CM30225: Parallel Computing

Shared Memory Architectures: Balena, pthreads and C

November 2018

Contents

1 Introduction

A *"multiprocessor architecture"* is a broad term applied to parallel architectures involving more than one full processor. A shared memory architecture is an approach to multiprocessor design which relies on a set of $\mathcal{N}$ processors, each connected via a shared memory bus to main memory. Access to memory is substantially slower than processor speeds, as all accesses must go through the shared bus which is the quintessential bottleneck in the design. Changes to values stored in shared memory are globally visible and access to shared values is incredibly slow when more than one processor wishes to perform an update [8]. For this reason, shared memory architectures are generally appropriate for small-scale implementations (less than 100 cores). Programmers must consider carefully the use of shared variables in programs which will reside in main memory. In order to combat slow memory accesses via the bus, processors will usually keep a cached copy of shared data, which introduces the cache coherency problem - maintaining consistent and updated copies of values in shared main memory is very difficult when values can be modified in any order by other processors [9]. This assignment requires consideration of the above in implementing matrix relaxation using a shared memory architecture.

Matrix Relaxation

Matrix relaxation involves repeatedly replacing a cell's value with the average of its four neighbours (excepting boundary values) until values settle down to a given precision. The task in question was to use C and the POSIX library to implement relaxation on a square matrix of dimension d, using t threads[1] to a precision of p. The solution was run on *Balena*, the High Performance Cluster (HPC) at University of Bath[2] using a variety of configurations to investigate the scalability and correctness of the parallel approach on various problem sizes. Matrix relaxation holds some inherent sequential properties which require careful synchronisation between threads, as the order of computations is crucial to the output:

1. **Consistency of neighbour cells:** For any *inner cell* $c_{i,j}$ (where i is the row and j is the column) to be correctly relaxed, the neighbouring cells ($c_{i+1,j}$, $c_{i-1,j}$, $c_{i,j+1}$, $c_{i,j-1}$) must be kept consistent values throughout the iteration. If any neighbouring cell values were changed (or *relaxed*) between reads for a certain cell, this would cause the value of the cells in the resulting matrix to be incorrect and unpredictable between iterations. The matrix needs to be divided in such a way without interfering with other threads' computations.

2. **Precision checks between iterations:** Before a new iteration can start, there needs to be a check across all inner cells as to whether the difference in corresponding matrix values before and after the current iteration is less than or equal to the chosen precision, p. There must be careful management over when computation can proceed as threads are very unlikely to be in lockstep.

[1] The number of `pthread` threads created, not including the program's main thread.
[2] `https://www.bath.ac.uk/corporate-information/balena-hpc-cluster/`

2 Sequential Approach

As in Algorithm 1, the sequential approach relied on working with two matrices of the same dimension d and swapping the references to these after each iteration. This way, the output matrix of iteration i becomes the input to iteration $(i + 1)$. The problem of neighbour cell inconsistency discussed previously is avoided with the introduction of this second (output) matrix. This comes at the cost of an increase in space complexity in memory on the heap, but this is still linear in d which is an appropriate trade off in order to avoid incorrect calculations (new space complexity is $O(2d) = O(d)$, where d is the dimension of the matrix). The fact that swapping input and output matrices comes at the end of each iteration means that no cell can be overwritten with a new relaxed value until every other cell's relaxed value has already been calculated. At step `9` of Algorithm 1 there is a potentially expensive operation in swapping matrices between every iteration. In order to reduce this overhead, the implementation instead swaps the *references* to each matrix, not the values of each and every inner cell.

Algorithm 1: Matrix relaxation (sequential).

Input: A $d \times d$ matrix with equal fixed edge values and chosen inner values.
Output: A $d \times d$ matrix with equal fixed edge values and ***relaxed*** inner values.
Data: d: Dimension of matrix, p: Precision

1 Extract program arguments
2 Create matrices `matA` and `matB` of dimension d
3 Initialise `matA` and `matB` with identical contents
4 **repeat**
5 **foreach** *inner cell* (i, j) **do**
6 `total = matA[i+1][j]+matA[i-1][j]+matA[i][j+1]+matA[i][j-1]`
7 `matB[i][j] = total ÷ 4`
8 **end**
9 Swap `matA` and `matB`
10 **until** *`abs(matB[i][j] - matA[i][j])`* $\leq p$ *for all* (i, j)

3 Parallel Approach

As with any algorithm, attempting to run operations in parallel brings initial overheads in the form of:

1. **Management of threads:** Creation, joining and destroying `pthreads` is an overhead [2] which was minimised by adopting a *"thread pool"* design, which fixes this overhead to once during the program lifetime [13]. Threads are re-used every iteration and destroyed together once precision is reached. If this was done on every iteration the overhead would be substantial (e.g. when p is small, and starting values are a large distance from the converged result causing lots of iterations).

2. **Management of data:** Allocating each thread a certain number of cells to relax in parallel is another overhead. When the problem size is very small this overhead is relatively large, but when the value of d increases this relative cost is reduced. Larger problem sizes (bigger matrices) should reasonably render this initial overhead small compared to the total runtime.

Algorithm 2 describes the parallel approach, which also uses two matrices of dimension d, initialised with identical values. The matrix is divided evenly as possible between threads so that cells can be relaxed in parallel. Once each thread has finished working on the number of cells it has been assigned, there is synchronisation across threads in order to check if another iteration is needed.

Algorithm 2: Matrix relaxation (parallel).

Input: A $d \times d$ matrix with equal fixed edge values and chosen inner values.
Output: A $d \times d$ matrix with equal fixed edge values and ***relaxed*** inner values.
Data: d: Dimension of matrix, p: Precision, t: Number of threads

```
1  Extract program arguments
2  Create matrices matA and matB of dimension d
3  Initialise matA and matB with identical contents
4  Distribute (d − 2)^2 inner cells of matA and matB between t threads
5  Create and start t threads to relax assigned cells
6  repeat
7      foreach inner cell (i, j) assigned to each thread do
8          total = matA[i+1][j]+matA[i-1][j]+matA[i][j+1]+matA[i][j-1]
9          matB[i][j] = total ÷ 4
10     end
11     Synchronise t threads
12     if abs(matB[i][j] - matA[i][j]) > p for any (i, j) across t threads then
13         Swap matA and matB
14     end
15 until abs(matB[i][j] - matA[i][j]) ≤ p for all (i, j) across t threads
```

Choice of Concurrency Primitives

Out of several choices of concurrency primitives (e.g. barriers, mutexes, semaphores, condition variables), the use of `pthread_barrier_t` barriers was adopted to synchronise computations across several threads, ensuring t threads had finished an iteration before precision values were calculated. Other primitives were considered but decided inappropriate for this task. Mutexes alone would provide mutual exclusion in accessing critical regions but not synchronization, and (although a candidate for synchronisation) condition variables have been subject to the problem of spurious "wake-ups" [4]. Use of semaphores is primarily appropriate in problems synchronising two threads rather than many, with the counting semaphores approach requiring a lot of inter-thread communication to signal and wait for when resources are available. Barriers require the least amount of code on the programmer's behalf in getting a specific number threads to wait for all to reach a certain stage in a computation before they can continue. Without barriers between iterations there would be race conditions and unpredictable results in computations [3], as matrix relaxation is an example of a problem where the calculated solution can be dependent on the execution speed of each thread if no synchronisation primitives are adopted. For these reasons, barriers were chosen as a suitable concurrency primitive for the problem.

Avoiding Race Conditions

A race condition is defined as the consequence of multiple threads reading and writing a shared value with the final result of operations dependent on the order of execution [5]. A data race is a type of race condition which occurs when two or more threads access a shared location and at least one access is a write which is not enforced using synchronization primitives [11]. In the parallel implementation, both of these situations are avoided using the below strategy.

