N

Parallel computing

Logbook - Adam Stock

University of the West of England

UFCFFL-15-M - Parallel computing

# First attempts 19/11/18

## Initial attempt 1.c

This initial version was created to explore the code given to us for reading and contains little in the way of student code, however reading it provides some insight into the workings of the encryption and decryption functions.

## Version 2.c

This version was created as a framework for future usage, it contains #includes for most things I could think of at the time that may be required based on work completed in previous lab sessions.   
the code itself has listing for items that will be used in all future versions of the program.

These are as follows –

char truekey [16] = "#####123456#####";

This is where that actual key we are trying to find will be stored, note this has been artificially  
limited by way of padding with ‘#’ and the usage of only numerical characters.

I have used 6 characters each containing 0-9 giving us 10^6 or 1 million possibilities – this could easily be expanded but the computational power required for this would be beyond the scope of this assignment.

char originaltxt [16] = "thisisatestxxxxx";

This is the text that will be encrypted by the AES function its actual contents matters little but must remain consistent in order to provide useful comparison.

char encryptedtxt [16];

This is the same text as above but after the AES function has been completed.

char work [16];

This will be where the candidate encrypted text will be stored while awaiting comparison to the encrypted text mentioned above.

char testkey [16] = "#####xxxxxx#####";

This is where the working key to be tested will be stored with ‘x’ as a placeholder to be generated later.

The ‘int main’ contains a basic nested for loop to generate the desired test key as well as some commented pseudo-code outlining further plans to continue this.

The function is designed to print out each test key sequentially so that its accuracy can be checked.

# 20/11/18

## Version 3.c

This is the first attempt at adding the encryption function. It attempts to encrypt the ‘originaltxt’ string then uses ‘strcmp’ to compare the newly encrypted string ‘work’ with ‘encryptedtxt’.   
‘strcmp’ places its output in to the int ‘ret’, if this value is equal to 0 then key is considered to be correct. Note this code does not work. In order to use the encryption functions the linker switch ‘-lcrypto’ must be used during compilation.

# 29/11/18

## Version 4.c

This version is a fixed version of 3.c above. The declarations of the char strings have been moved inside of main and pointers have been changed to direct links to ease troubleshooting. This version stores a clock into ‘time1’ variable before going into the nested for loop structure, when a match is found another clock variable is stored into ‘time2’. ‘time1’ is then subtracted from ‘time2’ and placed into ‘timer’, the result is then divided by the ‘CLOCKS\_PER\_SEC’ function in ‘time.h’ giving us the amount of time taken between beginning the loop and finding a match, this is then printed.

# 1/12/18

## Version 5.c

Added basic parallelism to the program, initially the variables ‘ix’, ‘testkey’, and ‘work’ were left outside of the #pragma comment, however this gave undesired results due to multiple threads modifying variables without blocking/synchronization with other threads.

In order to use said parallelism the header omp.h was included in the includes section and compilation was completed with the command – ‘gcc 5.c -o 5 -lcrypto -fopenmp’ , the linker switch ‘-fopenmp’ enabling the parallelism.

With the aforementioned variables moved inside the comment then then running code produced results but in this case each thread completed exactly the same workload in parallel (in effect being 4x less efficient). To combat this the outer for loop was modified to begin each threads search at an offset based on the number of the thread.

In this attempt 4 threads were used and the offset was set using ‘idstart’. Considering that each loop cycles through numbers 0-9 we have 10 total loops. The offset was set such that it was equal to 3 \* number of threads, meaning that thread one would start at 0, two at 3, three at 6 and the fourth at 9.

Using this produced work but was unbalanced with the threads completing workloads as shown in the graphic below –

This shows that thread 3 had a considerably smaller space to search, all threads would complete their tasks given time but with a password set to #####999999##### thread 3 would complete disproportionally faster than the other threads.

To clarify this version of the code has threads searching the following spaces –

Thread 0 - #####000000##### - #####999999##### = 999999 iterations   
Thread 1 - #####300000##### - #####999999##### = 699999 iterations  
Thread 2 - #####600000##### - #####999999##### = 399999 iterations  
Thread 3 - #####900000##### - #####999999##### = 99999 iterations

The actual result of this is that thread 3 would finish first followed by 1, 2 and finally 3 at proportional intervals.

## Version 6.c

With this attempt I tried moving the #pragma comment to the central most nested for loop. This initially proved to be an issue due to variables being in the incorrect places. When this was fixed the results given showed little difference in time between the threads. This is due to overheads, when run in this way the work is distributed into thousands of tiny threads with very little work to do as opposed to the previous method creating 4 large threads with relatively large workloads. In this case the overheads from switching threads greatly outnumber the workload in each thread giving a huge loss in efficiency.

Due to this the idea was scrapped.

# 2/12/18

## Versions 7-1thread.c, 7-2 thread.c and 7-4 thread.c

With the failure of the previous test the basis of code was returned to 5.c.

