Parallel Coursework 1

Testing Report

Introduction

This testing report details the testing performed for Parallel Computing, coursework 1. We begin by detailing the correctness of the solutions, and then look into further test cases and conclusion.

Below is the readme file included, for instructions on running the code.

```
Instructions for running the code:
Compile sequential.c or parallel.c using
    gcc -Wall -pthread filename.c -lrt

Run the program using ./filename, and possible flags:
-debug : The level of debug output: 0, 1, 2
-c : number of threads to use for the program
-d : length of the square array
-p : how precise the relaxation needs to be before the program ends
-g : (1 or 0) 0 to use values in from file specified in program, 1 to generate them
randomly
-f : string, path of the textfile to use

For example: ./parallel -debug 2 -c 16 -d 500 -p 0.01 -g 0 -f values.txt

If reading values from a file (-g 0), the values must first be computed by editing
numbergen.c to specify dimension size and output file, then compiling and running.
Computing a file with dimension 1000 can be used by any program for any dimension
<= 1000, but not for those with > 1000.
```

Tests

Correctness Testing – Manual vs Sequential vs Parallel

The correctness testing aims to demonstrate that both the sequential and parallel programs compute the correct answer.

A set square array is formed and relaxed manually to find the expected output. The same initial array is then fed into the sequential program and parallel programs. If the expected array is returned, the program has passed the test. The tests are then repeated with a different array of different size and desired precision. All tests are conducted 3 times to help increase accuracy.

Value set 1: valSet1.txt – dimension: 4x4, precision: 0.1, parallel threads: 1, 2, 4, 8
Value set 2: valSet2.txt – dimension: 6x6, precision: 1.0, parallel threads: 1, 2, 4, 8

	Value Set 1 – 4x4, 0.1 (Pass for result identical to manual)	**Value set 2 – 6x6, 1.0 (Pass for result identical to manual)**
Manual	-	-
Sequential	Pass x 3	Pass x 3
Parallel T:1	Pass x 3	Pass x 3
Parallel T:2	Pass x 3	Pass x 3
Parallel T:4	Pass x 3	Pass x 3
Parallel T:8	Pass x 3	Pass x 3

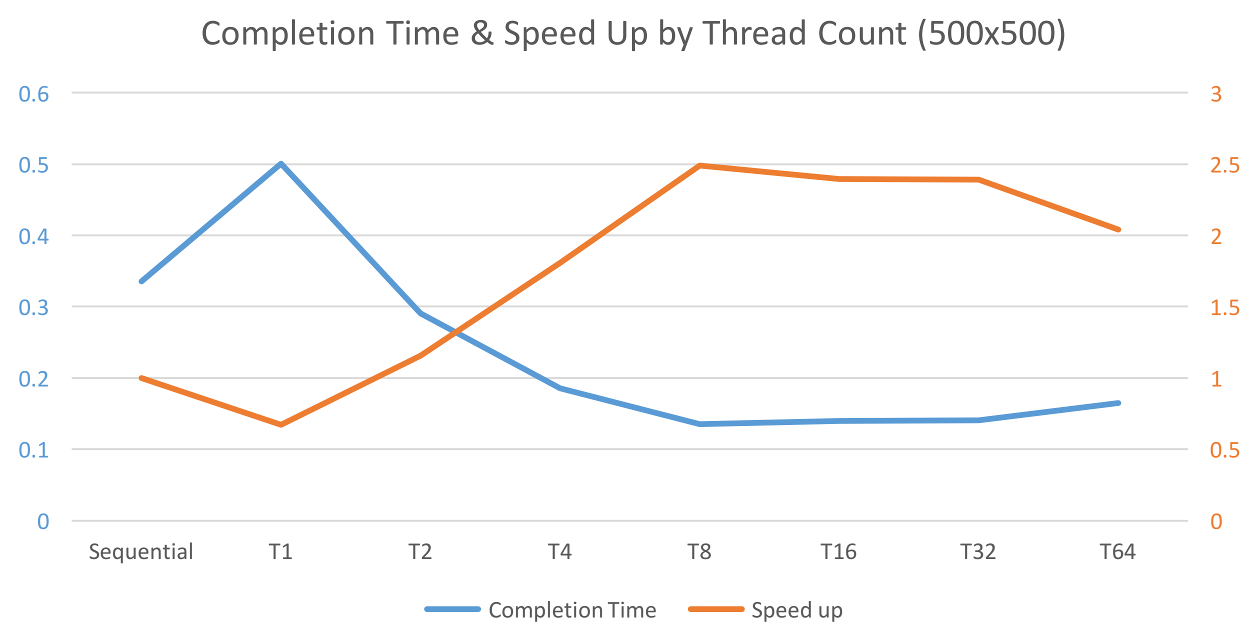
This suggests that both the Sequential and the Parallel code compute the correct answer, and that the Parallel code computes it correctly irrespective of number of threads used. This stands as the foundation for future tests, and is enough evidence to suggest the program is correct, irrespective of precision, threads or dimension size.

