6CS005
Learning Journal
Semester 1
2019/20

David Pearson
1725412

Contents

1 Parallel and Distributed Systems

1.1 Answer of First Question

What are threads and what they are designed to solve?

Threads are typically spawned by processes to carry out a section of code that is suitable for division into chunks. Each thread can process the section of code in a pseudo simultaneous fashion. Another attribute of threads is that they are inherently lightweight in terms of resource, also that they are designed to be spawned, carryout a process, then die. Depending on the program if a thread fails then it will not necessarily terminate the whole program, this is due to threads being created by processes.

1.2 Answer of Second Question

Name and describe two process scheduling policies. Which one is preferable and does the choice of policies have any influence on the behaviour of Java threads?

There are two policies that operating systems use regarding scheduling, pre-emptively and cooperatively. Cooperatively works on the basis that the individual threads have control of when they release control of their resources, run time, memory etc, this has the typical result that the thread runs to completion. Pre-emptively works by allowing the operating system to have a finer degree of control over execution time. For example, the scheduler is able to stop a thread before its completion then return to it at a later time, if at all.

In terms of which is preferable that is very much down to any given number of variables and the particular situation to which the scheduling policy is being applied. However, in the absence of any further detail to allow for a correctly reasoned answer and given the opinion shared in the lecture during week 3, I believe the desired response to be that "pre-emptive is good, co-operative is usually bad".

Initial research has indicted that with regards to Java the JVM works on a pre-emptive basis alongside a higher priority first system with resource allocation. It can therefore be concluded that the choice of policy would have very little influence on the behaviour of Java threads.

1.3 Answer of Third Question

Distinguish between Centralized and Distributed systems?

When attempting to distinguish between a Centralized and Distributed system, the key core difference is that within a Centralized system the data is stored in a single place and all computation takes place there. Whereas, in a Distributed system the computation to be carried out is spread across a number of different nodes/processors. Obviously in the case of a distributed system concurrency then enters the equation, with a need for all distributed elements to maintain integrity. A benefit being that any large problem spread around a distributed system can be completed faster than in a centralized system.

1.4 Answer of Forth Question

Explain transparency in D S?

In a distributed system if a feature is described with the term transparency, quite counter intuitively this means that it is hidden from the user. When a user is using a distributed system, they may be unaware that the data they are accessing is not being stored locally on the actual computer terminal they are at. This is a form of transparency known as access transparency. Another type of transparency is known as Failure transparency this is where if faults or errors occur in the distributed system, this is hidden from them. Ultimately It allows the user to work in ignorance of any possible hardware software errors. Other notable types of transparency are; Location, Concurrency, Replication, and Scaling.

1.5 Answer of Fifth Question

The following three statements contain a flow dependency, an anti-dependency and an output dependency. Can you identify each? Given that you are allowed to reorder the statements, can you find a permutation that produces the same values for the variables C and B as the original given statements? Show how you can reduce the dependencies by combining or rearranging calculations and using temporary variables.

B=A+C

B=C+D

C=B+D

I have, initially, as requested identified the data dependencies, please see appendix A. I have also re-ordered the three given statements into every possible combination, again shown in appendix A. If the statements are run in sequence or independently in parallel, then no basic re-ordering permutation that would give the same value for both the C and B variables, as initially provided

To attempt to address the final element and reduce dependencies, whilst being able to use temporary variables, I have created the following code. As you can see there are now no dependencies at all, and the B and C variables now hold the same values as they initially held.

```
int main() {
        int A,B,C,D;
        int tempC, tempB;

        A = 10;
        B = 4;
        C = 6;
        D = 4;

        tempB = B;
        tempC = C;

        B = A + C;
        B = C + D;
        C = B + D;

        B = tempB;
        C = tempC;
}
```

1.6 Answer of Sixth Question

1.6a The Left-Hand program

```c
#include <pthread.h>
#include <stdio.h>

int counter;

static void * thread_func(void * _tn) {
    int i;
    for (i = 0; i < 100000; i++)
        counter++;
    return NULL;
}

int main() {

int i, N = 5;
pthread_t t[N];
for (i = 0; i < N; i++)
    pthread_create(&t[i], NULL, thread_func, NULL);
for (i = 0; i < N; i++)
    pthread_join(t[i], NULL);
    printf("%d\n", counter);
return 0;
}
```

When running the above, provided, program the following results occurred.

```
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
168389
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
182020
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
172596
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
165019
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
156826
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
166102
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
148165
```

The results shown are due to how the threads are managed within the program. As they are created and joined in two separate and distinct loops. This means that they are effectively being run in parallel. Each subsequent thread being set away before the previous thread has completed. In addition, as there is no method such as Atomic or other such lock on the ‘counter’ variable then multiple dirty read errors are occurring.

