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 goal is to increase the speed at which the brute force algorithm can achieve its goal, in order to test this we have been pre-supplied with the necessary salted ciphertext encoded in Base64, as well as its plaintext equivalent. Given these two, this project aligns with the crypto-analytical *Known Plaintext Attack.*

Please refer to the next section for links to all the implementations detailed in this document, the methods of execution are detailed within the readme section of the gitlab project.

Gitlab Links

Logbook:

https://gitlab.uwe.ac.uk/jj6-williams/pc_coursework/tree/master/documentation

Serial:

https://gitlab.uwe.ac.uk/jj6-williams/pc_coursework/blob/master/serial/main.c

OpenMP:

https://gitlab.uwe.ac.uk/jj6-williams/pc_coursework/blob/master/openmp/main.c

OpenMPI:

https://gitlab.uwe.ac.uk/jj6-williams/pc_coursework/blob/master/openmpi/main.c

readme:

https://gitlab.uwe.ac.uk/jj6-williams/pc_coursework/blob/master/README.md

Table of Contents

Illustration Index

Index of Tables

Logs of Progress

Session One - 31st October, 2019

The goal of this session was to create working brute force algorithm and to demonstrate it working by having it crack the target serially. Once known that the program works serially then reconfiguring it utilising OpenMP and OpenMPI would be simpler, as the development wouldn't involve testing both the algorithm and the parallelisation simultaneously. If it is certain that the algorithm works serially, then any issues that arise in the later stages would be solely due to the newer additions.

The brute force algorithm is a modified version of the provided algorithms, being extended from checking three levels of permutations to five, this stage is much easier than previously anticipated due to the knowledge that the key we are attempting to find will certainly be five characters in length including "a-z, A-Z, 0-9". This fact removes the need to dynamically resize the string we are generating and instead just create a string of size 5 with permutations of the aforementioned character sets.

Resultantly, instead of using the method within the provided algorithm, it was instead opted to use a hard coded character set and arrange the string using that instead of calculating the value of an ASCII character like it was previous, this created the basic brute forcing algorithm.

```
const char *alphabet = "abcdefghijklmnopqrstuvwxyz"
           "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
           "0123456789";

clock_t begin = clock();
// retrive the slater from ciphertext (binary)
if (strncmp((const char*)ciphertext,"Salted__",8) == 0) {
    memcpy(salt,&ciphertext[8],8);
    ciphertext += 16;
    cipher_len -= 16;
}
// generate key and iv
for(int i=0; i<62; i++)
    for(int j=0; j<62; j++)
        for(int k=0; k<62; k++)
            for(int l = 0; l < 62; l++)
                for(int m = 0; m < 62; m++){
                    *password = alphabet[i];
                    *(password+1) = alphabet[j];
                    *(password+2) = alphabet[k];
                    *(password+3) = alphabet[l];
                    *(password+4) = alphabet[m];

            //printf("%s\n", password);

            initAES(password, salt, key, iv);
            unsigned char* result = decrypt(ciphertext, cipher_len, key, iv);
            if (success == 1){
                printf("%s\n", result);
                exit(0);
            }
            //else {printf("unsuccessful!\n");}
        }
```

Illustration 1: Changed Code

This was then tested as previously mentioned, however it was not as simple as previously thought.
At first the brute force algorithm began segmentation faulting, usually this would be due to a data structure being overflowed or possibly an incorrect input being passed somewhere. The latter is out of the question, as the program has hard

coded inputs and there are no varying type functionalities. The former is difficult to judge, as the program utilises few data structures and those that are used do not see the massive amount of data being generated by the brute force algorithm.

```
sdav@sdav:~/parallelcomp/pc_coursework/openmp$ make
gcc -fopenmp -c main.c -lssl -lcrypto
gcc -fopenmp main.o -o output -lssl -lcrypto
sdav@sdav:~/parallelcomp/pc_coursework/openmp$ ./output
Segmentation fault
sdav@sdav:~/parallelcomp/pc_coursework/openmp$
```

Illustration 2: Changed Code, Segmentation Fault

In order to test what was causing the segmentation fault I separated the AES decrypting code from the brute force code I had developed and ran the latter on its own to see if it was the one causing the fault, and it was not.

```
99994
99995
99996
99997
99998
99999

935.177046sdav@sdav:~/parallelcomp/pc_coursework/openmp$
```

Illustration 3: Change Code, Successful Generation

The brute force itself is capable of running to completion even when printing to the command line, which usually uses more resources. The only thing excluded from this sequence was the AES OpenSSL decrypting code, meaning the segmentation fault occurs there.

Session Two - 7th November, 2019

After some extensive communication with Dr Kun Wei the brute force algorithm was reviewed and fixed to the degree where it is now working as intended. The fix was freeing memory after attempting to decrypt the string each pass and adding a plaintext checker for two levels of authentication when trying to find the correct decryption.

With this solved the serial testing can begin. This is conducted solely so that there is something to compare the performance of the parallel versions of the algorithm to and so that we can review any differences in time taken between the three implementations. Due to the prolonged nature of brute force algorithms, we can assume that computing one in parallel rather than serially would drastically speed up the computation.

