**Parallel Computing Logbook**

**Introduction**

This logbook is the companion documentation for the UFCFFL-15-M UWE Parallel Computing module. Within this document are details surrounding the development, testing, and concluding points for the AES brute force algorithm developed with OpenMP and MPI. The aim is to increase the speed at which the brute force algorithm can achieve its target, in order to verify this, we have been pre- supplied with the necessary salted ciphertext encoded in Base64, as well as its plaintext equivalent. To perform this exercise, we have been provided with a serial version of the AES brute force algorithm and I will attempt to parallelize it.

**Table of Contents**

Introduction...................................................................................................... Gitlab Links........................................................................................................ Logs of Progress................................................................................................ Session One - 1st May 2021.............................................................................. Session Two – 3rd May 2021............................................................................. Session Three – 4th May 2021........................................................................... Session Four – 7th May 2021............................................................................. Session Five – 8th May 2021.............................................................................. Method of Parallelism....................................................................................... Performance Analysis........................................................................................ Testing Tables.................................................................................................... Serial Testing...................................................................................................... OpenMP Testing Table....................................................................................... OpenMP 2 Threads Reliability Testing............................................................... OpenMP 2 Threads Performance Metrics........................................................ OpenMPI Testing Table...................................................................................... OpenMP 2 Process Reliability Testing.............................................................. OpenMP 2 Process Performance Metrics......................................................... Performance Metrics Explanation:.............................................................. Comparative Analysis......................................................................................... Amdahls Law Vs Gustafsons Law...................................................................... Testing Environment…………….......................................................................... Conclusions....................................................................................................... References.........................................................................................................

**Logs of Progress  
Session One - 1st May 2021:**

The aim of this session was to confirm the brute force algorithm was working by having it crack the target password serially. After confirmation that the program runs serially, proceed to modify it using OpenMP and OpenMPI. Any new errors that occur after modification would only be as a result of the latest additions.

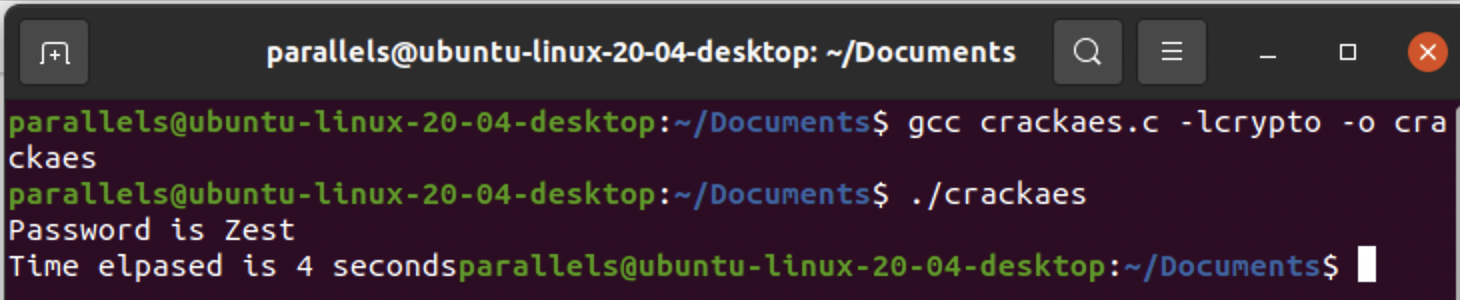
My understanding of the brute force algorithm is that it checks up to five levels of permutations to find a key which has five characters in length including “0-9,

a-z, A-Z”.

Now I go ahead and test the serial code in order to get time difference data for comparison after parallelization. This is due to the fact that I expect an improved runtime after parallelization. The algorithm cracks the first ciphertext perfectly serially with a password of four (4) character length in 4 seconds and the second ciphertext with a password of five (5) character length in 13 seconds.

Text

Description automatically generated



Since the search vector for this algorithm “Numbers – Upper Case – Lower Case” matched the ascending order that the ASCII characters would occur, I decided to try another ciphertext out of curiosity. One of the given ciphertexts in the program Mar10 does not match the search vector so I tried to crack it serially and it took considerably more time to crack

Text

Description automatically generated

**Session Two – 3rd May 2021**

I have now completed the OpenMP tutorial and I’m now trying to implement it on the brute force algorithm. I have started by including “omp.h” to the header and the “pragma omp parallel” construct above the for loop. I have decided to start by setting “omp\_num\_threads” to 2 and using the collapse construct, setting it at 5. When I try to run the program, it throws an error.

