Performance Analysis & Evaluation

Parallel Computing CW2 – Brute Force Encryption Cracking

Submission date: 6 December 2018 1400Hrs

Student ID: 09028091

Benchmark Tests

We have three programs to be tested – the serial algorithm, the OpenMP parallel algorithm, and the MPI parallel algorithm. These will be run on the UWE cluster (IP addresses 164.11.39.11-4), the serial and OpenMP variants on individual nodes and the MPI variant across all nodes.

The serial program will be run 10 times on a node, and both parallel programs will be run 10 times for each number of threads selected, and from these data mean averages will be calculated. Extra tests will be run every time we get unpredictable slowdown (caused by background processes, other users running code, or similar uncontrollable variables) – this is necessary in order to get results representative of real-world applications where uncontrolled access and interference would be minimal, if not impossible.

The key used to test the function is “AAAAA”, chosen arbitrarily to be near the middle of the search space, again as in the real world a password cracker’s speed would be a function of the key’s position in the search space – and its proximity to a given thread’s start point in that space. We could try to avoid this by choosing “ZZZZZ” which will always be at the end of the space, but again this would not be a true picture of parallel brute force operation in a real-world problem.

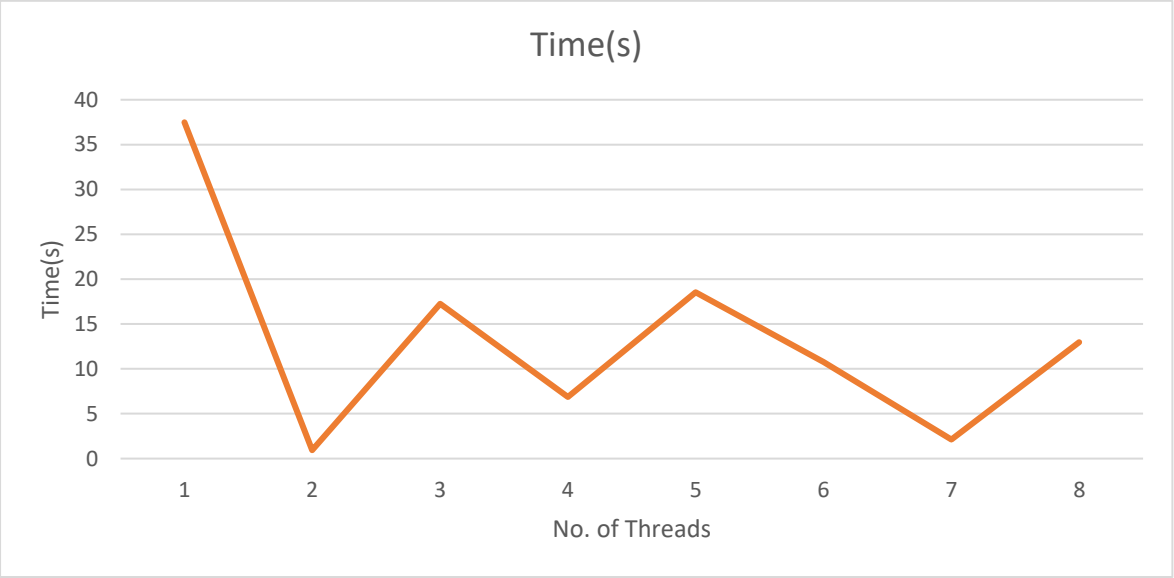
Results

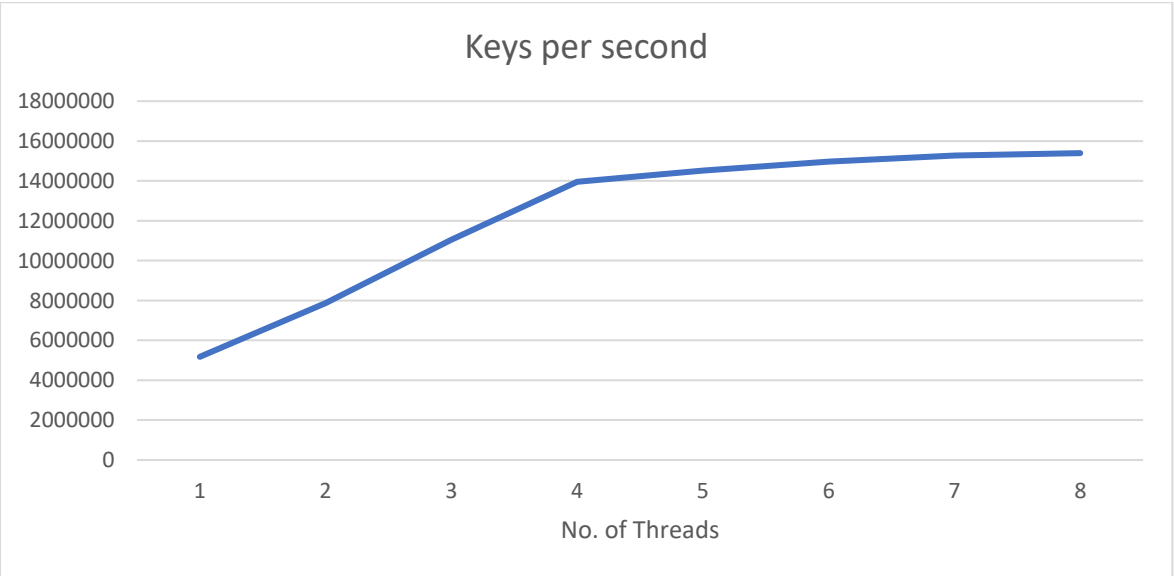
Serial

Time(s)	Keys per second
37.84958	5121048

OpenMP

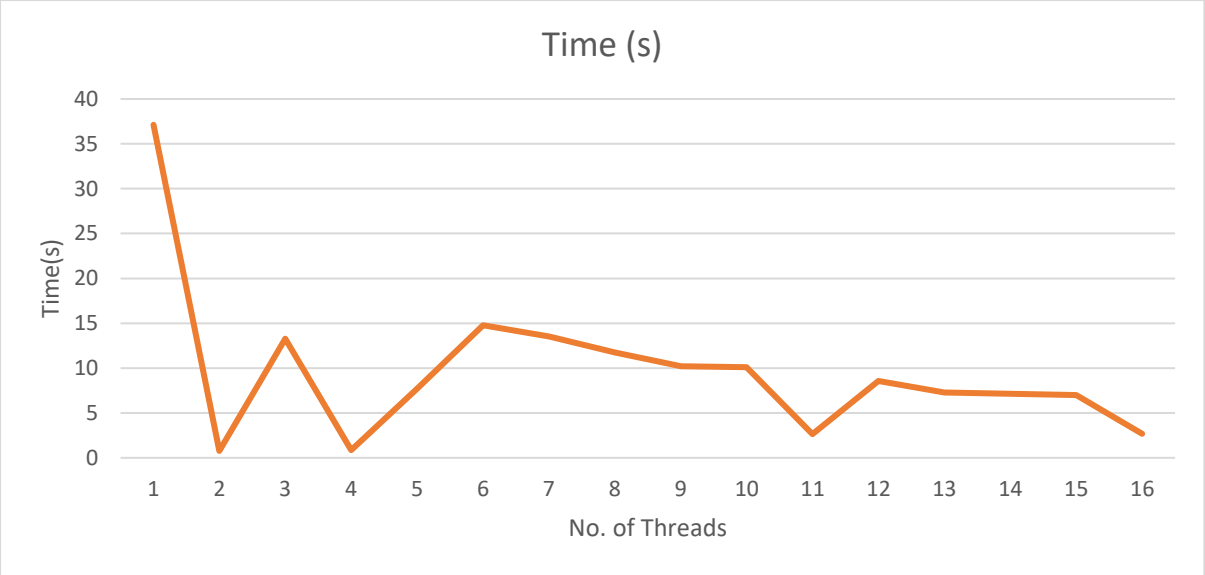
No. of Threads	Time(s)	Keys per second
1	37.48585	5176555
2	0.937575	7876695
3	17.23845	11050114
4	6.875596	13949802
5	18.53608	14511530
6	10.74828	14962967
7	2.134181	15267202
8	12.9622	15393068