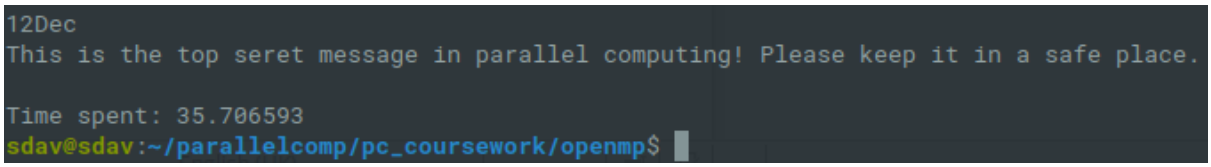


Illustration 4: Successful Serial Brute Force

The search vector for the algorithm works along “Numbers → Upper Case → Lower Case”, the search vector was decided to match the ascending order that ASCII characters would occur. The first run has revealed that the target password occurs relatively early in this search vector, with the password being 12Dec, meaning that the cracking only takes around 35 seconds.

For the sake of an active comparison, we will also test what the cracking time would be like if the target was towards the end of the search vector, that is with the vector occurring “Upper Case → Lower Case → Numbers”.

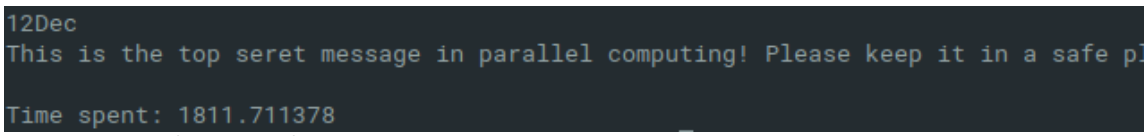


Illustration 5: Successful serial brute force, reversed search vector

The algorithm predictably takes considerably longer.

Session Three - 9th November, 2019

The method of OpenMP parallelisation decided upon was a fairly obvious one, the brute force algorithm utilises five nested for loops in order to achieve its function, and the OpenMP library introduces the concept of parallel for with the clause of collapse(n), with n being the amount of loops within that you wish to combine. Given this, each thread will handle a single iteration of all of the loops at once.

Testing this on the first search vector “Numbers → Upper Case → Lower Case” yielded some strange results. The algorithm was ran with three threads, and took longer to complete.

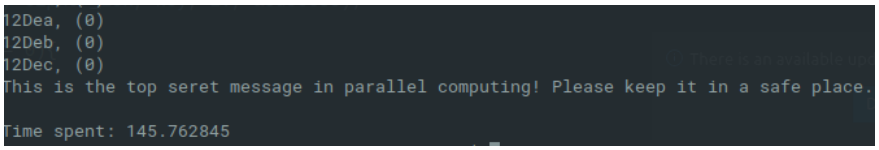


Illustration 6: Successful OpenMP parallel brute force, three threads

However, running the same three thread algorithm with the reverse search vector “Uppercase → Lowercase → Numbers”, actually causes it to find the solution faster.

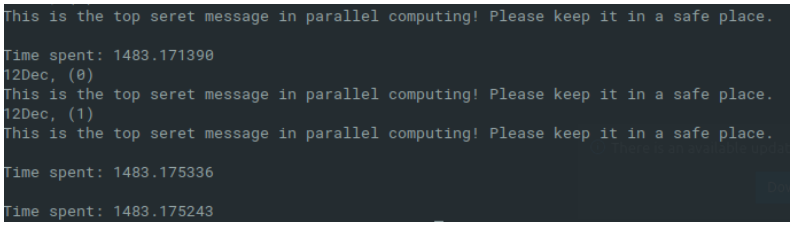


Illustration 7: Successful OpenMP parallel brute force, reversed vector, three threads

It was decided to continue researching this approach, and the amount of threads was increased from three to four. Once again, the first search vector proved slower, even than that of three threads, but not considerably.

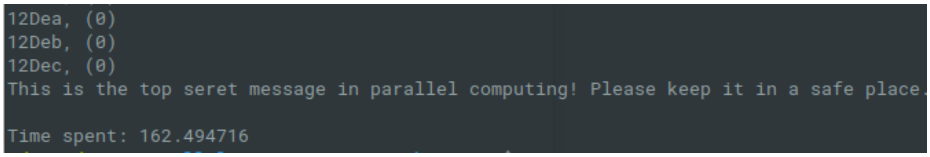


Illustration 8: Successful OpenMP parallel brute force, four threads

Then, with the reverse of the vector, it proved considerably faster than not only its serial version, but also faster than the three thread version.

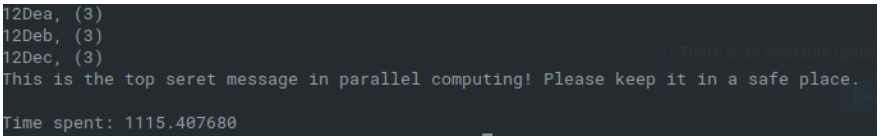


Illustration 9: Successful OpenMP parallel brute force, reversed vector, four threads

Session Four - 16th November, 2019

This session involved the planning of the OpenMPI implementation of the brute force algorithm. After researching the various OpenMPI commands a certain structure of the desired method of operation was determined.

The crux of the OpenMPI implementation was the rank system that it utilises. When initialising the algorithm you pass an argument which determines the the amount of processes that will be created to perform the action, each of these processes is assigned a rank within the default shared communication channel MPI_COMM_WORLD. We can utilise this in conjunction with the previously created alphabet lookup function, if we initialise the execution with sixty two processes then theoretically you can assign each process a seperate starting point utilising the ranks assigned by MPI_COMM_WORLD and using said rank to lookup its start point.