1.6b The Right-Hand program

```c
#include <pthread.h>
#include <stdio.h>

int counter;

static void * thread_func(void * _tn) {
    int i;
    for (i = 0; i < 100000; i++)
        counter++;
    return NULL;
}

int main() {

int i, N = 5;
pthread_t t[N];
for (i = 0; i < N; i++) {
    pthread_create(&t[i], NULL,thread_func, NULL);
    pthread_join(t[i], NULL);
    }
printf("%d\n", counter);
return 0;
}
```

When running the above, provided, program the following results occurred.

```
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
500000
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
500000
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
500000
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
500000
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
500000
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
500000
davidp@DESKTOP-KD7RD90:/mnt/c/Users/truea.DESKTOP-KD7RD90$ ./a.out
500000
```

The results shown are due to how the threads are managed within the program. As they are created and joined within a single distinct loop. This means that they are effectively being run sequentially. Each subsequent thread being set away only after the previous thread has completed. The main difference now being that there is no dirty read errors as with the previous program.

2 Applications of Matrix Multiplication and Password Cracking using HPC-based CPU system

Part A: Single Thread Matrix Multiplication

Answer of First Question

The analysis of the algorithm's complexity.

My analysis of the program showed that because there are three nested for loops within the program, the complexity in Big O notation would be $O(n^3)$.

Answer of Second Question

Suggest at least three different ways to speed up the matrix multiplication algorithm given here. (Pay special attention to the utilisation of cache memory to achieve the intended speed up).

1. By reordering the loops, a speed up can be achieved, this is due to how the arrays in C store data in memory and the subsequent transfers to the cache.
2. Remove the initialisation of the C matrix element to 0 between the second and third nested loops, this is one fewer line of code being executed each time the second loop is incremented.
3. Change the calculation method to be += instead of the existing method.

Answer of Third Question

Write your improved algorithms as pseudo-codes using any editor. Also, provide a reasoning as to why you think the suggested algorithm is an improvement over the given algorithm.

Pseudo code for the improved algorithm. Reasoning as to why is provided in the last question. However, the most pertinent element of my reasoning as to why this is an improvement is due to the fact that when run in comparison to the original, it is able to perform in a faster time.

```
int A[N][P], B[P][M], C[N][M];

for (int k = 0; k < N; k++) {
        for (int i = 0; i < M; i++) {
                for (int j = 0; j < P; j++) {
                        C[i][j] += A[i][k] * B[k][j];
                }
        }
}
```

Answer of Forth Question

Write a C program that implements matrix multiplication using both the loop as given above and the improved versions that you have written.

Please see file included in attached zip directory.

Filename: **1725412_Task2_A.c**

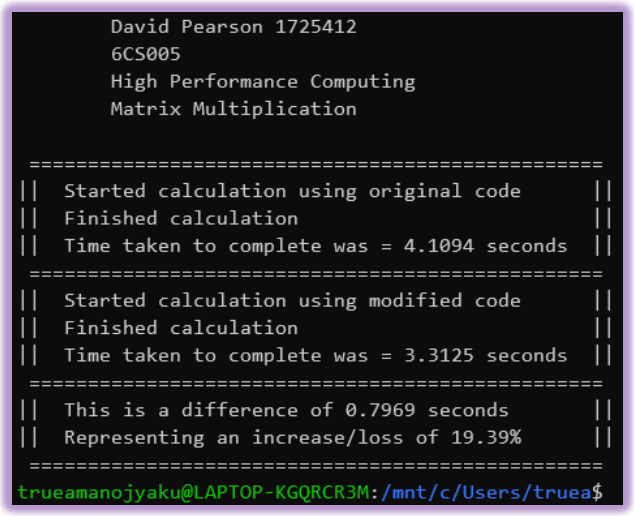
Compile command: **cc 1725412_Task2_A.c -o Task2_A**

