Parallel Coursework 2

Testing Report

Introduction

This testing report details the testing performed for Parallel Computing, coursework 2. We begin by detailing the correctness of the solutions, and then look into further test cases and conclusion.

Below is a partial extract from the included readme, for instructions on running the code.

> Compile sequential.c using: gcc -Wall -o sequential sequential.c -lrt
> Compile ompi-parallel.c using: mpicc -Wall -o ompi-parallel ompi-parallel.c
>
> Both programs can be run with these possible flags:
>
> - -debug : The level of debug output: 0, 1, 2, 3 (*default: 0*)
> - -d : integer length of the square array (*default: 10*)
> - -p : how precise the relaxation needs to be before the program ends, as a double (*default: 0.0001*)
> - -g : 0 to use values from file specified, 1 to generate them randomly (*default: 0*)
> - -f : string, path of the textfile to use (*default: ../Values/values.txt*)
>
> For example:
>
> - ./sequential -debug 2 -d 500 -p 0.01 -f values.txt
> - mpirun -np 16 ./ompi-parallel -debug 1 -d 10000 -p 0.001
>
> Excluding a flag will use the default value.

-lrt lets us use librt, which is the POSIX.1b Realtime Extension. We use it for timing the program.

-Wall enables all of the compilers warning messages, which may otherwise be suppressed.

-o designates the build output file

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 precision. All tests are conducted 3 times to help increase accuracy.

Value set 1: valueSet1.txt – dimension: 4x4, precision: 0.1, parallel threads: 1, 2, 4, 8
Value set 2: valueSet2.txt – dimension: 6x6, precision: 0.05, parallel threads: 1, 2, 4, 8

	Value Set 1 – 4x4, 0.1 (Pass for result identical to manual)	**Value set 2 – 6x6, 0.05 (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.

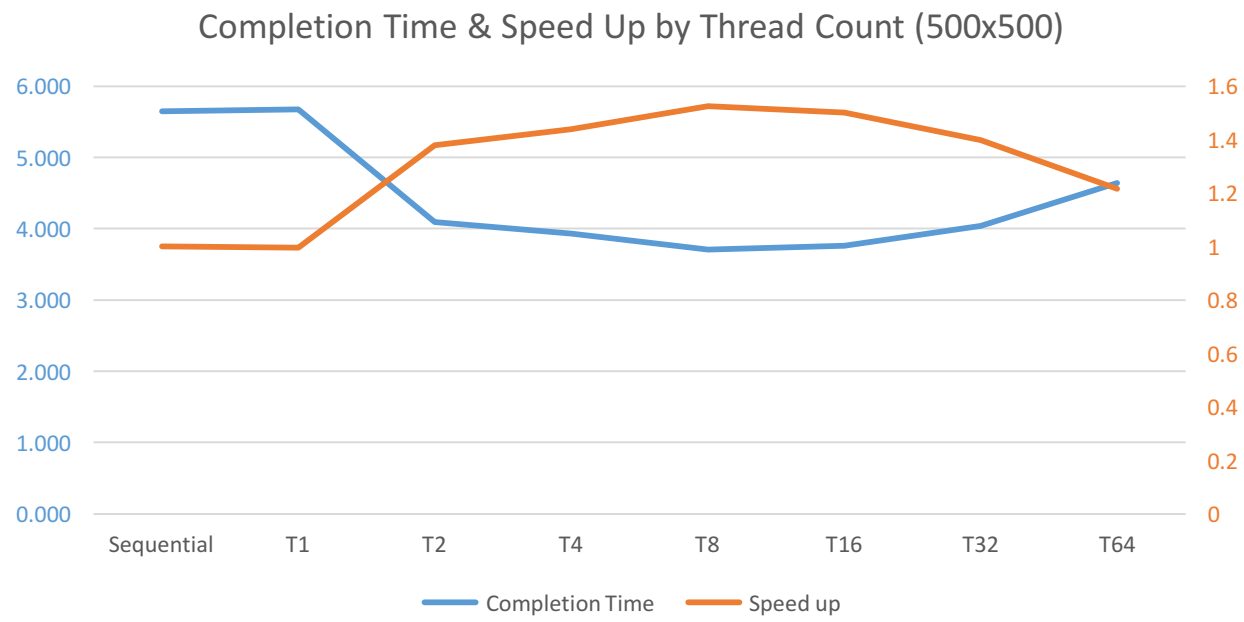
Throughout the rest of this Testing Report, we run a number of tests, focusing on changing Thread Count, Dimensions and Precision, and how they are inextricably linked. We also look at Speed up and Efficiency and how they change based on increasing hardware or increasing problem size, in line with Amdahl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

For all the tests computed throughout this report, we utilise the same set of numbers, which are provided for you within this zip file, at Values/values.txt. This is to help ensure accuracy of the testing process.