This is similar to one of the original plans for the OpenMP implementation, except with OpenMP threads the number assigned changes numerous times which causes a conflict of results, often resulting in multiple threads checking the same generation multiple times.

This doesnt happen within OpenMPI as the assigned rank within MPI_COMM_WORLD is unique to each process and doesnt change at any point in the runtime, unless you specifically create a new communication channel in which it would be assigned a new rank. Therefore, utilising this method each process can check one section of the search vector at the same time.

This handles the search part of the algorithm, the next is the result segment. The problem with the search implementation would be that even once one process finds the correct password the others would continue searching until reaching the end of the vectors. To work around this we can use the broadcasting functions given by OpenMPI, the standard functions are peer-to-peer, but the broadcast function allows you to promiscuously message all processes on the communication channel. Conjunctively, we can use OpenMPI's MPI_Irecv function which allows you to premeturely "post" your receive for a future send, after this you can use MPI_test with the request signature to check if the posted receive has been completed.Meaning the planned order of execution is:

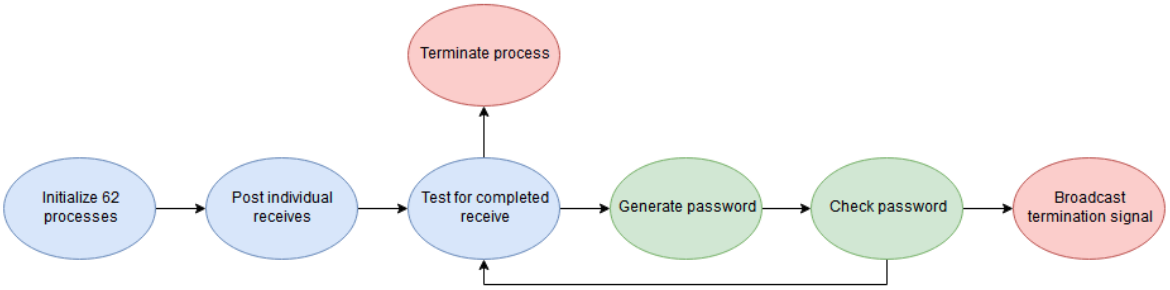


Illustration 10: MPI State Diagram

Session Five - 17th November, 2019

This session involved the development of the OpenMPI brute force algorithm, the search implementation of the algorithm works as intended, each process successfully begins searching its ranked part of the search vector.

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Irecv(&rbuf, count, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &req);
printf("Posted Recieve for %d\n", myrank);

    for(int j=0; j<dict_len; j++)
        for(int k=0; k<dict_len; k++)
            for(int l=0; l<dict_len; l++)
                for(int m=0; m<dict_len; m++){
                    MPI_Test(&req, &flag, &status);
                    if(flag == 1){
                        printf("Another process has found the key, exiting...\n");
                        MPI_Finalize();
                    }
                    *password = dict[myrank];
                    *(password+1) = dict[j];
                    *(password+2) = dict[k];
                    *(password+3) = dict[l];
                    *(password+4) = dict[m];

                    //printf("%s\n", password);

                    initAES(password, salt, key, iv);
                    unsigned char* result = decrypt(ciphertext, cipher_len, key, iv, &success);

                    if (success == 1){
                        if(checkPlaintext(plaintext, result)==0){
                            //MPI_Send(&sbuf, count, MPI_INT, myrank, 0, MPI_COMM_WORLD);
                            //sleep(10);
                            MPI_Bcast(&sbuf, count, MPI_INT, myrank, MPI_COMM_WORLD);

                            printf("%s\n", result);
                            gettimeofday(&end1, NULL);
                            double timetaken = end1.tv_sec + end1.tv_usec / 1e6 - start1.tv_sec  - start1.tv_usec / 1e6;
                            printf("\nTime spent: %f\n", timetaken);
                            //printTime(start, end);

                            MPI_Finalize();
                        }

                    }
```

Illustration 11: Session Five MPI Code

The recieves are all posted using promiscuous variables, also known as “wildcards” which allows the function to receive any messages sent on the MPI_COMM_WORLD. The function successfully finds the correct password, however the terminate function doesnt work entirely as of this session meaning that the program must be terminated manually.

```
This is the top seret message in parallel computing! Please keep it in a safe p
ace.

Time spent: 63.772221
```

Illustration 12: MPI Forward Vector

At current, the MPI implementation works as following forward vector:

Reverse vector:

```
This is the top seret message in parallel computing! Please keep it in a safe pl
ace.

Time spent: 1290.340512
```

Illustration 13: MPI Reverse Vector

Session Six - 19h November, 2019

The aforementioned method of stopping all processes when the correct result has been found was theoretically sound, but the physical implementation is refusing to work with very little to show as for why. Instead another operation was found, the process that finds the result will call “MPI_ABORT” which takes the arguments of the communication channel in which the processes should be aborted and an integer output for the error code passed.

Int MPI_Abort(MPI_Comm comm, int errorcode);

