PARALLEL COMPUTING LOGBOOK

INTRODUCTION

The logbook comprises the documentation part for Parallel Computing coursework. This document holds information about the AES Brute Force Algorithm's development, testing, and conclusion.

The main goal is to design a parallel Brute force search algorithm for obtaining the right secure key to crack the ciphers using shared memory and distributed memory paradigms, with Open MP and MPI parallelisation techniques. The purpose is to improve the speed at which the brute force algorithm achieves its aim, for doing that, we were provided with the necessary salted ciphertext encoded in Base64, as well as its plaintext equivalent.

LOGS OF PROGRESS

Session One – Serial test (15/4/2021)

To begin with the assignment, I have taken the code crackaes.c that was provided. The first task was to crack the given Ciphers in serial mode by using Brute force and to demonstrate it's operating by having it crack.

For performing the task, a search was done which includes the uppercase alphabets, lowercase alphabets and number from 0-9, with no special characters. This has been hardcoded in the program.

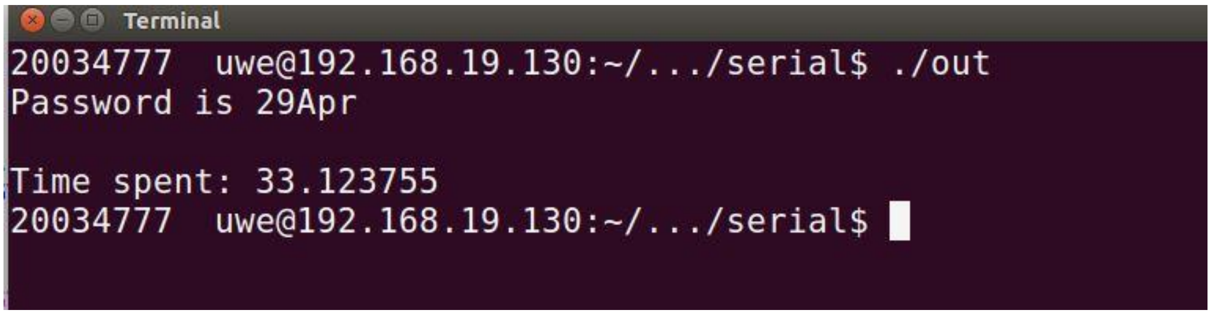
chardict[]=
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstu
vwxyz";

We have been provided with two ciphers to be cracked, cipher 2 was much easier to crack on the given program and the cracked password was 29Apr where the password length was 5. While trying to crack cipher1, it was facing difficulty, as we found the string length of cipher 1 will be different than cipher 2.

Serial Model result for cipher 2 cracked

Cracked Cipher	Time Spend
29Apr	33 seconds

The search vector works according to the given algorithm from 0-9, A-Z and a-z and when the search vector positions are been changed the cipher crack time also varies.



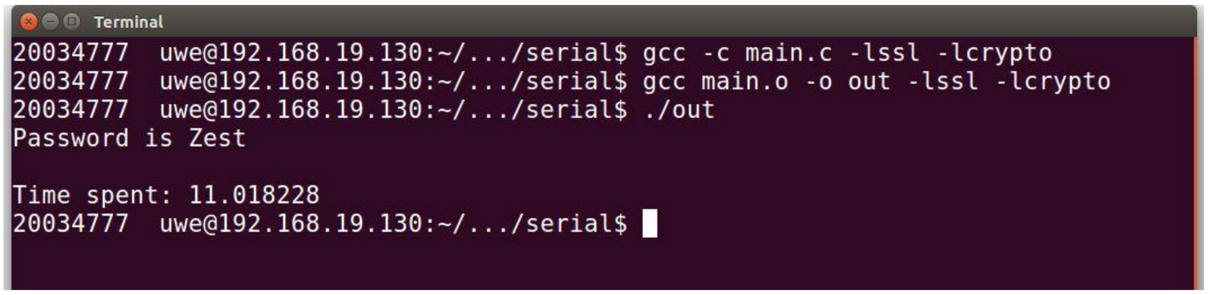
```
20034777  uwe@192.168.19.130:~/.../serial$ ./out
Password is 29Apr

Time spent: 33.123755
20034777  uwe@192.168.19.130:~/.../serial$ 
```

Figure 1: successful serial ciphe2 cracked

Session Two –Serial Test Cipher 1(19/04/2021)

After extended testing with different modification in the serial code, was able to crack the cipher 1 for which the cracked password was "Zest" with a string length of 4. The nested for loop was modified to crack the cipher1 and retrieve the password.

