UFCFFL-15-M
PARALLEL COMPUTING

LOGBOOK FOR COURSEWORK: BRUTE FORCE ENCRYPTION BREAKER

SUBMISSION DEADLINE: 6 DECEMBER 2018 1400Hrs

STUDENT ID: 09028091

6 November 2018

To begin the assignment, I have taken the code I have developed for a separate task which involved a known-plaintext dictionary attack on AES-128-CBC encryption[1].

This code first encrypts some plaintext using some key and some initialisation vector (IV), which outputs a ciphertext. Next the code takes a file of commonly used passwords, reads one word into a buffer, pads the buffer with '#'s up to the 16-char limit, null-terminates the buffer, and encrypts the same plaintext using the buffer as the key and the same IV as before. The two ciphertexts are then compared, and if the two ciphertexts match then we know we have found the key that was first used to encrypt the plaintext.

Due to the nature of AES encryption[2] there is no way of knowing if you are 'close' to a solution – for some key "ABC" you will learn as much from failing to decrypt using "ABB" as you will from "ZZZ". For a strong password (i.e. one not predictable or vulnerable to dictionary attack) we will not know whereabouts in the search space to focus our efforts. Therefore, a naive exhaustive search where we start from "a" and increment through the search space until we find "ZZZZZ" is reasonably the best approach.