```
MPI_Abort(MPI_COMM_WORLD, err);
```

Illustration 14: mpi_abort

This is generally considered a “rough” way to terminate the MPI processes but it is an extremely effective way to do so, it works nearly instantly and stops the MPI processes from using excessive resources.

After this was implemented the 62 process MPI implementation was tested for its consistency, which can be viewed in the performance analysis section of this document.

Method of Parallelism

In theory the brute force algorithm operates on a Single Instruction, Multiple Data (SIMD) basis according to the taxonomy propesed by Flynn (1966). Flynn (1966) describes the SIMD as “single instruction, steam multiple data stream, which includes most array processes”. The brute force algorithm being used operates on a linear basis, beginning with the variable initialisation then proceeding on to the base 64 decode and Salt removal. These processes cannot be parallelised due to the linear fashion in which they operate, and even if they could be it would be inconsequential due to the fact that they only run once per execution.

Next the program proceeds onto the password string generation and AES cracking attempt. This part also works in a linear fashion, generating the password one character increment at a time and then passing this onto the decryption algorithm. After this it is checked whether the decryption has returned a success value and the returned string is compared against the target plaintext, which is what makes this equivilent to that of a Known Plaintext Attack.

The aforementioned section is the part of the runtime that can benefit from parallelisation in a SIMD manner. Due to the fact that the password generation, decryption, and success checking must occur in a sequential order it is essentially a single instruction set. The multiple data occurs at the password generation, by having multiple threads start at scheduled points in the search vector you can create a parallel region where each thread is generating a unique password string to the other threads, processing it, and checking success.

Parallelising the aforemention sections creates Data Parallelism due to the fact that each thread is operating on a different variation of a password, or a different point in the search vector. In theory this will increase the speed of our brute force algorithm because its primary time constraint is the fact that without parallelism it can only generate and check once password attempt at a time. In our Data Parallelisation implementation no thread processes more than one password but each thread processes a unique password once in parallel with the other threads.

Putting all of this together, 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. Utilising the university resources it may be possible to utilise Cluster Computing or perhaps even Massively Parallel Processing, this would however require more research, as it is unsure if the university possesses these units, or if they are readily accessible.

Performance Analysis

Testing Tables

The highlighted sections throughout the following tables display results that are utilised in the perormance metrics section of the testing tables. Red denotes the worst case scenario for the vector, and green the best case scenario.

Serial Testing

Run	Forward	Reverse
1	37.213829	1610.340708
2	36.100590	1782.511293
3	36.752606	1917.821772
4	36.130878	1663.404843
5	36.236604	1807.652695
6	35.706593	1811.711378

OpenMP Testing Table

Test	Threads	Forward Vector	Reverse Vector
Serial	n/a	35.706593	1811.711378
Collapse()	2	238.907300	5205.874778
Collapse()	4	162.494716	1115.407600
Collapse()	6	237.772732	605.256776
Collapse()	8	329.376235	2384.765861

6 Thread Collapse() Reliability Testing

Run	Forward Vector	Reverse Vector
Initial	237.772732	605.256776
1	232.043586	506.874218
2	230.588988	505.406970
3	237.161950	542.416722
4	246.695829	505.871605
5	230.096937	561.564621

6 Thread Collapse Performance Metrics

Metric	Vector	Case	Result
Total Overhead	Forward	Best	1344.875029
Total Overhead	Forward	Worst	1442.961145
Total Overhead	Reverse	Best	1422.101112
Total Overhead	Reverse	Worst	1713.718884
Speedup	Forward	Best	0.15217391304
Speedup	Forward	Worst	0.15040650406
Speedup	Reverse	Best	3.18811881188
Speedup	Reverse	Worst	3.16859504132
Efficiency	Forward	Best	0.02536231884
Efficiency	Forward	Worst	0.02506775067
Efficiency	Reverse	Best	0.53135313531
Efficieny	Reverse	Worst	0.52809917355
Cost	Forward	Best	1380.581622
Cost	Forward	Worst	1480.174974
Cost	Reverse	Best	3032.44182
Cost	Reverse	Worst	3631.540656

OpenMPI 62 Processes

Test	Forward Vector	Reverse Vector
Serial	35.706593	1811.711378
1	63.772221	1290.340512
2	54.410518	1096.054339
3	50.861461	1206.687460
4	53.749121	1189.275201
5	50.324396	1210.348929

OpenMPI 62 Processes Performance Metric

Metric	Vector	Case	Result
Overhead	Forward	Best	1509.771844
Overhead	Forward	Worst	2036.05593
Overhead	Reverse	Best	66345.02831
Overhead	Reverse	Worst	78083.289972
Speedup	Forward	Best	0.70952849588

Speedup	Forward	Worst	0.58354293478
Speedup	Reverse	Best	1.46921612433
Speedup	Reverse	Worst	1.48629121861