Text

Description automatically generated

I have just realized that “return 0” throws an error in OpenMP so I have removed it and attempted to run it again. Now it appears that the program keeps running without finding the password and without stopping. I am now trying to manually exit the load by hard coding an exit loop and initializing it to 0 outside the parallel construct so I can set it to 1 when I want to exit the loop. I have now decided to use the “default(none)” construct after the pragma and collapse construct in order for me to declare each variable in the parallel region as shared, private and first private. This lists out all the variables that need to be declared in the parallel construct. After doing this and running the program, I’m getting a “segmentation fault” error.Text

Description automatically generated

This is probably due to wrong declaration of variables. I’m now trying to move the variables around shared, firstprivate and private, putting the variables that all threads should share a copy of in “shared”, variables that should be private to each thread in “private” and variables that should be private but has been initialized outside the parallel region in “firstprivate”. In the process of doing this, I realized I had to declare “end” and “result” outside the parallel area. I’m still getting segmentation fault error.

**Session Three – 4th May 2021**

After some analysis, I have decided to restart the declaring of variables as shared, private and firstprivate and initialize the password to memory location of plainplassword in the parallel region. This has now worked, and the cipher is cracking now but at the same time as the serial algorithm.

Text

Description automatically generated

I have tried to change the number of threads and I have discovered that when I increased the number of threads, the time taken to crack the cipher increases as well so I will be using 2 threads as this gives the best runtime. Now in order to further improve the parallelization, I have decided to add a schedule that determines the amount of work a thread does. I have tried using static, dynamic and runtime schedule and I got a better result with runtime schedule.

Now the ciphertext cracks in significantly lesser time.

Text

Description automatically generated

After finishing with this ciphertext, I tried to use the same algorithm to crack the second algorithm which has a character length of 4. Due to the fact that this second cyphertext has four characters, I had to add some minor modifications to the algorithm to accommodate this change and it cracks perfectly at a faster runtime that the serial run.Text