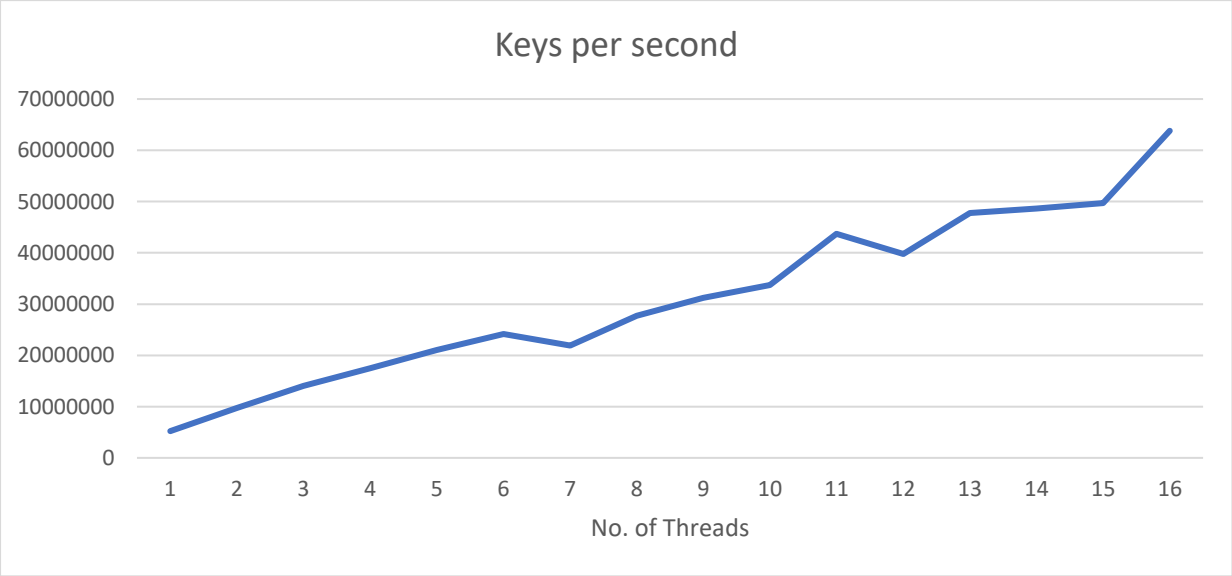




MPI

No. of Threads	Time(s)	Keys per second
1	37.12127	5221766
2	0.768716	9726674
3	13.29495	14037231
4	0.85223	17479129
5	7.722723	21029364
6	14.75881	24131806
7	13.50945	21924918
8	11.73166	27741286
9	10.19804	31197924
10	10.09434	33691946
11	2.652031	43691115
12	8.550959	39770551
13	7.279281	47795685
14	7.148836	48638473
15	6.984819	49736179
16	2.670422	63807476





Analysis

Overhead

Firstly, we should consider the amount of overhead introduced by making a program run in parallel form. Overhead can be caused by inter-process interactions, idling, and excess computation not performed in serial. It is calculated by taking the serial time T_s and subtracting it from the total time taken by all processes T_{all}, as given by:

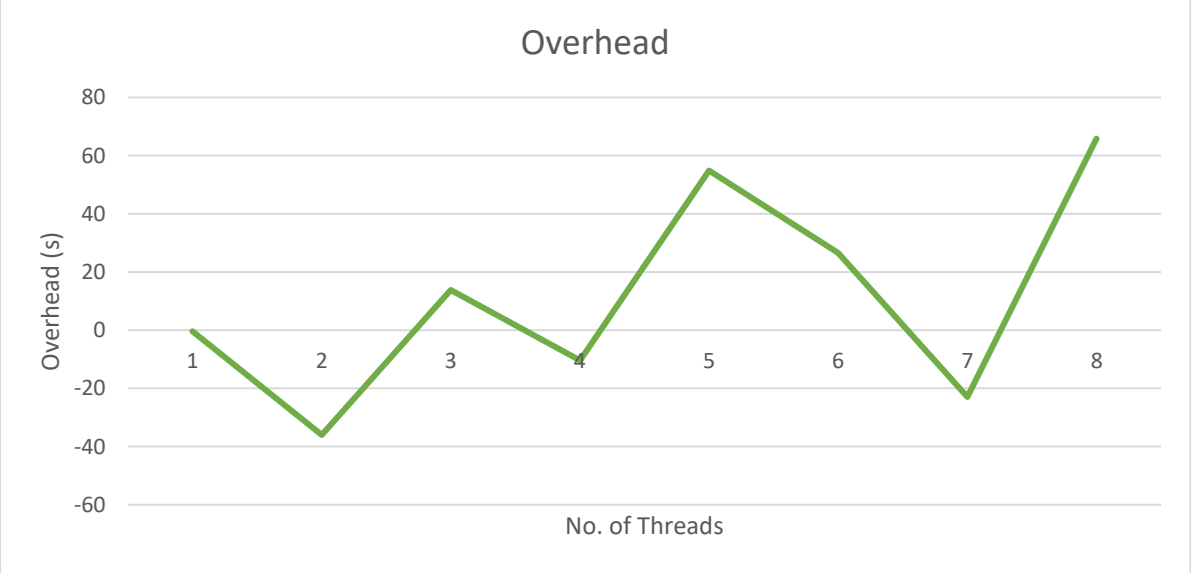
$$T_o = p\,T_P - T_S$$

Where p is the number of processes (or threads).

Doing this calculation gives us the following:

OpenMP

No. of Threads	T_O
1	-0.36374
2	-35.9744
3	13.86578
4	-10.3472
5	54.83083
6	26.64008
7	-22.9103
8	65.84805



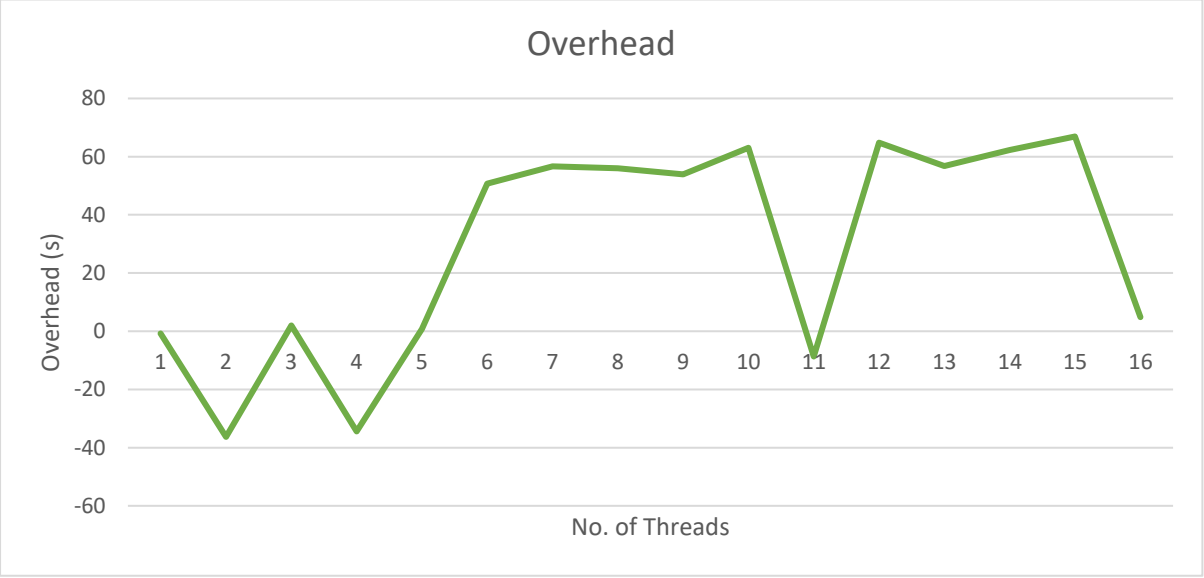
As we see, overhead is hard to calculate here due to the nature of the parallelism – if the correct key is near the start position of one thread's search space, we see such dramatic speed increases that there is a negative overhead compared to the serial version.