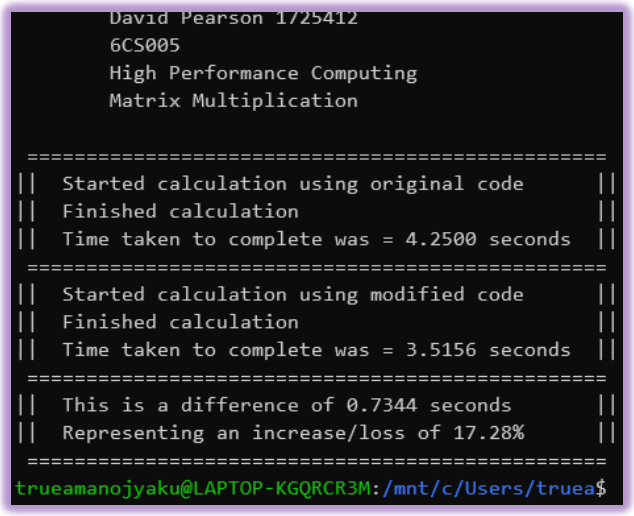
Execute command: **./Task2_A**

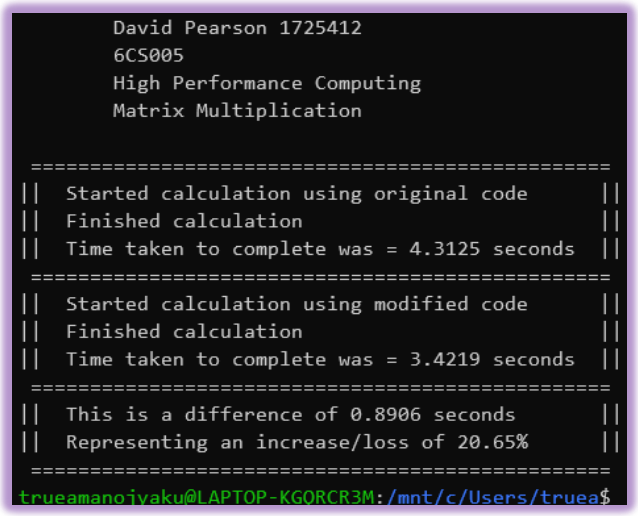
Answer of Fifth Question

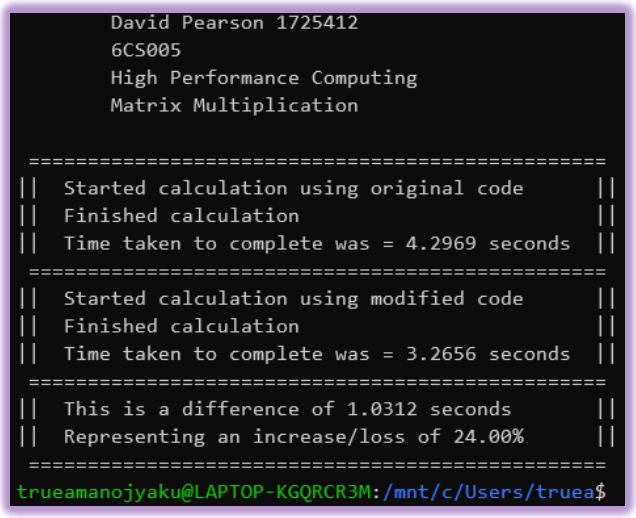
Measure the timing performance of these implemented algorithms. Record your observations. (Remember to use large values of N, M and P – the matrix dimensions when doing this task).

I have tested the two algorithms using matrix sizes of A[700]x[700] and B[700]x[700]. The program first populates the matrices using the rand() function with integers between 1 and 100. It then carries on to first calculate using the given algorithm, recording the time taken and displaying this in the console. Next it carries out the same matrix multiplication using the improved algorithm, again recording the time and displaying this in the console. Finally, the program calculates any time difference and reports this to the user in both seconds and as a percentile change. Please see below four separate examples of the output.









