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. The same values were used for all tests, and can be found at Values/values.txt.

Below is a partial extract from the included README, for instructions on running the code. View the full README for details about the ompi-parallel.c code's logic and how it runs.

> 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 granularity of debug output: 0, 1, 2, 3 (*default: 1*)
> - -d : integer length of the square array (*default: 1000*)
> - -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 sequential 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: 6x6, precision: 0.05, parallel threads: 2, 4, 8
Value set 2: valueSet2.txt – dimension: 10x10, precision: 0.1, parallel threads: 2, 4, 8

	Value Set 1 – 6x6, 0.05 (Pass for result identical to manual)	**Value set 2 – 10x10, 0.1 (Pass for result identical to manual)**
Manual	-	-
Sequential	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.

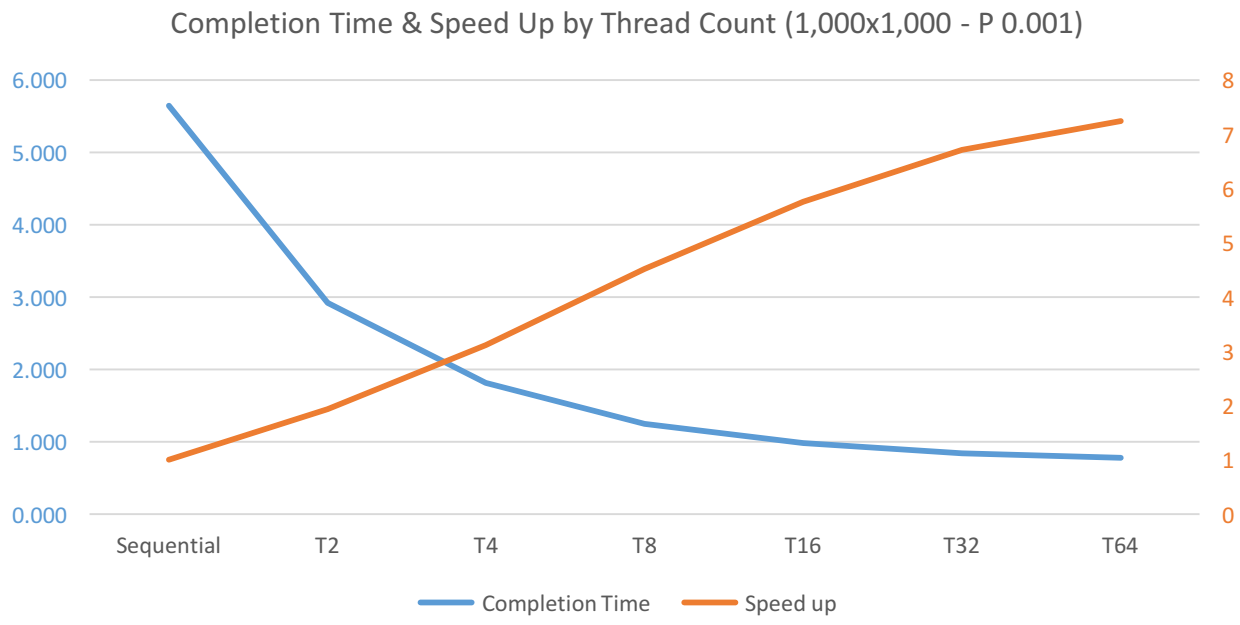
Throughout the rest of this Testing Report, we run a number of tests, focusing on changing Thread Count, Dimensions and Precision, and how they are inextricably linked. We also look at Speed up and Efficiency and how they change based on increasing hardware or increasing problem size, in line with Amdahl 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
2	1,000x1,000	0.001	2.916 seconds	1.935
4	1,000x1,000	0.001	1.812 seconds	3.114
8	1,000x1,000	0.001	1.250 seconds	4.514
16	1,000x1,000	0.001	0.981 seconds	5.754
32	1,000x1,000	0.001	0.841 seconds	6.712
64	1,000x1,000	0.001	0.779 seconds	7.247



For these tests, a maximum of 4 cores were used, with 16 threads per core.

