Parallel Relaxation on a Shared Memory Architecture

November 15, 2023

Contents

1 Introduction

The Relaxation Technique is a technique used for solving differential equations, by taking elements and replacing them by the average of their four neighbours. Boundary elements don't get overwritten. These values will converge, and this program will run until they converge to a level of precision we will set as a parameter. This document is designed to be read in conjunction with the fully-commented source code, which can be found in the `main.c` file that came with this submission. Code listings throughout are kept to a minimum, and the document tries to expand upon the comments to explain the theory, testing, and measurements made with the code.

2 Algorithm Design

2.1 Bulk Synchronous Parallel

To design this program, we should start off by considering the core computations that could be parallelised. To do this, we can think about what computations are stateless, and could be written in a functional-style, as pure functions are inherently parallelisable. This is because pure functions have no state, and so different function invocations shouldn't have any race conditions that can exist between them.

This problem at a higher-level does have state. Ultimately, each step of computation depends on the previous step, and we can only move on when all the neighbours of an element have been computed and are up-to-date before we compute the next value of the element.

To break this problem down into parts that don't share state, we can break it into time steps where at each time step T, all the values in the array have been computed correctly together, given a correct array for time step $T-1$. Our problem then becomes an iterative problem, where:

$$A_0 = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

$$A_n = some\_computation(A_{n-1})$$

With this equation, we can actually model each time step T as a superstep of computation, fitting the Bulk Synchronous Parallel model of computation, proposed by Valiant (1990). This suggests that later in our implementation, barriers may be a good choice of synchronisation.

This gives us the high-level overview, but we haven't actually decided on what *some_computation* is, and whether it is truly parallelisable and pure. To do this, we can think about how this problem might be solved sequentially with Algorithm 1.

Algorithm 1: Computing matrix A_n from A_{n-1}

Data: A_{n-1}
Result: A_n
for *each unfixed element in* A_{n-1} **do**
    get the neighbours of element;
    average them;
    write the result to the element position in A_n;

It is important to realise that this for-loop does not need to be executed in order, as it writes to a different matrix, A_n than it reads from, A_{n-1}. We don't need to start at the first element and end at the last element. That's what makes this for-loop a great candidate to be executed in parallel. To perform it in parallel, on an ideal machine, we would take *'each unfixed element in* A_{n-1}, and perform the body of the loop in separate, parallel processes.

2.2 The State

There is a clear condition that needs to be fulfilled when using this algorithm. That is to compute an element e_n in A_n, all the neighbours of e_{n-1} must be ready. This is the state that must be shared, or at least communicated, between processes. We could handle this by waiting on the neighbour elements to be computed and having those processes send a signal when they've finished computing. When the waiting process has received all four signals, it can compute the new element.

2.2.1 The Practicality

Unfortunately this approach isn't practical to implement, or run on a real machine. Will the signals include data about which time step has been computed? What if the four neighbours are ready before the waiting thread and send the signals too early?

For this to work, an immense amount of thought and detail would need to go into solving it. Is it possible that one value could go n time steps ahead, meaning that we would need to store N arrays each with size $m \times m$ in memory? $O(Nm^2)$ memory is a lot of memory as m gets larger and larger at scale.

This approach leans more and more into requiring proofs to solve efficiently, and once implemented, may be an inefficient solution anyway, due to all the messaging that must be done in between the simple calculations of "average four numbers". That is if a reliable and robust solution could even be developed in code that is still understandable.

2.2.2 Barriers in Action

Fortunately, there is an alternative solution which we can use. It is to use *barriers* to syncronise all the threads at the end of each time step T. This solution is a lot simpler to think about and implement, and the simplicity will allow us to consider flaws in our idea, and will help us constrain the amount of memory we use in our implementation, as the cost of *potentially* some speed.

Barriers are the only concurrency primitive that is used in this implementation, but we borrow ideas from other primitives later on to protect writing to shared state. Barriers are great as they are simple to use, simpler to reason about, and are clear memory walls, telling the compiler and hardware to maintain reads and writes such that they fall within the walls.

3 Implementing the Algorithm

3.1 The Naïve Approach

The approach that seems easy to start off with, is to parallelise the for-loop, do convergence checking, and then repeat if the matrix has not converged to our precision. This is the approach described by Algorithm 2.

Algorithm 2: Basic Parallelisation

Data: m, the size of the array
while *not converged* **do**
    for *i from 0 to* $(m-2)^2$ **do**
        start a thread to process e_i using Algorithm 3;
    update convergence;

Algorithm 3: Per Thread Computation

get the neighbours of $A_T[i]$: ($A_T[i+1]$, $A_T[i-1]$, $A_T[i+m]$, $A_T[i-m]$);
$A_{T+1}[i] = average$(neighbours);
$D[i] = abs(A_{T+1}[i] - A_T[i])$;

Where in Algorithm 3, D is an array keeping track of differences between time step T and $T-1$, and A_{T+1} is the next array to write to. We later optimise this algorithm, replacing D with a more parallel approach, and compare them in our scalability investigation.

This approach has many flaws when running on a real machine. The first flaw is that we start and destroy threads every single iteration. Creating threads in not free, and in some cases can be very expensive, especially when our main computation (the averaging of four numbers) is not very expensive[1]. We can solve this by reusing our threads, and adding our syncronisation technique.

The second problem is that we spawn $(m-2)^2$ threads, which will grow as the size of the matrix grows.

m	**Number of Threads**
10	$(10-2)^2 = 64$
100	9604
1000	996,004
10000	100 million threads

Table 1: Table showing that the number of threads spawned follows $\theta(m^2)$

This shows that if you wanted to solve a 10,000-sized square matrix, which is a small target for such a highly parallel solution, you could be spawning 100 million threads, every single iteration if you don't think about what you are doing.

This would actually slow our program down due to *context switching*. This is where a machine may only have a restricted number of physical processing cores that can run simultaneously, and so it must manage many virtual threads by waking them up, putting them to sleep, and switching between them all frequently, which are all expensive to do. This means that it would actually likely be faster to spawn less threads, (a number close to how many cores the machine has) than to have more. Later on in the document we analyse the data (such as in Figure 6) to determine if this is true for our implementation.

[1]The averaging of four double values can be expensive depending on hardware support for doubles, but so is calling any operating system function from a process

3.2 Writing the Algorithm

It is clear we can't just spawn threads infinitely, and we should test the performance of our code on many different thread counts, but that means when we write our solution, we need to think about how we split the data across the threads, as there may now be less threads than data points.