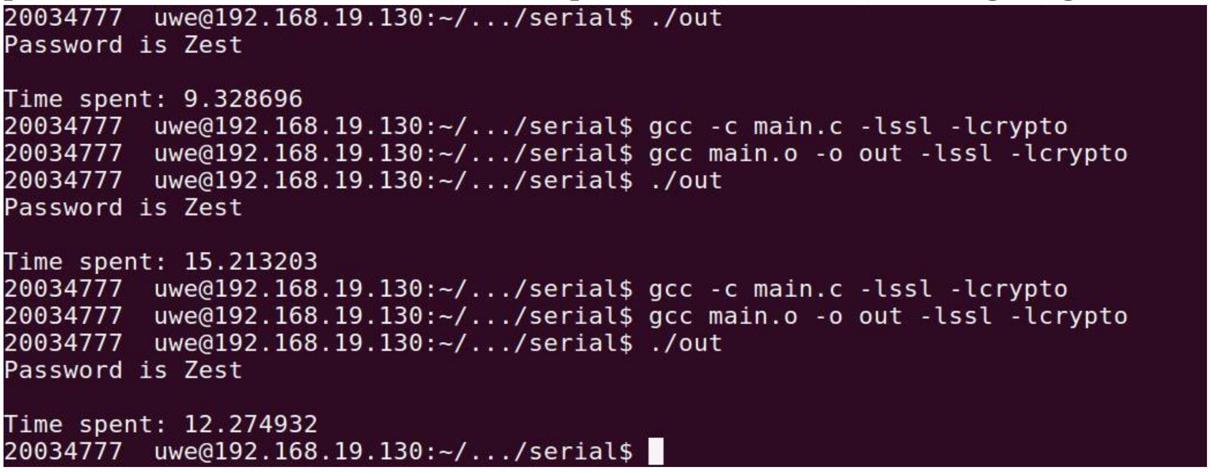
Cracked Cipher	Time Spend
Zest	11 seconds



```
20034777  uwe@192.168.19.130:~/.../serial$ gcc -c main.c -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ gcc main.o -o out -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ ./out
Password is Zest

Time spent: 11.018228
20034777  uwe@192.168.19.130:~/.../serial$ 
```

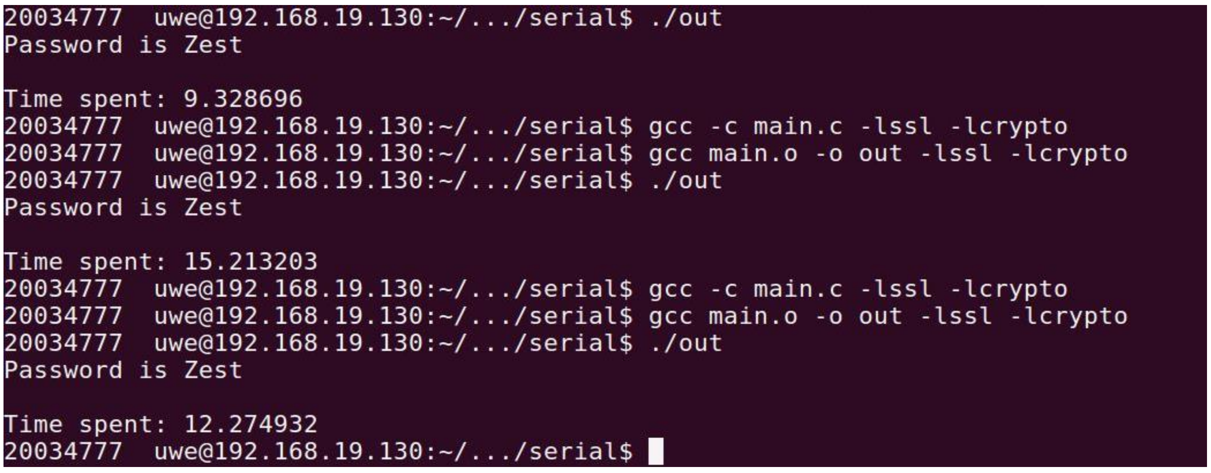
Figure 2: cracked cipher1 successfully in serial