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).

Cores	Array Dimensions	Precision	Average Completion Time over 3 Attempts	Speed up (s/p)
Sequential	1,000x1,000	0.001	5.644 seconds	1.000
1	1,000x1,000	0.001	5.671 seconds	0.995
2	1,000x1,000	0.001	4.095 seconds	1.378
4	1,000x1,000	0.001	3.926 seconds	1.438
8	1,000x1,000	0.001	3.704 seconds	1.524
16	1,000x1,000	0.001	3.760 seconds	1.501
32	1,000x1,000	0.001	4.040 seconds	1.397
64	1,000x1,000	0.001	4.641 seconds	1.216

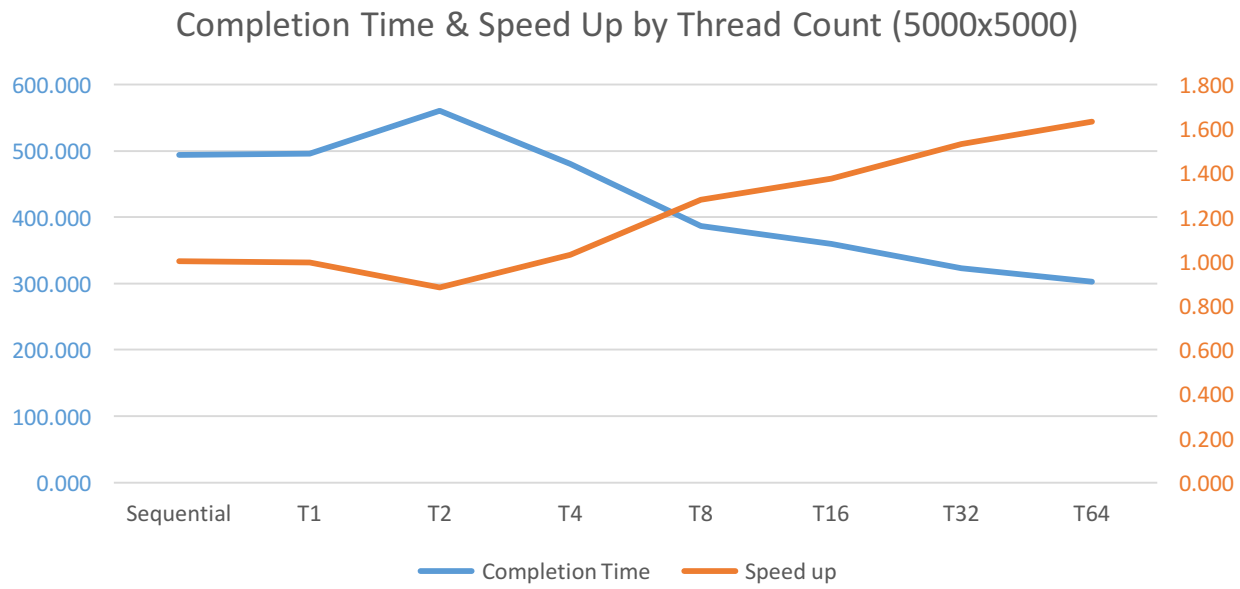


For an array of 500x500 and precision 0.1, the program continues to speed up with an increasing number of cores, until 16 cores are used where the program then slows. This shows the overhead generated exceeds the benefit of more cores at this array size. 32 and 64 cores increasingly slowed the program. A maximum of 4 nodes were used, with a total of 16 cores per node.

Surprisingly, it is shown that the overhead of creating an extra thread does not overshadow the benefits of computing the program over two cores, as opposed to sequentially completing the program, as the average completion time still lowers. However, this may be due where in the program the start timing is taken (MPI_Init).

The tests were repeated with an array size 10,000x10,000. It is expected that increasing the problem size for a fixed amount of cores will improve the relative speedup, and as such I hypothesize that 16, 32 and even 64 cores will run faster than 8 cores for this size problem.