Time does not permit us to alter the program to investigate this in greater detail, and it is uncertain if it would be of great benefit. We have enough data at least to know that, when we are not benefitting from key-related speedup, there is a significant overhead at 3, 5 and 8 threads and I would expect this to be reflected at all levels where we able to run enough tests with randomised keys. Doing this would take too long and would deprive other students of use of the cluster, so this cannot be done at this time.

The small negative overhead in single-threaded operation appears to be erroneous, but in fact was reproduced consistently in repeated test runs. It's not clear why this would occur as the code is logically the same when run this way, and the OpenMP code has to explicitly spawn a thread. My speculation is that the spawned thread is bound to an unused core on the node while the serial code runs on the master core, causing a minor speed increase, but I currently cannot check this so cannot know for sure.

MPI

No. of Threads	T_O
1	-0.72831
2	-36.3121
3	2.035255
4	-34.4407
5	0.764033
6	50.70325
7	56.71659
8	56.00369
9	53.93275
10	63.09384
11	-8.67724
12	64.76193
13	56.78107
14	62.23412
15	66.92271
16	4.877168



Again, here we have the same problem as with the OpenMP where the optimised search space obscures the real overhead of our program. However, due to the increased number of available cores and the more consistent speed increases as we go up, we can see a clear overhead of between roughly 55-65 seconds for all process levels above 5. My initial thoughts are that this is caused by inter-node communication processing, as it appears once MPI scales the code beyond the 4 cores of the first node (while 5 threads enjoy a key-related speedup which negates this) and can be seen to

remain constant. Intra-node overhead however appears minimal looking at the 3-thread data, and this is to be expected due to MPI essentially running 3 independent serial programs at once on 3 unused cores.

We see also a minor negative overhead in single-threaded performance, likely for the same reasons as in OpenMP.

Speedup & Efficiency

Speedup (S) is the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p identical processing elements.

Speedup can also be computed using the Keys Per Second throughput of the parallel algorithms, which we can see is more consistent than the time measured due to removing the significance of the key position. However, in MPI there is a latency between one thread finding the correct key, and this being communicated to other threads. During this latency the other threads are checking keys unnecessarily, which inflates the measurement, and there is no way to remove this surplus from the data. Therefore, I have only referred to time-taken as a better reflection of total program performance.

Efficiency is a measure of the fraction of time for which a processing element is usefully employed, given by:

$$E = \frac{S}{p}.$$

When analysing these data, we do so with reference to Amdahl's Law, which tells us that the maximum achievable Speedup is :