By changing the search vector positions in the code the time spent to crack the password was different for both the ciphers. The search was taking long time.



```
20034777  uwe@192.168.19.130:~/.../serial$ ./out
Password is Zest

Time spent: 9.328696
20034777  uwe@192.168.19.130:~/.../serial$ gcc -c main.c -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ gcc main.o -o out -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ ./out
Password is Zest

Time spent: 15.213203
20034777  uwe@192.168.19.130:~/.../serial$ gcc -c main.c -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ gcc main.o -o out -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ ./out
Password is Zest

Time spent: 12.274932
20034777  uwe@192.168.19.130:~/.../serial$ 
```

Figure 3change the search vector and testing



```
20034777  uwe@192.168.19.130:~/.../serial$ ./out
Password is Zest

Time spent: 9.328696
20034777  uwe@192.168.19.130:~/.../serial$ gcc -c main.c -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ gcc main.o -o out -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ ./out
Password is Zest

Time spent: 15.213203
20034777  uwe@192.168.19.130:~/.../serial$ gcc -c main.c -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ gcc main.o -o out -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ ./out
Password is Zest

Time spent: 12.274932
20034777  uwe@192.168.19.130:~/.../serial$ 
```

Figure 4changing the search vector and testing

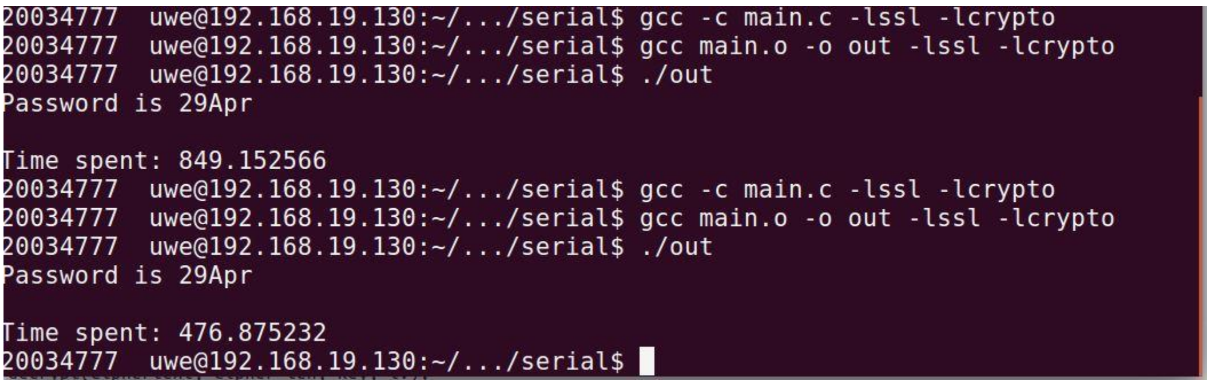

```
20034777  uwe@192.168.19.130:~/.../serial$ gcc -c main.c -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ gcc main.o -o out -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ ./out
Password is 29Apr

Time spent: 849.152566
20034777  uwe@192.168.19.130:~/.../serial$ gcc -c main.c -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ gcc main.o -o out -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../serial$ ./out
Password is 29Apr

Time spent: 476.875232
20034777  uwe@192.168.19.130:~/.../serial$ █
```

Figure 5 forward testing in OMP

The serial mode was working accuretly , the next stage is to cofigure the parallel verison using the OpenMP and MPI.

Session Three –openmp implementation(25/04/2021)

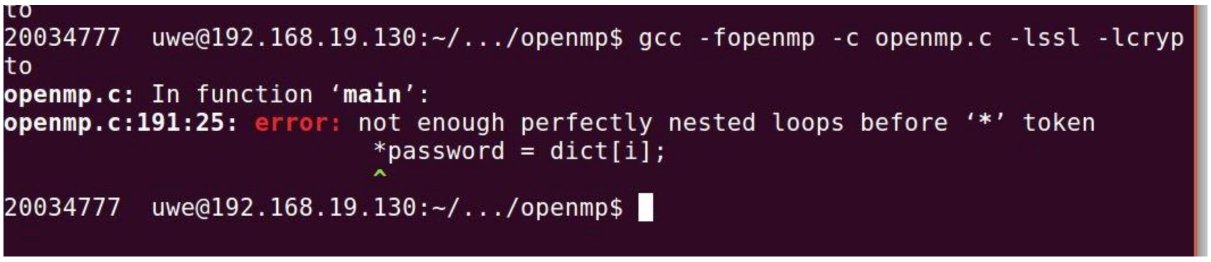
The OpenMP application program interface supports multiple platforms for shared memory parallel programming. Threading with OpenMP is considered simple but it can be used only in shared memory computer and after suffers from limited parallel scalability due to the limitations of memory architecture.

