Development of a Parallel Algorithm to Aid Partial Key Recovery Techniques

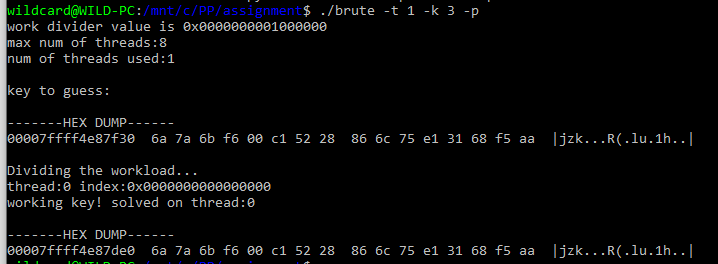
# Introduction

In this report we will cover the development of a parallel algorithm for recovering full AES 128 bit keys from partially recovered crypto materials. This algorithm will be implemented both for openMP and openMPI and an evaluation of its performance with regards to various laws and rules will be performed. A discussion regarding the design of this algorithm and the justification of its implementation will be carried out which will also explore ways in which it could be improved in the future. The structure of this report will follow in three parts, the flow of progress in developing the algorithm, the design discussion of the parallelization of the algorithm, and the performance evaluation of the algorithm through benchmarking methods. All findings will be compiled into a reflective conclusion on the project in which aspects of its development will be accessed in how they could have been done differently in order to reach other outcomes.

# Project Progression

**Serial Algorithm**

The course of developing both solutions for openMP and openMPI started with getting a serial version of the program working. Without much context on how we should solve brute forcing AES crypto materials, such as a 128bit key, the serial solution was started with the goal of cracking one or two bytes of a partially known 128bit key used in an aes-128-cbc decryption process. Reasoning behind this methodology was based on experience with Hardware security attacks such as Differential Power Analysis (DPA) or Correlation Power Analysis (CPA), where partial keys can be recovered. **(Cs.ru.nl, 2019)** Making a brute force attack a practical application to find these remaining bytes. These attacks are most often complicated with issues such as electrical noise that does not share a relationship with the cryptography operation, effectively poisoning the data set used in recovering an unknown key. **(Riscure.com, 2019)** More specifically the use of T-Tables for optimization of Crypto engines instead of traditional S-Box in embedded devices (ARM), can obscure the electrical leakage of certain bytes in the key, usually yielding a partially recovered key with 4 unconfirmed bytes. **(Eprint.iacr.org, 2019)**



Figure

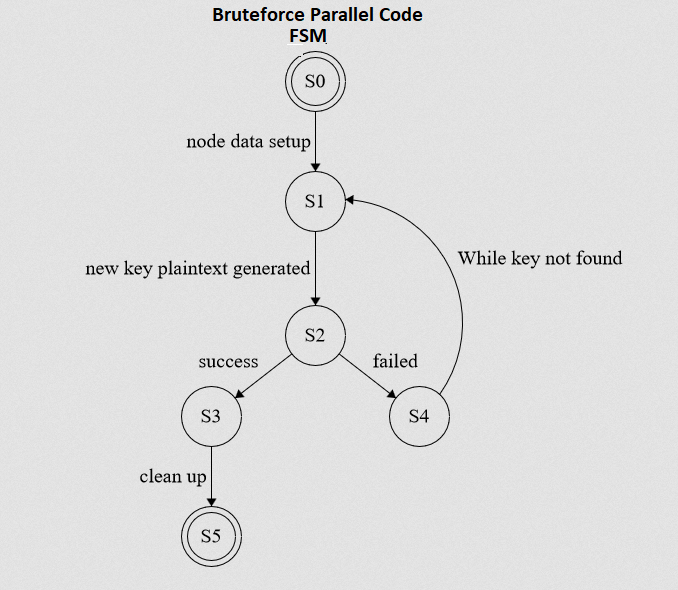
"Serial Algorithm solving the remaining unknown 3 bytes of a partially recovered key”

**Algorithm Parallelization**

Now that a serial solution was implemented and tested, work began on selecting which part of the program could be parallelized. It was determined at this time to parallelize the loop in which each new key byte would be incremented. Additionally, each thread would also need some setup code in order to correctly initialize itself for dealing with the problem. At this point no effort was made to divide the search space for each thread. Instead the goal then was to determine which resources were to be shared and which were to be private. Starting with the shared resources, we can assume most of the crypto materials will be good candidates. The original partial key would need to be shared since we will only be reading it in and making an internal copy of it in order to manipulate it. The cipher text and plaintext are can also shared since each thread will only need to read from them. We can keep an internal copy for each thread of its own buffer when decrypting the cipher text out to plaintext.