Throughout the rest of this Testing Report, we run a number of tests, focusing on changing Thread Count, Dimensions and Precision, and how they are inextricably linked. We also look at Speed up and Efficiency and how they change based on increasing hardware or increasing problem size, in line with Amdahl and Gustafson's Laws.

Thread Count Tests – Amdahl’s Law

These tests determine the average speed of running the program sequentially and in parallel for varying numbers of threads, using a fixed array and precision. The first set of tests demonstrate how speed up changes with a fixed problem size and increasing hardware (Amdahl’s Law), and the second set begin to demonstrate the same but for fixed hardware and an increased problem size (Gustafson’s Law).

Threads	Array Dimensions	Precision	Average Completion Time over 3 Attempts	Speed up (s/p)
Sequential	500x500	0.1	0.335612060 seconds	1
1	500x500	0.1	0.500871186 seconds	0.67
2	500x500	0.1	0.290115700 seconds	1.157
4	500x500	0.1	0.185762405 seconds	1.807
8	500x500	0.1	0.134877214 seconds	2.488
16	500x500	0.1	0.140009392 seconds	2.397
32	500x500	0.1	0.140220868 seconds	2.393
64	500x500	0.1	0.164402960 seconds	2.041

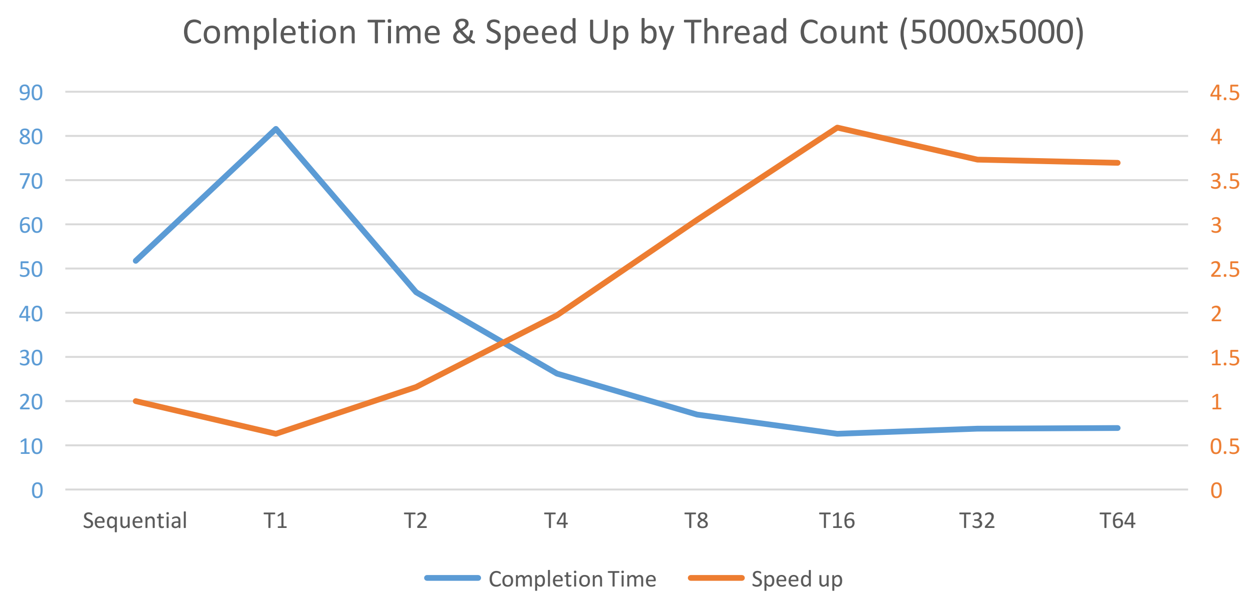


For an array of 500x500 and precision 0.1, the program continues to speed up with an increasing number of threads, until 16 threads are used where the program then slows. This shows the overhead generated exceeds the benefit of more threads at this array size. 32 and 64 threads increasingly slowed the program.

It is also seen the overhead of creating 1 thread to compute the program, as opposed to sequentially completing the program without creating any threads, as the average time increases slightly.