3.3 Load Balancing

Load Balancing is the idea of splitting a workload in some way across different resources. In our case, load balancing is the idea that regardless of our data size, our threads should have about the same amount of work to do each. The way this implementation performs load balancing was to keep track of the upper bound of elements each thread would need to have, and then assigning the lower bound until the rest of the data could be divided exactly into the upper bound by the remaining threads.

3.3.1 Load Balancing Example

For example, on a 10×10 matrix, the number of elements would be 64. If we decided to use 44 threads, the upper bound would be 2 elements per thread. The first 24 threads would take one element each, leaving 40 elements left to process, and 20 threads left to do it. As 40 / 20 = 2 (the upper bound), the remaining threads all take two elements each, leading each thread to have either one or two elements to process. This approach scales in the same way. This is show in Listing 1.

Listing 1: Load Balancing (per thread)

```
int max_could_allocate = threads_remaining * elements_per_thread;
int divides_exactly = max_could_allocate == data_points_remaining;

// to account for the fact that elements_per_thread is an
// overestimate, we subtract one unless it divides exactly
// end_index is the previous thread's end_index
int start_index = end_index;
end_index = start_index + elements_per_thread - !divides_exactly;
```

3.4 Infinite Memory

The theoretical algorithm doesn't consider the amount of memory used, or how long it takes to create such large matrices in the first place. A far more efficient way to do this, is to only have two matrices, as we don't care about producing every intermediate time step[2]. This implementation would then have a matrix we read from (the previous time step) and write to (computing the current time step). After every iteration, we swap the references of these two matrices after doing the convergence check, (which should be a protected operation as the pointers will be shared state) and do the processing again. This is shown in Figure 1.

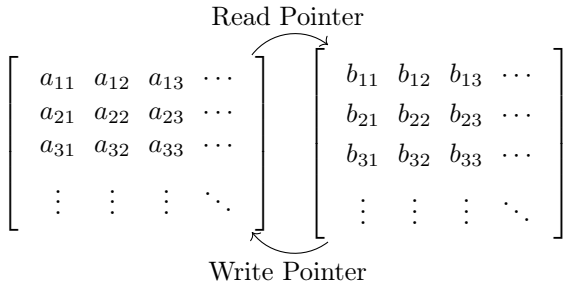

Figure 1: Swapping the pointers after each iteration

[2]Later on for testing correctness, printing every step will be useful, and we can add this in by printing at each iteration rather than keeping all the matrices in memory

3.5 Convergence Checking

The next problem to be solved was how should the convergence of all the values be checked. The implementation that this code uses is something that could be considered as one of the main bottlenecks of the program. During our scalability investigation, we can see if this is the case. The way this implementation works, is that when a number of threads is specified, the program spawns $n - 1$ threads, and the main thread becomes a special worker thread known as the 'coordinator thread', that takes on some of the compute, with some added responsibility.

The *coordinator thread* is exclusively responsible for doing convergence checking and writing to a convergence flag, representing whether all values in the matrix have converged. The bottleneck here is that the coordinator is the only thread that can do this, regardless of which thread arrives first. It also has to perform a linear search $\theta(m^2)$ elements in the worst case, each iteration.

An alternative implementation could allow any thread to check convergence, through the use of a *mutex* around the flag, and some way of scheduling (like a *signal*) that a thread has gone and done the check, but the potential speedup gained wouldn't be worth the additional overhead, as only one thread would still be checking $\theta(m^2)$ elements.

Figure 2 shows a high-level overview of the current implementation.

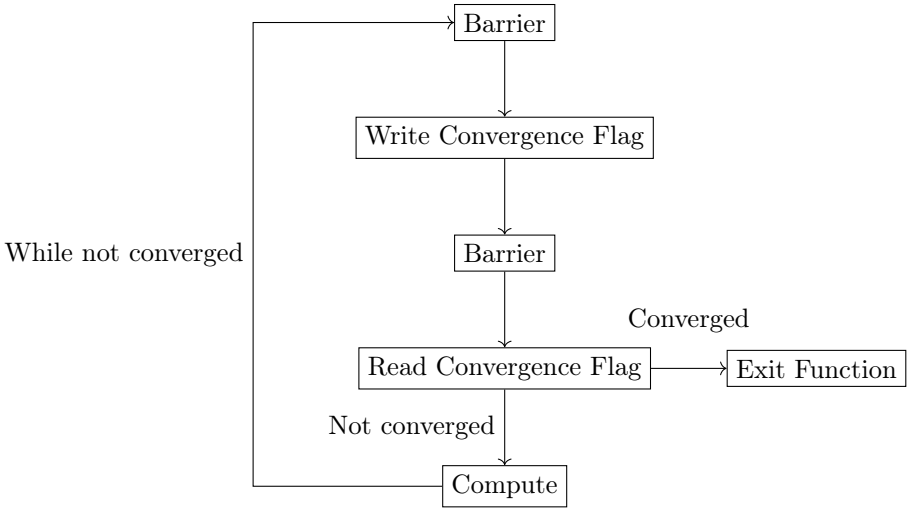

Figure 2: High Level Overview of Convergence Checking

Note that how 'Exit Function' works is that any non-main thread will terminate, and the main thread will leave and continue execution. This led to a very rare, but fatal bug in the program which we will look at later.

This implementation is fine; it contains no *shared mutable state*, as only the one thread can ever write to this flag, and the other threads cannot read from it until it is set, as the second barrier acts as a memory wall. The compiler and hardware should not pre-load the convergence flag for each thread before the barrier has been broken, which requires the main thread too. Relying on one thread shouldn't slow the program down by a noticeable amount, as all threads still need to be ready to perform the next computation step, despite which thread checks for convergence.

3.5.1 Better Implementation

An alternative **implementation detail** for the convergence checking however, could speed up the program significantly. Initially, what happened is that the main thread iterated through the 'differences' array, and if any differences were larger than the precision, we can say the converged flag is false. This means that the main thread is the only thread that can perform the comparisons between the old values and the new values, which is a clearly parallelisable operation. If we change our double array to a 'boolean' array, each thread can check for convergence when its done, and then main can just do a 'boolean' (integer) comparison. In the scalability investigation, we will compare the speedups of these two approaches to determine if it truly makes a difference.

3.5.2 Better Algorithm

A superior **algorithm** could go a step further and realise that after the first barrier, all the threads are ready to do some processing, so why don't we take half the threads, and compare the values in a tree-like structure, which would mean we do $log_2(n)$ parallel comparisons, rather than n sequential ones, as shown in Figure 3. This difference would become larger and larger for more threads.