Threads	Array Dimensions	Precision	Average Completion Time over 3 Attempts	Speed up (s/p)
Sequential	10,000x10,000	0.001	494.455 seconds	1.000
1	10,000x10,000	0.001	496.414 seconds	0.996
2	10,000x10,000	0.001	560.331 seconds	0.882
4	10,000x10,000	0.001	480.263 seconds	1.030
8	10,000x10,000	0.001	386.794 seconds	1.278
16	10,000x10,000	0.001	359.333 seconds	1.376
32	10,000x10,000	0.001	323.117 seconds	1.530
64	10,000x10,000	0.001	302.744 seconds	1.633



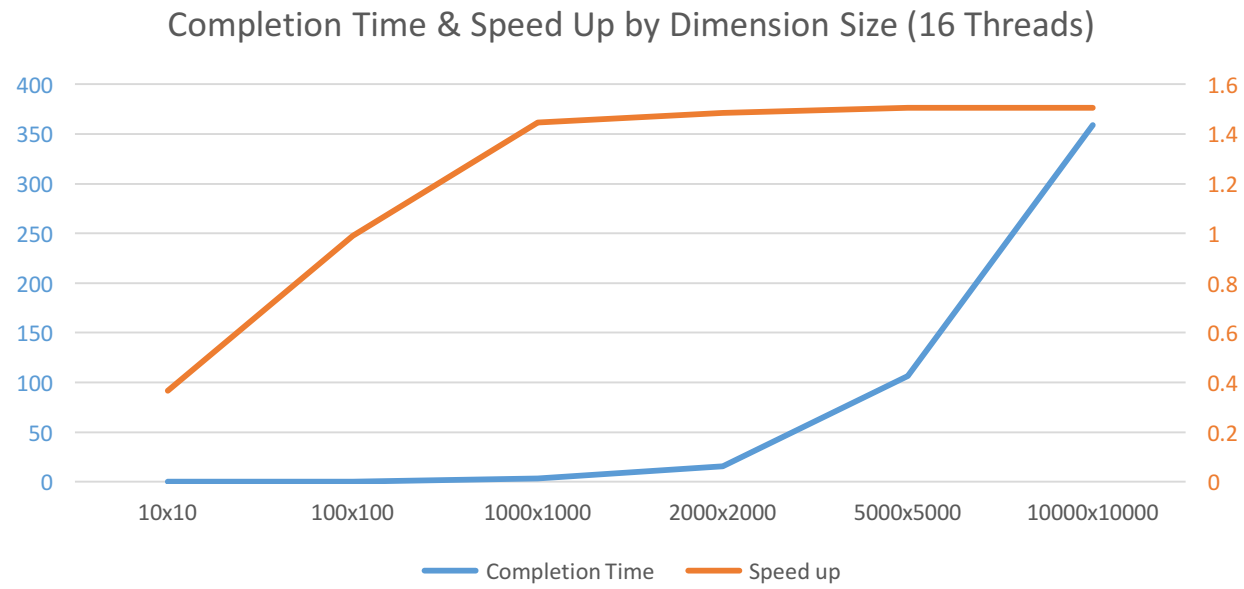
As shown above, my hypothesis was correct – an increasing problem size over a fixed set of hardware yields a greater benefit from that hardware. This is the beginning of Gustafson's Law, which will be detailed further in the coming tests.

Despite this, the speed up is far from ideal, reaching just 1.376 on 16 cores. This likely demonstrates a large sequential part of my application, which can be explained by looking at two large parts of the program:
Both reading in the initial array of numbers, and re-building the new array from slaves reply, are sequential parts of my program.

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 should increase 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	0.001	0.0069	0.368
16	100x100	0.001	0.1628	0.990
16	1,000x1,000	0.001	3.7835	1.447
16	2,000x2,000	0.001	15.5264	1.486
16	5,000x5,000	0.001	106.3269	1.505
16	10,000x10,000	0.001	358.6400	1.506