Ideally to prevent corruption issues, any resource that is being written to should be private. At this point it was theorized that each thread would have access to its own index within processing the search space so it was best to make this variable private. Additionally, during testing, the loop part of the code was changed from using statically set for loops, which hard coded the search space, to that of a while loop. Using a while loop allowed the program to dynamically set the search space of the loop via command line arguments. The only trick to keeping the loop running would be a control state, which would need to be shared in order for each thread to know when the correct key had been located.

During development a finite state machine was created to illustrate the control flow during the parallel portion of the code. It consists of a decryption/comparison process mainly.



Figure

“Finite State Machine for the parallelized loop of the algorithm”

**States**

S0: initialize nodes data (copy index etc.)

S1: decrypt cipher text with new key

S2: compare plaintext output with original plaintext(or check entropy)

S3: print the found key!

S4: increment key in search space

S5: clean up resources and exit program

**Resource Parallelization**

With a parallelized brute force program working, the focus soon became on how the search space for the missing key bytes could be divided in order to actually benefit from the parallelization. For each thread or process created we would have an index. This index is will be looped on to iterate each key guess. So for a search space of 256^4 we will move from 0 to the maximum value and try each and every combination until the decryption call produces the expected plaintext. If we have several threads or processes doing this, then it is pointless for them to try the same keysets. The index was then targeted to be divided by the number of threads/processes made available and would happen before we would enter any parallelization code. To perform this, it was thought to use a large integer such as a 64bit number, this number allowed the search space to be a maximum of 256^8 since it is an unsigned int. It would be incredibly unfeasible to try to crack an AES key if over 8 of the bytes weren’t known so this would not limit us in any way.

To divide the workload for each parallel loop, we would take the maximum value for a byte and shift it in units of 8bits by the number of missing key bytes. If say 3 bytes were missing then the index would be 0x1000000. This number could then be divided by how many threads/processes we would allocate, creating a divided unit that would be the search space range of each loop. For generating the key each byte in the index could be masked in relation to the position of the byte in the key being set. This way every possible combination of missing key bytes would be attempted.

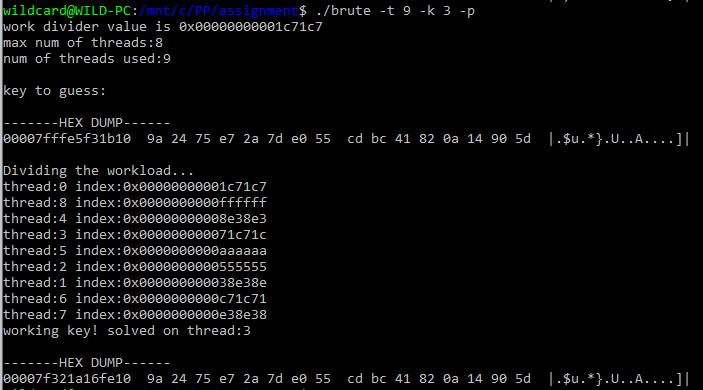


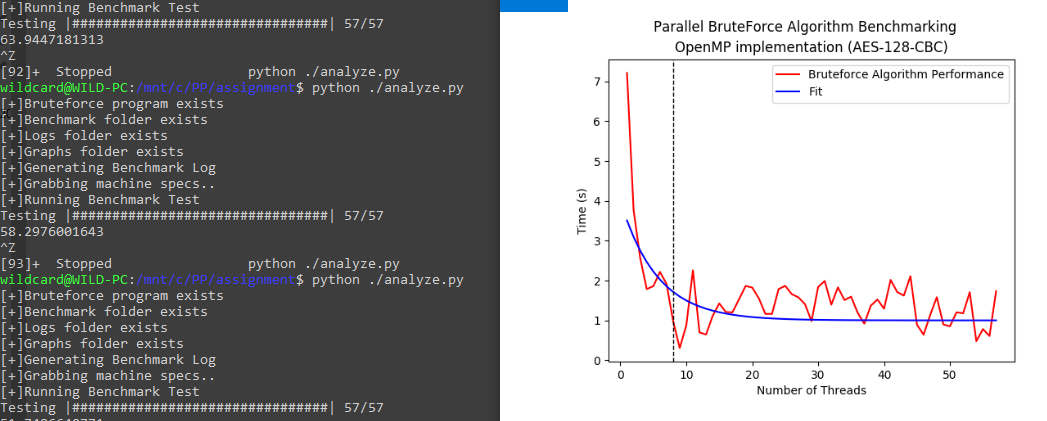
Figure 3

"openmp solution now running in parallel with a divided workload"

**Benchmarking Environment**

Now a new issue presented itself, how might we be able to evaluate this algorithms performance? Since the program had been already built to dynamically self-adjust based on the resources to be allocated. We could make use of this functionality by extending the program to handle command line arguments to change the behaviour of the algorithm. It was at this point that a python script was written to run our algorithm using various inputs.