The tests were repeated with an array size 5000x5000. It is expected that increasing the problem size for a fixed amount of threads will improve the relative speedup, and as such I hypothesize that 16 threads will instead run faster than 8 threads for this size problem.

Threads	Array Dimensions	Precision	Average Completion Time over 3 Attempts	Speed up (s/p)
Sequential	5000x5000	0.1	51.704399473 seconds	1.000
1	5000x5000	0.1	81.532672780 seconds	0.634
2	5000x5000	0.1	44.595699940 seconds	1.159
4	5000x5000	0.1	26.177496806 seconds	1.975
8	5000x5000	0.1	16.931951042 seconds	3.054
16	5000x5000	0.1	12.621520786 seconds	4.097
32	5000x5000	0.1	13.844825517 seconds	3.735
64	5000x5000	0.1	13.981570211 seconds	3.698

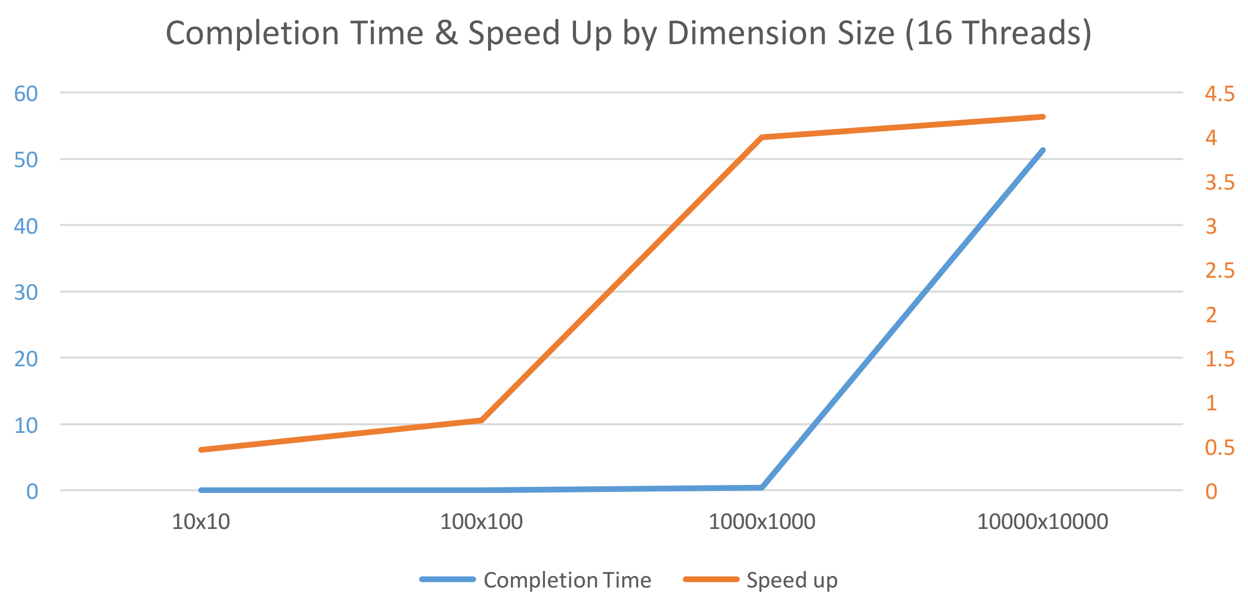


As hypothesized, the speedup increases for a fixed set of hardware, and as such 16 threads is now has a higher speed up than 8 threads, as the problem size grew larger. The hardware we're using is limited to 16 cores, so more than 16 threads begins to slow the program. The next tests focus on changing Dimension, and as such look at changing problem size in more detail.

Dimension Tests – Gustafson's Law

These tests demonstrate in more detail Gustafson's Law, by incorporating fixed hardware of 16 threads, but an increasing problem size. The speed up increases as the dimensions grow larger. The same problems were computed sequentially, and those speeds were used to calculate the speed up.