Efficiency	Forward	Best	0.01144400799
Efficiency	Forward	Worst	0.00941198281
Efficiency	Reverse	Best	0.02369703426
Efficiency	Reverse	Worst	0.023972439
Cost	Forward	Best	3120.112552
Cost	Forward	Worst	3953.877702
Cost	Reverse	Best	67955.369018
Cost	Reverse	Worst	80001.111744

OpenMPI Testing Table

Processes	Forward	Reverse
Serial	35.706593	1811.711378
2	0.647028	389.824451
4	0.696928	274.700083
6	3.635244	406.225463
8	3.875348	678.442976

OpenMPI 4 Process Reliability Testing

Run	Forward	Reverse
Serial	35.706593	1811.711378
Initial	0.696928	274.700083
1	0.788430	229.278593
2	0.672655	221.522366
3	0.736364	226.854755
4	0.719300	220.481466
5	0.679706	224.761184

OpenMPI 4 Process Performance Metrics

Metric	Vector	Case	Result
Overhead	Forward	Best	-33.015973
Overhead	Forward	Worst	-34.060109
Overhead	Reverse	Best	-728.414844
Overhead	Reverse	Worst	-819.02144

Speedup	Forward	Best	53.0830708164
Speedup	Forward	Worst	47.199915021
Speedup	Reverse	Best	7.30374637476
Speedup	Reverse	Worse	6.98151144279
Efficiency	Forward	Best	13.2707677041
Efficiency	Forward	Worst	11.7999787553
Efficiency	Reverse	Best	1.82593659369
Efficiency	Reverse	Worst	1.7453778607
Cost	Forward	Best	2.69062
Cost	Forward	Worst	3.15372
Cost	Reverse	Best	881.925864
Cost	Reverse	Worst	1098.800332

Brute Force Parallelism

Amdahls Law Vs Gustafsons Law

The aforementioned implementations fall into the sights of Amdahls Law. Amdahl (1967) says that a program that has had parallelisation implemented into it can still only be as fast as the critical regions which are unable to be parallelised. In the previous section, it was noted that the Base64 decode and salt removal was unable to be parallelised; as well as these, the IV and key initialisation occurs only once and is therefore is inconsequental to parallelise. These are obvious, but perhaps the less obvious consideration is the AES decryption function. Whilst multiple decryptions can occur at once, the decryption is a fairly lengthy algorithm where each part must occur in the order defined, and therefore is unparallelisable.

Taking these into consideration under Amdahls Law (1967) we can conclude that any parallelisation of the brute force algorithm will never speed the algorithm beyond the time that these critical sections take.

This is where Gustafsons Law comes into play. Gustafsons Law, proposed by Gustafson (1988) is a reevaluation of Amdahls Law. It acknowledges that a parallelised program being incapable of running faster than its sequential critical sections, but denies the assumption that one cannot speed the program beyond that point and identifies that higher performance hardware will cause an overall increase in execution time beyond parallelism.

In essence Gustafons Law looks at the broader scope of processing to correct for the gap in Amdahls Law. This is certainly true for our Brute Force parallel program, not only would the programs sequential critical sections greatly benefit from better hardware, but so would the parallel parts due to faster clock speeds.

Testing Environment

Due to the concepts acknowledged in the previous section, notably Gustafsons Law (1988), we have to acknowledge that any change in hardware would for all intents and purposes effect our testing, moving between home computer and university computers would in theory cause a noticable change in performance due to possible changes in processor speeds and memory frequencies.

Acknowledging this, it is concluded that testing must either take place on a single machine, or on a single virtual machine with specifications that are not subject to change. The latter is the best choice given the frequency at which development may change machines.

For the sake of transparency, the full settings on the virtual machine are below, running a 64bit Ubuntu 18.04 LTS with VMWare.

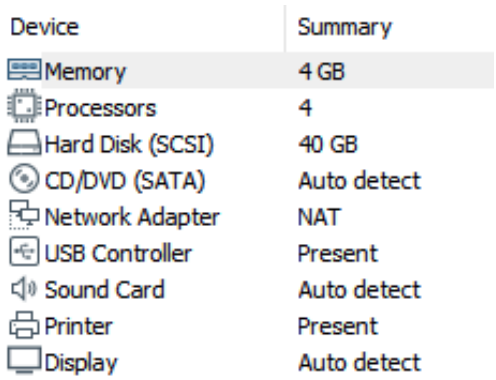


Illustration 15: Virtual Machine Settings

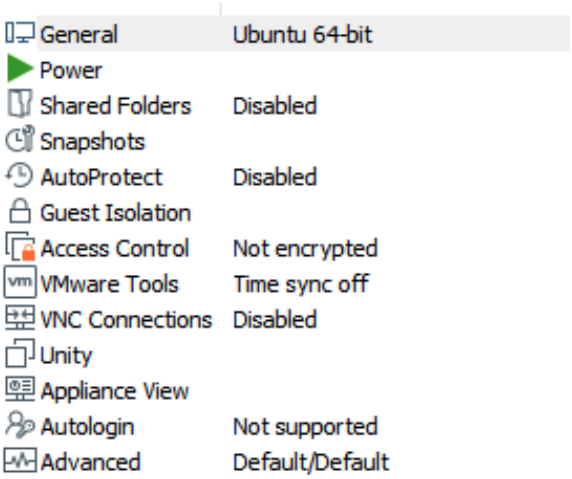
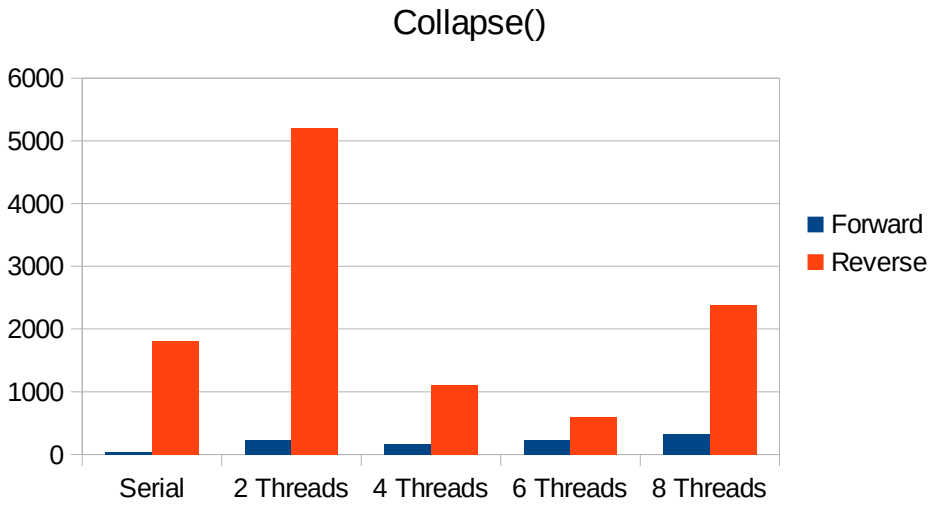


Illustration 16: Virtual Machine Settings Cont.

OpenMP Implementation Analysis

Collapse()

The first test conducted with Collapse() was the four thread forward vector, and the results were intially daunting. The time the algorithm took was considerably longer than that of the serial test, but it was quickly considered that the serial test was performing better only in this particular case due to the fact that it its search space was technically smaller than the parallel Collapse() version, and it wasn't bloated by the instruction processing time that came with the parallel implementation. Considering this, the next test was conducted, the four thread collapse() on a reverse of the search vector. This was successful, being over a whole ten minutes faster than the serial version.



The results of the four thread reverse vector created the inspiration to continue the use and testing of the Collapse() functionality. Next to see if there was a noticable increase in the forward vector and a decrease in the reverse vector, it was decided to test all the near values to four for a demonstratable comparison.