Starting with my Open MP configuration, the problem encountered was having separate passwords with different string length. The Cipher with 5 string length, the 5 nested loops works fine, but for Cipher1 where the password is 4 string length, was not working as expected.

Open MP library forwards the concept of parallelization for the clause of collapse (n), with n beings the number of loops that we wish to combine which helps in the iteration of a single thread for all the loops at once.

The password "Zest" which has been cracked in serial using four nested loops was throwing an error for me when I tried to run it in Open MP, for that, I tried to change the thread count from five to four, and it was successfully cracked.

Reconstructing the code each time to run each cipher made it hard, I did an alternative solution for that by adding an if condition check if the provided cipher is "cipher2" then run the program, else run the cipher1

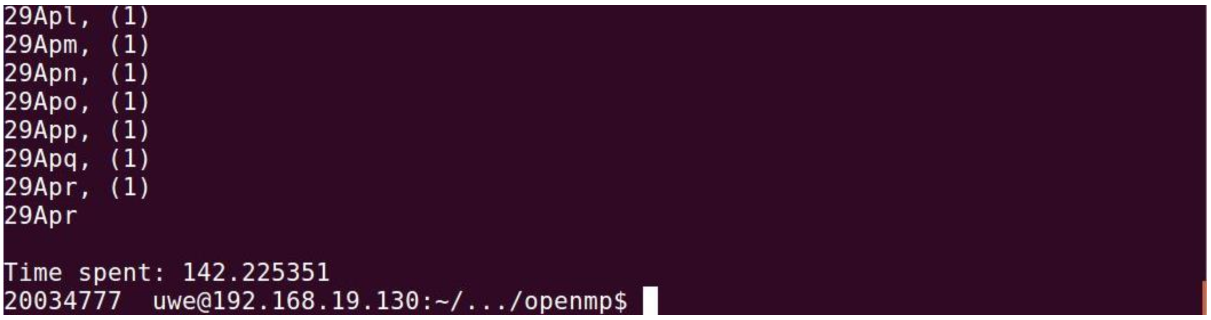


```
to
20034777  uwe@192.168.19.130:~/.../openmp$ gcc -fopenmp -c openmp.c -lssl -lcryp
to
openmp.c: In function 'main':
openmp.c:191:25: error: not enough perfectly nested loops before '*' token
                         *password = dict[i];
                         ^

20034777  uwe@192.168.19.130:~/.../openmp$
```

Figure 6 error occurred while trying to add no of threads

Session Four –openmp implementation(30/4/2021)

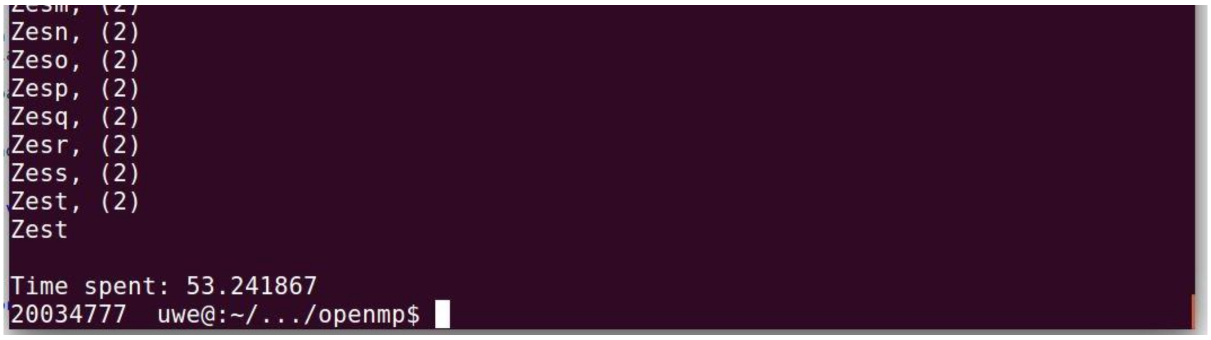
For Open MP both forward and backward vector search was done , where it was found that backward search found the solution faster for the cipher2. The vector search started from 0-9 , A-Z and finally a-z.



```
29Apl, (1)
29Apm, (1)
29Apn, (1)
29Apo, (1)
29App, (1)
29Apq, (1)
29Apr, (1)
29Apr

Time spent: 142.225351
20034777  uwe@192.168.19.130:~/.../openmp$
```