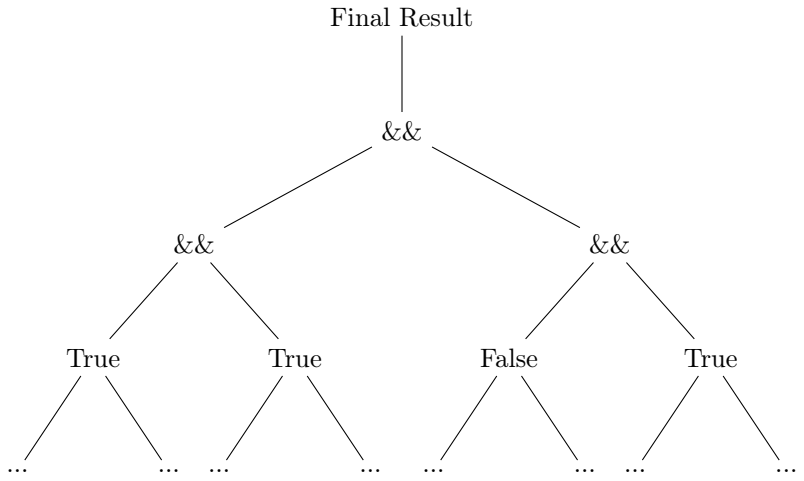

Figure 3: An example demonstrating how a convergence check could be parallelised. In this example, 'converged' would be False due to the right tree having a 'False'.

As with everything not *pathologically parallel*[3], there are problems with this. The main ones are:

1. We cannot exit early without some signalling mechanism, which may be slow for most iterations
2. We must have a way to decide which thread continues processing each time
3. This is (again) much harder to implement

Something to note is that with SIMD instruction sets (which are discussed in later in section 5.1 the sequential version could go extremely fast, and they could also be used in conjunction with the parallel algorithm for further speedup (and complexity).

3.5.3 The `pthread` API

Something that should be considered, is how the POSIX library somewhat forces you to write output to a structure if you want to 'return' values without killing the thread. In other languages, such as Go, channels can be created which are essentially streams of data where the thread can just add to the channel when a values has been computed, and a consumer of that channel can be notified and use the new value, without killing the thread. In POSIX, the way to return values is to pass them into the function `pthread_exit()`, which can be retrieved by another thread through `pthread_join`. The downside is that `pthread_exit()` does terminate the thread (The Open Group, 2017), so if you want to have long-living threads, you might decide to modify a shared resource (the struct reference passed in) instead, which is more dangerous code.

[3] ***Pathologically parallel*** is a term used to describe processes that are very easily parallelisable (Arm Developer, n.d.), such as ray tracing, where each pixel can be rendered completely independently of one another, also referred to as embarrassingly parallel, delightfully parallel, and others

4 Debugging the Program

4.1 Pathologically Buggy

Something else that is worth talking about is the debugging of parallel programs as opposed to the debugging of sequential ones. Over 50% of the time developing this code was spent purely on debugging the program and trying to find where it went wrong, why it went wrong, and what could be done to fix it. The debugging time is what produced the emotional attachment to this piece of code, and makes us far more grateful for languages designed around parallel safety.

These bugs ranged from small and noticeable, to large, rare, and hard to track down. Some of the smaller bugs included things such as forgetting that we pass a pointer to each thread, and so having *shared mutable state* between threads, as the start and end indices of each thread would be updated rather than each thread getting its own copy. Modern languages like *Rust* would attempt to prevent issues like this by making the passing of shared mutable state a compile-time error.

There were off-by-one errors too, since treating the main thread as a worker thread meant that there was duplicated code, so some variables were set differently leaving to having an incomplete `ThreadData` structure for main. This led to the creation of `initialise_thread_data()` so that the code could be 'type checked' at compile-time.

There was also a maximum size of array that could fit on the stack, and so the array had to be allocated on the heap for large enough matrices (approximately 7000^2 elements). However the most interesting bug was to do with having the coordinator thread, and making an assumption that wasn't true in parallel.

4.2 Main, the 'Safe' Coordinator

The most subtle, rare, and painful bug to track down was this one. It is a true example of a bug that would only happen in parallel, and so it gets its own section. This bug brings awareness to the edge cases that can occur in parallelism, and makes us realise that it's not always worth trying to be clever to save time. It's the bugs like these that bring some developers to just put a lock around everything and hope it works out. This bug only appeared once every 20 or so runs, and never once happened when running with a debugger.

To describe what was happening, we need to take a closer look at Figure 2. Here is that figure in context of the whole program as shown in Figure 4.

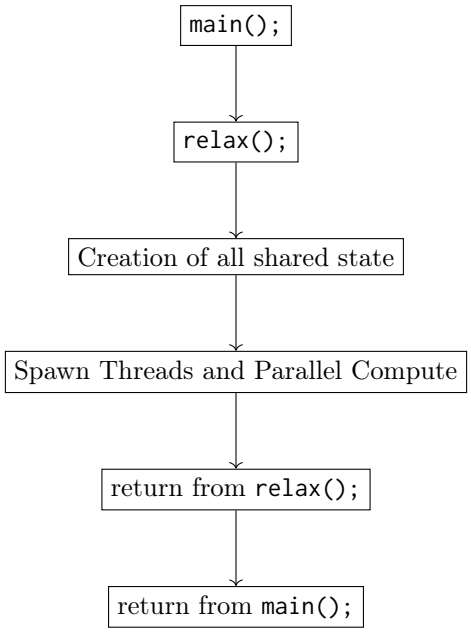

Figure 4: High level overview of the full implementation, Figure 2 is represented by 'Spawn Threads and Parallel Compute'

The bug itself was simple, a bad access on some variable, somewhere. To debug, we can turn off compiler optimisations, and use print statements to track down the variable being accessed. After many, many runs, waiting for the bad access to happen, we converge on a particular point in the program. The bug was happening after `relax();` had finished, but before the print statement that prints the result.

```
double **result = relax(...);
// error was happening exactly here on this line
 print_sqaure_matrix(result, size);
```

There is no code in between these two calls, so what could be happening?

What was really happening was as follows:

1. Create a boolean (int) on the call stack to track whether the matrix has converged
2. Share that boolean to each thread, so each thread can read it
3. The main thread at some point it sets it to true
4. Main then opens the second barrier, allowing all the threads to read the flag and exit

There is a special case in the algorithm above. If the main thread is able to unlock the barrier, read the convergence flag, and return from the call to `relax()` before one of the threads has read the flag, then the stack frame of `relax()` is popped, and the flag is no longer a valid address of memory to read from. If any other thread is too slow to read this flag for whatever reason, it may cause a bad access and crash the program.