In these attempts the code has been split into 3, this is to allow testing of the amount of threads available that are divisible by the numbers in the search space. In this case using number 0-9 (10 in total) and given the cpu I am working on being able to handle a maximum to 8 threads concurrently the amounts of threads chosen were 1, 2 and 5 as these are the only numbers divisible by the amount of potential cases. Also a Boolean was added as a flag so that only the thread that finds the correct result first would print its result and the time taken.

In 7-1.c no offset is used inside of the #pragma comment this is due to the system being limited to only using 1 thread – effectively making this a serial version.

In 7-2.c the offset was set so that threads would start at 5 \* thread number, in the case of thread 0 - 0, and thread 1 – 5. The idea of this is to split the workload in half. The actual result of this is that threads search the following –

Thread 0 - #####000000##### - #####999999##### = 999999 iterations   
Thread 1 - #####500000##### - #####999999##### = 499999 iterations

In 7-5.c the offset was set so that threads start at 2 \* thread number, resulting in the following –

Thread 0 - 0 \* 2 = 0  
Thread 1 - 1 \* 2 = 2  
Thread 2 - 2 \* 2 = 4  
Thread 3 - 3 \* 2 = 6  
Thread 4 - 4 \* 2 = 8

This results in the threads beginning at 200000 iteration intervals as follows –

Thread 0 - #####000000##### - #####999999##### = 999999 iterations   
Thread 1 - #####200000##### - #####999999##### = 799999 iterations  
Thread 2 - #####400000##### - #####999999##### = 599999 iterations  
Thread 3 - #####600000##### - #####999999##### = 399999 iterations   
Thread 4 - #####800000##### - #####999999##### = 199999 iterations

The result is that while the search space is being searched rapidly, it is not efficient.

## Versions 8-1 thread.c, 8-2 thread.c and 8-5 thread.c

This iteration of the code is mostly an update to increase efficiency of the system.

In each outer for loop the parameters have been changed to make each thread complete workloads without any overlap. This has been accomplished by changing the termination of the for loop to be proportional to the thread number.

In the case of 8-1.c this is unchanged as this is a serial program.

In the case of 8-2.c this is changed to idstart + 4 meaning threads will end as follows -

Thread 0 - 499999

Thread 1 - 999999.

Giving a ranges of the following per thread –

Thread 0 - 0 - 499999  
Thread 1 - 500000 - 999999

In the case of 8-5.c this is changed to idstart + 1 meaning threads will end as follows

Thread 0 -199999  
Thread 1 -399999  
Thread 2 -599999  
Thread 3 -799999  
Thread 4 -999999

Giving ranges as follows –

Thread 0 - 0 - 199999  
Thread 1 - 200000 - 399999  
Thread 2 - 400000 - 599999  
Thread 3 - 600000 - 799999  
Thread 4 - 800000 - 999999

This means that each thread has a range of 199999 iterations hence the chart from earlier will look like this -

# 3/12/18

## Versions 9-1 thread.c 9-2 thread.c and 9-5 thread.c

This version contains the revision of the timing portion of the program. Originally the function of clock() was stored into the double ‘time1’ before the for loop and stored again into ‘time2’ when the result was found. So calculate a result we use time2 – time1 and place this into timer, this is then divided by CLOCKS\_PER\_SECOND to give a final printed output.

In the multithreaded 9-x.c files this has been changed to use the omp\_get\_wtime() function. This is initially placed into time1 and again into time2 as before and again time2 – time1 is placed into timer however due to the nature of the function this can be displayed directly.

This decision was made due to results appearing more consistent with the latter method. Upon investigation this may be due to the ways in which they record time. The clock() function measures time spent actively on the cpu while omp\_get\_wtime measures time that has been used during execution.   
it may appear that the former is the best choice to measure performance in this instance but in practice it gave quite variable results.

# 4/12/18

## mpitest.c

As the name suggests this file was created using a tutorial as a test of the MPI compilation and environment. It is a simple multithreaded hello world program that can be compiled using the command - mpicc mpitest.c -0 mpitest

## Versions 10-1 thread.c, 10-2 thread.c and 10-5thread.c

These versions contain code designed to be compiled for the Message Passing Interface (MPI).

Changes to this code include replacing the #pragma comments with ‘MPI\_Init’ command, this functions in a similar way to the #pragma comments. Where #pragma’s scope is based on code being between { }, MPI’s scope goes from ‘MPI\_Init’ and ends at ‘MPI\_Finalize’. Doing this required a change of code to detect the thread number, this is now accomplished with the command ‘MPI\_Comm\_rank(MPI\_COMM\_WORLD, &threadno)’ this places the thread number into the integer ‘threadno’ for future usage.

## Makefile and automated upload scripts

At this time the inclusion of scripts was added to enable uploads to the cluster and faster compilation of multiple files.

The makefile was added with cases for each of the numbered threaded .c files with all the included flags and switches. At the top of this file a case for ‘all’ was added to allow the ‘make’ command to compile all of the .c files at once.

The upload.sh file automates uploads to the cluster, it uses ‘rsync’ to push the files to the cluster while at the same time using ‘chmod=0700’ to make them executable. Note this requires the environment have a working ssh key created and set with the cluster.

## 