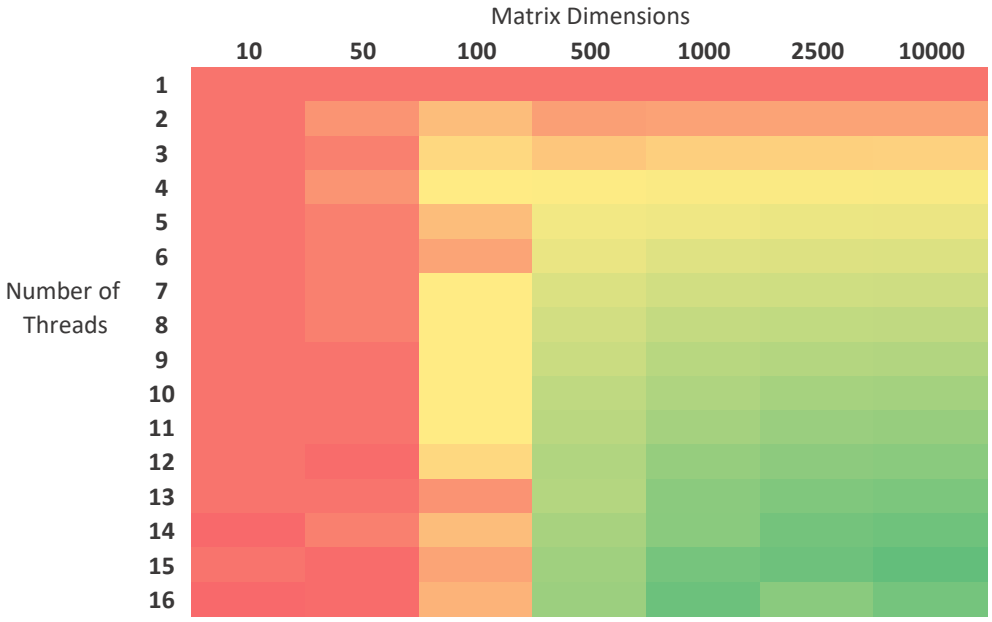


Figure 9: Speedup

Figure 9 shows that adding additional threads on smaller matrix dimensions has little to no effect. In some cases the overhead of thread management will outweigh any possible benefits. For instance, 15 threads on a matrix size of 50 has a speedup below 1 (0.83), showing that program performance has in fact deteriorated rather than benefited. This is reflected in efficiency values across Figure 10.

However, once the problem size has been scaled to at least a matrix size of 500, the advantages of a multi-threaded program become clear through the visualisation. With each successive fixed problem size, scaling processing power enables greater speedups. The greatest speedup achieved by my program was 13.6 with 15 threads using a matrix of size 10000. Even smaller numbers of threads at this problem size have a positive effect, corresponding with Amdahl's Law. Speedup metrics can additionally be used to find ideal ratios between increasing processing power and fixed problem sizes, so that the best economical cost for processing resources can be determined.