The initial solution was to just allocate the flag on the heap instead, but to be safe programmers we should free unneeded memory, which must be done in `relax()` as we were already returning something else. We add some `pthread_join()` calls in a loop to safely wait for all the threads and free it, only to realise if we kept those calls, we can leave the flag on the stack, as the stack frame won't be popped until all the threads have already read the flag and exited.

5 Further Optimisations

5.1 SIMD Architecture

The processor we are using is the The Xeon Platinum 8168 Skylake Processor, from Intel. The specification (Intel, n.d.) states that it supports Intel's *Instruction Set Extensions*: SSE4.2, AVX, AVX2, AVX-512. This means if our code is developed in a specific way, some of the operations will be compiled down to parallel instructions. It is a compiler optimisation that can be performed on loops that use simple instructions and are easily identifiable as SIMD loops. This is called *auto-vectorization* in GCC, and some examples and information can be found on the GNU project page (GCC Team, 2023).

These instruction set extensions essentially allow a compiler to transform code into SIMD operations that operate on specific sizes of vector. For example:

```
int n = size_of_array;
for (int i = 0; i < n; i++) {
    numbers[i] = 2 * numbers[i];
}
```

may get transformed into a single SIMD instruction representing:

$$\begin{bmatrix} a \\ b \\ \vdots \\ n \end{bmatrix} = 2 * \begin{bmatrix} a \\ b \\ \vdots \\ n \end{bmatrix}$$

as opposed to n separate, sequential instructions. When you compile code with the following compile statement, you get some information about what vectorisation the compiler has performed.

```
XXXXX@cm30225-login $ gcc -O3 -ftree-vectorize -fopt-info-vec -o exec source.c -pthread -lm

source.c:68:1: optimized: basic block part vectorized using 16 byte vectors
source.c:80:9: optimized: basic block part vectorized using 16 byte vectors
source.c:275:5: optimized: basic block part vectorized using 16 byte vectors
source.c:275:5: optimized: basic block part vectorized using 16 byte vectors
```

Unfortunately, this doesn't actually mean that any of the loops were vectorised, simply that some sequence of statements forming a 'basic block' were 'part vectorised'. Running a grep command to number our lines, we can see that in the code the only thing being vectorised, is the assignment to `thread_data` and calling `set_thread_data()`. It is possible to assist the compiler in performing this optimisation for the loops, however since it doesn't do it for 'simple' loops, such as initialising the matrices, it would require changing the code quite a bit.

5.2 Skipping Zeros

This implementation doesn't consider the fact that for matrices where the boundary elements are fixed at ones and zeros, and likely other forms, many of the values don't need to be computed. If all of a values neighbours are equal to zero, then it will be equal to zero in the next iteration. The average doesn't need to be computed.

5.3 Unnecessary Parallelism

There are other things that could be parallelised in this program. The initialisation of matrices (unmeasured), the averaging of the four values (way too small to be useful, and extremely expensive), and things like the convergence check mentioned earlier 3. This implementations aims to parallelise as much as it can, whilst keeping the code running fast, maintainable, and doesn't do anything necessary. The results later on, in the scalability investigation, show that the results are impressive without doing highly-intricate and advanced parallel programming.

6 Correctness Testing

There's a variety of approaches for testing the correctness of the program. There is a sequential version of the code (completely unparallel, doesn't include pthreads at all), which can be used that to generate files as the source of truth. The sequential code was checked manually for two small arrays and precision 0.1. The `print_square_matrix()` function was initially called every iteration, allowing intermediate steps to be checked. Both the parallel and the sequential versions of the code can print the number of iterations that have occurred, to ensure that the same amount of steps are taken.

To ensure that the testing was as robust as possible, we need a large dataset. The dataset should be varied, demonstrating edge cases, a variety of inputs, but not invalid inputs. We don't need to test how a negative size is allowed, we are measuring the reliability of our code on correct problems. Generating test examples, data, and more all turn out to be useful skills for generating scalability data too.

6.1 Gathering Data

To gather a substantial amount of data for testing, more than simple SBATCH scripts will be needed. Each script for a job should do more than one test, and generate data for varying thread counts, problem sizes, and precisions, as shown in Listing 2. The corresponding sources of truth can then be generated on a local machine. Files can be named with their metadata so that they can be compared automatically, for example, a file that had output for a 5000^2 matrix could be called `azure-5000-44-0.001000.txt`.

Listing 2: Example SBATCH commands

```
precisions=(0.1 0.01 0.001)
sizes=(1 10 100 1000 2746 3454 4973 5736 6633 7321 8743 9324 10000)
threads=(1 2 3 4 8 9 10 12 )

for i in "${sizes[@]}"; do
  for j in "${threads[@]}"; do
    for k in "${precisions[@]}"; do
        ./parallel-fine-ints "$i" "$j" "$k"
      done
    done
done
```

The C program will then be responsible for making timing measurements, and writing output to files with all the information needed for the data processing. The above example SBATCH script is a simplification of the process, as there are various fine-grained measurements and examples taken that each will take longer than 20-minutes to run. This led to the creation of many SBATCH scripts each with their own target points.

6.2 Python

After the code could write results, intermediate results, and iteration counts to a file, the files generated by the parallel code could then be compared with the files generated by the sequential (not single-threaded) code. For this we can develop a small Python script (see Appendix 3) that just goes through the output generated in parallel by the Azure machine, and the sequential output that was generated locally. It performs a diffing algorithm which is used to see if there are any differences between two files. If there are then we know something is different between our source of truth and the Azure output. This approach was very fast and practical until writing results of about 10000^2 elements, as the text files would simply become too large to finish writing on the supercomputer. For further testing, iteration counts and only the first few rows were used.

6.3 Metatesting

The testing could have been more rigorous, as there was no way to been sure that the sequential implementation was completely correct, and a bug may have been shared between the two. The diffing should be correct, however it's also possible that there are some edge cases, although some of my own files were correct and small changes in dummy files were picked up easily. For this, there is there is

an excerpt of a size 100×100, and a 5000×5000 result uploaded with this submission in `100.txt` and `5k.txt`, inside `testing.zip`. The smaller one is so that it can reasonably fit on a page, as the format is printed in matrix-style. These were both generated using a precision of 0.001, and using 44 threads on the Azure machine.