Description automatically generated

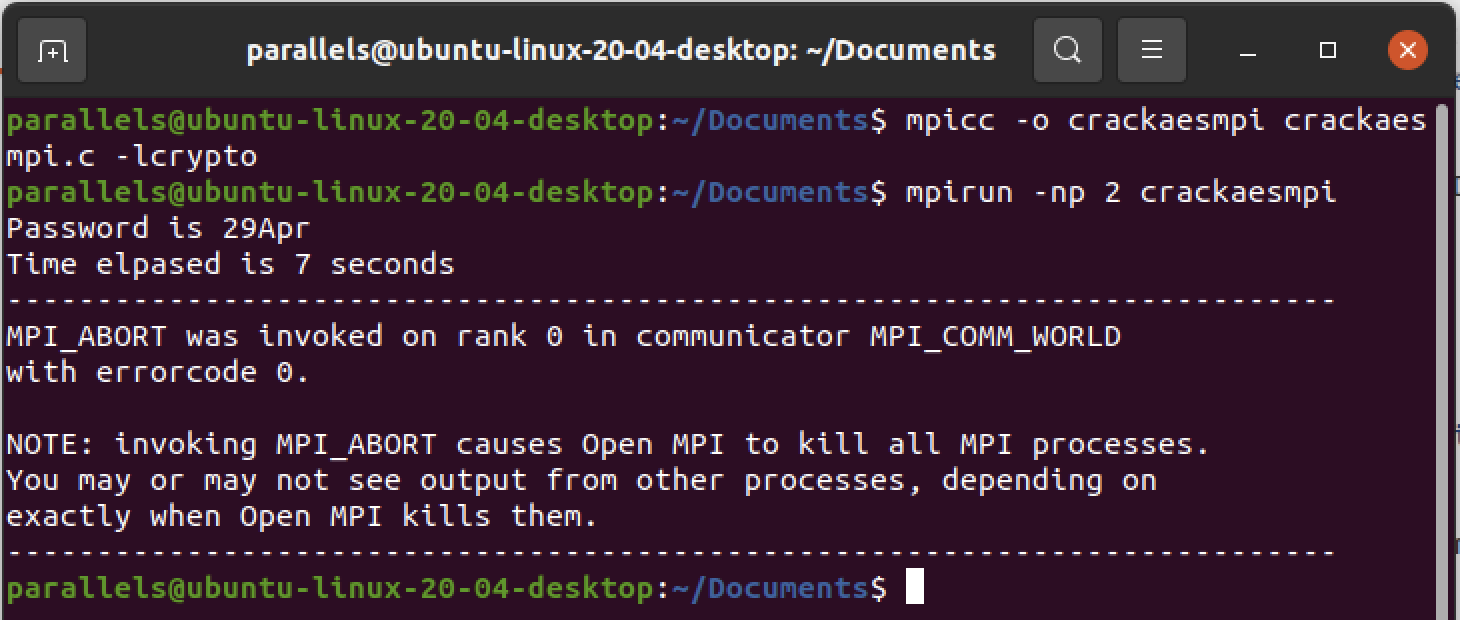
**Session Four - 7th May, 2021**

After successful parallelization of the brute force algorithm using OpenMP, I then proceed to parallelizing the serial code using OpenMPI. I have watched a few videos and read a number of articles on OpenMPI to better understand the language and how to properly implement it to achieve my goal. OpenMPI is quite different from OpenMP in that in OpenMPI,every thread can access only its own memory and to communicate between threads, OpenMPI commands need to be issued. I started by adding the MPI header file which contains MPI-specific definitions and function prototypes via an "include" statement and declaring the MPI variables in the main function. Then I went ahead to initialize the MPI process using the initialization routine after which I called the MPI communication routines. MPI routines are implemented as functions in C. In the communication routine, I used the communicator handle to represent a group of processors that can communicate with one another and rank of the processor in that communicator. I also used MPI\_COMM\_WORLD to ensure that every processor can communicate with every other processor. I then used size which determines the number of processors, of any communicator to which it belongs with a call to MPI\_COMM\_SIZE. By using this method, each process can check one section of the search data at the same time. One problem I discovered with this search pattern is that when one process finds the correct password the others would keep searching until they reach the end of their search data. To solve this, I used the broadcast function given by OpenMPI which allows you to message all processes on the communication channel. I’m also using the OpenMPI’s “MPI\_Irecv” function which allows early posting of receive for a future send. The request handle identifies the receive operation that was posted and can be used to check the status of the posted receive or to wait for its completion. None of the arguments passed to “MPI\_Irecv” would be read or written until the receive operation it invokes is completed. Thereafter, I used “MPI\_test” with the request signature to find out if the posted receive has been completed. Now I will put the finalize routine to end the MPI process. After compiling and running, the cyphertext cracked successfully but the process did not end.

Text

Description automatically generated

**Session Five - 8th May, 2021**

Currently, my brute force algorithm has been parallelized my OpenMPI and the cyphertext cracks in a faster time than the serial code, but the process does not terminate. After some further research, I have decided to add “MPI\_Abort” to kill all related processes. This is mostly considered a “hard” way to terminate the MPI processes, but it is a very effective way to do so, it works almost immediately and halts the MPI processes. The receives are all posted using wild variables, also known as “wildcards” which allows the function to receive any messages sent on the MPI\_COMM\_WORLD. The function locates the correct password but doesn’t terminate, that’s why the program must be terminated manually. After running the code again, I realized that the process was still not terminating, and it was related to the way I was displaying the time. So I decided to change the way I display the time from “time\_t” to “MPI\_Wtime”. After doing this the process terminates after cracking the code. 

Now that the brute force algorithm has been parallelized using OpenMPI, I tried to change the number of processors from 2 to 4 and this was throwing errors. I then tried several other numbers and discovered that my code only runs with 2 processors.

Text

Description automatically generated

**METHOD OF PARALLELISM**

There are two major components of parallel algorithm design. The first one is the identification and specification of the overall problem as a set of tasks that can be performed concurrently. The second is the mapping of these tasks onto different processors so that the overall communication overhead is minimized. The first component specifies concurrency, and the second one specifies data locality. The performance of an algorithm on a parallel architecture depends on both. The performance of a parallel algorithm depends not only on input size but also on the architecture of the parallel computer, the number of processors. In theory, the brute force algorithm operates on a Single Instruction Multiple Data (SIMD) architecture according to Flynn’s taxonomy (1966). An SIMD system is a multiprocessor machine capable of executing the same instruction on all the CPUs but operating on different data streams which come from different data flows, meaning that they execute programs in a lockstep mode, in which each processing element has its own data stream. This is what we were trying to do when we parallelized the serial brute force algorithm. In order to parallelize the serial code, we give different data streams of the same instruction to threads in OpenMP and processes in OpenMPI.