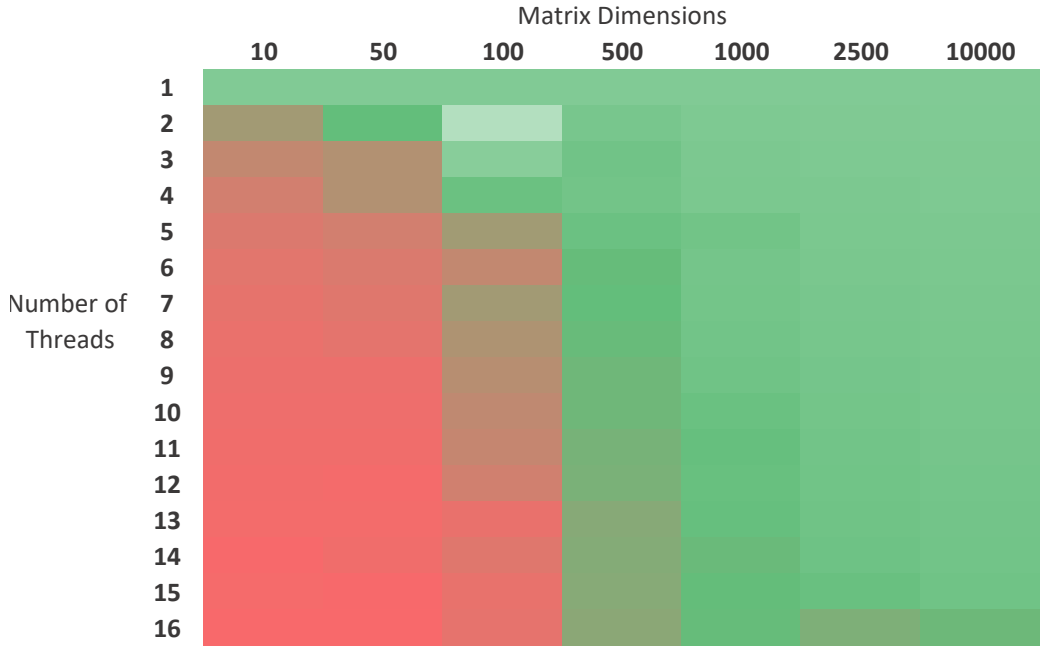


Figure 10: Efficiency

Superlinear speedup was observed on relaxing a a $100 \cdot 100$ matrix, with a speedup of 2.55 and 3.11 for 2 and 3 threads respectively. We have established that the same random seed was used throughout testing. Therefore in this scenario randomness shouldn't be an issue because every matrix will have been populated with the same values. It is of course possible that this is due to cache memory. A $100 \cdot 100$ matrix has a small space complexity associated with the algorithm relaxing it - it isn't unfeasible that this could fit in L1 and L2 memory. Recalling the introduction, these caches have very fast access times comparative to main memory, allowing the program to perform faster. Another explanation just as likely as using cached memory is that the algorithm used for computing with 2 and 3 threads isn't being compared to a true sequential algorithm as discussed previously. With many factors in play, it is ultimately difficult to pinpoint a conclusive reason for this observation of superlinear speedup and therefore challenging to understand what this data really represents.

Given the parallel tradeoff of thread management, efficiency clearly suffers with smaller problem sizes and additional threads. This is plain in Figure 10, where more than a couple of threads has a rapid effect on the efficiency of the program on small problem sizes such as 10, 50 and 100. Nevertheless, coherent with Gustafson's Law efficiency improves as the problem size grows. All results between 1-15 threads on a $10000 \cdot 10000$ matrix have an efficiency greater than 0.9 (90%). In some sense this is a good level of scaling, but the only reference point available for this report is an efficiency of 1 (100%) without looking at comparable algorithms and solutions on the same hardware. My program did start to see limitations above 15 threads, however. Efficiency dropped to 0.78 when 16 threads was specified for the same problem size. This is due to the structure of my program - more specifically, a main thread plus 16 pthreads running on a single node on Balena supporting 16 threads only. This means that there will be 1 thread descheduled (or split across 2 nodes if selected). Accordingly this affects efficiency and is a factor that needs to be considered carefully when scaling both the processing power and problem size.

References

A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *ACM SIGARCH Computer Architecture News*, volume 16, pages 280–298. IEEE Computer Society Press, 1988.

W. J. Bolosky and M. L. Scott. False sharing and its e ect on shared memory performance. In *Proceedings of the Fourth symposium on Experiences with distributed and multiprocessor systems*, 1993.

R. Bradford. Cm30225 parallel computing - slides 06, 2016. URL `https://people.bath.ac.uk/masrjb/CourseNotes/Notes.bpo/CM30225/slides06.pdf`.

J. Dean. Software engineering advice from building large-scale distributed systems. 2007.

D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. *The directory-based cache coherence protocol for the DASH multiprocessor*, volume 18. ACM, 1990.

5 Running the program

The parallel program use GNU's getopt library [2] to allow command-line arguments to be passed. First the program must be compiled with gcc:

```
gcc -std=gnu99 -Wall -pthread -o parallel parallel.c
```

Finally the compiled program can be executed with various command line arguments. For instance:

```
./parallel -S 10 -P 0.1 -T 2 -V
```

will run a parallel program using 2 threads, with a matrix of size $10 \cdot 10$, a precision of 0.1 and in verbose mode.

[2] https://www.gnu.org/software/libc/manual/html_node/Getopt.html

A full list of command line arguments is below:

Argument	Specification	Suggested Range	Default (if not provided)
T	**Number of Threads**	1-16	2
P	**Precision**	0.1-0.5	0.1
S	**Matrix Dimensions**	6-1000	6
V	**Verbosity**	n/a	Turned off by default

Further examples:

```
./parallel -S 10 -P 0.1 -T 2 -V
./parallel -S 150 -P 0.1 -T 8
./parallel -S 400 -P 0.1 -T 12
./parallel
```

Please note that full validation hasn't been completed for these command line arguments. A simple check is in place to ensure that the number of threads doesn't exceed the number of inner cells, but there are no checks regarding the data types of arguments specified. If an argument isn't passed, the default value will be used instead.

6 Appendix

$M_1 =$

1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
1.000000	0.946545	0.901489	0.876689	0.847923	0.874642	0.861564	0.923060	0.943429	1.000000
1.000000	0.902169	0.820876	0.753374	0.755979	0.714347	0.805123	0.806614	0.925887	1.000000
1.000000	0.868302	0.756249	0.697401	0.627148	0.695421	0.663959	0.815262	0.863544	1.000000
1.000000	0.852639	0.730894	0.632772	0.635382	0.581917	0.712437	0.721046	0.891854	1.000000
1.000000	0.849626	0.724982	0.653961	0.588433	0.653191	0.638036	0.791918	0.855283	1.000000
1.000000	0.867621	0.755120	0.676445	0.666337	0.642702	0.737347	0.767891	0.901859	1.000000
1.000000	0.896848	0.814536	0.760567	0.729810	0.758197	0.769846	0.854975	0.909902	1.000000
1.000000	0.945381	0.898023	0.868549	0.859712	0.858538	0.888943	0.910278	0.958504	1.000000
1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