For this exercise I am restricting my search to lowercase and uppercase alphabets, with no special characters, like so:

```
char alphabet[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

There are 52 characters in this alphabet, so the search space for a 5-character key would be 52^5 = 380,204,032 potential combinations. Of course, it will check for 1-4-character keys first, so the full amount searched will be:

$$52 + 2{,}704 + 140{,}608 + 7{,}311{,}616 + 380{,}204{,}032 = 387{,}659{,}012$$

for the key "ZZZZZ".

Using this approach, I have written an algorithm that runs in serial mode using code from the OpenSSL Wiki page[3]. In order to handle variable-length keys flexibly I have designed it to call itself recursively so that for an n-character key the algorithm will first test 1-character keys, then 2-character, up to n-character. In order to standardise performance, I will be using the keys composed of 'A' characters as the order of my alphabet array ensures these keys are near the middle of the search space.

Performance results are as follows:

Key length	Average time
1	<0.0001
2	~0.0004s
3	~0.014s
4	~0.75s
5	~37s

A key length of 5 here gives us a reasonable baseline to improve using parallelisation – extending the key-length to 6 would increase this time by 52, which would be about 32 minutes and far too slow to test.

Next steps: Parallelise the serial code and measure performance on the UWE cluster.

9 November 2018

The serial program spends most of its time in a for loop which runs 52 times (or until it finds a match). As each iteration of the loop works independently (i.e. there are no loop-level dependencies). Referring to the OpenMP tutorial[4] and videos[5], I see there should therefore be no issues with parallelising the code somewhat simply in OpenMP using the **`#pragma omp parallel for`** command.

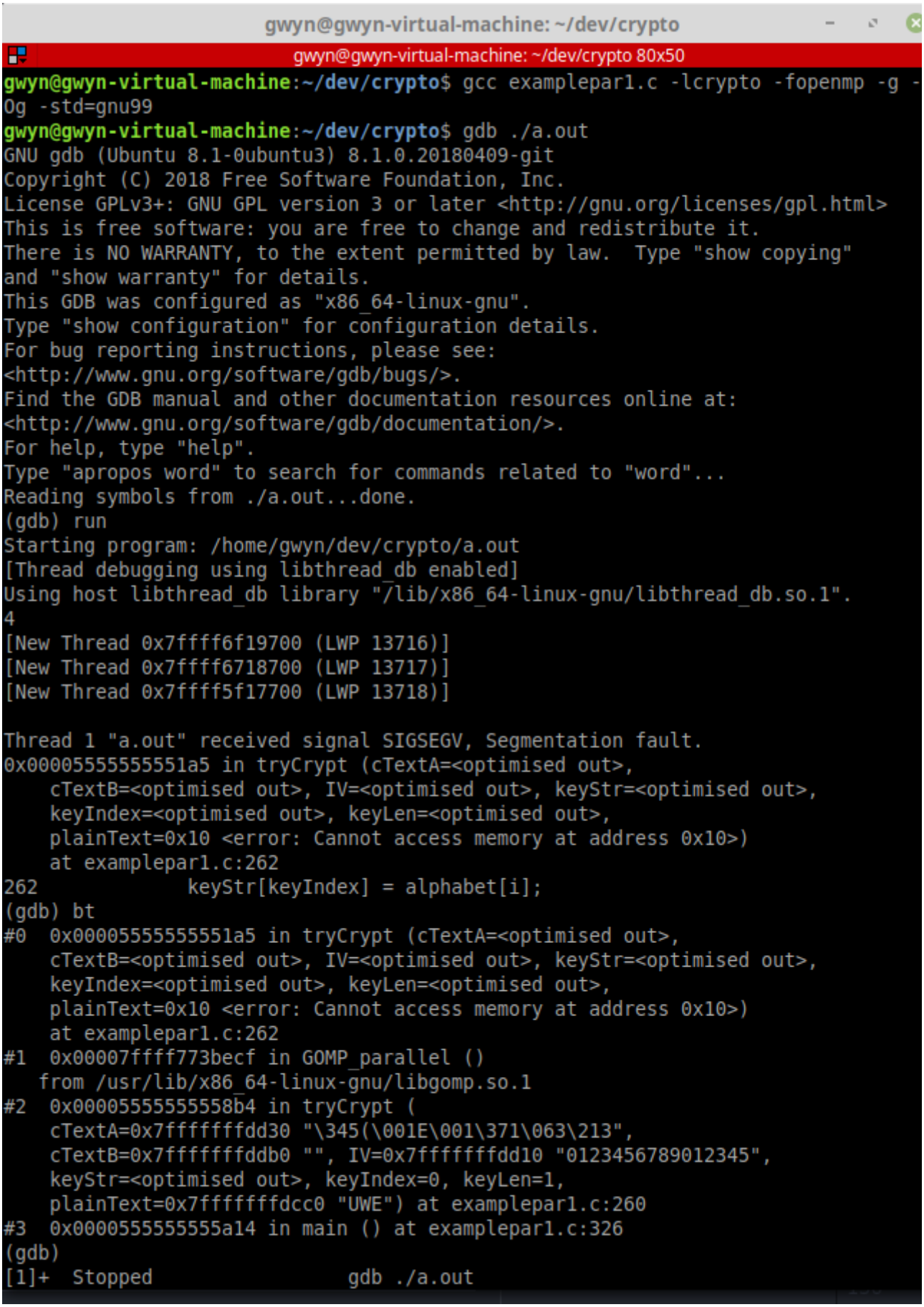
Inserting this has the immediate effect of invalidating the decryption routine, which now finds the 'success' condition and prints out an incorrect key and two incorrect ciphertexts in a seemingly random amount of time. This is to be expected as the key and ciphertext arrays are still shared and being overwritten continually by each thread.

The next step is to decide which variables need private copies within each thread – as only the key and ciphertext are written to in each loop we need to specify these as private (**`#pragma omp parallel for private (keyStr,cTextB)).`**

When this code is compiled and run we have a segmentation fault. If we recompile with the debug flags "**-g -Og -std=gnu99**" and run the code in gdb we find the following (see screenshot below).

The parallel program appears to be setting the pointer for the plaintext to the inaccessible address 0x10, even though we want this variable to be shared between all threads (it is not changed, after all). I don't know why OpenMP is causing this to happen, and if I explicitly make it shared in the pragma statement it makes no difference. OpenMP is taking a shared initialised pointer, replacing it with a private uninitialised pointer, and passing it to the encrypt function instead of the real one, all with no input from me. I assume this is due to an unterminated string.

I do notice that the variables I have set as private have not had their values initialised. Page 51 of the official tutorial is very helpful in this regard, as it shows that if you declare a variable outside the parallel block and make it private, the copy inside the parallel block will have an unspecified value. My code is doing this, so I have introduced global keyT and cipherTextT variables, and copied the contents of the key into keyT inside the parallel block. This way there is an up to date copy to process inside the loop.



“plainText=0x10 <error: Cannot access memory at address 0x10>”

Modifying the pragma statement to reflect the changes (**`#pragma omp parallel for private(keyT,ciphertextT)`**) appears to fix the issue as the program compiles and executes with no errors and some speedup.

Key length	OpenMP time	Serial time
1	~0.0004s	<0.0001
2	~0.0004s	~0.0004s
3	~0.008s	~0.014s
4	~0.41s	~0.75s
5	~20.4s	~37s

This is running on a Linux Mint 19 VM, on my Windows 10 laptop which has a Intel Core i7-5700HQ (4 cores/8 threads). The VM is set up to see 4 cores so we can expect the compiler to be using that as the default number of threads. Testing this by specifying omp_set_num_threads(4) confirms this as the timings are unchanged, compared to setting it to 2 which takes ~26 seconds.