$$\psi \equiv S(p) \leq \frac{1}{f + (1-f)/n} \leq \frac{1}{f}$$

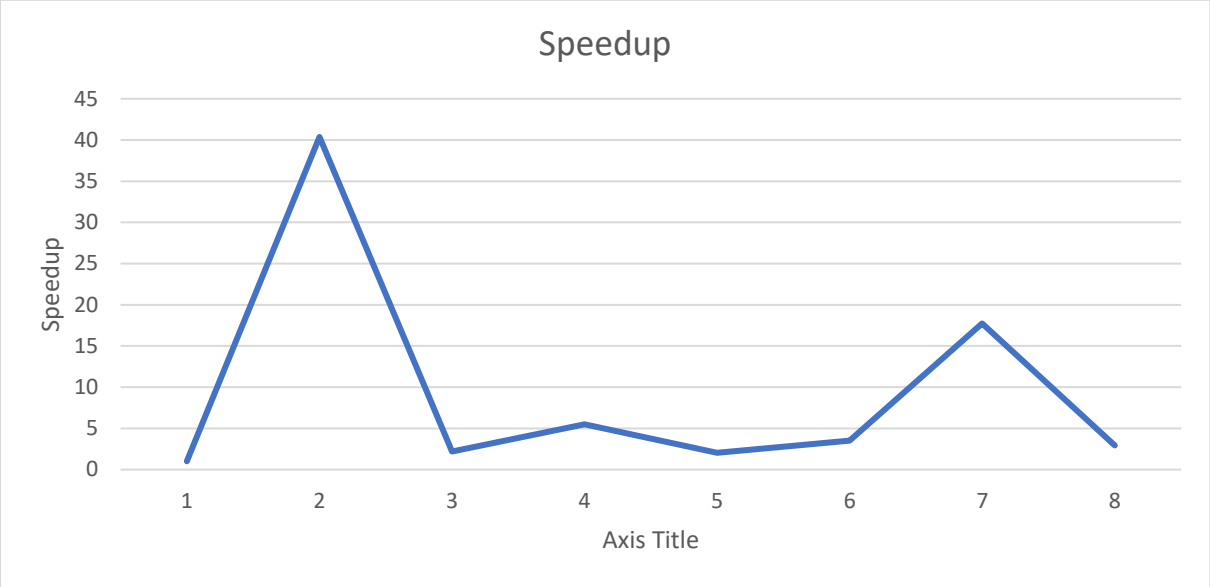
Where f is the proportion of running time, a parallel program spends outside of its parallel execution code. By commenting out the logic in the inner most loop of the code where my program spends most of its time, and running the resulting program, even using the most precise wall clock timer I know about I get an execution time of 0s. As there is clearly some setup code outside my loop structure, I will pick an arbitrary value of 0.01 (which is likely to be far higher than reality) in order to make an assessment, therefore:

$$\Psi = \frac{1}{0.01 + (1-0.01)/n} \leq \frac{1}{0.01}$$

Assuming f holds true, that means the maximum theoretical possible speedup for the algorithm I am using is 100 for an infinite number of cores, and for 16 cores the maximum speedup achievable would be **13.91**. As OpenMP on the cluster only has access to 4 physical cores, the maximum achievable speedup there would be **3.88**. However, both theoretical figures refer to worst-case scenarios, whereas our real-world problem will not approach this complexity.

OpenMP

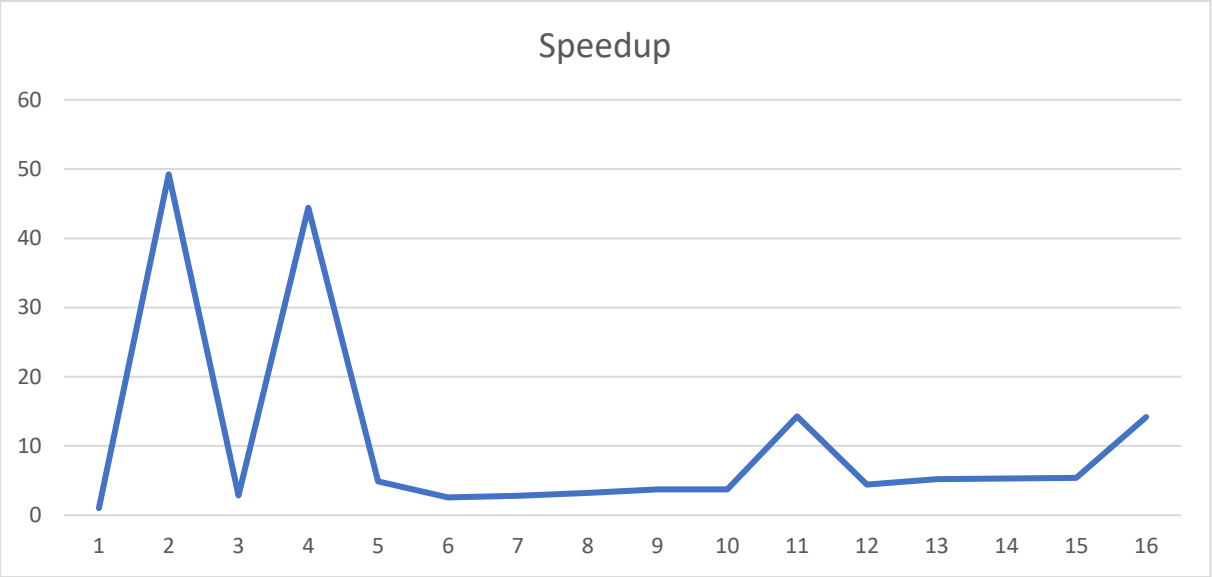
Threads	Speedup	Max Speedup	Efficiency
2	40.36965	1.98	20.18483
3	2.195648	2.94	0.731883
4	5.504917	3.88	1.376229
5	2.041941	3.88	0.408388
6	3.521456	3.88	0.586909
7	17.73494	3.88	2.533563
8	2.919996	3.88	0.364999