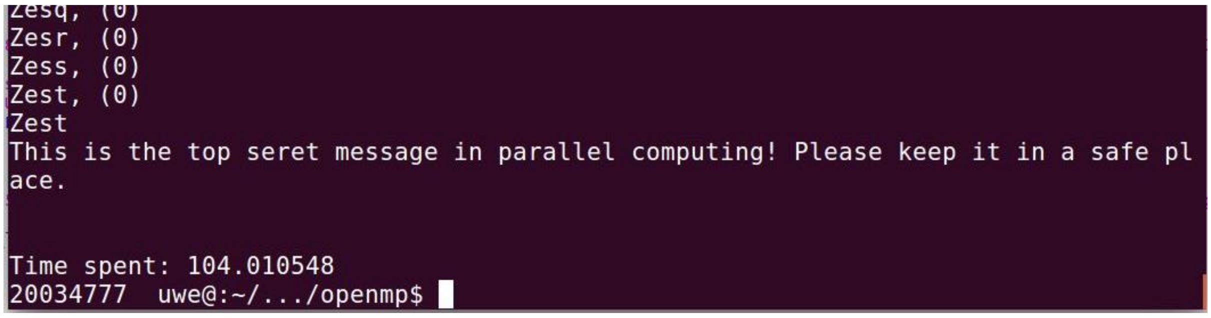
Figure 7 omp chiper2 cracked

With the backward search vector for the same Cipher2 , it was considerably taking longer time than the forward search. But for Cipher1 backward search was more faster when compared to the forward search vector.



```
Zesn, (2)
Zeso, (2)
Zesp, (2)
Zesq, (2)
Zesr, (2)
Zess, (2)
Zest, (2)
Zest

Time spent: 53.241867
20034777  uwe@:~/.../openmp$
```

Figure 8 omp ciper1 cracked successfully



```
Zesr, (0)
Zess, (0)
Zest, (0)
Zest
This is the top seret message in parallel computing! Please keep it in a safe pl
ace.


Time spent: 104.010548
20034777  uwe@:~/.../openmp$
```

Figure 9 changing the search vector and testing

Session Five- Open MPI(2/5/2021)

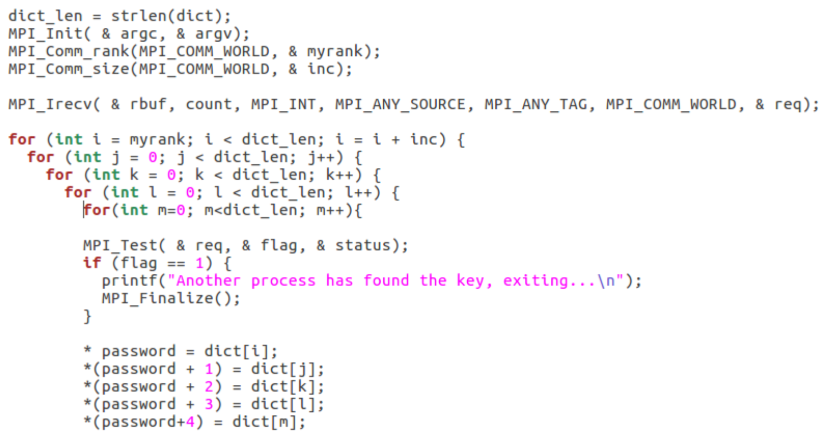
This session involves the planning and Open MPI implementation of the Brute force algorithm. After research on Open MPI it was determined that the underlying idea of the message passing model is that individual processes run independent copies of the program and these processes with other processes by exchanging message.