Setting the number of threads to 8 causes execution time to jump to 280 seconds, so to play with larger thread numbers the next step is to run code on the UWE cluster.

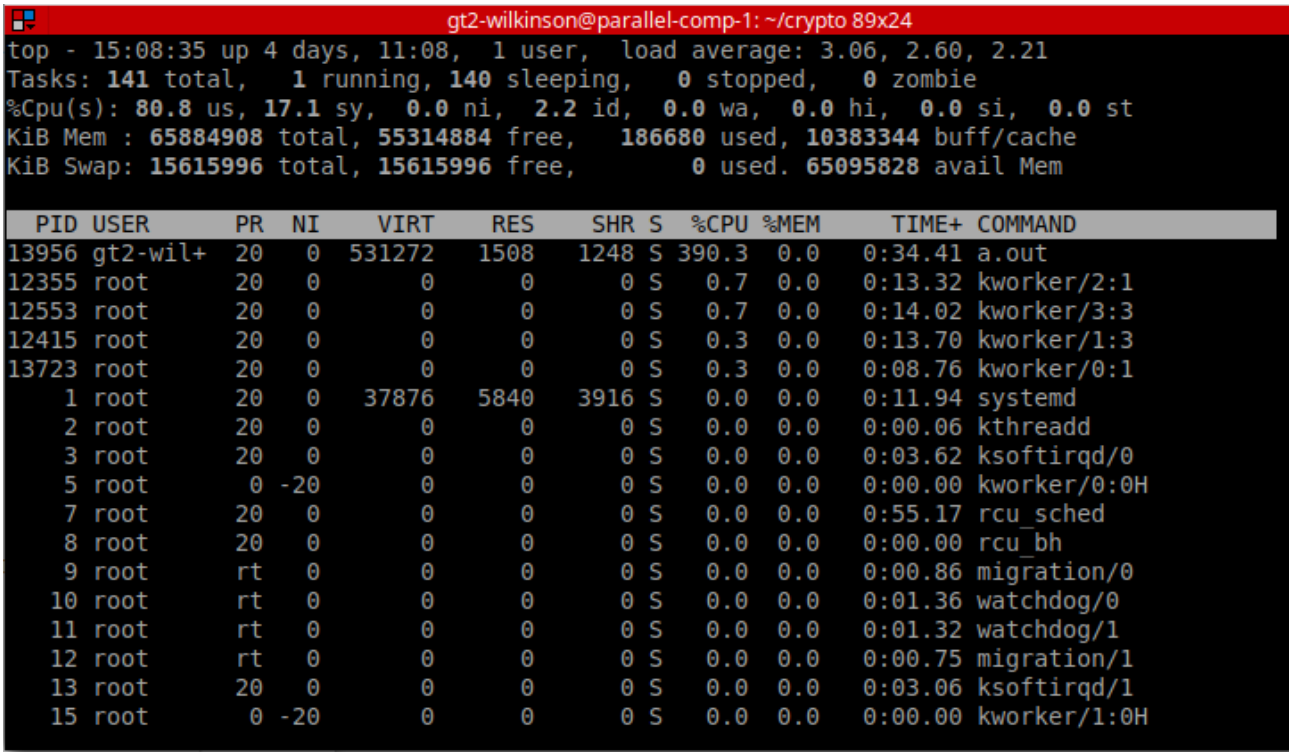
17 November 2018

I have added a command line argument that allows me to set the number of threads at runtime. This will allow me to more easily increment the threads used.

With this update, I have pulled the code onto the cluster and compiled using optimisation flag -O3 for the fastest performance. Results are as follows:

No. of Threads	Time elapsed
1	42.52s
2	30.85s
3	24.70s
4	23.01
5	121.16
6	162.98s
7	123.53s
8	79.75

Originally, I had intended to measure up to 16 threads, as /etc/cpuinfo reports 16 cores available to use on this node, but it seems clear from the above numbers that the system is not allowing me to use more than 4 physical cores. Running top while the 8-thread process is running appears to confirm this:



It is not clear why 8 threads are so much faster than 7 threads, but as it is still slower than the single-core execution I am content to assume this is a function of the key's position in the search space.

The above tests were run with schedule (static,4), which was arbitrarily chosen as I had previously run with no schedule specified, dynamic mode, guided mode, and static mode with a block size of 8, without any noticeable difference in processing speed for more than 4 threads.

The next task will be to investigate this and hopefully achieve speedups for more than 4 threads.

29 November 2018

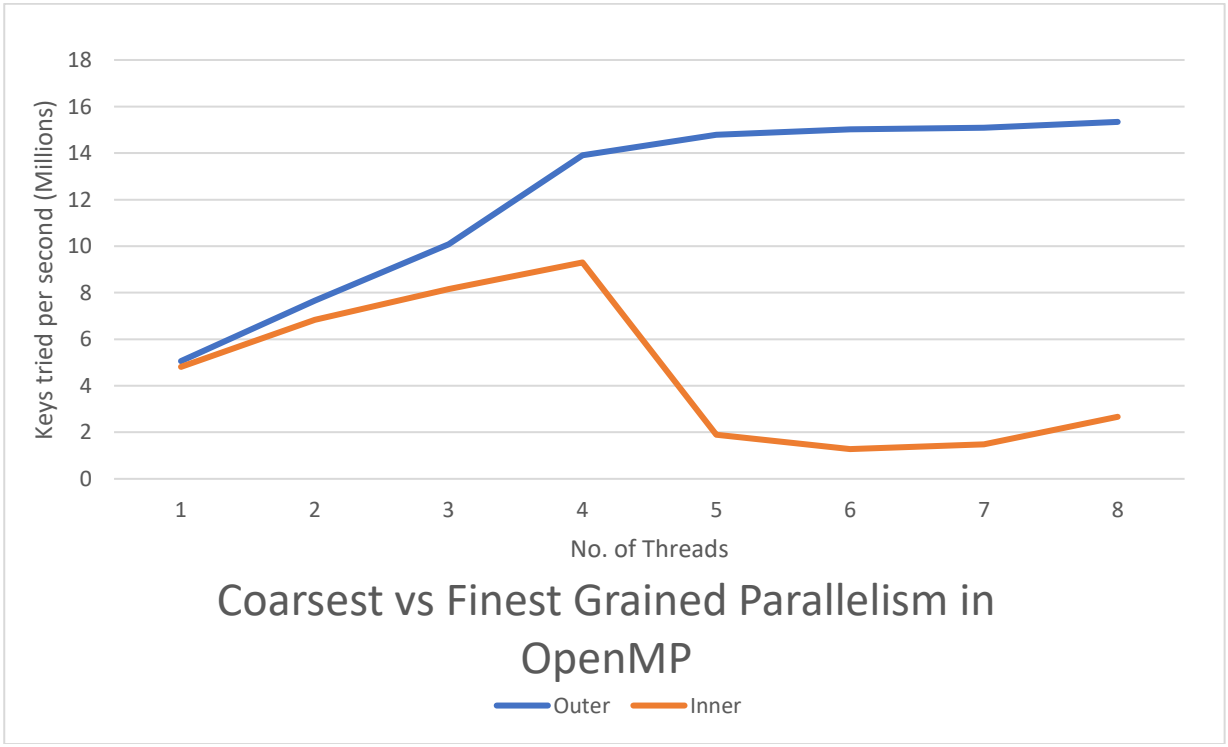
On seeing that classmates were achieving greater speedups in their OpenMP implementations which appeared to be working through the same logical steps as mine but parallelised with lower granularity (i.e. work sharing at the first letter of the key, rather than the last letter), I have decided to revisit my OpenMP logic to better understand it.

I wanted to try moving my OpenMP pragma statement to a less granular level, but after trying a couple of different approaches I had to give up. The recursive key-generation function at the core of my current algorithm is not easy to work inside and my attempts to 'shoe-horn' parallelism into the lower levels of the stack are resulting in memory read/write errors which will take time to locate.

It occurs to me that using a nested for-loop to generate fixed-length keys instead would be far simpler to code and understand, and it takes me only about an hour to have a new OpenMP solution. The added simplicity enabled me to add 'passwords tried' and 'passwords per second' calculations (using a simple reduction statement), to better contextualise the timing results, as well as having the ability to simply move my OpenMP pragma statement between loop structures so I can directly compare the performance of both approaches. The results gathered are as follows.

No. of Threads	Outer loop parallelised				Inner loop parallelised		
	Time (s)	Keys per second (Millions)	Max CPU usage		Time (s)	Keys per second (Millions)	Max CPU usage
1	38.20	5.06	100		40.27	4.81	100
2	0.93	7.66	N/A*		28.36	6.83	200
3	15.92	10.08	300		23.75	8.16	300
4	6.87	13.91	400		20.85	9.30	400
5	13.21	14.79	400		102.72	1.89	330
6	9.70	15.02	400		151.76	1.28	270
7	1.84	15.09	N/A*		130.87	1.48	305
8	13.37	15.34	400		72.97	2.66	400

* Indicates process terminated before figure could be displayed in the process manager.



As we see, when the outer loop is parallel, the keys per second continues to increase with no scaling issues beyond levelling out just over 15M keys per second, whereas when the inner loop is parallel, we hit an issue causing performance to greatly suffer. We also see the max CPU usage drops in this case, further affecting our throughput. One likely explanation is that this is caused by the overhead of starting/stopping threads so much more often, and this overhead can even be seen in the single-threaded performance difference of 2 seconds.

With this new understanding of my OpenMP code, and simplified core logic, I feel more confident in being able to develop a good MPI solution next.

30 November 2018

Starting with my serial code again, and taking a look at the lab tutorial notes for this module, I know I need to implement three things in MPI:

1. A scheme for dividing work between multiple processes.
2. A way to signal all threads to cease running when a solution has been found.
3. A way to add up the number of keys tested in all threads and output the keys per second.

There are two easy ways to do the first – we either use the thread rank as an array index and increment by the number of threads:

```
for(int i = rank; i < alphabetLen; i = i + size){
      //code goes here
}
```

Alternatively we can calculate the size of each thread's share and its starting index in the array, such as:

```
int rem = alphabetLen % size;
  for (rank=0; rank<size; rank++){
    if(rank<rem){
      chunkstart[rank] += rank;     //spread remainder evenly between threads
      chunksize[rank]++;
    }else{
      chunkstart[rank] += rem;
    }
  }
For(int i = chunkstart[rank]; i < (chunkstart[rank]+chunksize[rank]); i++){
      //code goes here
}
```

Both the above methods will split the workload between threads reasonably evenly. The first method however means we must start at the beginning of the search space and work across, each thread taking an alternating 'stripe', while the second means we can point our threads at blocks of near-equal width across the search space. While in a naïve search such as this it should make little difference, I consider the second option more elegant (as well as aesthetically pleasing) and will be implementing this.

The second task, the signal for thread termination, can also be implemented in two ways. The first is to simply call the error function MPI_Abort() on finding a correct result, which stops all processes and returns an error message. The second is to use non-blocking send and receive functions to signal all processes to end. When a correct result is found, the successful thread sends a success flag to the root process, which sends it to all processes. When a thread receives the success flag it ceases to iterate further and passes straight to MPI_Finalize().

The first is very fast and immediately ceases all processing, however by design it doesn't allow us to gain information from other threads (namely, number of keys tried) before exiting. Also the error message can't be disabled, and the abort function is not designed to be used in this fashion. The second sequence will exit cleanly and we will be able to calculate our performance, but each iteration introduces a communication overhead in every thread. We therefore need to find a balance between reducing overhead to reasonable

levels while not doing a lot of unnecessary processing while we wait for the success flag to reach all the threads. As the core logic is composed of 5 nested for-loops, we have 5 choices of where to place this code, and we can simply test all of them to find the best balance.

Send/Recv Loop Depth	Time (s)	Keys per second (Millions)
1	18.04	19.25
2	11.47	31.57
3	11.212	32.28
4	4.65	55.87
5	20.77	11.94

As the performance is best when the success signal code is in the second deepest loop, I will implement this. There is still a noticeable delay between finding the key and the program exiting, which must be considered during analysis.

In theory you can transmit the success flag using MPI_Reduce() (using a logical OR operation, so one true result returns true for all) combined with the broadcast function MPI_Bcast(), but in trying to implement this I found that these two blocking functions simply blocked every thread and never released. I still think this would be an elegant solution to this kind of parallel program, but I didn't want to spend more time debugging it. The second method works and is fast, which is enough for this task.

Thirdly to add up the numbers of keys tested across all threads, we use the MPI_Reduce() function which, similar to OpenMP's reduction statement, allows us to simply take every thread's copy of a variable and add them together. We can place this at the end of the program just before MPI_Finalize() as we will not need to worry about blocking by then.

As well as using the Practical lab session notes, I have found a very helpful guide called MPI For Dummies[6] which covers communication and datatypes very well, as well as the OpenMPI documentation pages [7]. I have also found the function index at DeinoMPI[8] useful for its example code snippets for each function.

After a few tries, I have a solution which works on the UWE cluster, exiting cleanly and outputting the correct key and some performance statistics as with the Serial and OpenMP code.

There are a few interesting aspects of executing MPI code which I have discovered. Firstly, the cluster and its nodes (located at 164.11.39.11, 164.11.39.12, 164.11.39.13 & 164.11.39.14 respectively) need to be prepared before trying to run code across them. To begin with, you must SSH into each of the 4 nodes and generate an SSH key to copy to each of its 3 neighbours, twelve key transfers in total. Without this, the machines will not communicate, and you will be restricted to one node only.

Secondly, after compiling using 'mpicc' you must use 'scp' to copy your program to the same directory on each other node. Without doing this, MPI will allocate processes on these nodes, and processes on the root node will behave as though they exist, but they will not have any code to run and this will lead to the correct key not being found.

Thirdly, each 'mpirun' command needs to specify the nodes used and number of process slots allocated for the program. Without doing this, each run command reads something like this:

```
$ mpirun -np 16 -bind-to none -host 164.11.39.11 slots=4 max-slots=4
164.11.39.12 slots=4 max-slots=4 164.11.39.13 slots=4 max-slots=4 164.11.39.14
slots=4 max-slots=4 mpi-brute-force
```

Fortunately, there is a -hostfile parameter which allows us to put the node information into a text file, like so:

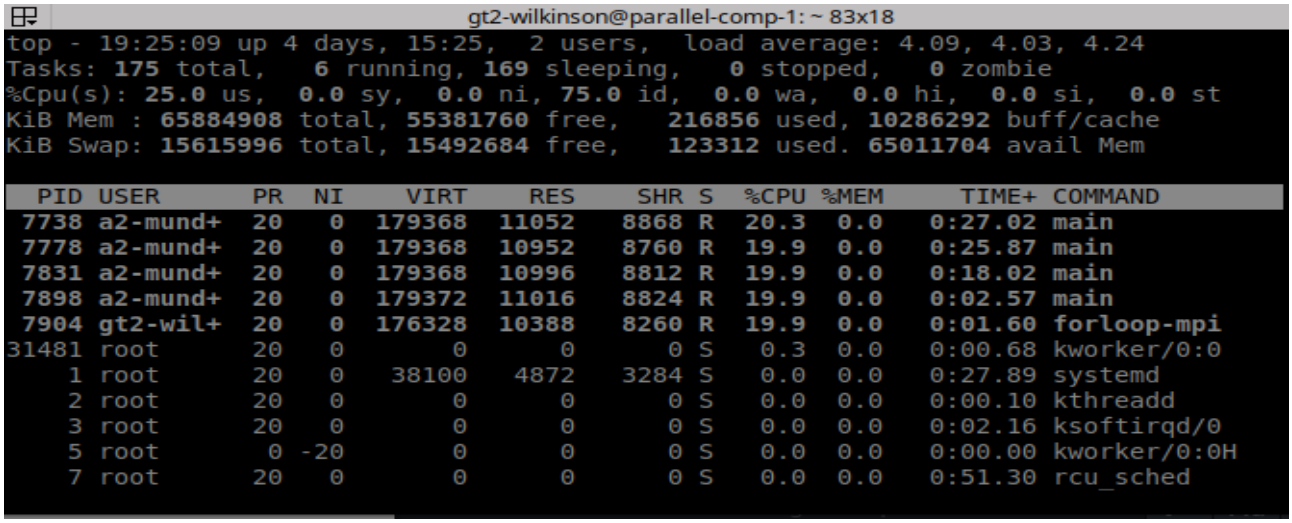
```
$ cat hostfile
```