The scripts goal was to produce timing related information to gauge the performance of the program. It could incrementally call the program with more threads/processes and then measure the timing of how quickly it finished or found the key. However there was a problem in how the key was set. Depending on where the missing key bytes fell in the search space, the algorithm performance would vary. To gauge its performance regardless of the target data, a performance evaluation mode was added to the program. In this mode the key would be randomly set and we could run the program many times to get an average of its general performance according to the resources available.

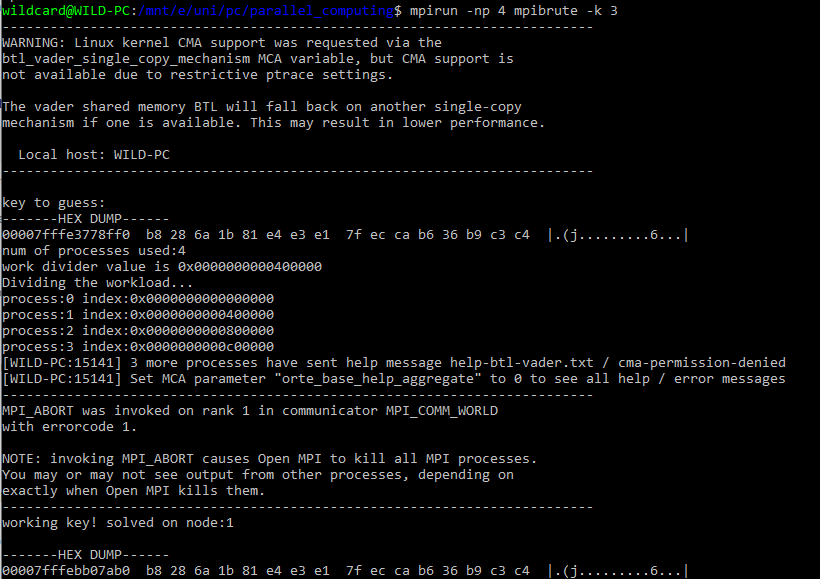


Figure

"Algorithm Benchmarking with the help of python and matplotlib"

**OpenMPI Modifications**

Finally, the algorithm was now in a state that it could be benchmarked for as an openMP solution. However, the project still needed an openMPI solution as well. Luckily to implement this, there was no need to change the algorithm, but to simply change how each process would be setup for parallelization. During developing this solution various methods of control using the openMPI API were examined. Methods like broadcasting messages to all processes, or the use of barriers was tested. Eventually the solution settled on using the 1st process as the one that would setup and distribute any variables to the other processes. All other processes would wait until they received this information and then would begin to look for the key within their respected search ranges. Each process would determine their search range based on their world id and the search space divider value. Instead of synchronizing all processes like we had done with the openMP solution via a control variable. We can simply print the key when found and abort the process. This will cause the main process to receive this signal and then kill all running processes. Doing so this way should reduce any latency experienced across the network since each world does not need to rely on the communication from another for the length of the search operation.

****

Figure

"openMPI solution running locally"

# Parallel Design

In developing our parallel algorithm in order to recover a full AES key from partially recovered key material, there was an examination of various parallelization models. In order to assess which parallelisation method was the most efficient and scalable for our problem set.