Each process has a reference

keeps a reference to the input and output matrices but only updates the values of the cells in the output matrix that it has been assigned. All threads are prevented from continuing to the next iteration using barriers, as described above. Critically, this approach means there is no need to use locks to protect access to the elements of either matrix. During a single iteration on any thread:

- Each element of the first (input) matrix can be read from any number of times without worrying about other threads updating any of its elements.
- Each element of the second (output) matrix is written to once per iteration. There is no potential for simultaneous read or write operations on the same element of the output matrix.
- Calculating the precision (difference between corresponding cells) is a read-only operation.
- Threads swap their references to input and output matrices and a precision check across all threads is done in a synchronised fashion. No thread starts a new iteration before the previous one has finished.

Dividing the Task Between Threads

Due to the "unfair" and unpredictable nature in which threads are scheduled and executed by the OS, I decided to split the number of cells to relax equally as possible between threads. Otherwise, if a single thread was given the bulk of the cells to relax and was de-scheduled by the OS in favour of a thread with less cells to relax, it may produce an overall execution time that is larger. This uneven split would cause runtime to vary drastically based on how the OS decides to schedule threads. Instead of relying on a "lucky" scheduling of tasks by the OS, the total number of cells was instead spread as evenly as possible so that the total runtime per thread should not vary significantly. For a $d \times d$ matrix, there are $(d-2)^2$ inner cells which need relaxing. It follows that by splitting the cells as equally as possible across t threads, then each thread should receive $(d-2)^2 \div t$ cells to relax. As this is just integer division, it may mean the number of cells does not divide exactly by the number of threads. If there is any remainder, this is given by $(d-2)^2 \mod t$. In keeping with the approach of an even distribution of cells across threads, this remainder is spread across threads in an even fashion too, to balance workload.

Once all cells have been divided across threads, the important single call to `pthread_create` can be made for each thread, passing in data to the `submatrix_relax` function where threads can start relaxing their part of the matrix.

Precision Checks

In order to determine when relaxation has finished across all threads, a global boolean flag called `all_within_precision` is used, along with a global boolean array of size equal to t, the number of threads (called `next_iteration_needed`). Each thread is responsible for updating its slot in the global `next_iteration_needed` array, indicating to the main thread whether the precision is reached for all cells allocated to that thread. I did initially consider use of a lock and counter variable to check how many threads had finished, however a simpler solution existed which was less costly in terms of runtime than maintaining locks around critical regions. Use of `pthread_barrier_t` barriers allows for synchronisation across multiple threads between iterations, in a "superstep" approach (as shown in Figure 1). As the purpose of the calls to `pthread_barrier_wait` is to create a synchronisation point where all threads must hit before they can continue, it was important to minimise the necessary work between each barrier call so that each thread could do as much work in parallel as possible. As such, each thread is not just responsible for relaxing the cells it has been assigned, but also calculating the precision.

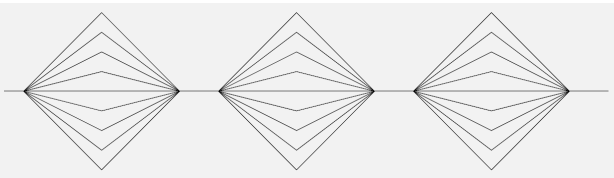

Figure 1: Diagram showing the "superstep programming" method with barriers.

If the precision for thread n is within the desired range, it updates its dedicated element in the global array - `next_iteration_needed[n]`. As all threads have a dedicated element in this array, it means that no data races occur for overwriting the same element in the array, preventing the need for locks around any regions including array updates. It also means that the work to compute precision is divided by t, the number of threads instead of giving the main thread $(d-2)^2$ cells to compute precision for in between `pthread_barrier_wait` calls. The only work that the main thread does between barrier calls is to check if any values in the `next_iteration_needed` array are set to `true`. If this is the case, the global flag `all_within_precision` that each thread checks for after an iteration is set to `false` and threads move to the next iteration, swapping their references to the input and output matrices. It is important that the main thread sets this flag before the second barrier is hit, ensuring all threads read the updated value before deciding if a new iteration is necessary. Eventually after a certain number of iterations it may be that all values in the `next_iteration_needed` array are set to `false`. When this happens, the matrix has been relaxed to the desired precision across all t threads, which means the single call to `pthread_join` can be made for each thread to terminate and join the main thread before the relaxed matrix is displayed. **Figures 2 and 3 show both the main thread and `pthread` flow of control.**

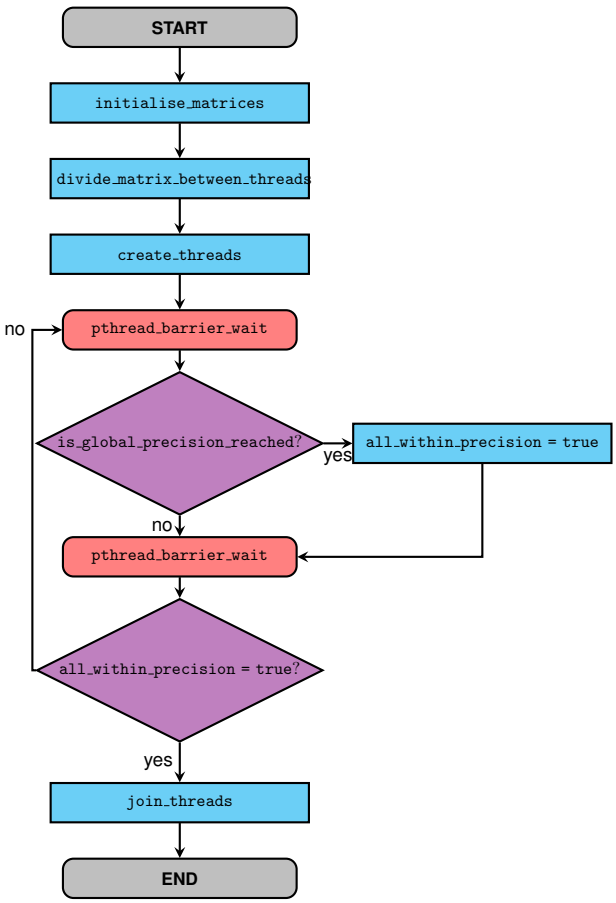

Figure 2: Diagram of the main thread execution life cycle.

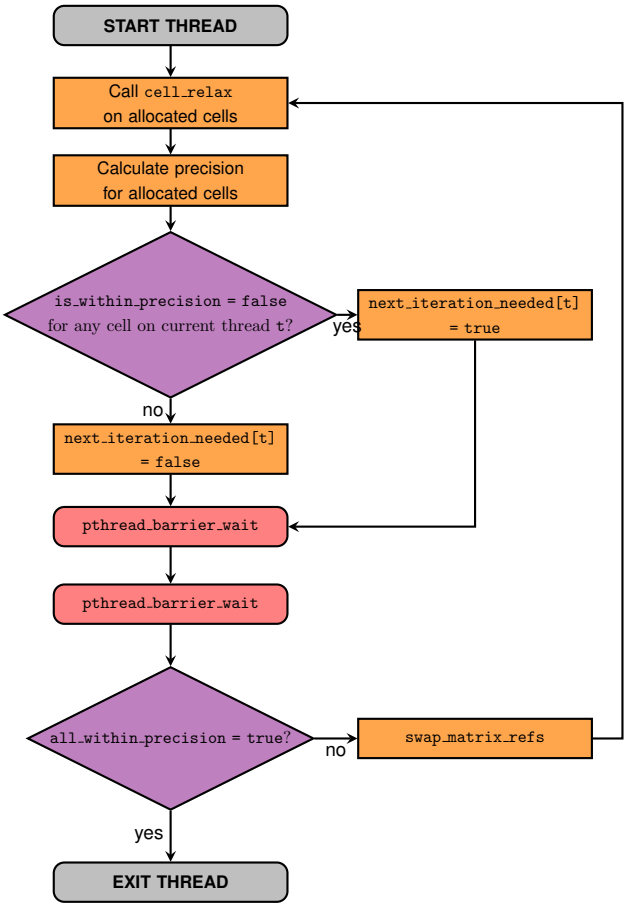

Figure 3: Diagram of the `pthread` execution life cycle.

