CM30225 Parallel Computing Assessed Coursework Assignment 1

Anonymous

November 2023

1 Introduction

In this project, we explore parallelism in a shared memory architecture by building and comparing two implementations of the same problem and investigating their differences. One implementation will be designed to execute sequentially, and the other in parallel. The goal is to develop an understanding of shared memory parallelism, its benefits, and various pitfalls.

The problem at hand is called finite differences which, according to Young (1971), is a form of relaxation method used to solve discrete differential equations. There are 3 hyperparameters to consider; the size of the problem; the number of threads or cores the problem should be solved across, in essence, how parallel the implementation is; and the precision, which is interpreted here as the maximum acceptable difference between an element a_{ij} and the same element a_{ij} in the previous iteration.

Unless stated otherwise, this investigation will use a value of precision of 0.001. This value has been chosen semi-arbitrarily as a number that provides demonstratable differences between implementations. In future work, it could be investigated how different requirements for precision could affect design decisions.

1.1 Problem Example

(1) Illustrates an example with a size of 4 and a precision of 0.1. By the third iteration, it can be seen that each element of the matrix differs by at most 0.1 from corresponding elements from the previous iteration. When this happens, the matrix is said to have *converged*.

$$\begin{bmatrix}1&1&1&1\\1&0&0&0\\1&0&0&0\\1&0&0&0\end{bmatrix} \mapsto \begin{bmatrix}1&1&1&1\\1&0.5&0.25&0\\1&0.25&0&0\\1&0&0&0\end{bmatrix} \mapsto \begin{bmatrix}1&1&1&1\\1&0.625&0.375&0\\1&0.375&0.125&0\\1&0&0&0\end{bmatrix} \mapsto \begin{bmatrix}1&1&1&1\\1&0.6875&0.4375&0\\1&0.4375&0.1875&0\\1&0&0&0\end{bmatrix} \tag{1}$$

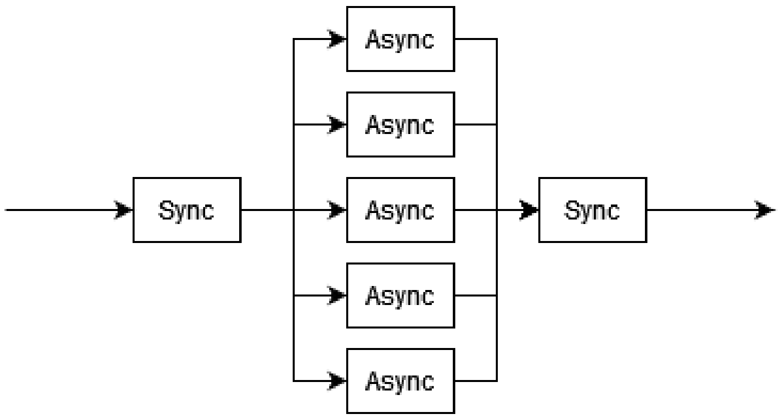

Figure 1: An illustration of a superstep.

2 Parallel Approach

In the chosen parallel design, computation is decomposed into supersteps, a concept originating from a model devised by Valiant (1990) called Bulk Synchronous Parallel (BSP). There are three parts to a superstep: first, sequential execution is paused; then many processors perform local computations in parallel; finally, all threads are synchronised and sequential computation is resumed (Figure 1). The main advantage of this, is that it provides a structured and easy-to-understand method of executing computations which depend on each other in parallel. This is the case with the described problem, where iterations cannot be calculated independently without first calculating the previous iteration.

Each iteration corresponds to a single superstep, and work is distributed whereby each processor operates independently on a separate part of the matrix. This approach completely avoids data races, a common issue in parallel computing caused by multiple processes mutating data simultaneously. Whilst it is possible to avoid race conditions by guaranteeing exclusive access to a single process while data is modified through mutexes, this approach avoids the need for mutexes by providing each process with separate data to operate on. All other shared data in this design, such as the matrix from the previous iteration, is immutable, avoiding the issue of concurrent changes entirely.