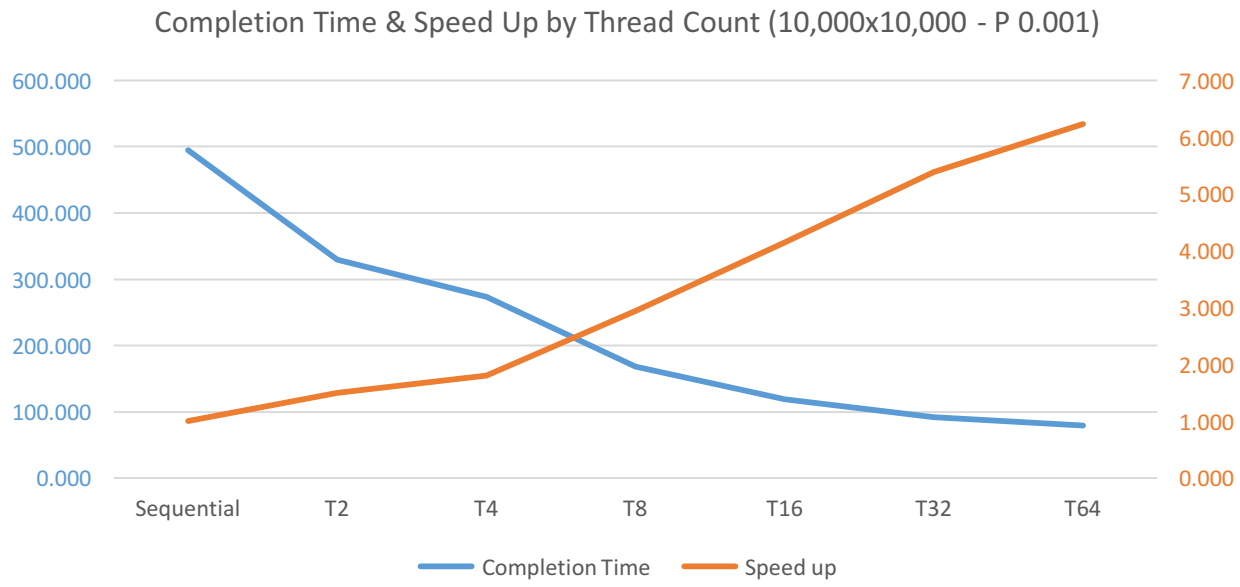
For an array of 1000x1000 and precision 0.001, the program continues to speed up with an increasing number of cores. We also see that the overhead of creating the extra threads doesn't overshadow the benefits of computing the program on two or more cores, as opposed to sequentially completing the program, as the speed is still faster. This is likely due to letting Open MPI handle the multiple processes, as opposed to POSIX.

The trade off is that communication in this system is a bigger concern, as seen in the next test when we increase the dimensions of this same test.

The tests were repeated with an array size 10,000x10,000, and then again with precision set to 0.000001. It is expected that increasing the problem size for a fixed amount of cores will improve the relative speedup.

It is also worth noting that this is a distributed memory system, and as such the cost of communication will play an important part. It is likely that the larger dimension tests will not achieve as good speed up as the finer precision tests, as not only is the problem size increasing, but also the amount of communication.