MPI is a standardized application programming interface for communication between separate processes. The MPI parallel program is launched as a set of independent identical processes, this process contains the same program code and instructions. MPI assigns each process rank which is used as an identification of the process.

The process can perform a separate task and handle different data based on rank. All the variables and data structures are local to the individual process. The process can exchange data by sending and receiving the message. Therefore, utilising this method each

the process can check one section of the search vector at the same time.

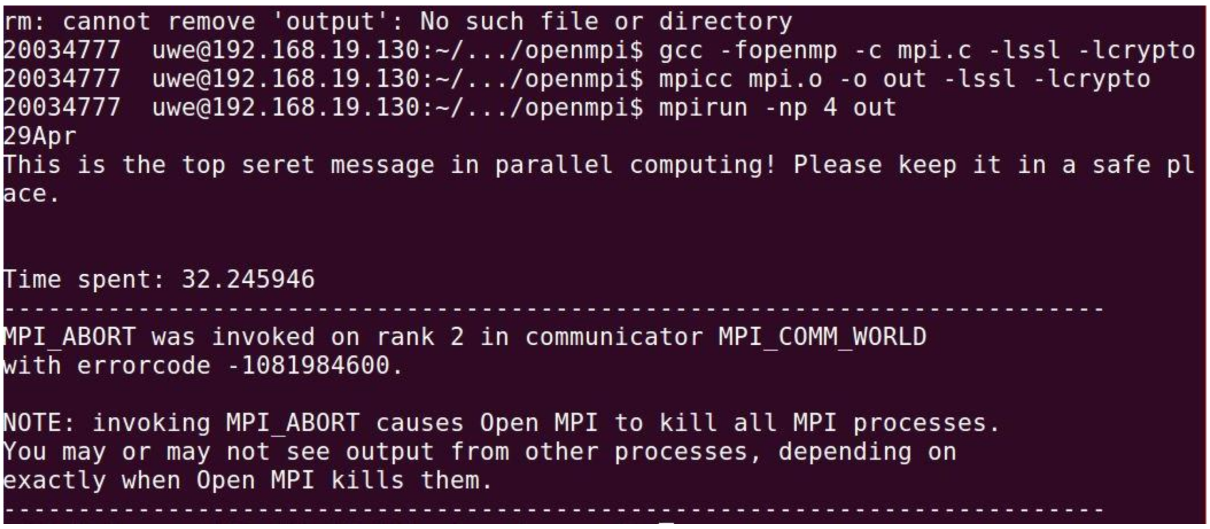
Here each process begins searching its ranked part of the search vector successfully**.**



```
dict_len = strlen(dict);
MPI_Init( & argc, & argv);
MPI_Comm_rank(MPI_COMM_WORLD, & myrank);
MPI_Comm_size(MPI_COMM_WORLD, & inc);

MPI_Irecv( & rbuf, count, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, & req);

for (int i = myrank; i < dict_len; i = i + inc) {
  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[i];
        *(password + 1) = dict[j];
        *(password + 2) = dict[k];
        *(password + 3) = dict[l];
        *(password+4) = dict[m];
```

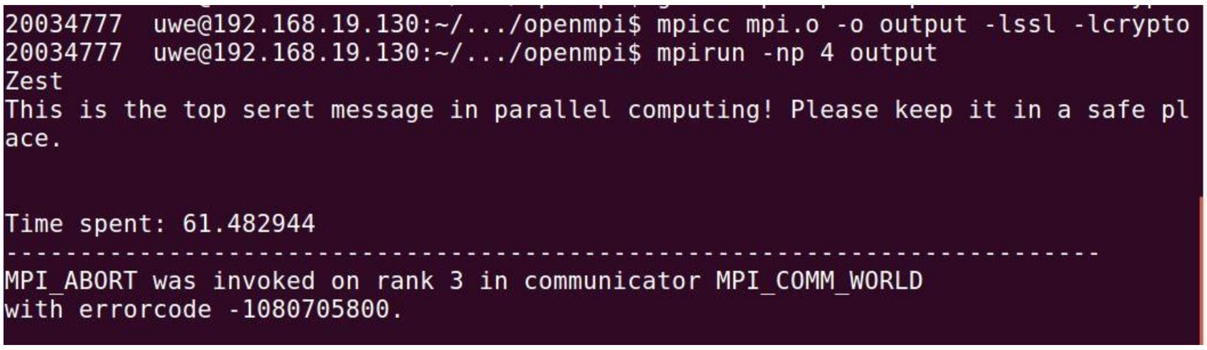
Session Six - Open MPI implementation(7/5/2021)