Here is a far smaller, 8×8 example:

1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
1.000000	0.952624	0.905579	0.855921	0.794173	0.696479	0.498335	0.000000
1.000000	0.905579	0.814960	0.725414	0.625777	0.494595	0.297522	0.000000
1.000000	0.855921	0.725414	0.607672	0.491596	0.360744	0.198346	0.000000
1.000000	0.794173	0.625777	0.491596	0.375519	0.261106	0.136599	0.000000
1.000000	0.696479	0.494595	0.360744	0.261106	0.174230	0.088423	0.000000
1.000000	0.498335	0.297522	0.198346	0.136599	0.088423	0.044047	0.000000
1.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

There is also the step-by-step calculations for the 8×8 matrix in `8x8-steps.txt`, inside `testing.zip`. This example is small so that each matrix and the steps can easily fit on one screen for clear viewing and verification.

6.4 Untested Behaviour

As mentioned above there are some behaviours that went untested. Some of these are more useful than others. The input matrices could be randomised, as long as they converge. The input matrices could contain negative numbers, as long as they converge. The input matrices may not need to be square. And so on. The testing done on this implementation was vigorous enough to not need to step out of the problem domain of having an $n \times n$ sqaure matrix, with fixed boundary elements of 1s and 0s.

7 Scalability Investigation

Measuring the scalability of parallel code is very important. If we our developing highly parallel code, we should be targeting really large problem sizes, and determining if our parallel solution does truly perform well on very large problems. For the scalability investigation, the single-threaded parallel data was considered to calculate metrics to make it a fairer test as it is certain the algorithm is the same. If interested, some of the CSV data can be found in the `data.zip` included with this upload.

7.1 Timing the Code

The code is timed in a way to give the most precise and meaningful times. This implementation has the timing purely around the `relax()` function, and is done so in the C code itself, using `time.h` with the `CLOCK_MONOTONIC` option. This is to avoid counting CPU cycles, which will multiply across the threads producing an incorrect real-time measurement. Counting CPU cycles may be more useful for calculating work efficiency, but this implementation focuses purely on seconds and milliseconds.

What **is** timed:

- Creation of an $(m-2)^2$ integer array for storing convergence booleans
- Creating the thread data, and spawning each thread
- The bulk of the processing, performing the algorithm

What **is not** timed:

- IO operations, opening files, printing results, counting iterations
- The creation of two m^2 matrices to read and write to
- Releasing the matrices from memory

There is also different sets of timings to measure how different properties impact the performance of our timings. These are the data sets produced:

- Parallel execution on a range of thread counts (including 1 thread), sizes
- Sequential execution on a range of sizes
- Compiler optimisations on and off for each of these
- Parallel execution on a fine range of thread counts (1 - 60, in steps of 1) for a 10000^2 matrix
- Parallel execution using an array of `int` to track convergence vs an array of `double` that records the differences (per-thread comparison to the previous array, vs the coordinating thread doing the comparison for each element)

7.1.1 Amdahl's Law and Gustafson's Law

The way that this code is being timed means that are a large proportion of our code is parallelisable. We run the code to initialise our thread data structures n times when we have n threads. Since n is a fixed value, we run this code in constant time in comparison to the problem size, m. We then run the parallelisable code once for each iteration. The iteration count is more related to the precision we choose rather than the size of the matrix, and we perform averaging for $(m-2)^2$ elements each iteration.

There is then some sequential code where the coordinating thread checks if all the elements have converged. The amount of steps is hard to say exactly due to the short-circuiting implementation, but it is very few in comparison there are many more steps in doing the reading and averaging of four double values, as opposed to reading if an array contains a 0 integer.

Amdahl's Law states that the natural speedup of the code is restricted by the proportion of the code that is actually running in parallel. This is true, and means that we expect on 44 core machine, that this code should run less than 44 times faster than it does on one thread as there is a sequential portion.

Gustafson's Law states that as our problem gets larger and larger, we spend more and more time doing parallel computation, and so the speedup should increase closer to the natural limit as our problem increases. For this code, that is also true, the setting of the threads only runs once, and the threads run many, many computations each.

7.1.2 Calculating Metrics

To calculate the metrices used by the later sections, a pandas (Python) script was developed that would process raw CSV files and add fill in missing information. That pandas script can be found in Appendix 4 for the precise calculations that were performed. Also note that when calculating metrics, this analysis uses the **single-threaded parallel implementation**. This is important as if we were developing actual sequential code, there may be things we could do to speed it up. The implementation of sequential code for this project didn't do anything special, and so using the parallel version is just an easier way to compute the data.

7.1.3 The Effect of Compiler Optimisations on our Data

In each case, we look at both compiler-optimised code, and unoptimised code. This is because we get two different stories from these separate sets of data. The compiler-optimised code graphs are telling us the true metrics we would get if we deployed our code to solve a massive problem. It is twice as fast, and is tested to be correct, so there wouldn't be any reason not to use it.

The unoptimised code is useful as it shows us the contribution we actually made to each of these metrics. For example, if our unoptimised code hit the maximum theoretical limit of (e.g.) speedup, then despite our optimised code looking like it has half the value, we did actually max out what we were in control of. The compiler takes our already fast code, and makes it run even faster, and more efficiently.

When the compiler optimises our code, the overall footprint gets smaller (less assembly instructions), however there are some external processes that don't get much faster, such as system calls and using libraries that are pre-compiled, which will now be a larger proportion of the code, making the 'sequential percentage' larger in optimised code. This effect is shown in Figure 5, and our data actually shows this theory in action when we compute our overhead in Figure 11.

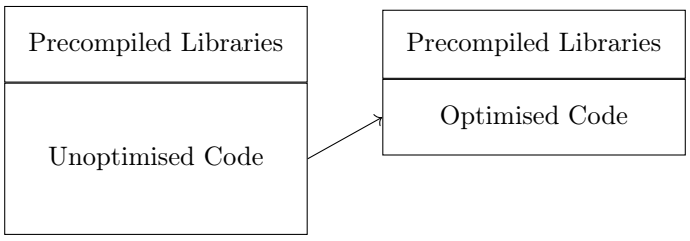

Figure 5: The footprint of our code, with and without compiler optimisations

7.2 Speedup

Speedup is the time taken sequentially vs the time taken in parallel. It shows us how many times faster our program runs, for a given number of processors. Figure 6 shows us that after 44 threads there is a significant fall off in speedup, which is expected as even creating one extra thread past that adds a massive amount of context swithcing that needs to be. The typical best speedup was between 41 and 44 threads, centering around 43 for the most part.