The given brute force algorithm operates sequentially, starting with the variable initialization then moving on to the base 64 decode and Salt removal. These processes cannot be parallelized due to their linear operation, and even if they could be it would be insignificant as that they only run once per execution. Then the program moves to the password string generation and AES cracking attempt. This portion also works in a linear fashion, generating the password one character increment at a time and passing it to the decryption algorithm. Thereafter the decryption is checked if it returns a success value and the returned string is compared against the target plaintext. This is the portion of the algorithm that can benefit from parallelisation in a SIMD manner. Because the password generation, decryption, and success checking must occur in a sequential order, it is essentially a single instruction set. The password generation is where we have the multiple data and by having multiple threads begin at scheduled points in the search data you can create a parallel region where each thread is generating a unique password string to the other threads, processing it, and checking if its correct. Parallelizing these portions of the algorithm results in Data Parallelism because each thread is operating on a different variant of a password, or at a different point in the search data. In theory this should increase the speed of the brute force algorithm because it’s primary time constraint is that it can only generate and check one password attempt at a time when run serially. In the implementation of the Data Parallelization, each thread processes a unique password once in parallel with the other threads. In addition, the machine we will be running the parallel implementations on is a single processor machine with multiple cores, and therefore our implementation falls under the concept of Multicore Computing.

**Performance Analysis**

**Testing Tables**  In the tables below, the lesser time is the best-case scenario, and the bigger time is the worst-case scenario. These parameters would be used in the calculations below. In this section, five metrics that are commonly used to measure performance are introduced. They are execution time, total overhead, cost, efficiency, and Speedup. An obvious performance parameter is a parallel program's execution time, or what is commonly referred to as the wall-clock time. The execution time is defined as the time elapsed from when the first processor starts executing a problem to when the last processor completes execution.

|  |  |  |
| --- | --- | --- |
| RUN | CIPHERTEXT 1 | CIPHERTEXT 2 |
| 1 | 3 seconds | 13 seconds |
| 2 | 4 seconds | 13 seconds |
| 3 | 3 seconds | 13 seconds |
| 4 | 3 seconds | 14 seconds |
| 5 | 3 seconds | 13 seconds |

**Serial Testing**

|  |  |  |
| --- | --- | --- |
| THREADS | CIPHERTEXT 1 | CIPHERTEXT 2 |
| 1 | 3 seconds | 13 seconds |
| 2 | 2 seconds | 7 seconds |
| 3 | 2 seconds | 7 seconds |
| 4 | 2 seconds | 7 seconds |
| 5 | 2 seconds | 7 seconds |

**OpenMP Testing Table**

|  |  |  |
| --- | --- | --- |
| RUN | CIPHERTEXT 1 | CIPHERTEXT 2 |
| 1 | 2 seconds | 7 seconds |
| 2 | 1 seconds | 7 seconds |
| 3 | 2 seconds | 6 seconds |
| 4 | 2 seconds | 7 seconds |
| 5 | 2 seconds | 7 seconds |

**OpenMP 2 Threads Reliability Testing**

I ran the code 5 times to test the reliability.

**OpenMP 2 Threads Performance Metrics**

|  |  |  |  |
| --- | --- | --- | --- |
| **METRIC** | **CASE** | **RESULT**  **CIPHERTEXT 1** | **RESULT**  **CIPHERTEXT 2** |
| Total Overhead | Best | -1 | -1 |
| Total Overhead | Worst | 0 | 0 |
| Speedup | Best | 3 | 2.167 |
| Speedup | Worst | 2 | 2 |
| Efficiency | Best | 1.5 | 1.0835 |
| Efficiency | Worst | 1 | 1 |
| Cost | Best | 2 | 12 |
| Cost | Worst | 4 | 14 |

**OpenMPI Testing Table**

|  |  |  |
| --- | --- | --- |
| NUM OF PROCESSORS | CIPHERTEXT 1 | CIPHERTEXT 2 |
| 1 | 3 seconds | 13 seconds |
| 2 | 1 seconds | 7 seconds |

**OpenMPI 2 Processors Reliability Testing**

|  |  |  |
| --- | --- | --- |
| RUN | CIPHERTEXT 1 | CIPHERTEXT 2 |
| 1 | 1 seconds | 7 seconds |
| 2 | 1 seconds | 7 seconds |
| 3 | 1 seconds | 7 seconds |
| 4 | 1 seconds | 7 seconds |
| 5 | 1 seconds | 7 seconds |

I ran the code 5 times to test the reliability.