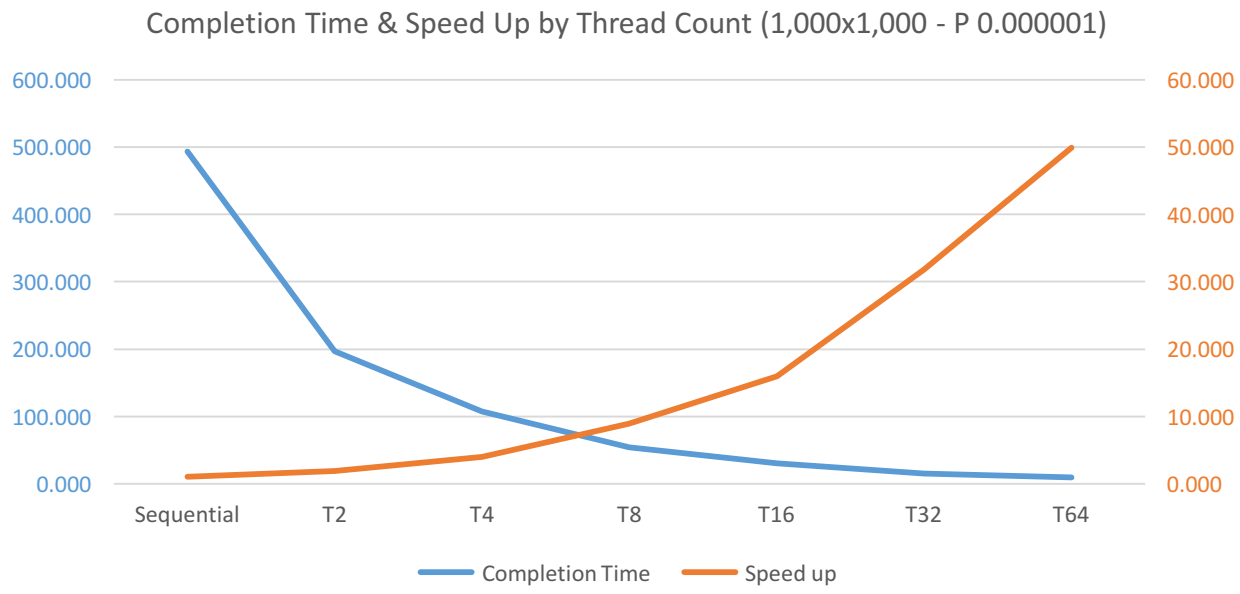
Threads	Array Dimensions	Precision	Average Completion Time over 3 Attempts	Speed up (s/p)
Sequential	10,000x10,000	0.001	494.455 seconds	1.000
2	10,000x10,000	0.001	329.546 seconds	1.500
4	10,000x10,000	0.001	273.742 seconds	1.806
8	10,000x10,000	0.001	168.073 seconds	2.942
16	10,000x10,000	0.001	119.250 seconds	4.146
32	10,000x10,000	0.001	91.790 seconds	5.387
64	10,000x10,000	0.001	79.368 seconds	6.230



We can see that increasing the dimensions has lowered the overall relative speed up, and this is most likely due to the communications cost or the large sequential file read at the beginning of the program, as the amount of data involved has increased 100 fold. Our next test will look at making the problem bigger through means of increasing the precision, which has the benefit of keeping the data cost the same per cycle, despite scaling the problem.

It is also worth noting that this is a distributed memory system, and as such the cost of communication will play an important part. It is likely that the larger dimension tests will not achieve as good speed up as the finer precision tests, as not only is the problem size increasing, but also the amount of communication.

Threads	Array Dimensions	Precision	Average Completion Time over 3 Attempts	Speed up (s/p)
Sequential	1,000x1,000	0.000001	492.944 seconds	1.000
2	1,000x1,000	0.000001	196.811 seconds	1.945
4	1,000x1,000	0.000001	107.012 seconds	3.951
8	1,000x1,000	0.000001	54.301 seconds	8.957
16	1,000x1,000	0.000001	30.859 seconds	15.974
32	1,000x1,000	0.000001	15.503 seconds	31.797
64	1,000x1,000	0.000001	9.884 seconds	49.873