4 Testing Program Correctness

In order to verify whether the approach taken was correct, it required a combination of manual verification and a set of varied test cases, where input and output were compared for correctness. See Appendix A for how to run the parallel implementation with different configurations.

Manual Verification

The first stage in testing program correctness was to check against a manually-calculated matrix relaxation in order to check that the parallel C implementation finished after an equal number of iterations and contained equal cell values to the manually-calculated relaxed matrix. The relaxation process was executed by hand on the below 5×5 matrix using the following arguments: $d = 5$, $p = 0.2$, $t = 3$. Using the parallel algorithm shown in Algorithm 2, the $(d-2)^2 = 9$ inner cells are divided equally across the 3 threads, as shown in Table 1. Between iterations, the maximum difference in cell values ($\max(|c_{new} - c_{old}|)$) for each thread was calculated, to check if this was within the desired precision of $p = 0.2$. **Rows and columns are indexed from 0.**

Table 1: Inner cell breakdown for $d = 5$, $t = 3$.

Thread	Inner cells to relax
1	$(1,1), (1,2), (1,3)$
2	$(2,1), (2,2), (2,3)$
3	$(3,1), (3,2), (3,3)$

The manual calculation was as follows:

$$\begin{bmatrix} 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \\ 1.000000 & 0.001251 & 0.563585 & 0.193304 & 1.000000 \\ 1.000000 & 0.808741 & 0.585009 & 0.479873 & 1.000000 \\ 1.000000 & 0.350291 & 0.895962 & 0.822840 & 1.000000 \\ 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \end{bmatrix}$$

Initial Matrix: The matrix of `double` values is set to 1.000000 on the edges, and random values between 0 and 1 on the inner cells.

$$\begin{bmatrix} 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \\ 1.000000 & 0.843081 & 0.444891 & 0.760865 & 1.000000 \\ 1.000000 & 0.484138 & 0.687040 & 0.650288 & 1.000000 \\ 1.000000 & 0.926176 & 0.689535 & 0.843959 & 1.000000 \\ 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \end{bmatrix}$$

Iteration 1:

Thread	$\max(\lvert c_{new} - c_{old} \rvert)$	$\leq p$?
1	$\lvert 0.843081 - 0.001251 \rvert$	`false`
2	$\lvert 0.484138 - 0.808741 \rvert$	`false`
3	$\lvert 0.926176 - 0.350291 \rvert$	`false`

$$\begin{bmatrix} 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \\ 1.000000 & 0.732257 & 0.822747 & 0.773795 & 1.000000 \\ 1.000000 & 0.864074 & 0.567213 & 0.822966 & 1.000000 \\ 1.000000 & 0.793418 & 0.864294 & 0.834956 & 1.000000 \\ 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \end{bmatrix}$$

$$\begin{bmatrix} 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \\ 1.000000 & 0.921705 & 0.768316 & 0.911428 & 1.000000 \\ 1.000000 & 0.773222 & 0.843520 & 0.793991 & 1.000000 \\ 1.000000 & 0.932092 & 0.798897 & 0.921815 & 1.000000 \\ 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \end{bmatrix}$$

$$\begin{bmatrix} 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \\ 1.000000 & 0.885385 & 0.919163 & 0.890577 & 1.000000 \\ 1.000000 & 0.924329 & 0.783607 & 0.919191 & 1.000000 \\ 1.000000 & 0.893030 & 0.924357 & 0.898222 & 1.000000 \\ 1.000000 & 1.000000 & 1.000000 & 1.000000 & 1.000000 \end{bmatrix}$$

Iteration 2:

Thread	$\max(\lvert c_{new} - c_{old} \rvert)$	$\leq p$**?**
1	$\lvert 0.822747 - 0.444891 \rvert$	false
2	$\lvert 0.864074 - 0.484138 \rvert$	false
3	$\lvert 0.864294 - 0.689535 \rvert$	true

Iteration 3:

Thread	$\max(\lvert c_{new} - c_{old} \rvert)$	$\leq p$**?**
1	$\lvert 0.921705 - 0.732257 \rvert$	true
2	$\lvert 0.843520 - 0.567213 \rvert$	false
3	$\lvert 0.932092 - 0.793418 \rvert$	true

Relaxed Matrix (4 iterations):

Thread	$\max(\lvert c_{new} - c_{old} \rvert)$	$\leq p$**?**
1	$\lvert 0.919163 - 0.768316 \rvert$	true
2	$\lvert 0.924329 - 0.773222 \rvert$	true
3	$\lvert 0.924357 - 0.798897 \rvert$	true

This step-by-step manual calculation shows checks for precision across each thread's cells after they have been relaxed, which is used to decide whether another iteration is needed. Once the manual calculation had taken place, I created a C function to populate the initial matrix with identical values and ran the program to compare outputs:

```
$  ./sharedrelax -d 5 -t 3 -p 0.2 -v
```

The resulting output included:

```
INFO: ___Relaxed Matrix (4 iterations)_______
1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.885385 0.919163 0.890577 1.000000
1.000000 0.924329 0.783607 0.919191 1.000000
1.000000 0.893030 0.924357 0.898222 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000
```

Upon observing that that the parallel C implementation matched the manual example at each step, I varied the value of t from 1 to 9 (the maximum value of t possible for a matrix of this size). Outputs from these tests were saved to files and compared using the Unix `diff` tool to verify that the total number of iterations required and the matrix values at each stage were identical, which was a success. These outputs are visible in the `tests` directory of this submission. The manual verification stage was then performed on a variety of other small ($d \leq 6, t \leq 5$) examples, with values compared to the result of the sequential

and parallel implementations. This made me confident that for small matrices with a small number of threads, the result of the computation could be trusted as correct.

Automated Test Cases

For tests with larger matrices and more threads involved, I was not as confident because scaling the problem or splitting it across more threads could indeed expose more race conditions that weren't as detectable on small examples. For this reason, I went on to produce automated tests where the number of iterations and values of matrix cells at each iteration could be compared at larger scales.

In order to verify the program's correctness in a more automated sense at larger scale, I created a short test script to compare outputs when running the program under different configurations on increasingly large matrices. I modified the program to only print the state of the matrix after each iteration, as well as the number of iterations it took to relax the matrix, using the `-i` argument for "information mode". This full test script is viewable in Appendix C. Its output is saved to a file in the `tests` directory of this submission.

```
$ ./correctness_test.bash
```

This script computed the matrix for a given dimension d using 1 `pthread` ($t = 1$), then compared this output to the result of running under the same conditions with increasing numbers of threads ($t = \{2, ..., 50\}$). This was then repeated for increasingly large matrices ($d = \{10, ..., 1000\}$). As initial matrix inner cell values are randomly chosen between 0 and 1, the same seed was used in all trials of each matrix at size d, to produce consistent and reproducible results. To ensure each test case was reproducible, I also ensured each was run 100 times to account for potential race conditions occurring that can go undetected if a test was only run once. It must be noted that whilst all tests passed after 100 repetitions and no bugs or race conditions were detected, this does not guarantee or prove full program correctness. For extra assurance, the output of the parallel implementation for these configurations were compared using the `diff` tool to the sequential implementation's output which also showed no difference in matrix values at each iteration.