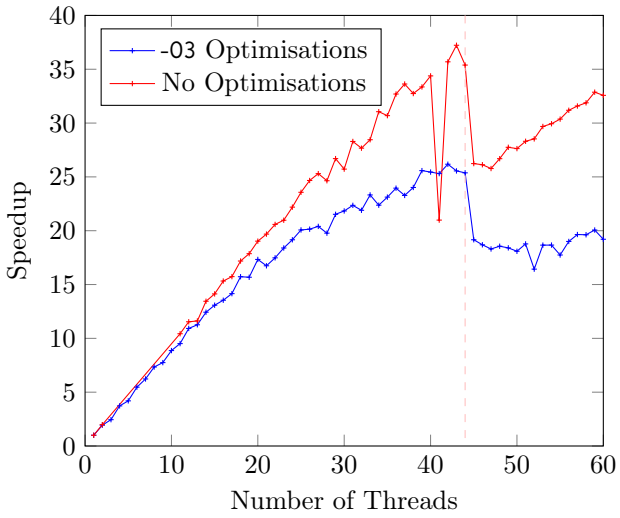

Figure 6: Demonstrating speedup measured on a 10000×10000-sized matrix

The maximum speedup values are 26.17× at 42 threads with optimsations on, and 37.23× at 43 threads without optimsiations, which is very close to Amdahl's limit of 44. Speedups are compared to the single-threaded version in each case, and if we look at the actual time taken, the optimised version of the code was just over twice as fast as the unoptimised code. This shows us that whilst speedup is a useful metric, it doesn't tell us everything we need to know (such as pure speed). If we want to be compare two different implementations, by comparing their speedup, we should use the same initial speed to compare against.

Now that we have looked at speedup, some data that is interesting to look at is how much switching from an array of doubles representing difference (single-threaded convergence computing) to an array of integers representing convergence (multi-threaded convergence computing) actually affected the speedup of our program. This is shown in Figure 7.

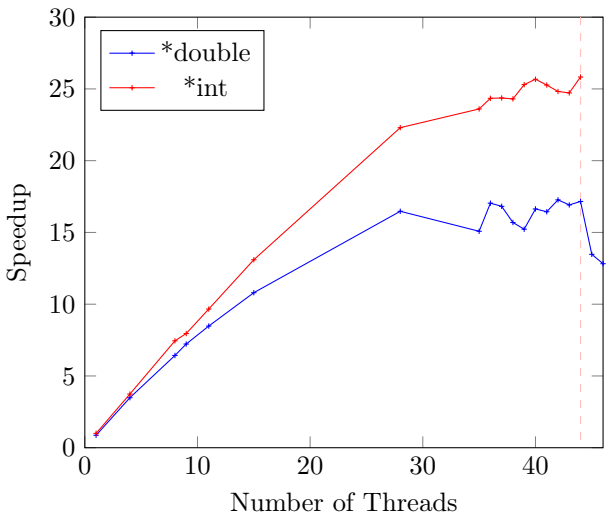

Figure 7: Demonstrating speedup measured on a $9,485 \times 9,485$-sized matrix

This figure is incredibly insightful. Not only does it show us that switching to an array of integers, and allowing each thread to perform the comparison operation was definitely the right choice, but shows us how much of an impact such a small change to our code can make. The speedup increased from 17.26× to 25.67× which is an insane 48% increase in speedup for a roughly $10,000^2$ sized matrix.

Finally we should check to see that our speedup does actually increase as the problem gets larger.

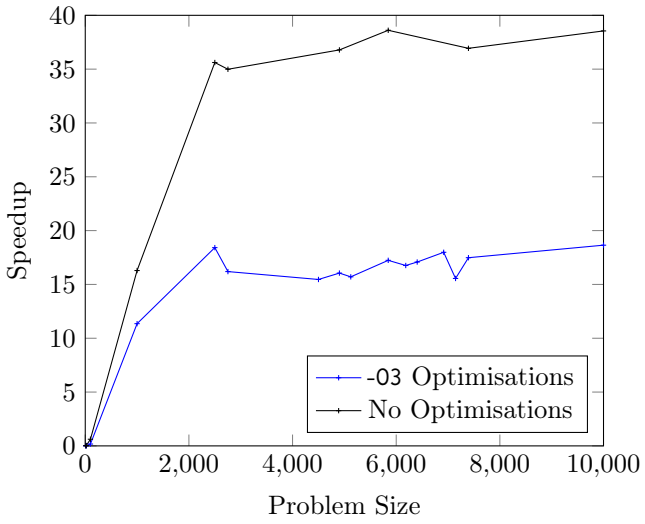

Figure 8: Demonstrating speedup measured using 43 threads

Figure 8 shows us what we would expect. The speedup does increase with the problem, size, and starts to converge up to Amdahl's limit. Looking at speedup has given us valuable insight, which is that as we add more cores, we are generally getting faster and faster. Our parallel code is behaving as it should. To see how we are making use of our resources, we should now measure the efficiency of our solution.

7.3 Efficiency

Efficiency measures the speedup per processor. Efficiency demonstrates what proportion of our added processing power we are actually using to perform our computation. Efficiency in this case won't be 100%, as we do have that chunk of our code where only the main (coordinator) thread is active, however due to our more parallel integer-checking implementation, our efficiency should be fairly strong.

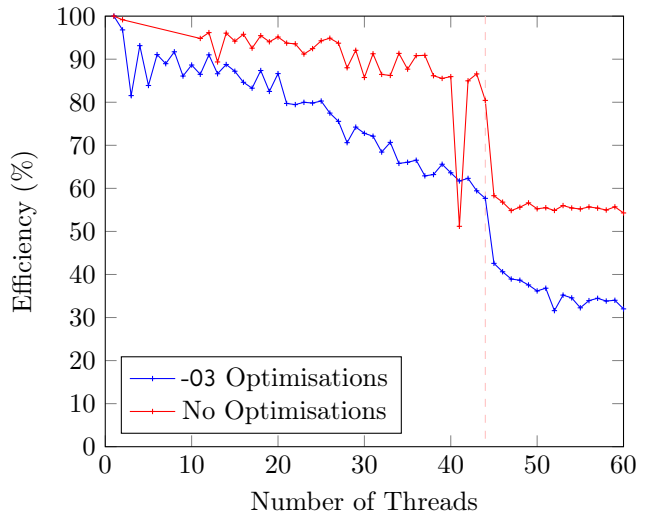

Figure 9: Demonstrating efficiency measured on a 10000 × 10000-sized matrix

Figure 9 shows us that we don't get the same amount of speedup as we introduce new processors. It remains fairly high at about 58% on the optimised code at 44 threads, and 86% on the unoptimised code at 44 threads. Eventually however, throwing CPUs at our implementation would barely increase our speedup. This does perform better than a typical efficiency graph, as the decline in efficiency isn't steep, but is slow and linear. Again after 44 threads we see a sharp decline in efficiency due to context switching overhead.