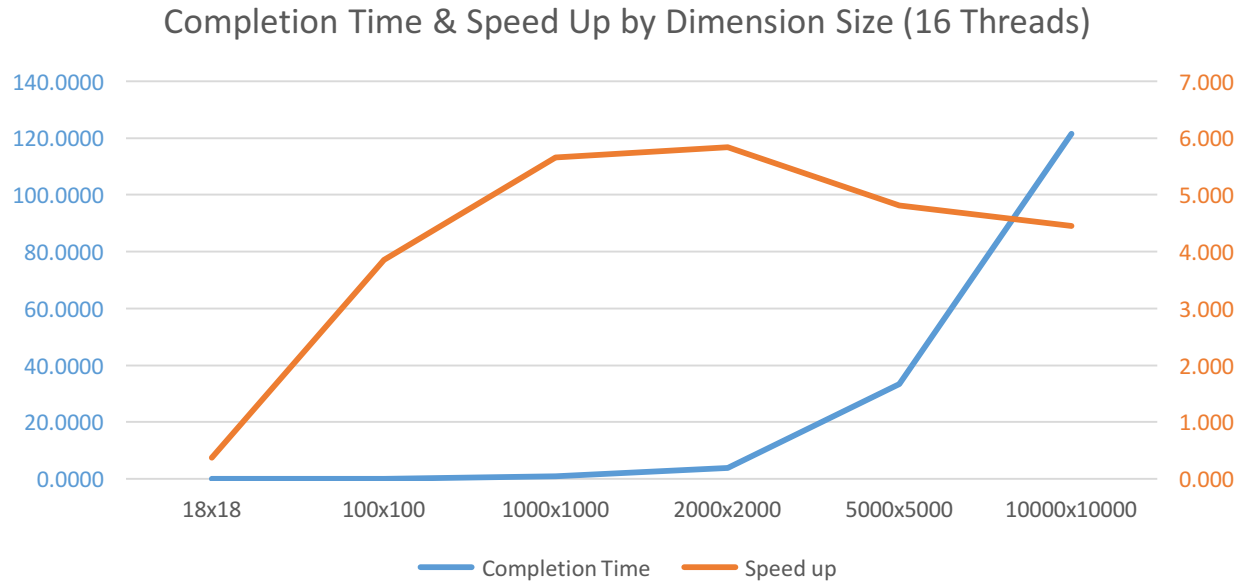


As shown above, my hypothesis was correct – an increasing problem size over a fixed set of hardware yields a greater benefit from that hardware. To note especially is that up to 32 threads, this example achieved near perfect speed up. This is the beginning of Gustafson's Law, which will be detailed further in the coming tests.

Dimension Tests – Gustafson's Law

These next tests attempt to 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. It is useful to remember that the larger problem sizes had a negative impact, most likely due to communications costs or the initial file input, and this can be seen reflected in this first test.

Threads	Array Dimensions	Precision	Average Completion Time over 3 Attempts	Speed up (s/p)
16	18x18	0.001	0.0068 seconds	0.369
16	100x100	0.001	0.0418 seconds	3.850
16	1,000x1,000	0.001	0.9685 seconds	5.654
16	2,000x2,000	0.001	3.9480 seconds	5.843
16	5,000x5,000	0.001	33.2618 seconds	4.812
16	10,000x10,000	0.001	121.3832 seconds	4.453