More specifically, each processor is assigned a number of consecutive rows to operate on. In other words, the mxm matrix A is subdivided into p partitions where p is the number of processors. Each partition $s \in A; |s| = \frac{m}{p}$ is a unique set of consecutive rows of A. For example, if (1) were to be solved with two processors p_1, p_2, they would be assigned partitions $s_{p1} = \{R_1, R_2\}, s_{p2} = \{R_3, R_4\}$ where R_i is row $i \in [0 \dots p]$ of A.

We could theoretically define s as any unique denomination of A, for example columns could be used instead of rows. Rows were chosen because they tend to relate better to the memory representation of arrays in C. C-style arrays store rows as consecutive memory addresses, meaning accesses to consecutive values in a row can potentially lead to better cache-time locality, improving performance by reducing cache misses. While at first this might seem unrelated to parallelism, it is important to remember to reduce synchronous costs of a parallel implementation, improving speedup. This is discussed further in Section 3.1 when considering Amdahl's Law.

3 Scalability Investigation

Scalability has been investigated in several ways for the project, including speedup, efficiency, and relations to Amdahl's law as well as Gustafson's law.

Speedup in this context refers to the measured improvement in execution time achieved through parallelisation of the problem, which can be written as (2). Table 1 shows investigations into speedup with respect to both thread count and problem size, we will cover the differences between these comparisons later. The maximum speedup observed for the problem is $S_{32} = \frac{1012}{265} = 3.8(2s.f.)$, which is the speedup using 32 threads for a problem size of 20000x20000.

It would be sensible to assume that the greatest speedup would be found when the thread count is equal the number of available cores on the machine, which in this case is 44, however the investigation shows that 32 threads consistently outperformed 44. This is assumed to be because, concerning this specific implementation and this specific problem, the overheads of instantiating more than 32 threads are higher than any potential speedup benefit. This leads to the conclusion that speedup scales well up to 32 threads, but then begins to plateau or even decrease.

$$S_p = \frac{T_{seq}}{T_{par}} \tag{2}$$

3.1 Amdahl's Law

Amdahl's law states that every program has a natural limit on the maximum theoretical speedup it can attain from parallelisation. This can be more formally quantified with respect to sequential and parallel processing time, (3) considers exactly this as the number of processors $p \to \infty$. Amdahl's law is relevant because it helps to describe why further parallelisation past a point doesn't lead to increased speedup.

This effect can be observed in Figure 2, where past 32 threads there is no increase in speedup. This will be due to parts of the program being inherently sequential, for example, IO and thread synchronisation. In the described implementation in Section 2, threads may only proceed to the next superstep after all threads have finished their computation. Whilst reducing the proportion of the program which runs sequentially will increase speedup, it is not able to scale indefinitely with thread count.

$$S_\infty \leq \frac{T_{seq} + T_{par}}{T_{seq}} \tag{3}$$

3.2 Gustafson's Law

Gustafson's law considers speedup as proportional to the problem size rather than the number of processors. If this is the case then it would be reasonable to expect a problem with a fixed processor count to experience a larger speedup when the problem size is increased. This precisely matches the findings in Figure 2, where the speedup of a problem with size 20000 is consistently higher than that of a problem with size 20000. This shows that the provided solution scales well with respect to problem size, up to the maximum tested problem size.