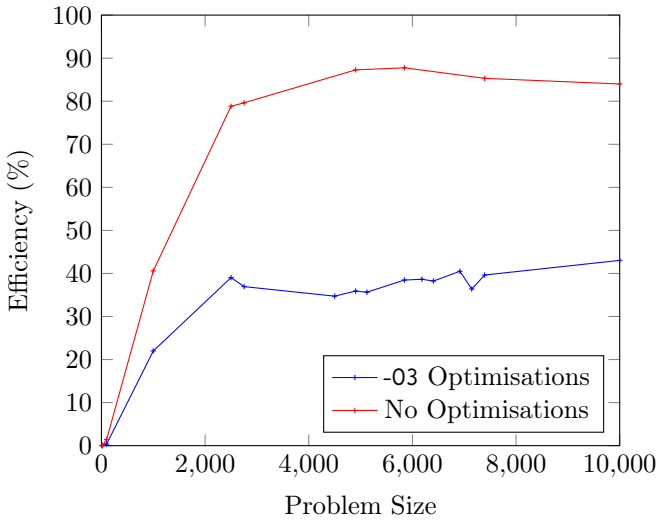

Figure 10: Demonstrating efficiency measured on 44 threads

Figure 10 tells us about how the efficiency scales with the size of the problem. It resembles the figure we saw earlier for speedup against problem size, Figure 8. The optimised code looks as if it has a little more room to grow but both graphs are converging. This suggests the opposite of what Figure 9 tells us, and shows the dynamic between Amdahl's Law and Gustafson's Law. Amdahl would tell us to not bother adding CPUs after a particular point, and Gustafson would say as the problem gets larger, those extra CPUs will make a more noticeable difference. *Isoefficiency* is a metric that can help us get a more broad perspective, but first we need to calculate some other metrics.

7.4 Work Efficiency

We can determine how cost efficient our implementation is by determining how long it would take to run our parallel code, if it ran on each processor. Subtracting the sequential time from this provides us with the amount of time that is overhead for our parallel implementation. It's like measuring the proportion of our code that is sequential, but as a value in terms of time instead. Instead of using the standard way of measuring parallel overhead, you can divide by the number of processors, and calculate the amount of overhead as a proportion of a parallel run. This is shown in Figure 11.

$$T_{overhead} = pT_{par} - T_{seq} \tag{1}$$

$$\frac{T_{overhead}}{p} = T_{par} - \frac{T_{seq}}{p} \tag{2}$$

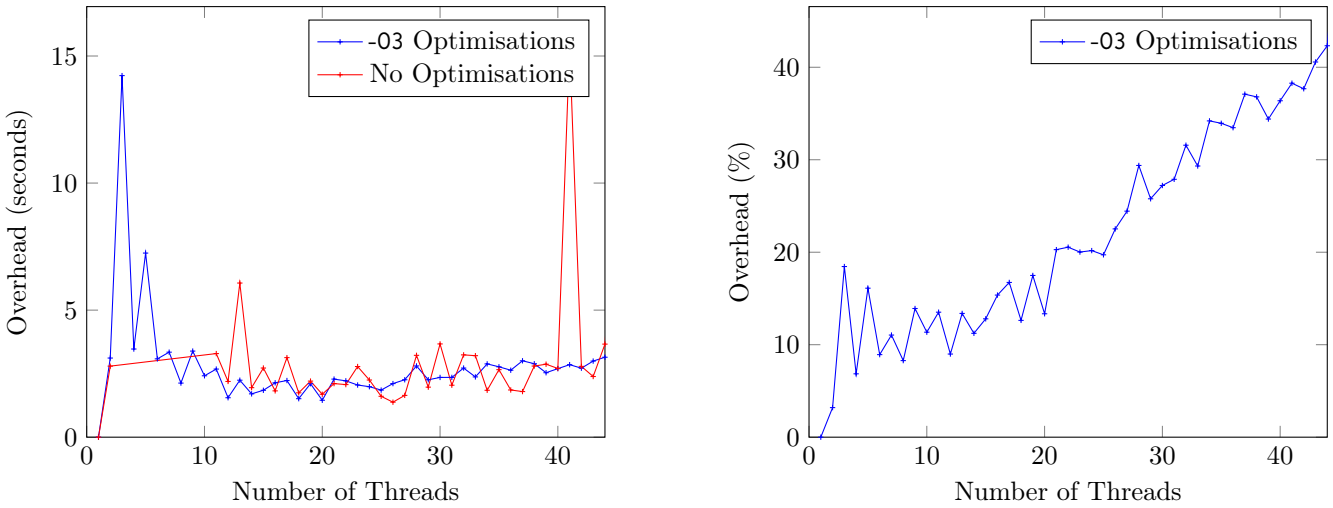

Figure 11: Overhead measured on a 10000×10000-sized matrix

This is informative to us as we can now see that when we used 44 threads, approximately 43% of our runtime was parallel overhead. However, as we increase the number of cores used our overhead remains as a slow-growing value. As we increase the number of cores there is always going to be some amount of overhead. When the overhead hits 90% of the computation, our parallel code is running so fast that throwing new cores at it is going to make a marginal difference compared to the cost of creating that thread, computing it's data to process on and more. This suggest that there will be at some point an optimal number of cores to run this implementation on.

Note how the unoptimised version of this overhead is essentially the same graph, which means it takes the same number of seconds to run whether optimsied or unoptimised. Earlier we claimed that the optimised code ran twice as fast, so why wouldn't the parallel overhead be twice as fast? This tells us that the precompiled code is a majority of the parallel overhead. This is the code that can only run as fast as the implementation of POSIX pthreads we use. This is a crucial finding as it tells that this is the amount that we truly have no control over, unless we modify the pthread library, or write our own.

7.5 Isoefficiency

Now that we know how about the parallel overhead of our program, we can begin to ask how this program scales. We want to calculate how much our problem size needs to grow by, in order to maintain its efficiency. We can do this by plotting the problem size, as a function of the number of cores, for various levels of efficiency. To do this, we can create a dataset that groups efficiencies into 'bands'. Band 3 efficiency could be 25% to 35% efficiency for example. We can then plot each band of efficiency, with problem sizes on the vertical axis, and thread count on the horizontal axis. Unfortunately this result was incorrect, or not helpful with the datasets, as shown in Figure 12.