Threads	**Array Dimensions**	**Precision**	**Average Completion Time over 3 Attempts (seconds)**	**Speed up (s/p)**
16	10x10	0.1	0.005228751	0.457
16	100x100	0.1	0.012128109	0.793
16	1000x1000	0.1	0.475328199	3.998
16	10000x10000	0.1	51.290416832	4.229

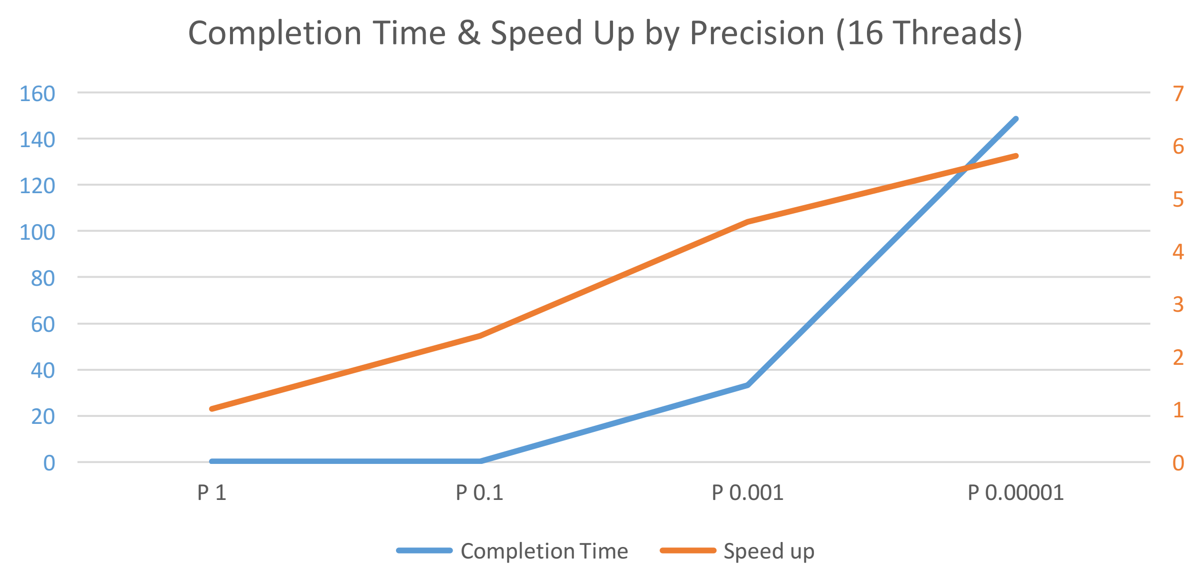


As shown above, for a fixed hardware size, the speed up increases as the problem size increases, as per Gustafson's Law.

Precision Tests – Gustafson’s Law

These tests again demonstrate Gustafson’s Law, with fixed hardware of 16 threads, but this time increasing the problem size through means of precision. The speed up increases as the program aims to be more precise. The same problems were computed sequentially, and those speeds were used to calculate the speed up.

Threads	Array Dimensions	Precision	Average Completion Time over 3 Attempts (seconds)	Speed up (s/p)
16	500x500	1	0.083644602	0.9998
16	500x500	0.1	0.139587870	2.3860
16	500x500	0.001	33.199836617	4.5450
16	500x500	0.00001	148.644991221	5.8039



Again, the speed up increases as the problem gets more difficult. However, throughout these tests, we have only examined speed up. There is another important measure which is efficiency of the hardware use.

Efficiency Results

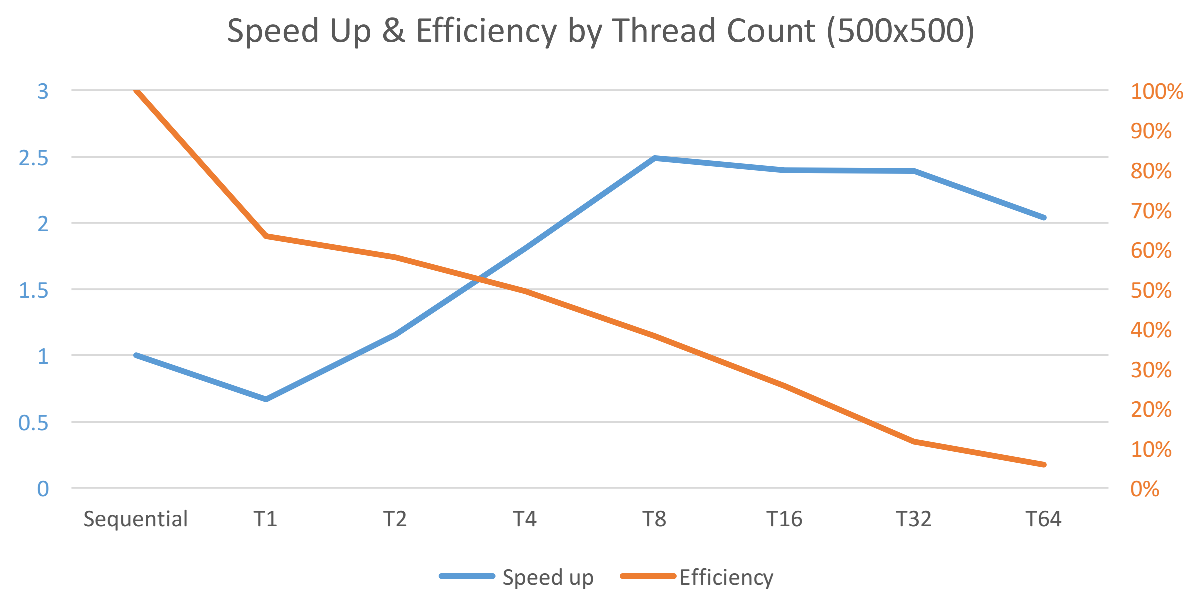
Using the above tests and their computed speed ups, we will look into efficiency. Efficiency gauges the cost of a parallel system – the higher the efficiency, the better the utilisation of the processors.