It was soon determined that our algorithm should focus on data parallelization rather than task parallelization, since if we used task parallelization then we would experience a significant delay and sync issues. The specifics of our problem rely on locating the correct answer within a large data space, the process in which to do so isn’t very task heavy, but data intensive in the sense it has to pivot through lots of data via a simple task structure. If the algorithm was made to be task parallelized as well, then there would significant challenges introduced regarding synchronization of processes/threads handling each task set to complete one data set. **(Cs.umd.edu, 2019)**

Evaluating our algorithms design now against what is known as Flynn’s taxonomy, which is a classification of parallel computer architectures. It can be described with the use of different CPU instruction execution methods and different data processing methods. Each method comes in two forms, either singular or multiple making up 4 possible architectures. SISD (Single Instruction Single Data), SIMD (Single Instruction Multiple Data), MISD (Multiple Instructions Single Data), and MIMD (Multiple Instructions Multiple Data). SISD can be thought of as a single core processor with one data stream. SIMD can be described as vector processor, where multiple streams of data are processed in one instruction. MISD is equivalent to each core of a process processing a single stream of data. And MIMD is usually defined as a multi core processor processing multiple streams of data. **(www2.cs.uh.edu, 2019)**

With our problem, we have a large set of data, but only a few tasks to complete on it. As explained previously, there won’t be a good trade off if we make our algorithm task parallelized, so that can eliminate any design relating to MISD or SIMD. MISD isn’t valid because of the poor trade-off for performance, and SIMD isn’t valid due to the operation of encrypting the data needs to be checked for every data set, unless changes were made to completely customize the AES library to implement this functionality, it isn’t practical for our algorithm. So that leaves us with 2 taxonomies, SISD and MIMD, which are thought of as a single core processor and a multi core process with a divided workload. SISD is the simplest since it is the serial form of the algorithm used prevalently. Whereas MIMD is if we take that same algorithm and make sure that each execution of its instructions has access to a unique data set. In MIMD there is also what is defined as shared or distributed memory, for our algorithm it wouldn’t make sense to use a shared memory space since on process does not rely on the data of another process during runtime. So our algorithm can be seen to be data parallelized MIMD-distributed memory design, on a singular machine or spread across many machines. **(www2.cs.uh.edu, 2019)**

# Algorithm Benchmarking and Analysis

At this point in the project, we had a usable version of both openMP and openMPI solutions. We could pass in arguments to now tailor their behaviour or control the availability of their resources. Using the benchmarking script developed to gauge performance, we can now plot the timing related information regarding how well the openMP solution handled cracking the partial key vs the openMPI version. A bit more information on the benchmarking procedure, in the script for each thread/process allocated, times were collected in and then averaged. Around a 100 or so runs were performed in order to stabilize the results. These average completion times are what make up the performance graphs, and are recorded for each incrementing of number of threads/processes made available.

The results from these tests proved very enlightening, and during testing moves to improve the performance were evaluated, but not much could be done without redesigning the crypto library for the algorithm. The parallel algorithm running is mostly in 2 parts, generate the key and check the output of decryption. However the decryption function call to the AES library functions produces the most significant delay in the algorithm. It requires several long chunks of assembly needed to re-initialize crypto materials and perform decryption on the data. These two chunks of assembly, for both CBC decrypt and set key would need to be manually implemented for the parallel algorithm to reach significant improvements.

Without modifying much, we can take a look at how well the openMP solution performed natively on a quad core CPU with 8 threads made available. All algorithm benchmarking was performed on a windows 10 machine through the Linux subsystem. A detailing of the specs of the machine were recorded every time the benchmarking script was run, so that if needed the data could be contextualised to the machine it ran on.

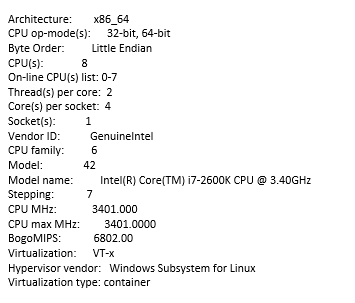


Figure 6

"Machine specs exported every time benchmarking tests were performed,

i.e. host machine specs running the openMP solution"

Graphing the benchmarking results was split into two formats, the first shows the overall performance of the algorithm based on how quickly it could find the key, for each thread/process allocated. And the second will reflect the speed up experienced due to parallelization both theoretical and practical according to both Amdahl’s and Gustafson’s laws. For the 1st section, both graphs for openMP and openMPI are split to show a range just over the max amount of available resources, as well as another showing if allocating is extended beyond the limits of the hardware. This should illustrate how the algorithm might suffer when dealing with issues such as latency.

**OpenMP solution**

Starting with openMP, we can see a significant drop in time of completion on the initial parallelisation. For each thread allocated, the increase is exponential until just before the max thread limit for the machine is hit, this is denoted by dotted line. This isn’t out of the ordinary at all since one thread or core is needed to maintain the operating system and run the scheduler. The data does still vary slightly due to the randomization of the bytes being cracked for the partial key aligning more closely to the search space divider value. To mitigate this in expressing performance, a best fit line has been calculated in blue.