Whilst the test cases are not fully exhaustive, nor guaranteed to show full program correctness in themselves, they do provide greater confidence in the program producing correct and consistent output and handling race conditions at these scales. Combined with the successful manual verification to check for correctness on smaller examples, and the consistency in these outputs seen with larger matrices and thread counts, the program can be regarded as producing correct outputs within the context of this assignment. **Full results are detailed in Appendix B.1.**

5 Scalability Investigation

After the manual and automated correctness testing stage, experiments were performed to understand how scalable the parallel implementation was using *Balena*, under different program configurations. The `real` value given by the Unix `time` command was used to measure execution time, e.g.:

```
$ time ./sharedrelax -d 1000 -t 4 -p 0.1
```

Full results of testing can be seen in Appendix B.

Comments on Reproducibility and Speedup

Some uncontrollable factors can affect the runtime of the problem, including the distance of values from a converged result. Because of this, for scalability tests which required **varying** the size of the matrix, a **different seed** was used each time when generating random initial matrix values and several trials were performed before averaging the total time taken. This approach was adopted to minimise the chance of unfairly comparing the runtime of a matrix which of size d_1 which was close to a converged result against a matrix of size d_2 which was far from a converged result and attributing the difference in timings wholly to the difference in matrix size. For scalability tests which required use of a **fixed** size matrix, the **same seed** was used each time when generating random initial matrix values to ensure values remained equal whilst varying the number of threads t and precision value p.

In the following experiments, **speedup calculations use the time for $t = 1$ as the "sequential" execution time**, because no relaxation is done on the main thread. The reason for this is because by changing the original sequential algorithm (Algorithm 1) to run in parallel, it has fundamentally changed the way the algorithm operates (by dividing work etc. - see Algorithm 2), which makes comparisons of runtime measurements difficult, as it is not comparing a like-for-like algorithm. This does also mean that the overhead of managing a single `pthread` is attributed to the "sequential" time measurement, but is a better baseline than comparing two different algorithms and concluding prematurely that the parallel version is best. Other limiting factors on speedup can include a problem's inherent lack of parallelism or an implementation's poor management of resources, causing unnecessary contention which leads to synchronization delays [7].

5.1 Fixing the Matrix Dimension d

The initial scalability tests involved taking a fixed size matrix ($d = 1000$) and varying either the thread count t (Figures 4, 5, 6 and 7), or the precision value p (Figure 8) to see how this affected timings. These tests allowed reflection on the validity of Amdahl's Law which assumes a fixed problem size when claiming that every program's maximum attainable speedup is bounded by the inverse of its sequential fraction, regardless of number of processors used [1]. This is different from the maximum speedup attainable from hardware (bound by the number of cores). Amdahl simply argues that an upper bound exists for the problem which may show different speedup trends on different hardware.

Changing the Thread Count t

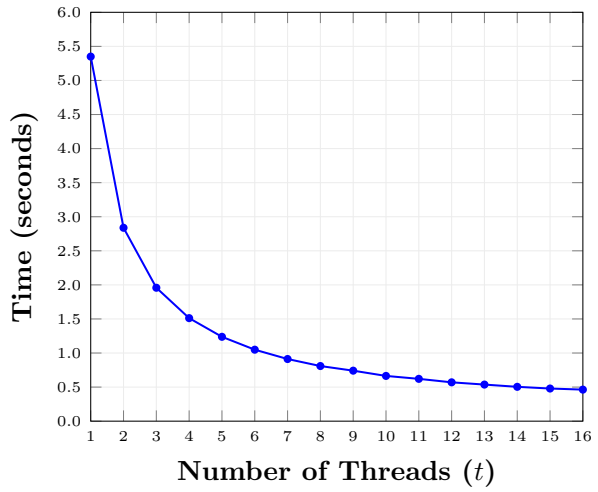

Figure 4: Time (seconds) against Number of Threads for $d = 1000, p = 0.05, t = \{1, ..., 16\}$

As the number of threads increase from 1 to 16, the resulting execution time decreases in smaller intervals (referred to as "diminishing returns" [12]) which seems to show accordance with Amdahl's Law. For the fixed problem size there is indeed a fixed sequential overhead of thread creation, dividing cells between threads and so on, which is a limiting factor on the speedup obtained. Later experiments (e.g Appendix B.4) show that *larger problem sizes* can produce larger speedup and efficiency values, which also confirms Gustafson's Law [10]. In other words, both Laws are correct but Amdahl considers a fixed size problem whereas Gustafson acknowledges that problem sizes usually increase with the number of processors available.

As the matrix is subdivided into more parts, it seems that increasing the parallel part of the problem positively impacts the runtime, as *Balena* can schedule each thread on a separate core all the way up to $t = 16$. See Appendix B.2 for raw data. **Using the raw times calculated above, a graph of speedup and efficiency can be plotted.**

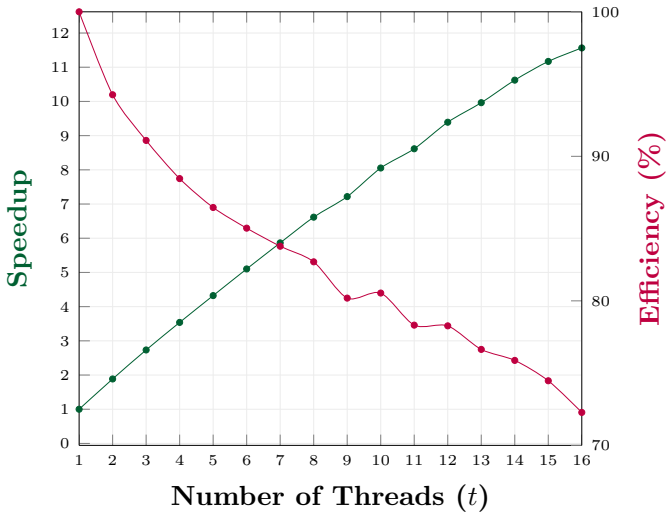

Figure 5: Speedup and Efficiency (%) against Number of Threads for $d = 1000, p = 0.05, t = \{1, ..., 16\}$

Results show that for a fixed matrix size, the speedup values were almost linear with t. As more threads are added, more can be done in parallel across t cores (scheduling 1 thread per core), which results in increased speedup values. As speedup values increase in a sub linear fashion, there is actually a decrease in efficiency from 100.00% to 72.28%. This can be attributed to the increase in scheduling overhead managing multiple threads - overall utilisation of the hardware is not as efficient as a larger fraction of time is spent scheduling than doing computation when t increases. See Appendix B.2 for raw data.

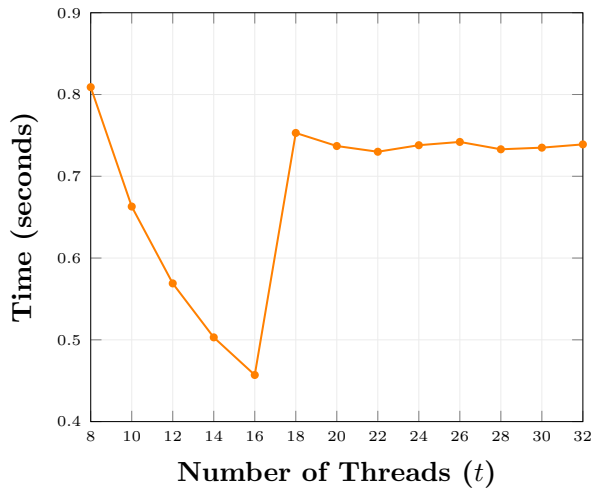

Figure 6: Time (seconds) against Number of Threads for $d = 1000, p = 0.05, t = \{8, ..., 32\}$

Balena has 16 cores, so whilst creating more threads than this is possible, it means that there is a visible cost in terms of scheduling and management that causes the speedup and efficiency values to decrease. By increasing the value of t beyond the number of cores available, there is a clear overhead in scheduling and waiting for idle threads to complete. The problem can't be as efficiently parallelised as before when $t > 16$ because there are simply not enough cores to run 1 thread per core. This confirms the reason behind the abrupt increase in execution time observed when $t = 18$ and above using the same size matrix with the same initial values as in the previous experiment using $t = \{1, ..., 16\}$. See Appendix B.3 for raw data.