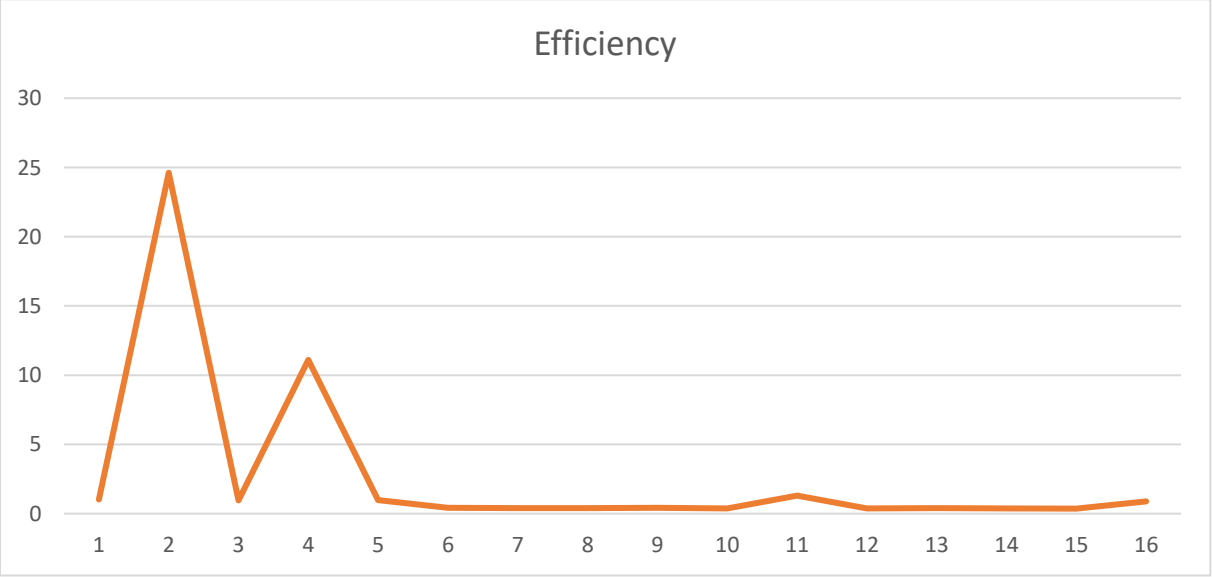


As with the analysis on overhead, it is hard to get a true picture of the speedup and efficiency of this process due to the result being so key-dependent. We have good results for 3 and 6 threaded operations in our speedup measurements, and our efficiency calculations only have 2, 4 and 7 spoiling them, but as this gives us little insight we will move on.