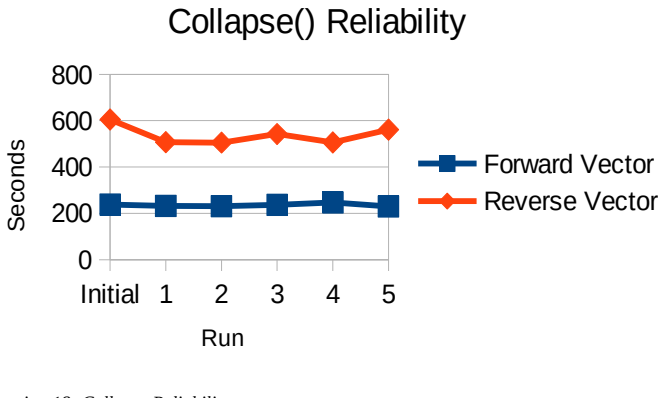
The next test involved two threads, which didnt proceed entirely as predicted. The forward vector indeed took more time than its serial and four thread counterparts. However, the reverse vector took considerably longer than not only its four thread counterpart but also its serial counterpart. The conclusions drawn from this were that even though the two thread implementation starts at a point that is technically closer to the result in the vector it doesn't start close enough to make up for the extra computational time that the parallel implementation requires, this coupled with the threads being slowed by one another, creates a situation where the implementation looks like it should work faster, but in practice takes longer.

Not being deterred by the previous results the testing continued with the six thread implementation. The forward vector test took longer than its serial counterpart and also longer than its four thread counterpart, this was predicted due to the aforementioned hypothesis where the parallel counterpart requires more processing to find a result closer to the start of a vector than the serial does. The reverse test was once again a success, with it being not only twenty minutes faster than the serial

counterpart but also ten minutes faster than its four thread counterpart. We can assume this functions in a similar nature to the two thread implementation working slower, whereas in the two thread version the extra instruction and wait time outweighs the increase in processing, in the six thread version the increase to data processing speeds outweighs the instruction processing and wait time. However there is still a noticable delay caused by the instruction and wait time, this is evidencable by the fact that the forward vector still takes considerably longer than its serial counterpart.

The final rung of testing was the 8 thread implementation. The forward search was predictably slower but the reverse search was as well, at nearly ten minutes slower than its serial counterpart. After some discussion with professors it was determined that this wasnt entirely out of the ordinary, due to increasing increasing the amount of threads increasing resources required as well as the amount of instructions processed by the CPU, creating a noticable “optimum” amount of threads.

The reliability testing for the six thread collapse implementation proved fruitful. Whilst it was consistent that the forward search vector took longer than the serial forward counterpart, the reverse vector always found the result in one third of the time it takes the serial counterpart to do so.



Despite the rather disappointing results, the average of the times taken between the serial and six thread collapse proves that the collapse implementation is genuinely faster in most cases; just not in the case of the result being towards the start of the search vector.

Performance Metrics

The total overhead for the OpenMP 6 thread collapse implementation of the brute force algorithm is fairly consistent up until a singular point. In the best cases for the forward and reverse vector as well as the worst case in the forward vector are all results within ~100 of each other. The consistency of this overhead is interesting, as it would imply that even when the resulting password is towards then end of search vector the increase in overhead is minimal. This is immediately brought into question by the result of the worst case reverse vector however, which is nearly an entire ~300 above its best case counterpart. This does however give a possible cause to our increase in processing time, something is causing an excess of overhead, but only in particular scenarios. If it was to be guessed as to what may cause said overhead, the most likely would be idle time the threads have to spend waiting for alloted processor

time not only after the other threads but also any programs running in the background. Another likely culprit is resource contention, the program utilises little in the way of memory allocation but avoiding allocation totally is difficult, and therefore the longer the algorithm runs the more likely you are to encounter said resource contention.

The speedup results for the forward vector are ignorable in both best and worse case, this was evident without needing to calculate them, the forward vector simply takes much more time in its OpenMP implementation. The reverse vector however is a different story, with the best and worst case scenarios possessing a near identical speedup value. Furthering from this, the forward vector searches can be considered horribly inefficient, but the reverse vectors pass the figurative halfway mark for efficiency, a result that one would consider quite the feat in comparison to the forward vector counterparts.

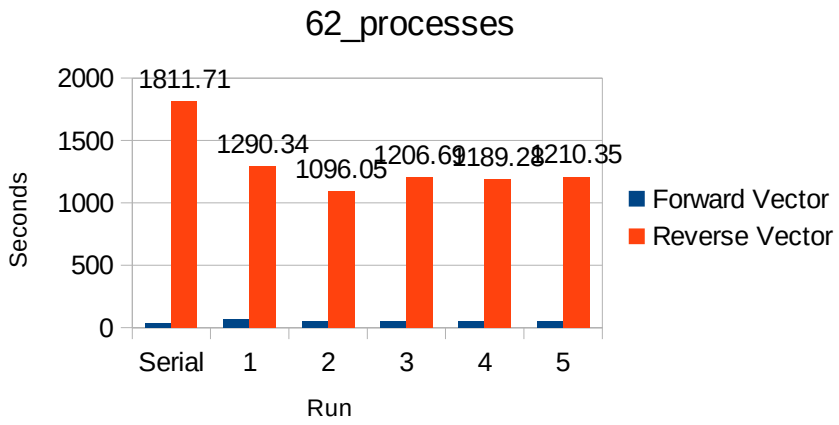