3.3 Efficiency

Efficiency attempts to quantify how effective a parallel program is at utilising multiple processors. It can be thought of as the speedup per processor, or as a percentage which describes how much of the total collective compute power is utilised (4). Typically efficiency drops as the number of processors increases, because as p increases so do communication overheads, and hence so

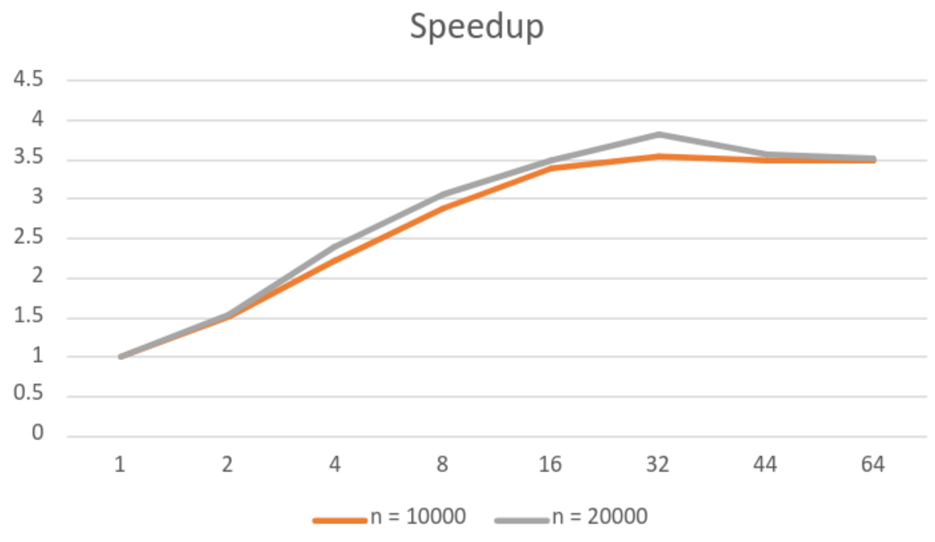

Figure 2: Speedup for problems of size 10000 and 20000 derived from the investigation described by Table 1.

does the amount of time spent processing sequentially. This is the same behaviour described by Amdahl's law. This trend can be seen in Figure 3.

Another observation which can be made from Figure 3 is that efficiency seems to increase with problem size, which can be seen because the grey line is consistently below the orange line. This is consistent with Gustafson's law since the scalability of the problem increases as the problem size increases.

$$E_p = \frac{S_p}{p} = \frac{T_{seq}}{p + T_{par}} \tag{4}$$

Table 1: Solve time in seconds with respect to thread count and problem size for a matrix in the same form as (5). Values are averages of 3 runs with identical inputs.

	Problem Size				
Thread Count	10	100	1000	10000	20000
Single	0	0	2	252	1012
2	0	0	1	166	661
4	0	0	1	113	419
8	0	0	1	87	330
16	0	0	1	74	289
32	0	0	1	71	265
44	0	1	1	72	283
64	0	1	1	72	286

$$A := \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \dots & 0 \end{bmatrix} \tag{5}$$

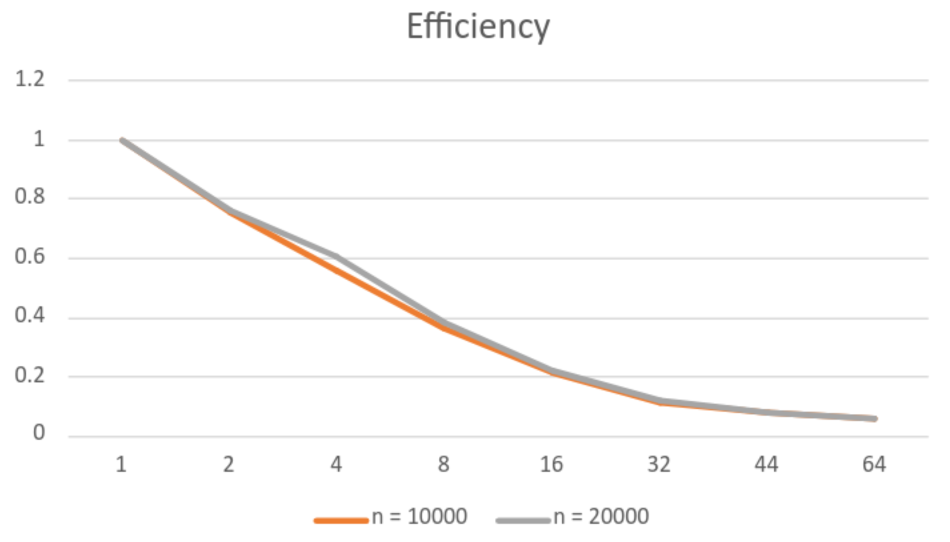

Figure 3: Efficiency for problems of size 10000 and 20000 derived from the investigation described by Table 1.