This cut-off point of $t = 16$ is hardware specific to *Balena*, but the same pattern would be observed on any general shared memory multicore architecture - with n cores, using a value $t > n$ causes the problem to be divided in a way that does not minimise the parallel fraction of the total runtime. **Using the raw times calculated above, a graph of speedup and efficiency can be plotted.**

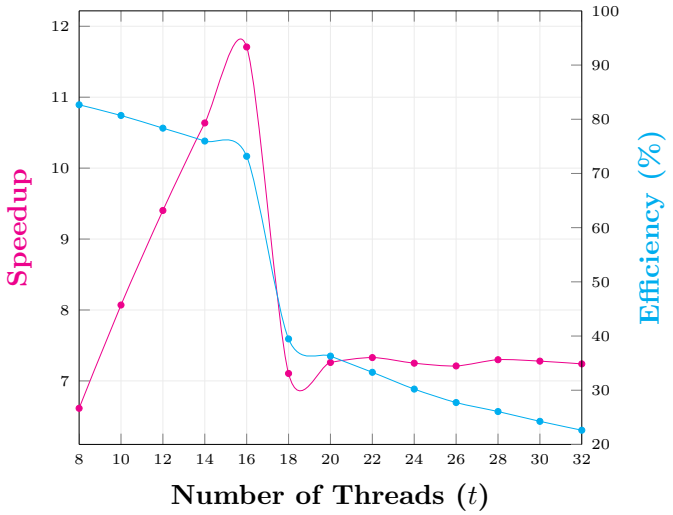

Figure 7: Speedup and Efficiency (%) against Number of Threads for $d = 1000, p = 0.05, t = \{8, ..., 32\}$

From the 72.28% efficiency observed at $t = 16$, the value unsurprisingly decreases a substantial amount once t is increased further due to the overhead of scheduling more threads than there are cores available for the process. If the algorithm itself allowed for $t > (d-2)^2$, then increasing the value of t to a value much larger would eventually cause symptoms of thrashing [6] - the scheduling overhead would dwarf the overall computation time. Using Amdahl's reasoning, the maximum speedup value is 11.708 so this follows that the sequential fraction of the program (and limiting factor on speedup for this fixed problem size) for $d = 1000, p = 0.05$ is $\frac{1}{11.708}\times \approx 8.5\%$. This sequential fraction limits the maximum attainable speedup no matter how many processors are used, which confirms Amdahl's Law. See Appendix B.3 for raw data.

Changing the Precision p

The next scalability experiments observed the effects of keeping a fixed problem size and thread count, whilst varying the precision value p. It was observed that precision values of $p > 0.1$ do not reap a lot of benefits from the parallel approach, as the difference in execution time between using 1 thread compared to 8 threads is not significant until p is set to a value less than 0.1 (Figure 8).

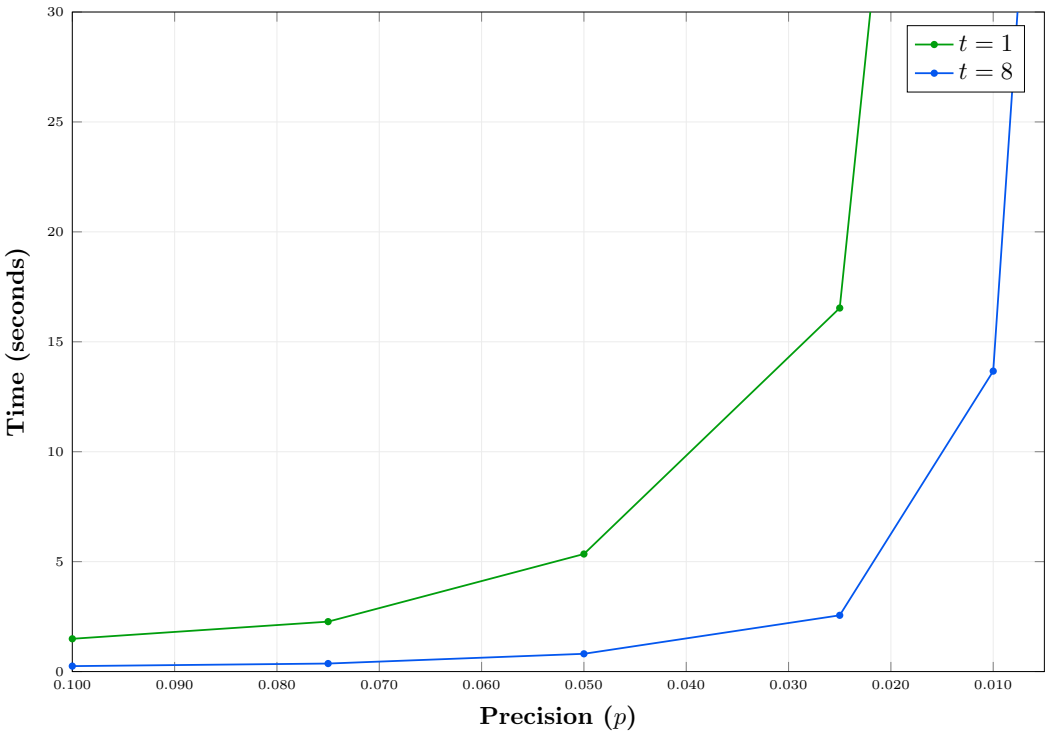

Figure 8: Time (seconds) against Precision (p) for $t = \{1, 8\}$, $d = 1000$, $p = \{0.1, ..., 0.005\}$

A greater value for the precision p generally causes a larger number of iterations to be required for a matrix to be relaxed. This means (for $t = 8$ specifically) an increased fraction of the total runtime is spent by the threads themselves doing computations, which as seen before in Figure 5 allows more of the total program time to be executed in parallel across t cores where $t \leq 16$. As a consequence, there is a reduction in total execution time and an increase in speedup and efficiency values for a fixed problem size when the value of p is reduced below 0.1. By increasing the thread count such that $8 < t \leq 16$, the same trend is observed, but overall runtimes are significantly reduced as p is decreased below 0.1 at larger fixed values of t within these limits. As before, this limit of $t \leq 16$ is required for this pattern to be observed due to the specific architecture of *Balena*. This general pattern would be observed until $t = n$ for any shared memory architecture with n cores. It must be noted that using very small precision values (e.g. $p < 0.001$) could potentially lead to a number of iterations so large that *Balena* "times out". Furthermore, some combinations of initial matrix values may never terminate given this value if averages never converge. See Appendix B.5 for raw data.

5.2 Varying the Matrix Dimension d

The next scalability tests involved varying the size of the matrix (size of problem) under a fixed thread count t, and constant fixed precision value p (Figure 9) to see how this affected timings. The effect of an increase in matrix dimension from d_1 to d_2 is an increase in inner cell count by $(d_2 - 2)^2 - (d_1 - 2)^2$. In other words, the matrix dimension is not linear with the number of cells to relax.

As mentioned, Gustafson's Law states that the problem size scales with the number of processors n which means that speedup is not bound by n if larger problem sizes are considered [10]. The raw data in Appendix B.4 shows this too - values for speedup and efficiency increase as the matrix dimension increases to larger sizes (scales). This is consistent with the graph of dimension d against time for various fixed thread counts $t = \{1, 4, 8, 16\}$ shown in Figure 9 - larger fixed values of t produced reduced execution times as the matrix size scales to larger values, with the difference in execution times in multithreaded configurations ($t \geq 4$) becoming more pronounced as $d > 2000$. It is clear that dividing the matrix between threads is a much more scalable approach for larger matrix dimensions than simply executing the problem sequentially ($t = 1$). For smaller problem sizes ($d < 1000$) the difference in execution times between sequential and parallel implementations is far less pronounced, reinforcing the point that parallelism is only worth the initial overhead in thread management and work allocation if the problem is sufficiently large such that the initial parallel overheads do not dominate.