**OpenMPI 2 Processor Performance Metrics**

|  |  |  |  |
| --- | --- | --- | --- |
| **METRIC** | **CASE** | **RESULT**  **CIPHERTEXT 1** | **RESULT**  **CIPHERTEXT 2** |
| Total Overhead | Best | -1 | 0 |
| Total Overhead | Worst | -1 | 0 |
| Speedup | Best | 3 | 2 |
| Speedup | Worst | 3 | 2 |
| Efficiency | Best | 1.5 | 1 |
| Efficiency | Worst | 1.5 | 1 |
| Cost | Best | 2 | 14 |
| Cost | Worst | 2 | 14 |

**Performance Metrics Explanation**:

**Execution Time:**

Serial runtime of a program is the time elapsed between the beginning and the end of its execution on a sequential computer.

The parallel runtime is the time that elapses from the moment the first processor starts to the moment the last processor finishes execution.

**Total Overhead:** is the total time spend by all processors combined in non-useful work.

**Speed-up**: is defined as 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 processors

**Efficiency**: is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processors.

**Cost**: The cost of a parallel algorithm (or program) is Cost = Parallel running time × #processors

**Performance Metrics Calculation**

Total Overhead, To = PTp – Ts

Speedup, S = Ts/Tp

Execution Time, ET= Ts x Tp

Efficiency, E = S/P

Cost, C = Tp x P

Where P = number of processors;

Tp = Parallel Time;

Ts = Serial Time;

S = Speedup.

**Comparative Analysis**

**Serial:**

I ran the serial brute force algorithm on the first cipher and the worst run was 4 seconds while the best run was 3 seconds. Then I proceeded to run the second cipher and the worst run was 14 while the best run was 13.

**OpenMP:**

To parallelize the brute force algorithm using OpenMP I started by using the “Pragma omp parallel for” construct in front of the nested for loops. Then I decided to use the collapse construct which collapses nested for loops into one loop and I didn’t get the result I was after. Then I used the default(none) construct to make sure I declare all the variables in the parallel region accurately. I had to move the variables around shared, private and firstprivate multiple times until I got the arrangement properly in order for the program to run without errors. I then had to initialize the password variable in the parallel region in order for the password to be private to each thread. All private variables initialized outside the parallel region were put in firstprivate and all variables I wanted the threads to share a copy of were put in shared. Changing the set number of threads between 2 to 5 impacted the runtime of the program in that the more threads set, the slower it takes to crack the cipher. After achieving this, I still wasn’t satisfied so I decided to try the schedule construct to schedule chunks of the work to each thread in a specified manner. I tried static schedule with chunk of three (3) and it gave a better reason. Still unconvinced, I then decided to use runtime schedule so the sharing of work can be done by OpenMP in the runtime. Changing the set number of threads between 2 to 5 did not have any impact on runtime after using the schedule construct. After parallelizing the brute force algorithm using OpenMP, I ran the first cipher five (5) times using two (2) threads and the worst run was 2 seconds while the best run was 1 second. Then I proceeded to run the second cipher five (5) times and the worst run was 7 while the best run was 6. Increasing the number of threads between from 2, 3, 4 to 5 gives the same result.

**OpenMPI:**

I took a lot of time to figure out a way to start this as the general understanding of OpenMPI was a little bit difficult to grasp. After going through different video tutorials and articles on MPI, I finally found a strategy to tackle the problem. I used MPI variables like Non-blocking receive to acknowledge any future sent message, then I use “MPI\_Broadcast” to broadcast when a process finds the correct password to the rest of the processes. I had an issue where my program wasn’t terminating, and I used “MPI\_Abort” to resolve that issue. After parallelizing the brute force algorithm using OpenMPI, I ran the first cipher five (5) times using two (2) processors and it gave a stable runtime of 1 second. Then I proceeded to run the second cipher five (5) times and it also gave a stable runtime of 7 seconds.