3.4 Investigatory Limitations

A consequence of performing investigations on a shared cluster is that jobs are given a maximum elapsed time they cannot exceed. This affected the investigation because it made certain tests not possible to carry out. Table 2 shows a partially carried out investigation, where it was not possible to perform tests over a certain problem size. This is because exceeding this problem size causes the program to exceed the provided time limit. It's worth noting that the problem used for the investigation outlined in Table 2 takes many more iterations to converge than the problem used for the investigation outlined in Table 1. Since the investigation provided is limited to a single data set, the results shown throughout Section 3 are not proveably conclusive.

Another limitation is that the majority of problem sizes considered throughout the investigation all execute within a second, which is the resolution of the timer used when collecting data. A consideration for future work could be to perform the same investigations over a wider range of problem sizes above 10000. This could provide a higher degree of confidence in the findings provided.

Table 2: Solve time in seconds with respect to thread count and problem size for a matrix in the same form as (6)

	Problem Size				
Thread Count	10	100	1000	10000	20000
Single	0	1	-	-	-
2	0	1	-	-	-
4	0	1	-	-	-
8	0	1	-	-	-
16	0	2	-	-	-
32	0	8	-	-	-
44	0	13	-	-	-
64	0	20	-	-	-

$$B := \begin{bmatrix} 0 & 2 & 4 & \dots & 2j \\ 1 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & \dots & 0 \\ 0 & 2 & 4 & \dots & 2j \end{bmatrix} \tag{6}$$

4 Considerations of Correctness

4.1 Debugger and Compiler Flags

The project was developed using the GCC compiler, which supports the compiler flags '-Wall' and '-Wextra'. These enable a wider range of warning messages at compile time, which allows bugs to be caught earlier in development leading to more robust and correct code. The project was built with these flags enabled at all times, ensuring that any problems detected by the compiler are fixed.

GDB, the GNU Debugger, was used at times to debug errors in the project. These included a variety of runtime errors, mostly consisting of misallocating of memory or segmentation faults. GDB allows the execution of code line-by-line and provides information on variables in the current stack frame such as their labels and current values. It is a useful tool for gathering additional information on errors, especially errors which happen non-deterministically. Another benefit is that it increases confidence that proposed fixes fix the underlying problem rather than some higher-level symptom.

4.2 Unit Tests

Unit tests validate that individual parts of a program are working as intended. The project makes use of these to provide some level of guarantee that the implementations provided are correct. One form of test used was implemented by providing the same problem inputs to both the synchronous and concurrent implementations and subsequently comparing their outputs. This test was performed a substantial number of times, including at every step of the Scalability Investigation (Section 3).

These tests alone cannot guarantee correctness, clearly, the two implementations could both contain the same logical error and still pass the test. However, it verifies that the two implementations are observationally equivalent, a concept introduced by (Morris (1968)) whereby two programs are observationally (or extensionally) equivalent if they provide the same set of outputs for any set of inputs. This is a powerful reasoning tool, because it implies that when these tests pass, it is possible to reason about the program logic of the parallel implementation by inspecting the conceptually much simpler sequential implementation. This makes finding logical errors easier to find and further guarantees the correctness of the program.

To ensure unit tests are readable and written in a consistent style, they all follow the 'Arrange, Act, Assert' methodology. Test data is first arranged, then the test is performed, and finally, it is asserted whether the test passed successfully; demonstrated by Figure 4. The output of this test's success case can be found in Appendix A.2, while the output of the failure case can be found in Appendix A.1.