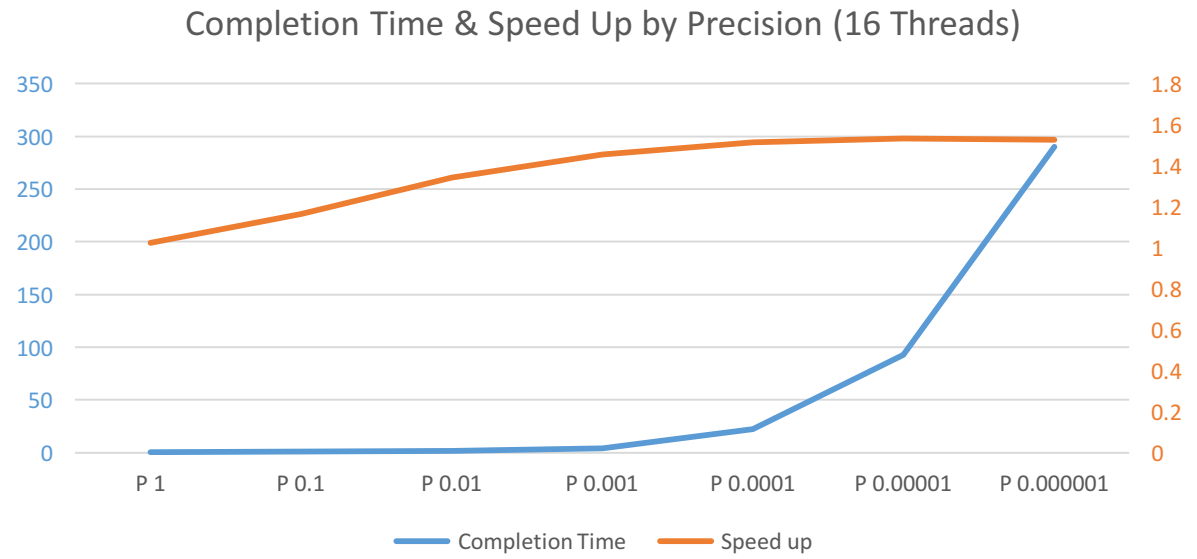


Again, the speed up increases as the problem gets more difficult. Interestingly, the speed up seems to level off after the 1000x1000 mark. Again this could potentially be due to the nature of part of my program being largely sequential when reading in from the initial file, or correlating the results from the slaves. If these areas of the program were parallelised, the speed up would likely increase further for larger problems.

Precision Tests – Gustafson's Law

These tests again attempt to 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	1,000x1,000	1	0.697	1.023
16	1,000x1,000	0.1	0.841	1.163
16	1,000x1,000	0.01	1.403	1.342
16	1,000x1,000	0.001	3.773	1.455
16	1,000x1,000	0.0001	22.130	1.513
16	1,000x1,000	0.00001	92.593	1.531
16	1,000x1,000	0.000001	290.094	1.545



Again, the speed up increases as the problem becomes greater, as expected by Gustafson's Law, though once again the speed up is minimal.

However, throughout these tests, we have only examined speed up. There is another important measure which is efficiency of the hardware use, and we look at that in the final tests.

Efficiency Results

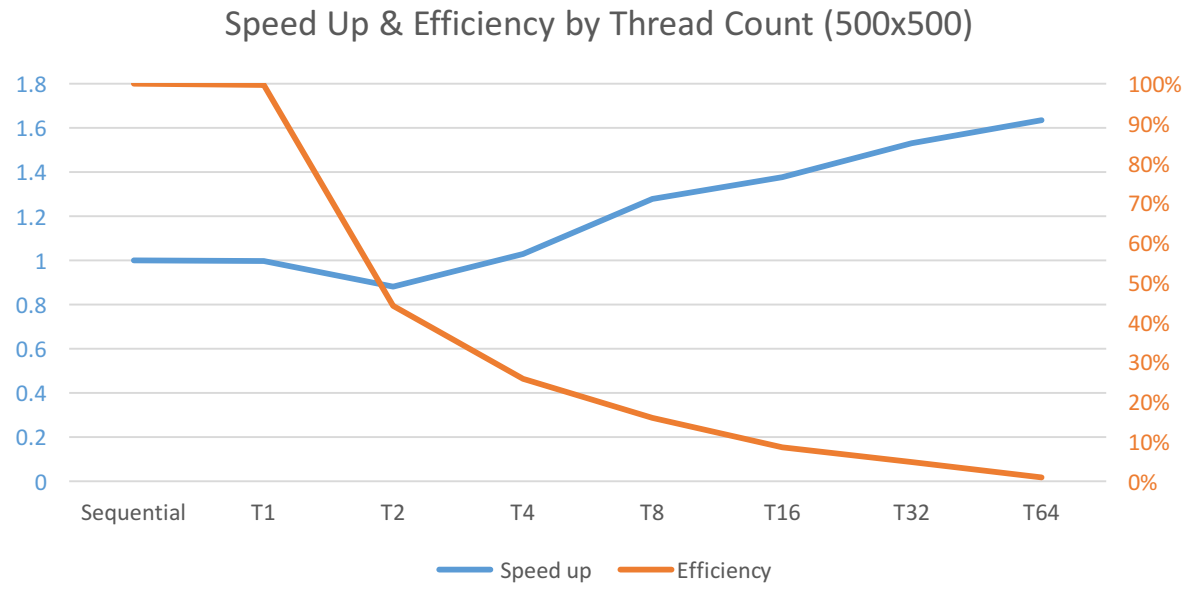
Using the above tests and their computed speed ups, we will look into efficiency. Efficiency gauges the cost of a parallel system – the higher the efficiency, the better the utilisation of the processors.