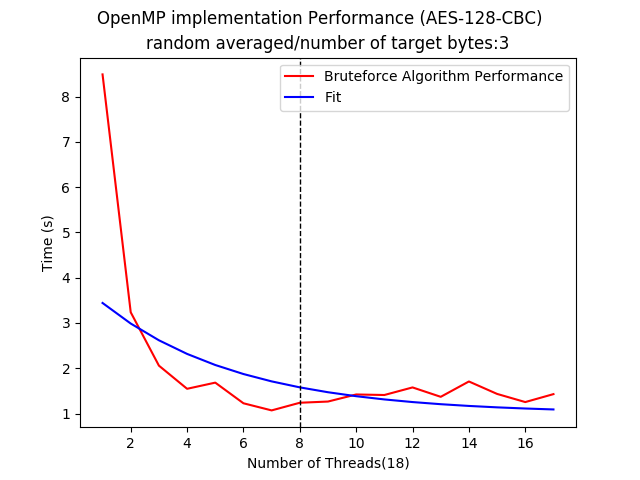


Figure 7

"openMP Benchmarking Completion times(short run)"

Again for openMP the solution was run for a bit longer with more threads allocated, far more than the hardware can provide. But doing so shows how the algorithms performance never decreases, it only stabilizes along. Initially it was expected to negatively impact the timings significantly soon after the max number of threads, but it was then noticed that just not enough threads were being set. Allocating more than 50 or so got to the point that would freeze the machine completely. For the period of 8 threads and up till the point of freezing, the operating system was able to allocate memory for all the threads, but it would continually swap in and out various threads execution in the scheduler. This allowed the answer to continuously be found regardless of not having enough resources to allow the algorithm to scale, instead it would maintain its performance until there was no more memory to allocate.

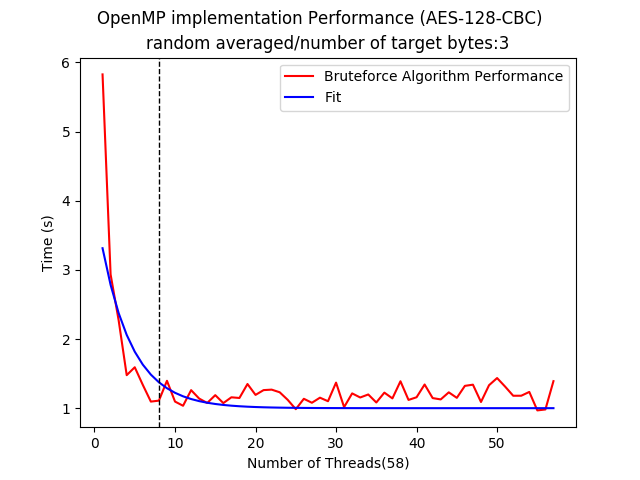


Figure 8

"openMP Benchmarking Completion times(long run)"

The next goal in benchmarking was to evaluate more specifically the speed up experienced from parallelizing the algorithm. We can measure the speedup of our parallelization by using Amdahl’s law. **(Web.engr.oregonstate.edu, 2019)**

To do so we need to estimate the portion of our program that is capable of being parallelized. To do this we can take the time of completion of the serialized run of the algorithm, and subtract it with the time taken when the algorithm is split into 2 parallel runs. The difference should show an estimation of how much of the program is effected by parallelization, thus giving use key information needed to calculate how our algorithm will be sped up by parallelising it to the nth degree. In the case of our openMP solution it was around 78% of the program was capable of being run in parallel, or in other words 78% of the program performance was improved using parallelization. We should then expect that for each new time of completion for each new thread allocated, that performance will increase around 78% of the previous runs performance. This increase should then taper off when the resources are then exhausted such as the total thread count-1 is allocated to the algorithm.

The speed up under Amdahl’s law was calculated using the following algorithm where T represents the time taken for the single thread run of the program to complete. B was calculated as a measurement of the proportion of the program effected through parallelization, this was the 78% that we determined. Iterating on each new time of completion gives us the correct speed up values for n times parallelized. **(Puget Systems, 2019)**



Figure 9

"Amdahl's Law calculation formula"

The results of plotting Amdahl’s law against our openMP performance results sheds light on how time of completion in regards to the expected scalability will taper off based on the impact of resource limitations. Using Gustafson’s law however can show how that scalability would impact the performance if these limitations were eliminated. This can gives us a nice trend to compare our performance against.