The final metric considered was cost. The cost of the forward vectors were close in relation to eachother, once again within a ~100 mark difference, this is a somewhat pleasing result as it would for all intents and purposes mean that despite the forward vector search being horribly inefficient its computational cost remains relatively the same. The best case reverse vector result is predictable, it is roughly double the two forward vector results, the worst case however is that and more, being nearly 1.1 times more costly than the best case scenario of the reverse vector, and 2.4 times more costly than the worst case forward vector. One could say that the reverse results are typical however, given that the program would predictably run for a longer time when the result is at the end of the vector and therefore an increased cost is inevitable no matter what degree the implementation appears as, calling into question if this result truly displays anything at all.

OpenMPI Implementation Analysis

62 Processes

After conducting research into the operations and functions within the OpenMPI library it was immediately known what form the implementation should take. The ranking system was so conveniently set up to work into the string generation with sixty-two processes starting at individual parts of the alphabet character array. Because of this, the outermost ring of the for loops was able to be removed, narrowing the amount of nested for loops from five to four.

The exit functionality planned was to broadcast that the password had been found from the process that entered the final state, after which the other processes would check to see if their posted recieves had been fufilled and if they had then they would finalize, however this didnt work as intended. Whilst working on paper, the recieves simply wouldnt be fufilled by the broadcast no matter what. This method is still believed to be possible to implement, and perhaps with further time and research a functioning implementation could be found.



The results of the testing were surprising. Operating on the forward vector the OpenMPI implementation came much closer to the serial value than the OpenMP implementation ever did, with OpenMPI at its longest taking 63 seconds compared to serials 37 seconds and greatly contrasted by OpenMPs 246 seconds. At its shortest, OpenMPI took 50 seconds, compared to Serials 35 seconds, and OpenMPs 230 seconds.

The reverse vector is a different story however, with OpenMPI consistently performing faster than the serial implementation, but not nearly as fast as the OpenMP implementation. At its slowest, OpenMPI weighs in at 1290 seconds (21 ½ minutes) compared to Serials 1917 seconds (31 minutes) but falls short of OpenMP at 605 seconds.

62 Process Performance Metrics

The metrics for the 62 processes implementation are greatly divisive, this is due in part to the fact that running 62 processes is highly resource intensive. The overhead for forward vector in both cases is higher than the OpenMP implementation, the

reason for this is obvious. Despite the fact that it runs faster than the OpenMP implementation it executes more simultaneous processes than OpenMP and therefore there has more wasted computation. The reverse vector suffers from the exact same issue, and is inflated even further by the fact that the algorithm runs for longer. This causes the reverse vectors overhead to be nearly 43 times in the best case, and 38 times in the worst case, as for why the best case seems to have a larger increase in overhead than the worst vector there is no evidencable reason.

The implementation performs better in regards to SpeedUp, this was evident without the metric calculations, the forward vector works faster than the OpenMP version and scores incredibly close to the serial algorithm too. Whilst the speedup results for the forward vector dont pass into a considerable value (1 or higher) the algorithm is still evidently faster than OpenMP when comparing the speedup metrics. The results for the reverse vector tell a different story however, whilst surpassing the 1 boundry and is as a result evidently better than its serial counterpart it doesnt perform as well as the OpenMP implementation, which surpasses the 3 boundry.

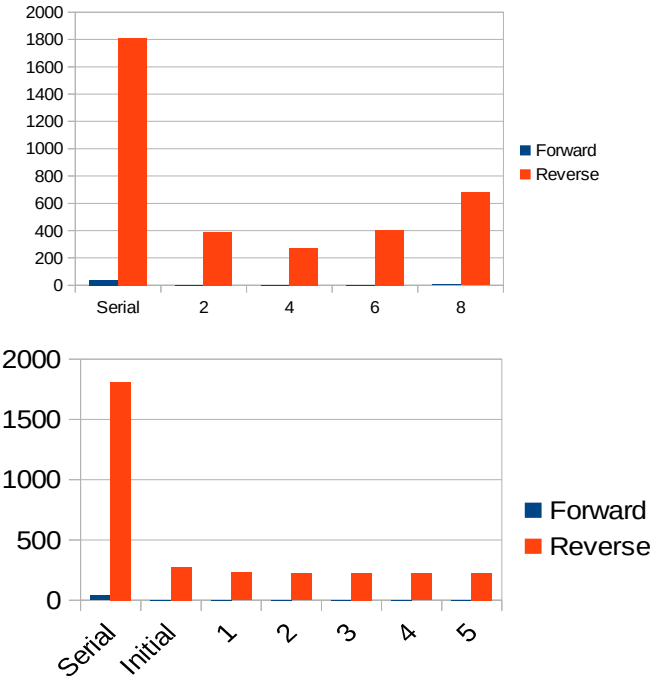
Efficiency is by far the worst category for the OpenMPI implementation. The efficiency is quite simply terrible, this is due entirely to the fact that the implementation utilises 62 processes to achieve its goals. Theoretically this creates a huge amount of contention and wasted computation. Whether this is a worthy trade off for the fact that the program functions somewhat better than the serial implementation would be down to opinion and dependent on a case by case basis.

The cost of the OpenMPI implementation is also extremely high, and it is once again due to the fact it utilises 62 processes. The excessive amount of computation paired with a longer run time causes by the reverse vector causes the cost to be 22 times more costly than OpenMP in both the best and worst cases.

Despite the fact that the metrics quite obviously point to the OpenMPI implementation being worse than OpenMP, it still has particular cases where it performs better than it, such as processing the forward vector and it processes the reverse vector faster than serial does.