This session involves Open MPI impelemtation of the Brute force algorithm. The program finds the accurate password successfully and the time taken was almost near to Openmp backward search vector.



```
rm: cannot remove 'output': No such file or directory
20034777  uwe@192.168.19.130:~/.../openmpi$ gcc -fopenmp -c mpi.c -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../openmpi$ mpicc mpi.o -o out -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../openmpi$ mpirun -np 4 out
29Apr
This is the top seret message in parallel computing! Please keep it in a safe pl
ace.


Time spent: 32.245946
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 2 in communicator MPI_COMM_WORLD
with errorcode -1081984600.

NOTE: invoking MPI_ABORT causes Open MPI to kill all MPI processes.
You may or may not see output from other processes, depending on
exactly when Open MPI kills them.
--------------------------------------------------------------------------
```

Figure 10 MPI successfully for cipher1



```
20034777  uwe@192.168.19.130:~/.../openmpi$ mpicc mpi.o -o output -lssl -lcrypto
20034777  uwe@192.168.19.130:~/.../openmpi$ mpirun -np 4 output
Zest
This is the top seret message in parallel computing! Please keep it in a safe pl
ace.


Time spent: 61.482944
--------------------------------------------------------------------------
MPI_ABORT was invoked on rank 3 in communicator MPI_COMM_WORLD
with errorcode -1080705800.
```

Figure 11 MPI successful for Cipher2

The use of MPI_ABORT routine is intended to ensure that an MPI process exits (and possibly entire job), it cannot wait for a thread to release a lock.

METHOD OF PARALLELISATION

Parallelism offers one way to solve these computational problems quickly by creating and coordinating multiple execution processes. SIMD basis according to the taxonomy proposed by Flynn (1966). The most important characteristic of this paradigm is that the processors are synchronized at the instruction level, meaning that they execute programs in a lock-step fashion in which each processor has its own data stream. Hence, this paradigm is also called data-parallel programming.

In the Brute Force approach, while solving any problem we generate all possible solutions to that problem and then from those solutions, we find the best and feasible one. This method actually takes a lot of time in solving problems because in this we generate all those solutions also which are actually not a solution M. In our program the brute force algorithm is being used to operate on a liner basis, beginning with the variable declaration and then proceeding on to base64 decode and salt removal. Moving on to the next level the program proceeds onto the password string generation and AED cracking. By incrementing one character at a time the password is cracked here and then it’s passed on to the decryption algorithm. If the decryption algorithm is returned success then it is compared against the plaintext that’s been already provided.

Some or all parts of the problem can be solved or executed simultaneously, the machine will be running the parallel implementation on the single processor machine with multi-cores and thus it falls under the concept of multicore computing. No thread processes more than one password in our Data Parallelisation, but each thread processes a single password once in parallel with the other threads.

PERFORMANCE ANALYSIS & EVALUATION

Benchmark Tests

We have three programs to be tested – the serial algorithm, the OpenMP parallel algorithm, and the MPI parallel algorithm. The serial mode for the Brute Force algorithm to crack the 2 different Ciphers provided was cracked and tested in forward and backward vector search for calculating the time taken for each search. The same was done using Open MP and MPI methods, the results stated as below.

Testing Tables

Serial testing

No of Runs – Zest	Forward search	Backward Search
1	9.213203	11.752102
2	11.44486	19.556543
3	10.44656	18.556444
No of Runs – 29Apr	Forward search	Backward Search
1	142.225351	33.123755
2	156.875232	36.752565
3	145.655618	35.706593

OpenMP Testing

No of Runs – Zest	Forward search	Backward Search
Collapse(4)	47.036992	73.625788
Collapse(3)	44.169046	66.219002
Collapse(2)	42.670179	71.892123
No of Runs – 29Apr	Forward search	Backward Search
Collapse(5)	155.862799	1523.840761
Collapse(4)	157.760281	1653.889422

OpenMPI Testing

Processes – Zest	Forward search	Backward Search
4	31.564892	62.521138
Processes – 29Apr	Forward search	Backward Search
4	32.93448	19.36994

Amdahls Law Vs Gustafsons Law

The prior mentioned execution falls into the sights of Amdahl's Law. Amdahl (1967) says that if the number of processors is increased, then the speed can be increased. The Base64 decode and salt removal were unable to be parallelised; the IV and key initialisation occur only once and is, therefore, is insignificant 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 fairly lengthy where each part must occur in the order defined, and therefore is parallelisable.