**Analysis**:

After the tests, I discovered that the brute force algorithm cracks the cipher in parallel in about half the time it takes it to crack the cipher serially. I have also discovered from my testing that in this case, the OpenMPI parallelization is more efficient than the OpenMP parallelization as it gives a more stable result upon multiple trials. A fundamental difference between OpenMPI and OpenMPis that in the latter all threads can access the same memory (so called shared memory available to all threads), while in OpenMPIevery thread can access only its own memory.

**Amdahls Law Vs Gustafsons Law**

The above implementations follow Amdahl’s Law (1967) which states that “in a program with parallel processing, a relatively few instructions have to be performed in sequence will have a limiting factor on program speedup such that adding more processors may not make the program run faster.”

From the analysis of the brute force algorithm, it can be found that the Base64 decode, salt, IV and Key are not parallelized as it would be irrelevant. The AES decryption function as well is not parallelized because decryption is lengthy and sequential in a defined order. By taking Amdahl’s law into consideration, we can deduce that the speed of a parallelized brute force algorithm will never go beyond the time these critical sections take.

Gustafson’s Law (1988) states that increase of problem size for large machines can retain scalability with respect to the number of processors. This is a re-evaluation of Amdahl’s Law. What Gustafson’s law says is that the true parallel power of a large multiprocessor system is only achievable when a large parallel problem is applied which means that the greater the number of resources, the faster the solution of large problems.

In essence, Gustafson’s Law is based on a broader scope of processing to breach for the gap in Amdahl’s Law. This is certainly true for our Brute Force parallel program, not only would the program’s sequential critical sections greatly benefit from better hardware, but so would the parallel parts due to faster clock speeds.

**Testing Environment**

In line with the concepts explained above, notably Gustafson’s Law (1988), we have to agree that any change in hardware would for all intents and purposes affect our testing, moving between computers would cause a measurable change in performance due to possible changes in processor speeds and memory frequencies.

Acknowledging this, it can be deduced that testing the program must either take place on a single virtual machine with specifications that are not subject to change or on a single machine

For the sake of transparency, the full settings on my virtual machine are below, running a 20.04 ARM64 Ubuntu with Parallels Desktop.

Graphical user interface

Description automatically generated.

**CONCLUSION**

In conclusion, the implementation of the cracking of the given ciphers using brute force in OpenMP and OpenMPI was successful and the journey was an interesting one. I can acknowledge that using both OpenMP and OpenMPI the given cipher can be cracked in significantly less than time than when run in serial. OpenMP is used only for parallelization across multiple cores on the same node and OpenMPI is primarily used for parallelization across multiple nodes (although it works also on a single node). An important difference between OpenMPand OpenMPIis that in OpenMP all threads can access the same memory (so called shared memory available to all threads), while in OpenMPI, every thread can access only its own memory. In order for communication between threads to occur, OpenMPI command needs to be issued, which is typically very expensive.

On the hardware level, all OpenMPIcommands results in the network traffic between nodes, which is expensive (latency problem). In OpenMP all threads are running on the same motherboard, and hence access the same RAM.

After execution of the task, I discovered that OpenMPI is a more mature way of parallelizing a program as according to my tests and report, the results were more stable than OpenMP where the execution time fluctuates more.

Some major realization from this exercise is that parallel job is very hard to debug and some algorithms are easy to parallelize while some are impossible as seen with the likes of key, salt and IV. A good practice would be to test if more processors gives you better performance and you’d find that sometimes it even gives you worse!

.

**REFERENCE:**

Amdahl, G.M. (1967) Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. Afips Spring Joint Computer Conference [online]., pp. 1- 4. [Accessed 29th April 2021].

Flynn, M.J., (1966) Some Computer Organizations and their Effectiveness. Ieee Transactions on Computers [online]. c-21 (9), pp. 948-960. [Accessed 29th April 2021].

Gustafson, J.L (1988) Reevaluating Amdahls Law. Afips Communications of the ACM [online]. 31 (5), pp. 532-533. [Accessed 29th April 2021]

Rutgers University (2019) Computational Physics article “Parallel Programming” [online]. [Accessed 26th April 2021]

Chapman, B.M. and Massaioli, F. (2005) Openmp. Elsevier [online]. 31 (10), pp. 957- 959. [Accessed 14 November 2019].