4 Processes

The 4 process implementation was a direct progression from the 62 process implementation. Where the 6 process algorithm was simplistic to create but performed less than stellar, the 4 process algorithm required slightly more tweaking to get working properly but when it was finished the results were worth the time spent.



The 4 process implementation far out-performs everything else created so far, it is faster than serial, OpenMP, and the OpenMPI 62process on both the forward and the reverse vector. OpenMPI was determined to be capable of creating a fast implementation even at the 62 process stage; whilst it was slower than serial or OpenMP collapse in some regards it still displayed the possibility of speedup in some areas.

Dragging the 62process down was its work distribution. The algorithm splitting start points along the 62 characters in the search vector looked as if it would create potential speedup by having a broad breadth-first search operation, but the excessive amount of computation required was what caused it to slow down on conventional machines. The 4 process works similarly but this time narrows the width of the search space, and uses OpenMPI ranking systems alongside the MPI_WORLD_COMM to coordinate positions in the search vector.

4 Process Performance Metrics

Comparison of the performance metrics with this is relatively simplistic, all of the values created from testing this implementation are better than all other performance metrics encountered so far. All of the total overhead calculations result in negative numbers, meaning that even with multiple processes the overhead is no where near excessive enough to be out-performed by the serial implementation. As for Speedup, all of the numbers are significantly higher than any other implementation, the best case forward vector is calculated to be roughly 53 times faster than its serial counterpart, compared to 62processes and OpenMP Collapse, where neither are faster and possess no value for Speedup.

In regards to efficiency, even with its 4 process count it possess a postive result for efficiency, whereas before all efficiency results were below 1, with some even being at numerous lower decimal values, being as high as they are in comparison is obvious enough. Finally, cost; cost is a metric that is designed to seemily intentionally pull down parallel implementations due to the fact that one of the ways to decrease it is to decrease parallelism, but when comparing it to other parallel implementations it becomes slightly more useful. The 4 process cost is nearly 80 times more resource efficient than 62 process in every category, and outperforms OpenMP in every category aswell being nearly 3 times cheaper than OpenMP in most regards.

Conclusions

During the first two implementations, OpenMP collapse and OpenMPI 62process, the concept of creating an implementation that works faster than serial in every regard was looking unlikely, numerous tweaks were made and much research conducted. It was at this time a point made by Chapman and Massioli resonated, saying *“Although it is often straightforward to add OpenMP directives to an existing program, it requires much more effort to obtain high levels of performance”*, it was figured that this rang true for OpenMPI aswell. The results for OpenMP Collapse and OpenMPI 62process were bleak, whilst they certainly performed well consistently on the reverse vector testing, they could never beat serial on the forward vector search.

Not being deterred, it was decided to continue working with OpenMPI, as it had shown much promise, and after a lot of remodelling and tweaking the implementation I desired was created. Not only was it faster than serial, it was considerably faster. In essence the 4process algorithm was the culmination of what had been learned along the path that lead to it, it utilised data parallelisation, but not to the excessive degree that 62process did, because from experimenting with 62process it was learned that dividing up the space too much, and therefore having too many threads or processes is bad, more power does not equate to better performance.

In finality, the conclusions derived from this coursework are that it is certainly possible to create an efficient and well working system utilising parallel computing libraries OpenMP and OpenMPI, it just takes a lot of experimentation, tweaking, and consideration of the fundamental properties of parallel computing to achieve. Whilst the creation of an all-round better algorithm was only achieved in this project with OpenMPI, there is no doubt that given more time it would be possible to create one in OpenMP too.

References

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 14 November 2019].

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

Flynn, M.J., (1966) Some Computer Organizations and their Effectiveness. Ieee Transactions on Computers [online]. c-21 (9), pp. 948-960. [Accessed 14 November 2019].

Gustafson, J.L (1988) Reevaluating Amdahls Law. Afips Communications of the ACM [online]. 31 (5), pp. 532-533. [Accessed 14 November 2019].