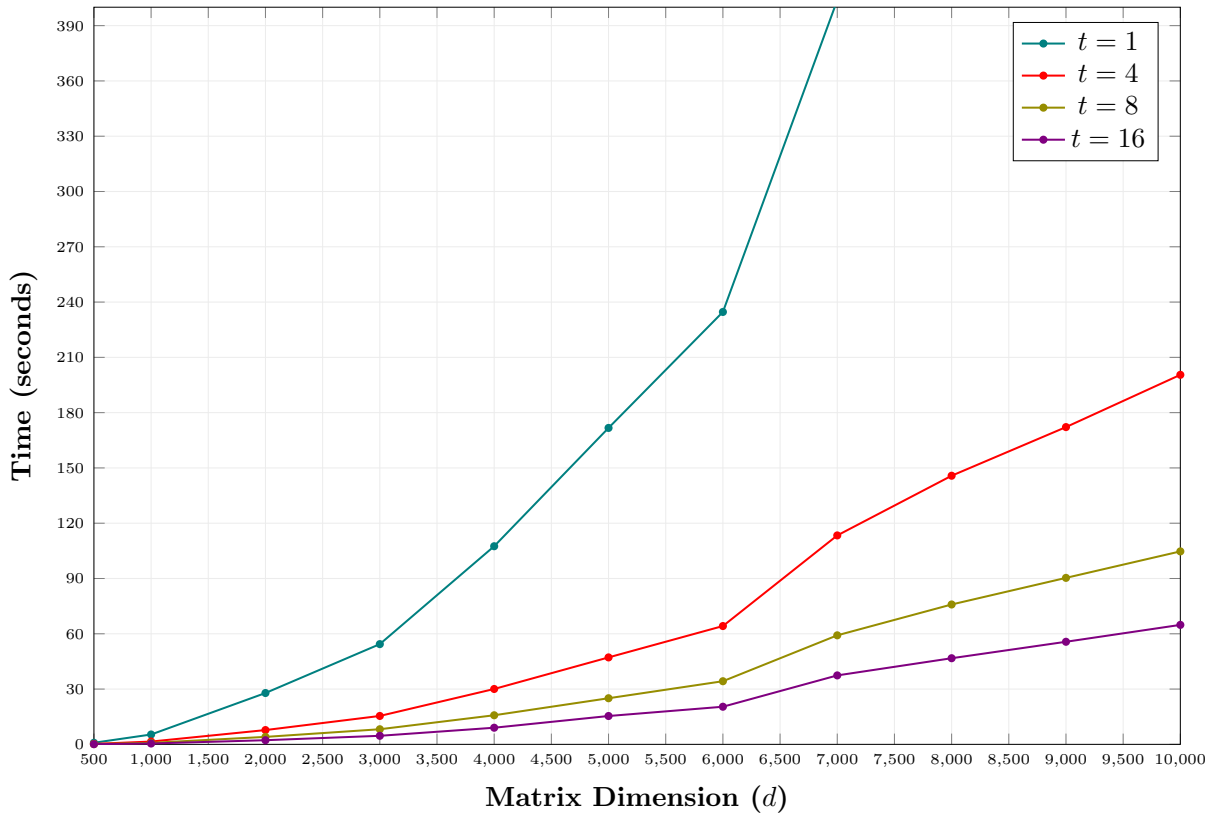

Figure 9: Time (seconds) against Matrix Dimension (d) for $d = \{500, ..., 10000\}, p = 0.05, t = \{1, 4, 8, 16\}$

5.3 Further Speedup and Efficiency Calculations

The final scalability tests involved varying either the size of the matrix or the thread count t whilst the other variable was fixed, with a constant fixed precision value p to see how this affected speedup (Table 2) and efficiency (Table 3) measurements. The below table cells have been coloured to indicate a particularly **low** or **high** value of speedup or efficiency respectively, to make it easier to see the trend in how both values vary across varying matrix dimensions and thread counts. **Values have been calculated using the raw data in Appendix B.6.**

The speedup value using n processors, S_n, is calculated as: $\frac{T_s + T_p}{T_s + \frac{T_p}{n}}$ where T_s and T_p account for the sequential and parallel execution times respectively [10]. The efficiency value using n processors, E_n, is calculated as $\frac{S_n}{n}$. It is clear from these experiments that the value of n which produces the largest E_n (i.e. $n = 1$) is not the same value of n which produces the maximum S_n (in this case $n = 16$ using *Balena*). A careful consideration is needed by system designers depending on whether efficient use of hardware or reduced execution time is a priority. The rationale behind the equal division of the matrix between threads, the thread pool approach and the minimised use of mutexes was to parallelise the problem as efficiently as possible. It is clear that this efficiency only shown in sufficiently large problem sizes. Matrix dimensions of $d < 100$ don't show much benefit from this effort. In fact, for $d \leq 50, t \geq 13$, $S_n < 1$ which means the increase in overheads by increasing t has actually made it quicker to solve this size of problem sequentially at precision 0.2. For a fixed problem size there is indeed a natural limit on speedup, confirming Amdahl's Law.

It is only once the dimension d increases beyond 1000×1000 that the benefits of parallelism are observed in the form of larger speedups whilst more threads are executed in parallel across t cores (1 thread per core), confirming Gustafson's Law. For example, the greatest value of total speedup observed was 10.247 using 15 threads on a 15000×15000 matrix where the problem size is very large - $\sim \frac{(15000-2)^2}{15}$ cells are given to each thread to relax.

Table 2: Speedup across range of thread counts ($t = \{1, ..., 16\}$) and matrix dimensions ($d = \{10, ..., 20000\}$), with $p = 0.2$.

	10	50	100	250	500	1000	2500	5000	10000	15000	20000
1	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
2	1.000	1.000	1.250	1.556	1.793	1.790	1.844	1.903	1.840	1.891	1.929
3	1.000	1.000	1.667	2.000	2.476	2.495	2.646	2.613	2.694	2.705	2.752
4	1.000	1.000	1.667	2.333	2.971	3.108	3.319	3.430	3.378	3.496	3.547
5	1.000	1.000	1.667	2.800	3.467	3.127	4.028	4.088	4.054	4.216	4.232
6	1.000	1.000	1.667	2.800	3.852	3.734	4.618	4.740	4.817	4.918	5.024
7	1.000	1.000	1.250	3.111	4.160	4.289	5.275	5.379	5.404	5.569	5.647
8	1.000	1.000	1.667	3.111	4.522	4.806	5.787	5.967	6.045	6.301	6.308
9	1.000	1.000	1.250	3.500	4.727	5.296	6.324	6.545	6.484	6.889	6.900
10	1.000	1.000	1.250	3.500	4.952	5.767	6.800	7.087	7.077	7.519	7.426
11	1.000	1.000	1.250	3.500	5.200	6.179	7.227	7.610	7.590	8.092	8.089
12	0.667	1.000	1.250	3.500	5.474	6.570	7.623	8.077	8.158	8.662	8.631
13	0.500	0.750	1.000	3.500	5.474	6.920	7.988	8.546	8.613	9.214	9.146
14	0.667	0.750	1.250	3.500	5.778	7.310	8.348	8.995	9.096	9.719	9.535
15	0.667	0.750	1.250	3.111	5.778	7.632	8.810	9.450	9.528	10.247	9.926
16	0.667	0.750	1.000	3.111	6.118	8.238	8.265	9.396	8.352	10.013	10.204

In support of Gustafson, the general trend seen in Table 3 is that a growing problem size d causes a general increase in efficiency.