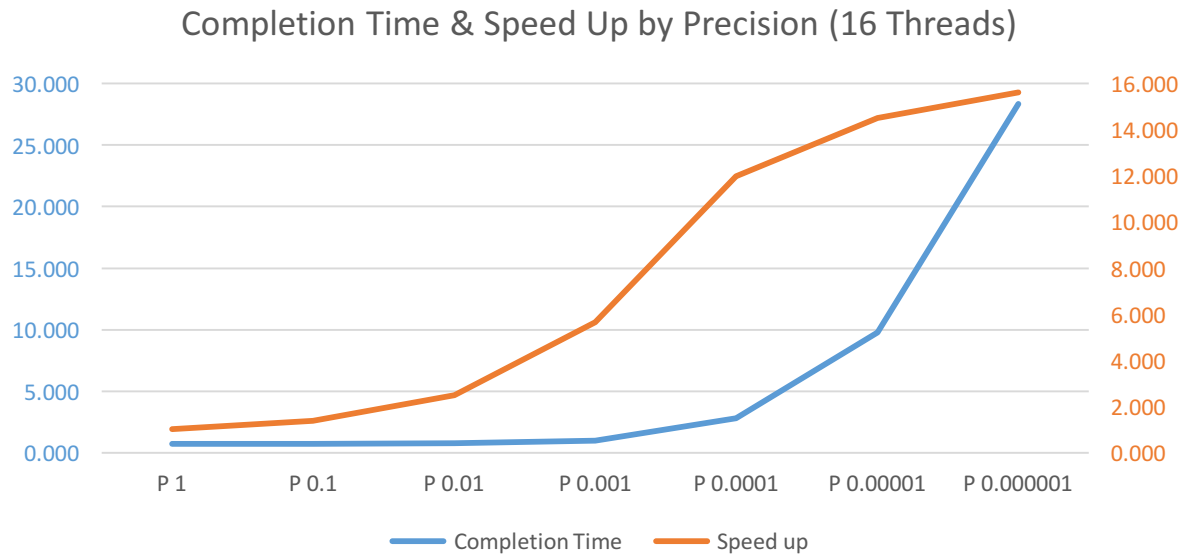


Again, the speed up increases as the problem gets more difficult. Interestingly, the speed up peaks at 2,000x2,000, and then begins to decrease. Again this could potentially be due to the cost of communication or the sequential file input, and how it increases with a larger problem size. The next test demonstrates stepped increments of precision granularity, in the hope to display an increasing problem size without increasing overheads.

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 should increase as the program increases the precision. 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	0.696 seconds	1.024
16	1,000x1,000	0.1	0.705 seconds	1.386
16	1,000x1,000	0.01	0.755 seconds	2.496
16	1,000x1,000	0.001	0.973 seconds	5.643
16	1,000x1,000	0.0001	2.796 seconds	11.971
16	1,000x1,000	0.00001	9.771 seconds	14.511
16	1,000x1,000	0.000001	28.329 seconds	15.619



Here we can clearly see Gustafson's Law at work – as the problem size increases via the precision granularity, we can see the speed up clearly tending towards p, in this case 16, which is expected.

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

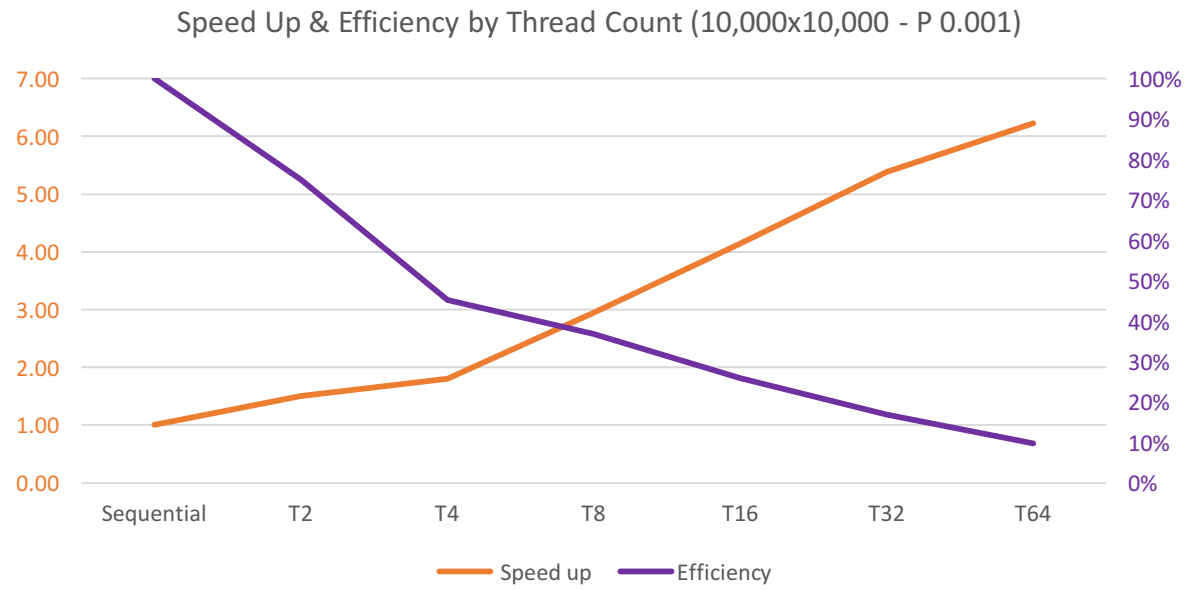
Using the above tests and their computed speed ups, we will look into efficiency. Efficiency gauges the cost of a parallel system – the higher the efficiency, the better the utilisation of the processors.