4.3 Valgrind

Valgrind is a collection of tools used for profiling and finding memory-related bugs, including memory leaks. It does this by analysing the program's machine-level instructions at runtime. Specifically, the memcheck tool monitors the program's memory read and write instructions.

```c
// Arrange - allocate memory, initialise variables
int rc = 0;
double(*result_async)[size][size];
double(*result_sync)[size][size];

rc = array_2d_try_alloc(size, &result_async);
if (rc != 0)
    return rc;

rc = array_2d_try_alloc(size, &result_sync);
if (rc != 0)
{
    free(result_async);
    return rc;
}

load_testcase_1(size, result_async);
load_testcase_1(size, result_sync);

// Act - perform the tested action
solve(size, result_async, thread_count, precision);
solve_sync(size, result_sync, precision);

// Assert - assert that the action performed successfully
rc = memcmp(result_async, result_sync, sizeof(*result_async));
if (rc == 0)
    // Pass condition
else
    // Fail condition

// Free allocated memory
free(result_sync);
free(result_async);

```

Figure 4: An example unit test following the 'Arrange, Act, Assert' (AAA) methodology.

Since manual memory management is more prone to introducing memory-related bugs, it is sensible to use tools such as Valgrind's memcheck to boost confidence in the correctness of the program. This is especially important in a parallel environment, where memory-related bugs are much harder to find.

Valgrind was used throughout the project to ensure that no memory leaks were occurring. One instance of an issue Valgrind brought to attention was the use of an allocated but uninitialised buffer, for which the full output can be found in Appendix B.1. As seen in Figure 5, three pointers have memory allocated with the function solve_try_alloc, both handles and args are initialised later on, but prev_values is never explicitly initialised. The fix is quite simple, initialise prev_values, as seen on line 9 in Figure 6.

See Appendix B.3 for a run of Valgrind on the finished project, showing no reported memory-related errors.

4.4 Helgrind

Helgrind is a module for Valgrind which provides tools for identifying race conditions and thread synchronization errors. This was used throughout the project in complement to Valgrind, to find any potential issues specific to the multithreaded implementation. Appendix B.2 shows the output for Helgrind on the finished project, giving a high confidence that the parallel implementation

```c
double (*prev_values)[size][size];
pthread_t *handles;
solve_args *args;

// solve_try_alloc correctly allocates memory for prev_values, handles, args,
    and guarantees that either all pointers will have memory allocated or none
    of them will.
rc = solve_try_alloc(size, &prev_values, thread_count, &handles, &args);
if (rc != 0)
    return rc;
```

Figure 5: Problematic C code where memory is correctly allocated but ‘prev_values‘ is not being initialised.

```c
double (*prev_values)[size][size];
pthread_t *handles;
solve_args *args;

rc = solve_try_alloc(size, &prev_values, thread_count, &handles, &args);
if (rc != 0)
    return rc;

memcpy(prev_values, values, sizeof(*prev_values));
```

Figure 6: An amended version of Figure 5 where ‘prev_values‘ is being correctly initialised.

doesn’t include any of the aforementioned classes of bug.