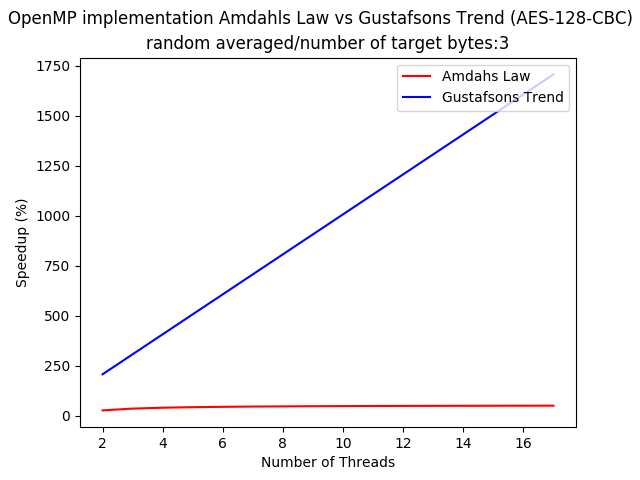


Figure 10

"openMP Benchmarking - Amdahl's Law vs Gustafson's Law(short run)"

In the previous graph, comparing the Amdahl’s Law performance plot to that of the trend it should be following, yielded a graph without any detail on the representation of our experienced speed up. A single plot of this speed up can be seen for the short run (around 20 threads). It can be seen that up to 3 threads our performance follows more closely to the trend however starts to taper off around a 40% speedup when resource restrictions begin to effect the algorithms performance. Interestingly enough the algorithm speed up still climbs, albeit slowly, towards around 50% where it maxes out.

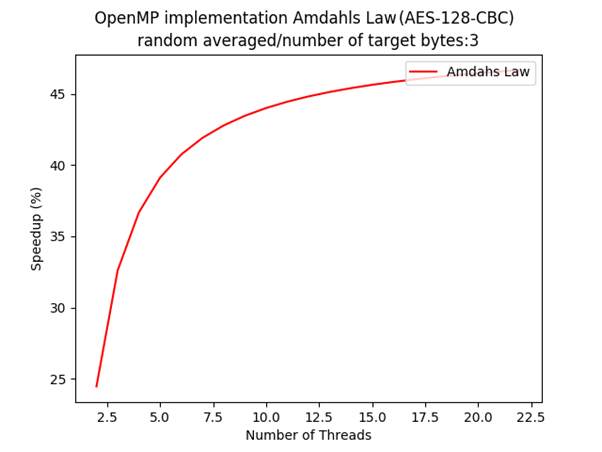


Figure 11

"openMP Benchmarking - Amdahl's Law(short run)"

In our expanded run up to around 50 threads we can see it reach this 50% speed up margin. If the hardware is expanded, we can assume that our speed up curve may look narrower and align to the trend with some skewing due to the inevitable latency.

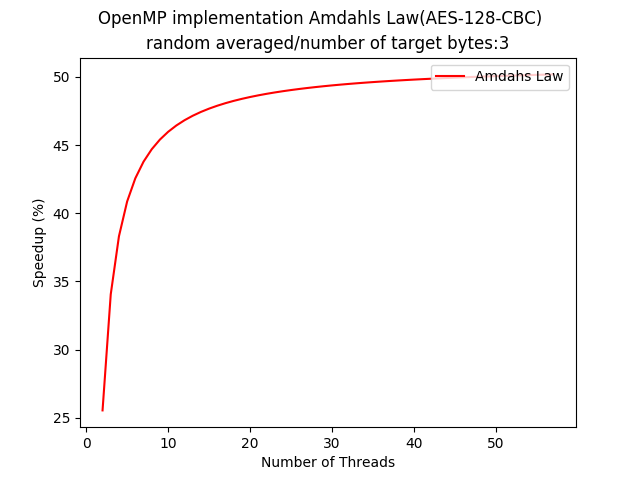
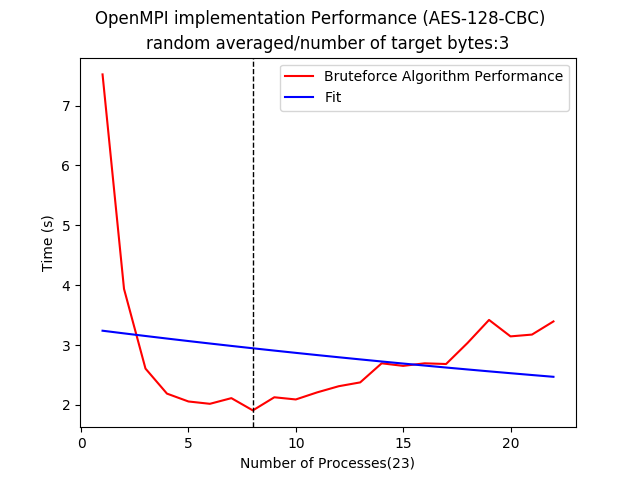