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, the Dimension Tests and the Precision Tests. Although both the Precision tests and Dimension tests demonstrate increasing problem size, the Dimension Test also includes increase in communication costs.

Thread Count Tests:

Threads	Array Dimensions	Precision	Speed up (s/p)	Efficiency % (speed up/threads)
Sequential	10,000x10,000	0.001	1.000	100%
2	10,000x10,000	0.001	1.500	75%
4	10,000x10,000	0.001	1.806	45.15%
8	10,000x10,000	0.001	2.942	36.78%
16	10,000x10,000	0.001	4.146	25.91%
32	10,000x10,000	0.001	5.387	16.83%
64	10,000x10,000	0.001	6.230	9.73%

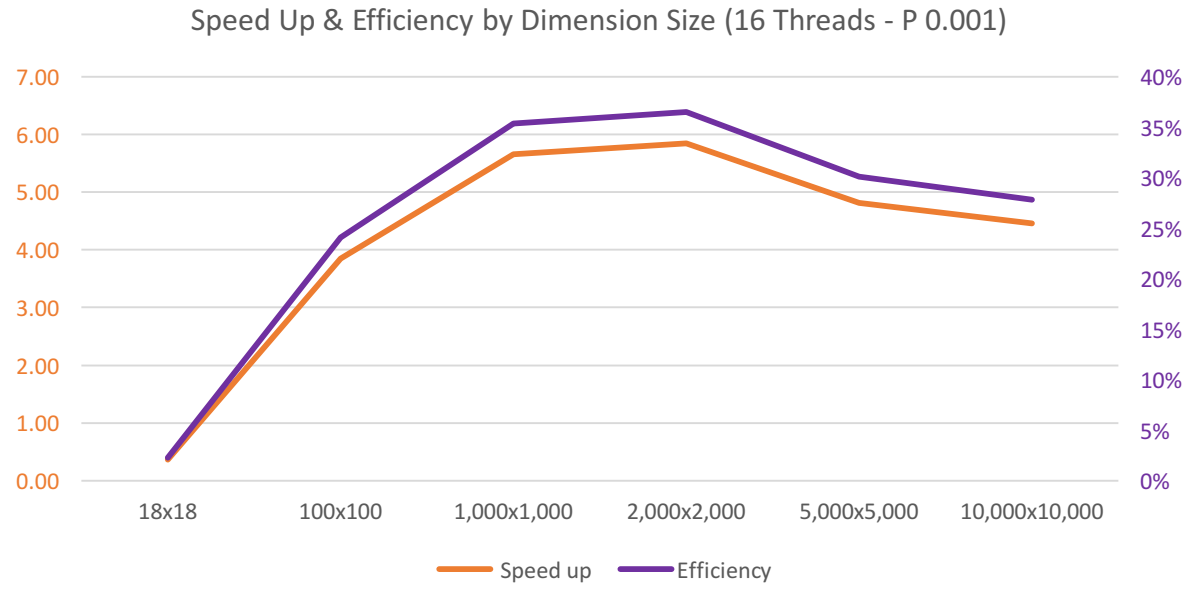


As expected, the efficiency decreases as the number of threads increases, but the extra hardware does increase the speed up of the problem, inline with Amdahl's Law.