A Unit Testing

A.1 A Failed Test Output

The first three lines describe each thread and its allocated rows. More on the parallel implementation can be found in Section 2.

```
thread: 0; start_row: 1; end_row: 3
thread: 1; start_row: 3; end_row: 5
thread: 2; start_row: 5; end_row: 9
[ASYNC] solved in 23 iterations
[SYNC] solved in 27 iterations
FAIL solution doesn't match synchronous implementation

sync impl result
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.953414 0.909445 0.869637 0.833224 0.796197 0.748335 0.000000
1.000000 0.909445 0.824526 0.748498 0.681020 0.616507 0.542981 0.000000
1.000000 0.869637 0.748498 0.642309 0.551285 0.470350 0.388935 0.000000
1.000000 0.833224 0.681020 0.551285 0.444816 0.356434 0.277098 0.000000
1.000000 0.796197 0.616507 0.470350 0.356434 0.268052 0.196162 0.000000
1.000000 0.748335 0.542981 0.388935 0.277098 0.196162 0.135560 0.000000
1.000000 0.665423 0.438778 0.292090 0.195986 0.131473 0.086573 0.000000
1.000000 0.482658 0.269402 0.163889 0.104004 0.066977 0.042587 0.000000
1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.0000000.000000
async impl result:
1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000
1.000000 0.946841 0.897378 0.854027 0.816295 0.780147 0.734910 0.000000
1.000000 0.897378 0.802463 0.719899 0.650116 0.587214 0.518526 0.000000
1.000000 0.854027 0.719899 0.605394 0.511405 0.432682 0.357562 0.000000
1.000000 0.816295 0.650116 0.511405 0.401912 0.316019 0.243542 0.000000
1.000000 0.780147 0.587214 0.432682 0.316019 0.230125 0.164818 0.000000
1.000000 0.734910 0.518526 0.357562 0.243542 0.164818 0.109729 0.000000
1.000000 0.655862 0.421394 0.269822 0.172263 0.109361 0.068448 0.000000
1.000000 0.477711 0.260405 0.152398 0.091767 0.055619 0.033281 0.000000
1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
```

A.2 A Passed Test Output

```
thread: 0; start_row: 1; end_row: 3
thread: 1; start_row: 3; end_row: 5
thread: 2; start_row: 5; end_row: 9
[ASYNC] solved in 27 iterations
[SYNC] solved in 27 iterations
PASS solution matches synchronous implementation
```

B Valgrind Testing

B.1 Valgrind Identifying an Uninitialised Value

```
==116944== Memcheck, a memory error detector
==116944== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==116944== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright
   info
==116944== Command: ./relaxation
==116944==
matrix size: 1000
precision: 0.010000
thread count:8
thread: 0; start_row: 1; end_row: 125
thread: 1; start_row: 125; end_row: 249
thread: 2; start_row: 249; end_row: 373
thread: 3; start_row: 373; end_row: 497
thread: 4; start_row: 497; end_row: 621
thread: 5; start_row: 621; end_row: 745
thread: 6; start_row: 745; end_row: 869
thread: 7; start_row: 869; end_row: 999

==116944== Conditional jump or move depends on uninitialised value(s)
==116944==    at 0x10A71C: matrix_has_converged (in /home/jay/src/shared-
   memory-parallel-computing/relaxation)
==116944==    by 0x109D03: solve (in /home/jay/src/shared-memory-parallel-
   computing/relaxation)
==116944==    by 0x10967D: main (in /home/jay/src/shared-memory-parallel-
   computing/relaxation)
==116944==  Uninitialised value was created by a heap allocation
==116944==    at 0x4848899: malloc (in /usr/libexec/valgrind/
   vgpreload_memcheck-amd64-linux.so)
==116944==    by 0x10A95D: array_2d_try_alloc (in /home/jay/src/shared-
   memory-parallel-computing/relaxation)
==116944==    by 0x10A8B0: solve_try_alloc (in /home/jay/src/shared-memory
   -parallel-computing/relaxation)
==116944==    by 0x1099B9: solve (in /home/jay/src/shared-memory-parallel-
   computing/relaxation)
==116944==    by 0x10967D: main (in /home/jay/src/shared-memory-parallel-
   computing/relaxation)
[ASYNC] iteration: 38 (23s)
[ASYNC] solved in 38 iterations in 23s
[SYNC] iteration: 37 (7s)
[SYNC] solved in 37 iterations in 7s
==116944== Conditional jump or move depends on uninitialised value(s)
==116944==    at 0x485207E: bcmp (in /usr/libexec/valgrind/
   vgpreload_memcheck-amd64-linux.so)
==116944==    by 0x10978F: main (in /home/jay/src/shared-memory-parallel-
   computing/relaxation)
==116944==  Uninitialised value was created by a heap allocation
==116944==    at 0x4848899: malloc (in /usr/libexec/valgrind/
   vgpreload_memcheck-amd64-linux.so)
==116944==    by 0x10A95D: array_2d_try_alloc (in /home/jay/src/shared-
   memory-parallel-computing/relaxation)
==116944==    by 0x10A8B0: solve_try_alloc (in /home/jay/src/shared-memory
   -parallel-computing/relaxation)
==116944==    by 0x1099B9: solve (in /home/jay/src/shared-memory-parallel-
   computing/relaxation)
```

```
==116944==    by 0x10967D: main (in /home/jay/src/shared-memory-parallel-
    computing/relaxation)
==116944==
PASS solution matches synchronous implementation
==116944==
==116944== HEAP SUMMARY:
==116944==     in use at exit: 0 bytes in 0 blocks
==116944==   total heap usage: 163 allocs, 163 frees, 32,043,840 bytes
    allocated
==116944==
==116944== All heap blocks were freed -- no leaks are possible
==116944==
==116944== For lists of detected and suppressed errors, rerun with: -s
==116944== ERROR SUMMARY: 2032028 errors from 2 contexts (suppressed: 0
    from 0)
```