Table 3: Efficiency (%) across range of thread counts ($t = \{1, ..., 16\}$) and matrix dimensions ($d = \{10, ..., 20000\}$), with $p = 0.2$.

	10	50	100	250	500	1000	2500	5000	10000	15000	20000
1	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
2	50.0	50.0	62.5	77.8	89.7	89.5	92.2	95.1	92.0	94.6	96.4
3	33.3	33.3	55.6	66.7	82.5	83.2	88.2	87.1	89.8	90.2	91.7
4	25.0	25.0	41.7	58.3	74.3	77.7	83.0	85.8	84.5	87.4	88.7
5	20.0	20.0	33.3	56.0	69.3	62.5	80.6	81.8	81.1	84.3	84.6
6	16.7	16.7	27.8	46.7	64.2	62.2	77.0	79.0	80.3	82.0	83.7
7	14.3	14.3	17.9	44.4	59.4	61.3	75.4	76.8	77.2	79.6	80.7
8	12.5	12.5	20.8	38.9	56.5	60.1	72.3	74.6	75.6	78.8	78.8
9	11.1	11.1	13.9	38.9	52.5	58.8	70.3	72.7	72.0	76.6	76.7
10	10.0	10.0	12.5	35.0	49.5	57.7	68.0	70.9	70.8	75.2	74.3
11	9.1	9.1	11.4	31.8	47.3	56.2	65.7	69.2	69.0	73.6	73.5
12	5.6	8.3	10.4	29.2	45.6	54.8	63.5	67.3	68.0	72.2	71.9
13	3.9	5.8	7.7	26.9	42.1	53.2	61.5	65.7	66.3	70.9	70.3
14	4.8	5.4	8.9	25.0	41.3	52.2	59.6	64.3	65.0	69.4	68.1
15	4.4	5.0	8.3	20.7	38.5	50.9	58.7	63.0	63.5	68.3	66.2
16	4.2	4.7	6.3	19.4	38.2	51.5	51.7	58.7	52.2	62.6	63.8

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors, *Readings in Computer Architecture*, pages 79–81. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-539-8. URL `http://dl.acm.org/citation.cfm?id=333067.333076`.

[2] E. T. Anderson, Edward Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *Performance Evaluation Review*, 17:49–60, 04 1989. doi: 10.1145/75108.75378.

[3] Steven Brawer. *Introduction to Parallel Programming*. Academic Press Professional, Inc., San Diego, CA, USA, 1989. ISBN 0-12-128470-0.

[4] D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley professional computing series. Addison-Wesley, 1997. ISBN 9780201633924. URL `https://books.google.co.uk/books?id=_xvnuFzo7q0C`.

[5] Steve Carr, Jean Mayo, and Ching-Kuang Shene. Race conditions: A case study. In *Race Conditions: A Case Study*, 09 2002.

[6] Peter J. Denning. Thrashing: Its causes and prevention. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 915–922, New York, NY, USA, 1968. ACM. doi: 10.1145/1476589.1476705. URL `http://doi.acm.org/10.1145/1476589.1476705`.

[7] Kenneth W. Dritz and James M. Boyle. Beyond "speedup": Performance analysis of parallel programs. *University of North Texas Libraries*, 02 1987.

[8] H. El-Rewini and M. Abd-El-Barr. *Shared Memory Architecture*, chapter 4, pages 77–102. Wiley-Blackwell, 2005. ISBN 9780471478386. doi: 10.1002/0471478385.ch4. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/0471478385.ch4`.

[9] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, pages 124–131, New York, NY, USA, 1983. ACM. ISBN 0-89791-101-6. doi: 10.1145/800046.801647. URL `http://doi.acm.org/10.1145/800046.801647`.

[10] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31:532–533, 1988.

[11] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. *ACM SIGPLAN Notices*, 47, 04 2012. doi: 10.1145/2150976.2150997.

[12] David Padua, editor. *Law of Diminishing Returns*, pages 1006–1006. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_2185. URL `https://doi.org/10.1007/978-0-387-09766-4_2185`.

[13] Mark Syer, Bram Adams, and Ahmed E. Hassan. Identifying performance deviations in thread pools. In *Identifying performance deviations in thread pools*, pages 83–92, 09 2011. doi: 10.1109/ICSM.2011.6080775.

A Compiling and Running the `sharedrelax.c` program

Compiling the source code can be done with the gcc compiler (recommended). Once compiled into a binary file `sharedrelax`, the program is executable with customisable CLI parameters. Arguments are parsed with the getopt library[3]. An example compilation and run of the program might look like:

```
$ gcc -std=gnu99 -Wall -pthread sharedrelax.c -o sharedrelax
$ ./sharedrelax -d 1000 -t 16 -p 0.5 -v
```

The above runs the program using a 1000×1000 matrix with a precision of 0.5 across 16 threads, with verbose mode enabled.

Table 4: Program command line arguments.

Name	Description	Type	Default Value
`d`	Matrix dimension	`Integer`	`50`
`p`	Precision	`Double`	`0.1`
`t`	Number of threads	`Integer`	`1`
`v`	Verbose mode	`Boolean`	`false`
`i`	Info mode	`Boolean`	`false`

Verbose mode (`-v`) prints details such as how workload is split between threads. Info mode (`-i`) just prints the state of the matrix after each iteration as well as the number of iterations it took to relax.

Other acceptable uses:

```
$ ./sharedrelax -d 15 -p 0.2 -t 1 -v
$ ./sharedrelax -d 15 -p 0.2 -t 1 -i
$ ./sharedrelax -d 125 -p 0.7 -t 5
$ ./sharedrelax -d 1000 -p 0.6 -t 20 -i
$ ./sharedrelax
```

A check is in place to validate whether $t > (d-2)^2$, as this is invalid in accordance with the way work is divided amongst threads using my approach. A minimum of 1 cell must be allocated to each thread.

[3] `https://www.gnu.org/software/libc/manual/htmlnode/Getopt.html`

B Testing Data and Results

B.1 Automated Correctness Testing for $t = \{2, ..., 50\}$, $d = \{10, ..., 1000\}$, $p = 0.1$

Dimension (d)	Precision (p)	Threads (t)	Repetitions	Result
10	0.1	2	100	Pass
10	0.1	5	100	Pass
10	0.1	10	100	Pass
10	0.1	15	100	Pass
10	0.1	20	100	Pass
10	0.1	30	100	Pass
10	0.1	50	100	Pass
20	0.1	2	100	Pass
20	0.1	5	100	Pass
20	0.1	10	100	Pass
20	0.1	15	100	Pass
20	0.1	20	100	Pass
20	0.1	30	100	Pass
20	0.1	50	100	Pass
50	0.1	2	100	Pass
50	0.1	5	100	Pass
50	0.1	10	100	Pass
50	0.1	15	100	Pass
50	0.1	20	100	Pass
50	0.1	30	100	Pass
50	0.1	50	100	Pass
100	0.1	2	100	Pass
100	0.1	5	100	Pass
100	0.1	10	100	Pass
100	0.1	15	100	Pass
100	0.1	20	100	Pass
100	0.1	30	100	Pass
100	0.1	50	100	Pass
200	0.1	2	100	Pass
200	0.1	5	100	Pass
200	0.1	10	100	Pass
200	0.1	15	100	Pass
200	0.1	20	100	Pass
200	0.1	30	100	Pass
200	0.1	50	100	Pass
500	0.1	2	100	Pass
500	0.1	5	100	Pass
500	0.1	10	100	Pass
500	0.1	15	100	Pass
500	0.1	20	100	Pass
500	0.1	30	100	Pass
500	0.1	50	100	Pass
1000	0.1	2	100	Pass
1000	0.1	5	100	Pass
1000	0.1	10	100	Pass
1000	0.1	15	100	Pass
1000	0.1	20	100	Pass
1000	0.1	30	100	Pass
1000	0.1	50	100	Pass