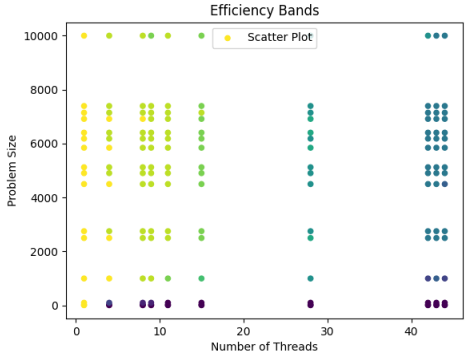

Figure 12: The rendering of problem size vs thread count, tracking bands of efficiency

7.6 Karp-Flatt

The Karp-Flatt metric is used to determine what proportion of our program is inherently sequential. The smaller it is the better.

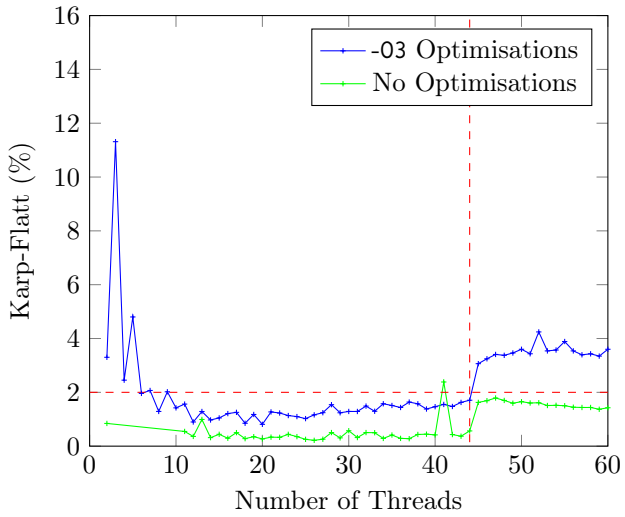

Figure 13: Demonstrating Karp-Flatt measured on a 10000×10000-sized matrix

Figure 13 shows us that less than 2% of our code is running sequentially. This is great as it means over 98% of our code is running in parallel. The value doesn't decrease, but this is likely because our coordinator thread makes up about 2% of each iteration. As we increase the problem size, the number of elements that need to be checked for convergence increases at the same rate, so is reliably constant proportion of our code.

7.7 Conclusion

To conclude, we developed a parallel solution for the relaxation technique for a shared memory architecture. We discussed the algorithm design for parallel programs, and the implementation details, including load balancing, and managing memory. We looked at some of the bugs that can occur in parallel programs, such as the main thread popping a stack frame too early and discussed some ways we could further optimise our solution. We looked at how one could measure parallel programs, and the role that compiler optimisations have in skewing our results. Finally we looked at a variety of different metrics for our program and determined that they show signs of a useful, strong parallel implementation, that worked to make use of all 44 cores available.

A Appendix A

A.1 The Python Script for a Diff Algorithm

Listing 3: Testing Script

```
import difflib
import os

# comparison function
def compare_files(file1_path, file2_path):
    with open(file1_path, 'r') as file1:
        file1_contents = file1.readlines()

    with open(file2_path, 'r') as file2:
        file2_contents = file2.readlines()

    differ = difflib.Differ()
    diff_result = list(differ.compare(file1_contents, file2_contents))

    # Check if there are differences, print them if there are
    if any([line.startswith('-') or line.startswith('+') for line in diff_result]):
        print(f"Files␣{file1_path}␣and␣{file2_path}␣are␣different:")
        print('\n'.join(diff_result))


# running the script in the current directory
azure_dir = os.getcwd() + '/azure-matrices/'
local_dir = os.getcwd() + '/local-matrices/'

sizes = [...]
thread_counts = [...]
precision = '...'

for size in sizes:
    result_file = local_dir + f'local-{size}-{precision}.txt'
    azure_files = [azure_dir + f"azure-{size}-{thread_count}-{precision}.txt"
                   for thread_count in thread_counts]

    for file in azure_files:
        compare_files(result_file, file)
```

A.2 The Python Script for a Data Processing

Listing 4: Data Processing Script

```
import os
import pandas as pd

# function that operates per file
def modify_file(filename):
    df = pd.read_csv(filename)

    single_threaded_df = df[df['␣threads'] == 1]
    merged_df = pd.merge(df, single_threaded_df[['size', '␣seconds']],
                                    on='size', suffixes=('', '_ref'))

    df['speedup'] = merged_df['␣seconds_ref'] / df['␣seconds']

    # then add speedup per threads, using single-threaded timing * 100.0
    df['efficiency'] = df['speedup'] / df['␣threads'] * 100.0

    # then add karpflatt as a percentage, which is
    # (1/speedup(thread) - 1/threads) / (1 - 1/threads)
    df['karpflatt'] = (1.0 / df['speedup'] - 1.0 / df['␣threads'])
                            # TODO: This new line is only for readability
                            # it is not valid Python
                            / (1.0 - 1.0 / df['␣threads']) * 100.0

    # add work-efficiency where
    # overheadTime = threads * seconds - single-threaded timing
    df['overhead'] = df['␣threads'] * df['␣seconds'] - merged_df['␣seconds_ref']

    # write over file
    df.to_csv(filename, index=False)


timings_dir = os.getcwd() + '/timing/'
for file in [f for f in os.listdir(timings_dir)
                if os.path.isfile(os.path.join(timings_dir, f))]:
    modify_file(timings_dir + file)
```

References

Arm Developer, n.d. Embarrassingly parallel applications [Online]. Available from: https://developer.arm.com/documentation/dui0538/f/parallel-processing-concepts/embarrassingly-parallel-applications.

GCC Team, 2023. Auto-vectorization in gcc [Online]. Available from: https://gcc.gnu.org/projects/tree-ssa/vectorization.html.

Intel, n.d. Intel xeon platinum 8168 processor [Online]. Available from: https://www.intel.com/content/www/us/en/products/sku/120504/intel-xeon-platinum-8168-processor-33m-cache-2-70-ghz/specifications.html.

The Open Group, 2017. The open group base specifications issue 7, 2018 edition [Online]. Available from: https://pubs.opengroup.org/onlinepubs/9699919799/functions/pthread_exit.html.

Valiant, L.G., 1990. A bridging model for parallel computation. *Commun. acm* [Online], 33(8), p.103–111. Available from: https://doi.org/10.1145/79173.79181.