```
164.11.39.11 slots=4 max-slots=4
164.11.39.12 slots=4 max-slots=4
164.11.39.13 slots=4 max-slots=4
164.11.39.14 slots=4 max-slots=4
```

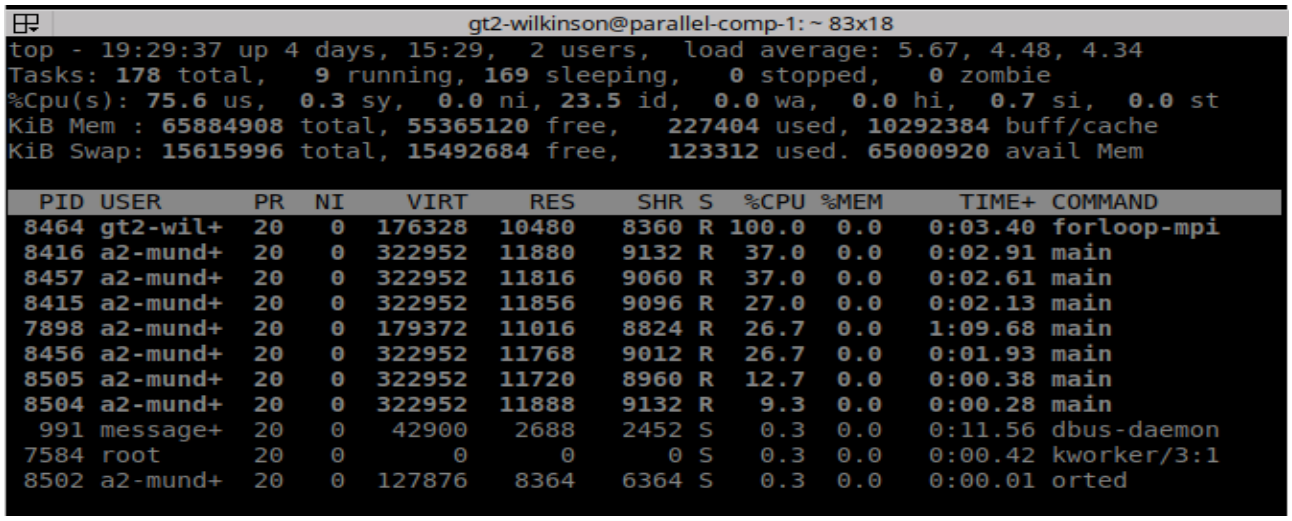
This lets us get the same result by running:

```
$ mpirun -np 16 -hostfile hostfile mpi-brute-force
```

Lastly, if we run the above with -np 1, and another process is running on processor 1, we see the following:



The single MPI process is sharing CPU time on a single core with another program. This is intended behaviour by MPI which automatically binds processes to cores unless asked to do otherwise. If we add the parameter -bind-to none to the mpirun command, instead we see:



The process can run at full speed on an unused core, meaning we can rely on its full performance.

As the above pictures show however, we might have a problem with congestion on the node which could cause total performance to suffer when we come to perform our analysis.

With this implemented and running across all 4 nodes, I believe this code is completed and I am ready to start the performance analysis and evaluation.

References

1. Syracuse University (2018) *SEED Labs – Secret-Key Encryption Lab*. Available from: http://www.cis.syr.edu/~wedu/seed/Labs_16.04/Crypto/Crypto_Encryption/Crypto_Encryption.pdf [Accessed 1 December 2018].
2. National Institute of Standards and Technology (NIST) (2001) - *FIPS 197 Advanced Encryption Standard (AES).* Available from: https://csrc.nist.gov/publications/detail/fips/197/final [Accessed 1 December 2018].
3. OpenSSL (2017) *EVP Symmetric Encryption and Decryption*. Available from: https://wiki.openssl.org/index.php/EVP_Symmetric_Encryption_and_Decryption [Accessed 1 December 2018].
4. Mattson T. & Meadows I. - OpenMP (2008) *A "Hands-on" Introduction to OpenMP.* Available from: https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf [Accessed 1 December 2018].
5. Mattson T. - *Introduction to OpenMP - Tim Mattson (Intel)*. Available from: https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG [Accessed 1 December 2018].
6. Balaji P. & Hoefler T. (2013) *MPI For Dummies.* Available from: https://htor.inf.ethz.ch/teaching/mpi_tutorials/ppopp13/2013-02-24-ppopp-mpi-basic.pdf [Accessed 1 December 2018].
7. Open MPI (2018) *Open MPI Documentation*. Available from: https://www.open-mpi.org/doc/ [Accessed 1 December 2018]
8. DeinoMPI (2009) *MPI Functions*. Available from: http://mpi.deino.net/mpi_functions/index.htm [Accessed 1 December 2018].