B.2 Time Taken (s), Speedup and Efficiency (%) for for $t = \{1, ..., 16\}$, $d = 1000$, $p = 0.05$

Threads	Time (s)	Speedup	Efficiency (%)
1	5.350	1.000	100.00
2	2.838	1.885	94.26
3	1.958	2.733	91.10
4	1.512	3.539	88.46
5	1.238	4.323	86.46
6	1.049	5.102	85.03
7	0.912	5.864	83.78
8	0.809	6.616	82.70
9	0.741	7.217	80.19
10	0.664	8.054	80.54
11	0.621	8.616	78.32
12	0.570	9.392	78.27
13	0.537	9.963	76.64
14	0.504	10.623	75.88
15	0.479	11.170	74.47
16	0.463	11.564	72.28

B.3 Time Taken (s), Speedup and Efficiency (%) for $t = \{8, ..., 32\}$, $d = 1000$, $p = 0.05$

Threads	Time (s)	Speedup	Efficiency (%)
8	0.809	6.614	82.67
10	0.663	8.070	80.70
12	0.569	9.403	78.36
14	0.503	10.637	75.98
16	0.457	11.708	73.17
18	0.753	7.105	39.47
20	0.737	7.260	36.30
22	0.730	7.329	33.31
24	0.738	7.250	30.21
26	0.742	7.211	27.73
28	0.733	7.299	26.07
30	0.735	7.279	24.26
32	0.739	7.240	22.62

B.4 Time Taken (s) for $t = \{1, 4, 8, 16\}$, $d = \{500, ..., 10000\}$, $p = 0.05$

Dimension d	Time (s) for $t = 1$	Time (s) for $t = 4$	Time (s) for $t = 8$	Time (s) for $t = 16$
500	0.863	0.249	0.144	0.097
1000	5.348	1.513	0.808	0.456
2000	27.857	7.754	4.048	2.250
3000	54.388	15.416	8.203	4.641
4000	107.513	30.051	15.789	9.027
5000	171.704	47.180	25.027	15.384
6000	234.639	64.201	34.260	20.415
7000	403.821	113.377	59.162	37.415
8000	530.570	145.767	75.899	46.737
9000	*Timeout*	172.161	90.340	55.666
10000	*Timeout*	200.526	104.710	64.832

From this, the speedup and efficiency can be calculated:

Dimension d	Speedup for $t = 1$	Speedup for $t = 4$	Speedup for $t = 8$	Speedup for $t = 16$
500	1.000	3.466	5.993	8.897
1000	1.000	3.535	6.619	11.728
2000	1.000	3.593	6.882	12.381
3000	1.000	3.528	6.630	11.719
4000	1.000	3.578	6.809	11.910
5000	1.000	3.639	6.861	11.161
6000	1.000	3.655	6.849	11.493
7000	1.000	3.562	6.826	10.793
8000	1.000	3.640	6.990	11.352
9000	1.000	*N/A*	*N/A*	*N/A*
10000	1.000	*N/A*	*N/A*	*N/A*

Dimension d	Efficiency (%) for $t = 1$	Efficiency (%) for $t = 4$	Efficiency (%) for $t = 8$	Efficiency (%) for $t = 16$
500	100.00	86.65	74.91	55.61
1000	100.00	88.37	82.74	73.30
2000	100.00	89.81	86.02	77.38
3000	100.00	88.20	82.88	73.24
4000	100.00	89.44	85.12	74.44
5000	100.00	90.98	85.76	69.76
6000	100.00	91.37	85.61	71.83
7000	100.00	89.04	85.32	67.46
8000	100.00	91.00	87.38	70.95
9000	100.00	*N/A*	*N/A*	*N/A*
10000	100.00	*N/A*	*N/A*	*N/A*

B.5 Time Taken (s) for $t = \{1, 8\}$, $d = 1000$, $p = \{0.5, ..., 0.005\}$

Precision p	Iterations	Time (s) for $t = 1$	Time (s) for $t = 8$	Speedup	Efficiency (%)
0.500	5	0.118	0.034	3.471	43.382
0.400	8	0.151	0.042	3.595	44.940
0.300	14	0.246	0.054	4.556	56.944
0.200	32	0.519	0.097	5.351	66.881
0.100	95	1.491	0.248	6.012	75.151
0.075	146	2.271	0.365	6.222	77.774
0.050	328	5.347	0.808	6.618	82.720
0.025	1030	16.536	2.559	6.462	80.774
0.010	5333	84.118	13.666	6.155	76.941
0.005	18313	295.644	48.023	6.156	76.954

B.6 Time Taken (s) for $t = \{1, ..., 16\}$, $d = \{10, ..., 20000\}$, $p = 0.2$

	10	50	100	250	500	1000	2500	5000	10000	15000	20000
1	0.002	0.003	0.005	0.028	0.104	0.519	3.339	14.733	62.029	175.406	265.536
2	0.002	0.003	0.004	0.018	0.058	0.290	1.811	7.743	33.704	92.738	137.685
3	0.002	0.003	0.003	0.014	0.042	0.208	1.262	5.639	23.026	64.837	96.477
4	0.002	0.003	0.003	0.012	0.035	0.167	1.006	4.295	18.360	50.168	74.867
5	0.002	0.003	0.003	0.010	0.030	0.166	0.829	3.604	15.299	41.600	62.742
6	0.002	0.003	0.003	0.010	0.027	0.139	0.723	3.108	12.877	35.664	52.856
7	0.002	0.003	0.004	0.009	0.025	0.121	0.633	2.739	11.478	31.496	47.020
8	0.002	0.003	0.003	0.009	0.023	0.108	0.577	2.469	10.262	27.839	42.098
9	0.002	0.003	0.004	0.008	0.022	0.098	0.528	2.251	9.567	25.460	38.484
10	0.002	0.003	0.004	0.008	0.021	0.090	0.491	2.079	8.765	23.329	35.758
11	0.002	0.003	0.004	0.008	0.020	0.084	0.462	1.936	8.173	21.676	32.826
12	0.003	0.003	0.004	0.008	0.019	0.079	0.438	1.824	7.603	20.249	30.764
13	0.004	0.004	0.005	0.008	0.019	0.075	0.418	1.724	7.202	19.037	29.033
14	0.003	0.004	0.004	0.008	0.018	0.071	0.400	1.638	6.819	18.048	27.848
15	0.003	0.004	0.004	0.009	0.018	0.068	0.379	1.559	6.510	17.117	26.751
16	0.003	0.004	0.005	0.009	0.017	0.063	0.404	1.568	7.427	17.518	26.022

Calculated speedup and efficiency values using this data can be seen in Section 5.3.

C Automated Correctness Test Script

```
#!/bin/bash

# Initialise variables
prec=0.1
repeat=100
nthreads=(2 5 10 15 20 30 50)
dims=(10 20 50 100 200 500 1000)
pass=1

for dim in "${dims[@]}"
do
    # Set expected output for current dimension (using t=1 as benchmark)
    ./sharedrelax -d $dim -t 1 -p $prec -i > in.txt

    # Check output matched by increasing number of threads
    for tcount in "${nthreads[@]}"
    do
        # Repeat each test to ensure some confidence in reproducibility
        for (( i=1; i<=$repeat; i++ ))
        do
            ./sharedrelax -d $dim -t $tcount -p $prec -i > out.txt

            # If files differ, failure!
            if [[ $(diff in.txt out.txt) ]]; then
                pass=0
                echo "FAILED TEST for d=$dim, t=$tcount"
            fi
        done
        # Else if no trials fail, print success for this test case
        if [ "$pass" -eq "1" ]; then
            echo "PASSED TEST for d=$dim, t=$tcount"
        fi
    done
done

if [ "$pass" -eq "1" ]; then
   echo "All passed.";
   exit;
fi
```