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 at a lower rate, but the efficiency will increase. 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	10,000x10,000	0.001	1.0000	100%
1	10,000x10,000	0.001	0.996	99.6%
2	10,000x10,000	0.001	0.882	44.1%
4	10,000x10,000	0.001	1.030	25.75%
8	10,000x10,000	0.001	1.278	15.96%
16	10,000x10,000	0.001	1.376	8.6%
32	10,000x10,000	0.001	1.530	4.78%
64	10,000x10,000	0.001	1.633	1.02%

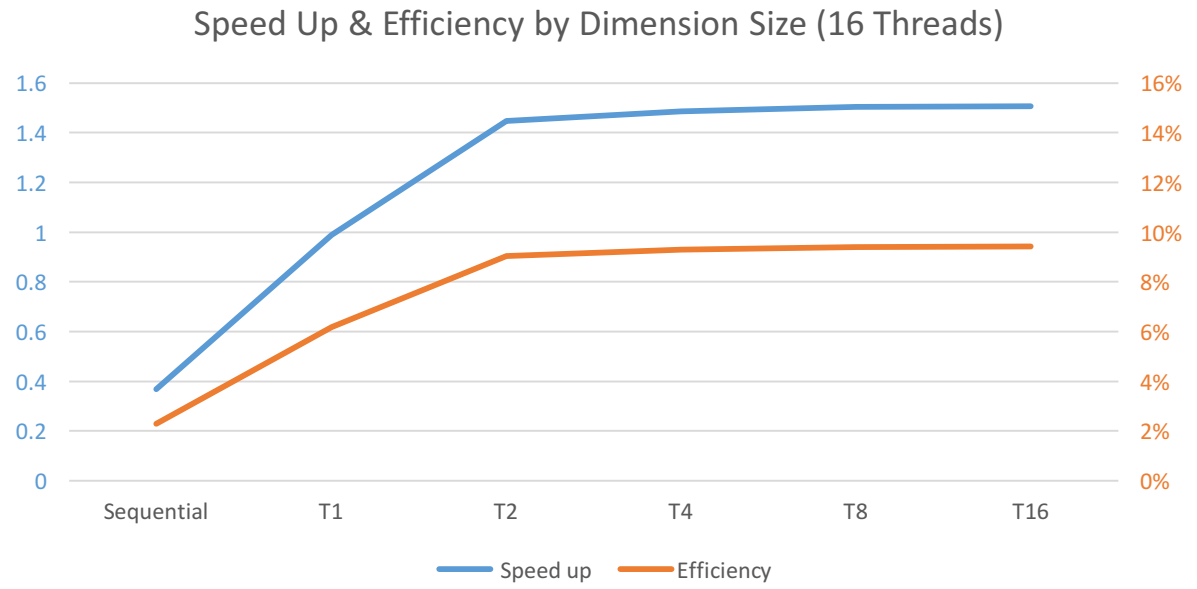


The efficiency remains close to 100% with just the master thread running, but swiftly drops after that, describing the obvious sequential parts of my application. As expected by Amdahl's Law, the efficiency here decreases as more cores are added, while speed up increases.

Next we go on to look at a fixed hardware set on an increasing problem size. The hypothesis, according to Gustafson's Law, will be that the Efficiency will increase, as the problem size increases.

Dimension Tests:

Threads	Array Dimensions	Precision	Speed up (s/p)	Efficiency (speed up/threads)
16	10x10	0.001	0.368	2.30%
16	100x100	0.001	0.989	6.18%
16	1,000x1,000	0.001	1.447	9.05%
16	2,000x2,000	0.001	1.486	9.29%
16	5,000x5,000	0.001	1.505	9.41%
16	10,000x10,000	0.001	1.507	9.42%



Here we see that the initial hypothesis is correct, despite the poor efficiency, even at a large problem size. The efficiency shown here details that my program seems to cause the processes to wait around unused for long periods of time. It's possible this happens when the master thread is combining the results of the slaves sequentially, or distributing the initial array, though the efficiency is still surprisingly low despite the large problem size in the final test.