MPI

No. of Threads	Speedup	Max Speedup	Efficiency
2	49.23739	1.98	24.6187
3	2.846915	2.94	0.948972
4	44.41239	3.88	11.1031
5	4.901067	4.81	0.980213
6	2.564542	5.71	0.427424
7	2.801711	6.6	0.400244
8	3.226277	7.48	0.403285
9	3.711458	8.33	0.412384
10	3.749584	9.17	0.374958
11	14.27192	10	1.297448
12	4.426355	10.81	0.368863
13	5.199632	11.61	0.399972
14	5.29451	12.39	0.378179
15	5.418835	13.16	0.361256
16	14.17363	13.91	0.885852





As before, MPI gives us a much better picture of the algorithm, and we can see that apart from key-related speedups there is an efficiency of around 0.35-0.4 which stays quite constant as soon as we enter multi-node operations. Speedups achieved are likewise below half of the theoretically achievable maximum but do climb linearly at a similar rate.

Gustafson's Law

Gustafson's law is an observation about workloads in parallel, where ***p*** is the amount known to be parallelisable and (1-***p***) is not parallelisable.

$$W = (1-p)W + pW.$$

If a system has a speedup of ***s*** then we get:

$$W(s) = (1-p)W + spW.$$

If we divide the second by the first then we get the speedup in latency, which in theory should be linear.

$$S_{\text{latency}}(s) = \frac{TW(s)}{TW} = \frac{W(s)}{W} = 1 - p + sp.$$

The amount known to be parallelisable is, in the worst case, 380,204,032 iterations of calling a 128-bit encryption routine, compared to initialising variables and performing one encryption routine. As 1-***p*** in our case is almost zero, we end up with ***sp***. What this means is that with more computing resources made available we can expect to increase the task size by the same multiple (such as increasing key complexity or more difficult encryption routines) without noticing a decrease in performance.

Algorithm Cost

Finally, let us quickly consider the algorithm cost of the serial and parallel programs. This is defined as the parallel running time multiplied by the number of processors and compared to the serial time taken. This lets us know whether it is in fact efficient to run the parallel algorithm over the serial if we are required to pay for overall CPU usage.

OpenMP

No. of Threads	Cost
2	1.87515
3	51.71536
4	27.50238
5	92.68041
6	64.48966
7	14.93927
8	103.6976

From this data it appears that the OpenMP algorithm is not cost-optimal outside of when the key is in a nice place to be found quickly. This means it would be more efficient to run a linear algorithm to find the password, or even multiple linear algorithms side by side working on restricted search spaces.

MPI

No. of Threads	Cost
2	1.537432
3	39.88484
4	3.408921
5	38.61362
6	88.55284
7	94.56618
8	93.85327
9	91.78233
10	100.9434
11	29.17234
12	102.6115
13	94.63065
14	100.0837
15	104.7723
16	42.72675

MPI similarly has a problem with being cost-optimal once it passes the 4-core single-node threshold, as we can see when it doesn't have an easily found key it spends 90-100 CPU seconds in parallel compared to under 40 for the serial algorithm.

Conclusion

OpenMP and MPI can both be used to dramatically speed up a serial brute-force password cracker over what is possible in single-threaded algorithms. This is unfortunately not efficient in comparison with a serial algorithm, as shown by the above data. However, this should be put in context – where manpower is a greater cost to an organisation than CPU time, it will likely be that the balance of costs falls in favour of using parallel processing across many cores, to save workers' time and reduce time wasted waiting for outputs. Also, in applications where response time is a significant factor, such as Cybersecurity or Weather Forecasting, the benefit of fast operations will outweigh the clock-for-clock inefficiency of parallel computation, especially where an algorithm can be made to perform at or near optimal efficiency.

As well as the above, the implementation used in this project was scaled down hugely from what we would see in the real world, due to the time and resource constraints of the assignment. Rather than a problem set of 380,204,032 iterations, we would have been working with a 16-character key and an acceptable alphabet of 94, giving us a search space of **3.715742908341009e+31**. As we know from the **Amdahl effect,** we would get more speedup from a larger problem size, and I feel a useful further project would be to investigate this using a larger cluster and Cuda parallelism using a GPU system.