Taking these into consideration under Amdahl’s Law (1967) we can conclude that any parallelisation of the brute force algorithm won’t be able to speed up the time of the algorithm beyond the time that these critical sections take.

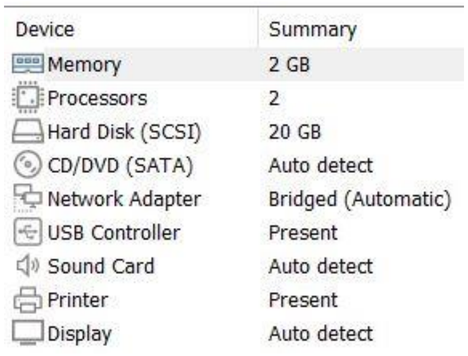
This is where Gustafson’s Law become operative. Gustafson’s Law, proposed by Gustafson (1988) is a re-evaluation of Amdahl’s Law. It acknowledges that this law says that an increase in problem size for large machines can retain scalability with respect to the number of processors. What Gustafson’s law says is that the true parallel power of a large multiprocessor system is only achievable when the large parallel problem is applied. In substance of

Gustafson's Law looks at the broader scope of processing to correct for the gap in Amdahl's 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

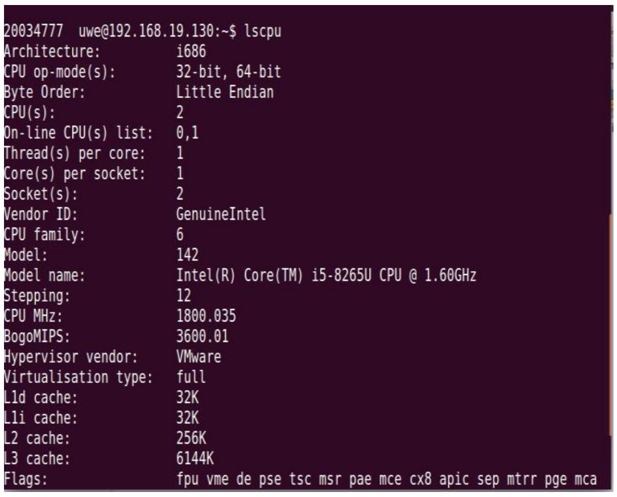
Due to the concepts recognised in the previous section, notably Gustafson's Law (1988), we can accept that any alteration that's been done on the hardware side will affect the testing part. It is concluded that testing must either take place on a single machine, or on a single virtual machine with specifications that are not subject to change.

For the parallelization tools like OpenMP, while testing with threads, where if 'x' is the number of threads, we should be sure to select the correct number of threads (not greater than the number of threads that your CPU supports.)The full settings on the virtual machine are below, running a 64bit Ubuntu version 16.1.1 with VMWare.



Device	Summary
Memory	2 GB
Processors	2
Hard Disk (SCSI)	20 GB
CD/DVD (SATA)	Auto detect
Network Adapter	Bridged (Automatic)
USB Controller	Present
Sound Card	Auto detect
Printer	Present
Display	Auto detect

Figure 13



```
20034777  uwe@192.168.19.130:~$ lscpu
Architecture:          i686
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 142
Model name:            Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
Stepping:              12
CPU MHz:               1800.035
BogoMIPS:              3600.01
Hypervisor vendor:     VMware
Virtualisation type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
```

Figure 12

CONCLUSION

Whilst for the implementation of OpenmP and OpenMPI , the concept of creating an implementation that works faster than serial in every regard was looking unlikely, numerous modifications were made and much research conducted. 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.

It had been decided to continue working with OpenMPI, because it had shown much promise, and after lots of remodelling and tweaking the implementation I desired was created. Not only was it faster than serial, but it had also been considerably faster. In essence, the 4process algorithm was the culmination of