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	18x18	0.001	0.369	2.30%
16	100x100	0.001	3.850	24.06%
16	1,000x1,000	0.001	5.654	35.34%
16	2,000x2,000	0.001	5.843	36.52%
16	5,000x5,000	0.001	4.812	30.08%
16	10,000x10,000	0.001	4.453	27.83%

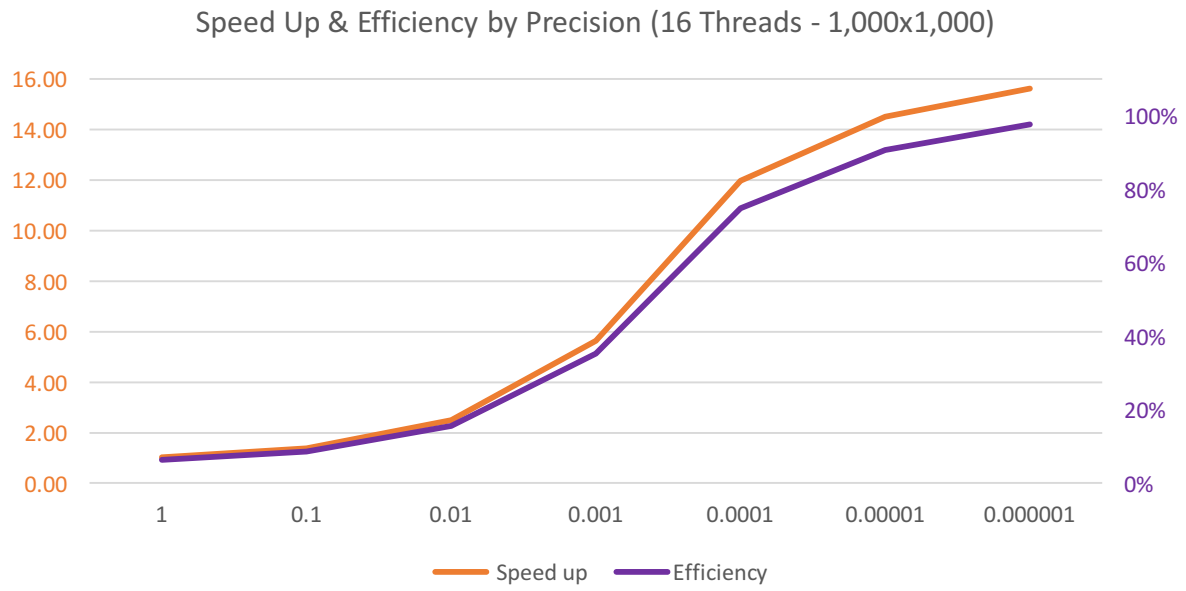


We can see here that, to begin with, increasing the problem size increases speed up and efficiency, but the overheads of the large problem size begin to exceed the benefit of the threads. This could be due to communication costs, or the initial reading in of the input file, which is sequential. We therefore look to the final test of precision to see how efficiency is truly affected by an increasing problem size on a fixed set of hardware.

Finally, we look at the relationship between an increasing problem size, speed up and efficiency, utilising increasing precision.

Precision Tests:

Threads	Array Dimensions	Precision	Speed up (s/p)	Efficiency (speed up/threads)
16	1,000x1,000	1	1.024	6.40%
16	1,000x1,000	0.1	1.386	8.66%
16	1,000x1,000	0.01	2.496	15.60%
16	1,000x1,000	0.001	5.643	35.27%
16	1,000x1,000	0.0001	11.971	74.82%
16	1,000x1,000	0.00001	14.511	90.69%
16	1,000x1,000	0.000001	15.619	97.62%



As demonstrated here, an increasing problem size without the extra burden of more communication allows the program's efficiency to increase and, in this case, tend towards 100%, as well as speed up tend to the perfect P, being the number of processes.