Figure 12

"openMP Benchmarking - Amdahl's Law(long run)"

**OpenMPI solution**

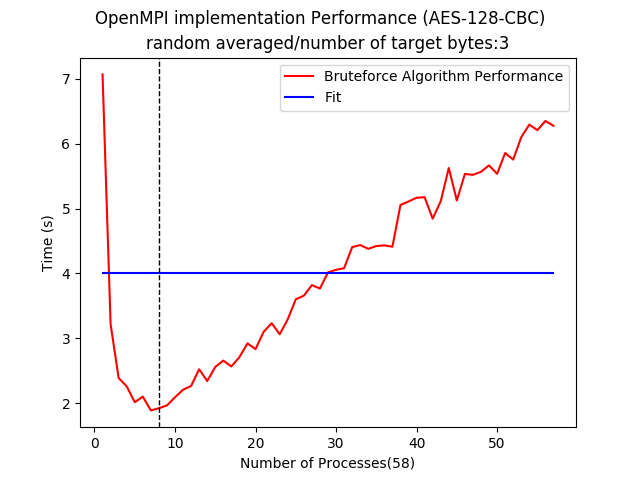
Now with the openMP solution tested and analysed, efforts began to benchmark and evaluate the openMPI solution in hopes that it would give us some insight into how the two schemes may differ. Starting as we did with openMP we will do a benchmarking of the algorithm for a short run. A note on the platform, due to issues accessing the provided cluster computer, all openMPI benchmarking results will reflect that of the same platform that openMP was tested on. Since the platform is quite restricted, we wont see how both algorithms will scale when distributed across a large number of resources, however we can determine factors that we can use to make an educated guess on how that might look. With the solution running in the same fashion as before, we notice a difference right away. The openMPI solution or rather the infrastructure that this solution runs on has greater latency issues to overcome. With openMP we saw no reduction in performance after the resources were stressed, and with openMPI it seems that any exceeding of these resources produces significant performance issues. Reading a performance comparison of openMP vs openMPI enlightened to the fact that openMPI had no shared memory in its messaging system, it is intended to run on a network, so using it locally produced a larger overhead and delay when accessing the local machines network resources. **(hindawi.com, 2019)** This overhead was enough to create a significant delay that could not be mitigated by parallelisation of the algorithm.

****

Figure

"openMPI Benchmarking Completion times(short run)"

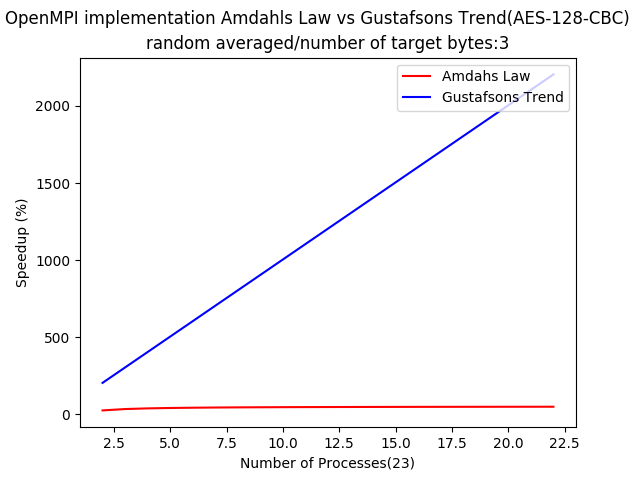
This overhead can be seen upon running it for longer, where performance is lost at a constant at around 5-10%. Interestingly enough our best fit line which is based on the change in regards to each point in comparison with the rest of the data ends up being completely fixed. The algorithm due to the experienced overhead works out to around a 4 second average for completion for any process in the 50 allocation range.