I hypothesize that with a fixed problem size but an increasing set of hardware, the speed up will increase, but the efficiency will decrease. With a fixed set of hardware but an increasing problem size, I believe the speed up will increase, and so too will the efficiency. These hypotheses draw on Amdahl and Gustafson's laws.

For this section, we will use the results from the Thread Count Tests, and the Dimension Tests, as both the Precision tests and Dimension tests demonstrate increasing problem size.

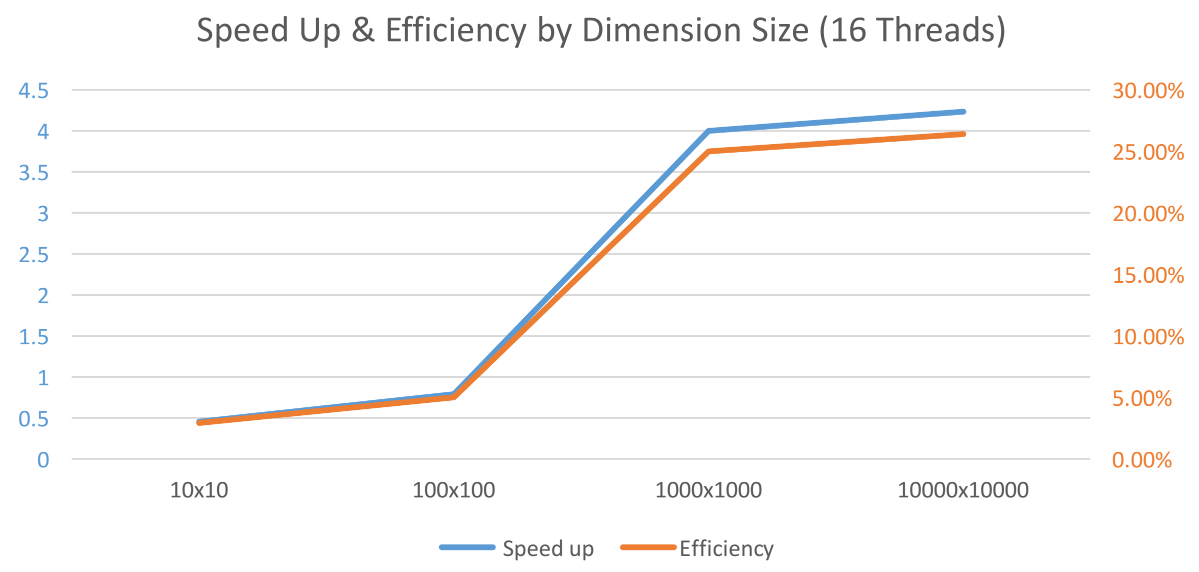
Thread Count Tests:

Threads	**Array Dimensions**	**Precision**	**Speed up (s/p)**	**Efficiency % (speed up/threads)**
Sequential	5000x5000	0.1	1.000	100%
1	5000x5000	0.1	0.634	63.4%
2	5000x5000	0.1	1.159	58%
4	5000x5000	0.1	1.975	49.4%
8	5000x5000	0.1	3.054	38.2%
16	5000x5000	0.1	4.097	25.6%
32	5000x5000	0.1	3.735	11.7%
64	5000x5000	0.1	3.698	5.8%



Dimension Tests:

Threads	Array Dimensions	Precision	Speed up (s/p)	Efficiency (speed up/threads)
16	10x10	0.1	0.457	2.9%
16	100x100	0.1	0.793	5%
16	1000x1000	0.1	3.998	25%
16	10000x10000	0.1	4.229	26.4%



As hypothesized, the efficiency decreases as the hardware set grows larger on a fixed problem, whereas on fixed hardware and an increasing problem set, the efficiency increases.