Figure 11: Expected Output Matrix

6.1 Speedup for all sizes and threads

Number of Threads \ Matrix Dimensions	10	50	100	500	1000	2500	10000
1	1	1	1	1	1	1	1
2	1	1.66667	2.54545	1.89691	1.96663	1.9805	1.98605
3	1	1.25	3.11111	2.72593	2.91604	2.94409	2.95499
4	1	1.66667	3.5	3.68	3.85913	3.87842	3.91829
5	1	1.25	2.54545	4.38095	4.56573	4.82906	4.85686
6	1	1.25	2	4.90667	5.58908	5.7499	5.79731
7	1	1.25	3.5	5.84127	6.48333	6.64332	6.71735
8	1	1.25	3.5	6.45614	7.33962	7.53842	7.62544
9	1	1	3.5	6.9434	8.13808	8.41045	8.52074
10	1	1	3.5	7.66667	8.72197	9.27055	9.41425
11	1	1	3.5	8	9.35096	10.0937	10.2836
12	1	0.83333	3.11111	8.55814	10.3457	10.9336	11.1361
13	1	1	1.64706	8.36364	11.0511	11.7305	11.9945
14	0.75	1.25	2.54545	9.2	11.1143	12.542	12.8318
15	1	0.83333	2	9.68421	12.3885	12.9503	13.6074
16	0.75	0.83333	2.33333	9.94595	13.0537	11.0859	12.4506

Figure 12: Speedup

6.2 Efficiency for all sizes and threads

Number of Threads \ Matrix Dimensions	10	50	100	500	1000	2500	10000
1	1	1	1	1	1	1	1
2	0.5	0.83333	1.27273	0.94845	0.98332	0.99025	0.99303
3	0.33333	0.41667	1.03704	0.90864	0.97201	0.98136	0.985
4	0.25	0.41667	0.875	0.92	0.96478	0.9696	0.97957
5	0.2	0.25	0.50909	0.87619	0.91315	0.96581	0.97137
6	0.16667	0.20833	0.33333	0.81778	0.93151	0.95832	0.96622
7	0.14286	0.17857	0.5	0.83447	0.92619	0.94905	0.95962
8	0.125	0.15625	0.4375	0.80702	0.91745	0.9423	0.95318
9	0.11111	0.11111	0.38889	0.77149	0.90423	0.93449	0.94675
10	0.1	0.1	0.35	0.76667	0.8722	0.92706	0.94142
11	0.09091	0.09091	0.31818	0.72727	0.85009	0.91761	0.93487
12	0.08333	0.06944	0.25926	0.71318	0.86215	0.91113	0.92801
13	0.07692	0.07692	0.1267	0.64336	0.85009	0.90235	0.92265
14	0.05357	0.08929	0.18182	0.65714	0.79388	0.89586	0.91655
15	0.06667	0.05556	0.13333	0.64561	0.8259	0.86335	0.90716
16	0.04688	0.05208	0.14583	0.62162	0.81586	0.69287	0.77816

Figure 13: Efficiency

6.3 Raw Data

6.3.1 Time (s) for S = [10, 50, 100, 500]

Number of Threads	10x10	50x50	100x100	500x500
1	0.003	0.005	0.028	0.368
2	0.003	0.003	0.011	0.194
3	0.003	0.004	0.009	0.135
4	0.003	0.003	0.008	0.10
5	0.003	0.004	0.011	0.084
6	0.003	0.004	0.014	0.075
7	0.003	0.004	0.008	0.063
8	0.003	0.004	0.008	0.057
9	0.003	0.005	0.008	0.053
10	0.003	0.005	0.008	0.048
11	0.003	0.005	0.008	0.046
12	0.003	0.006	0.009	0.043
13	0.003	0.005	0.017	0.044
14	0.004	0.004	0.011	0.040
15	0.003	0.006	0.014	0.038
16	0.004	0.006	0.012	0.037

6.3.2 Time (s) for S = [1000,2500,10000,25000]

Number of Threads	1000x1000	2500x2500	10000x10000	25000x25000
1	1.945	14.323	251.464	*Timed Out*
2	0.989	7.232	126.615	*Timed Out*
3	0.667	4.865	85.098	771.8495
4	0.504	3.693	64.1777	581.572
5	0.426	2.966	51.775	467.196
6	0.348	2.491	43.376	391.6
7	0.300	2.156	37.435	337.233
8	0.265	1.9	32.977	296.296
9	0.239	1.703	29.512	264.732
10	0.223	1.545	26.711	239.224
11	0.208	1.419	24.453	218.579
12	0.188	1.310	22.581	201.494
13	0.176	1.221	20.965	186.733
14	0.175	1.142	19.597	174.18
15	0.157	1.106	18.48	163.967
16	0.149	1.292	20.197	157.257
17	0.264			
18	0.246			
19	0.237			
20	0.232			
21	0.238			
22	0.240			
23	0.247			
24	0.244			
25	0.247			
26	0.244			
27	0.242			
28	0.238			
29	0.238			
30	0.233			
31	0.232			
32	0.229			