****

Figure

"openMPI Benchmarking Completion times(long run)"

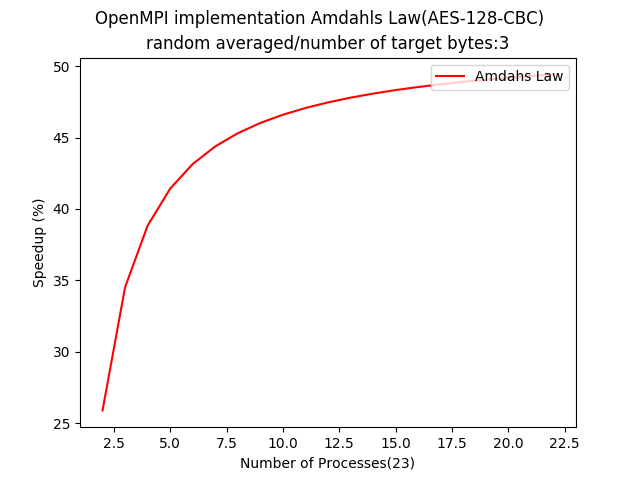
Following through with Amdahl’s Law and Gustafson’s Trend we can see where our algorithm should have taken off, now if we compare that trend to that of openMPs we get almost identical speedups.



Figure

"openMPI Benchmarking - Amdahl's Law vs Gustafson's Law(short run)"

Here is a look at the results of just plotting via Amdahl’s Law calculation, we can see by comparing back to openMP that in openMPI we might get a bump in the speed up if we are truly free from our resource restriction.



Figure

"openMPI Benchmarking - Amdahl's Law(short run)"

Gathering another plotting for running it longer we can see how our speed up curve is nearly identical to the openMP implementation. However the curve starts to form at a lower speedup percentage this can be explained by the additional overhead aswell as a slight difference in the proportion of parallel to serial code.

# C:\PP\assignment\benchmark\graphs\20191129-195910-bruteforce_performance_openmpi.png

Figure

" openMPI Benchmarking - Amdahl's Law(long run)""

# Conclusions

With the project concluded and the results compiled, where are we left? Well without testing the openMPI solution on the cluster computer and instead on the local machine, we still have enough to make some conclusions on the scalability and performance of both algorithms. It can be said that our openMP solution is far better for a local setup as you would expect, openMP is designed to make use of the hardware its running on rather than across a network. However scaling that solution up has its restrictions since local hardware can only contain so many cores/threads. The openMPI solution even though it didn’t perform as well as openMP for the local setup, it showed results that suggest its gain over openMPs implementation. The openMPI solution will actually be able to be scaled without issue on a network which would if cracking for 4 bytes instead of 3 could dwarf the performance of the openMP solution. Both algorithms do have their trade-offs, if the problem required more communication and task parallelization then openMP would be a better solution since it deals with shared memory and machine code parallelisation better. But in this case openMPI is the better option since it can do data parallelisation better due to the scalability of a cluster computer network.

# References

Cs.ru.nl. (2019). [online] Available at: https://www.cs.ru.nl/bachelors-theses/2019/Maaike\_van\_Leuken\_\_\_4641493\_\_\_Comparing\_Correlation\_Coefficient\_and\_Difference\_of\_Means\_in\_a\_Differential\_Power\_Analysis\_Attack.pdf [Accessed 27 Nov. 2019].

Cs.umd.edu. (2019). [online] Available at: http://www.cs.umd.edu/class/fall2013/cmsc433/lectures/concurrency-basics.pdf [Accessed 27 Nov. 2019].

Eprint.iacr.org. (2019). [online] Available at: https://eprint.iacr.org/2016/714.pdf [Accessed 27 Nov. 2019].

Puget Systems. (2019). *Estimating CPU Performance using Amdahls Law*. [online] Available at: https://www.pugetsystems.com/labs/articles/Estimating-CPU-Performance-using-Amdahls-Law-619/ [Accessed 27 Nov. 2019].

Riscure.com. (2019). [online] Available at: https://www.riscure.com/uploads/2017/09/eu-15-sanfelix-mune-dehaas-unboxing-the-white-box-wp\_v1.1.pdf [Accessed 27 Nov. 2019].

Web.engr.oregonstate.edu. (2019). [online] Available at: http://web.engr.oregonstate.edu/~mjb/cs575/Handouts/speedups.and.amdahls.law.2pp.pdf [Accessed 27 Nov. 2019].

Www2.cs.uh.edu. (2019). [online] Available at: http://www2.cs.uh.edu/~gabriel/courses/cosc6374\_s08/ParCo\_02\_Arch.pdf [Accessed 27 Nov. 2019].

Downloads.hindawi.com. (2019). [online] Available at: http://downloads.hindawi.com/journals/am/2015/575687.pdf [Accessed 29 Nov. 2019].