B.2 Valgrind Reports No Errors

```
==117768== Memcheck, a memory error detector
==117768== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==117768== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright
    info
==117768== Command: ./relaxation
==117768==
matrix size: 1000
precision: 0.010000
thread count:8
thread: 0; start_row: 1; end_row: 125
thread: 1; start_row: 125; end_row: 249
thread: 2; start_row: 249; end_row: 373
thread: 3; start_row: 373; end_row: 497
thread: 4; start_row: 497; end_row: 621
thread: 5; start_row: 621; end_row: 745
thread: 6; start_row: 745; end_row: 869
thread: 7; start_row: 869; end_row: 999
[ASYNC] iteration: 37 (19s)
[ASYNC] solved in 37 iterations in 19s
[SYNC] iteration: 37 (7s)
[SYNC] solved in 37 iterations in 7s
PASS solution matches synchronous implementation
==117768==
==117768== HEAP SUMMARY:
==117768==     in use at exit: 0 bytes in 0 blocks
==117768==   total heap usage: 159 allocs, 159 frees, 32,042,752 bytes
    allocated
==117768==
==117768== All heap blocks were freed -- no leaks are possible
==117768==
==117768== For lists of detected and suppressed errors, rerun with: -s
==117768== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

B.3 Helgrind Reports No Errors

```
==118976== Helgrind, a thread error detector
==118976== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==118976== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright
    info
```

```
==118976== Command: ./relaxation
==118976==
matrix size: 1000
precision: 0.010000
thread count:8
thread: 0; start_row: 1; end_row: 125
thread: 1; start_row: 125; end_row: 249
thread: 2; start_row: 249; end_row: 373
thread: 3; start_row: 373; end_row: 497
thread: 4; start_row: 497; end_row: 621
thread: 5; start_row: 621; end_row: 745
thread: 6; start_row: 745; end_row: 869
thread: 7; start_row: 869; end_row: 999
[ASYNC] iteration: 37 (20s)
[ASYNC] solved in 37 iterations in 20s
[SYNC] iteration: 37 (11s)
[SYNC] solved in 37 iterations in 11s
PASS solution matches synchronous implementation
==118976==
==118976== Use --history-level=approx or =none to gain increased speed, at
==118976== the cost of reduced accuracy of conflicting-access information
==118976== For lists of detected and suppressed errors, rerun with: -s
==118976== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

References

Morris, J.H., 1968. *Lambda-calculus models of programming languages* [Online]. Ph.D. thesis. Massachusetts Institute of Technology. Available from: https://dspace.mit.edu/handle/1721.1/64850.

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

Young, D.M., 1971. Iterative solution of large linear systems [Online]. Academic Press. Available from: https://doi.org/https://doi.org/10.1016/B978-0-12-773050-9.50009-7.