Part B: Write a code to implement matrix multiplication using multithreading

Answer of First Question

Write a C program to implement matrix multiplication using multithreading. The number of threads should be configurable at run time, for example, read via an external file.

Please see file included in attached zip directory.

Filename: **1725412_Task2_B.c**

Compile command: **cc 1725412_Task2_B.c -pthread -o Task2_B**

Execute command: **./Task2_B**

Please note that the program will ask you to specify how many threads at runtime. After you have chosen, the program will run a total of five times, showing you the time taken for each run as well as the average time across all runs.

Answer of Second Question

The code should print the time it takes to do the matrix multiplication using the given number of threads by averaging it over multiple runs.

When creating the program, it has been made so that the matrix multiplication takes place a total of five times, each time reporting the time taken to the user, followed by an average time.

```
David Pearson 1725412
6CS005
High Performance Computing
Matrix Multiplication 2B

Enter number of threads = 2
Time taken to complete with 2 Threads was = 36.1875 seconds
Time taken to complete with 2 Threads was = 34.5156 seconds
Time taken to complete with 2 Threads was = 33.2969 seconds
Time taken to complete with 2 Threads was = 36.7031 seconds
Time taken to complete with 2 Threads was = 35.9062 seconds
Average time taken to complete with 2 Threads was = 35.3219 seconds
```

Answer of Third Question

Plot the time it takes to complete the matrix multiplication against the number of threads and identify a sweet spot in terms of the optimal number of threads needed to do the matrix multiplication.

When calculating the 'sweet spot' I used matrices of [2000] x [2000] in size, populated with numbers between 1-100. Starting with 1 thread and incrementing to a total of 6 threads each calculation was carried out five times with the duration recorded. For additional data, I completed this on two separate computer systems.

Running on a 6 core i7-6800 @ 3.40ghz

	Time Taken in seconds					
Thread count	Run 1	Run 2	Run 3	Run 4	Run 5	Average
1	27.5	27.4531	28.2031	27.5312	27.6562	27.6687
2	17.7188	18.0625	18.0469	17.9062	17.8906	17.925
3	17.0312	17.0781	16.9219	16.9688	17.0469	17.0094
4	17.3125	17.1406	17.2188	17.4688	17.4062	17.3094
5	17.8125	18.0312	18.7188	19.0318	17.75	18.2688
6	20.1097	19.4844	20.25	19.6562	19.4062	19.7812

Running on a 2 core i3-6006h @ 2.00ghz

	Time Taken in seconds					
Thread count	Run 1	Run 2	Run 3	Run 4	Run 5	Average
1	51.9844	50.8438	50	50.1875	50.3906	50.6812
2	26.2344	37.25	39.5938	35.875	39.2188	37.6344
3	43.75	42.8125	42.9219	41.125	40.75	42.2719
4	51.2656	51.3125	51.3594	51.125	51.3125	51.275
5	50.9062	51.2188	50.9062	51.1094	50.9375	51.0156
6	51.125	51.0156	51	51.0938	47.4688	50.3406

Part C: Password cracking using POSIX Threads

Answer of First Question

Run the program 10 times and calculate the mean running time.

I added a timing element to the program and placed it into a loop to repeat the required ten times, each time returning the time taken to crack the password. Finally, the program then outputs the mean running time.

```
trueamanojyaku@LAPTOP-KGQRCR3M:/mnt/c/Users/truea$ ./CrackAZ99
#1813    AS12 $6$AS$RpBqfxKaD5wax.zvXKR/mSE0xfLqIawh1hJe6wqsv4yAvijb9.5Pzi.3He9RXuSR8F6D6brdsSuTANEkbvQ7J/
        Time taken to complete run 1 was = 7.48 seconds
#1813    AS12 $6$AS$RpBqfxKaD5wax.zvXKR/mSE0xfLqIawh1hJe6wqsv4yAvijb9.5Pzi.3He9RXuSR8F6D6brdsSuTANEkbvQ7J/
        Time taken to complete run 2 was = 8.38 seconds
#1813    AS12 $6$AS$RpBqfxKaD5wax.zvXKR/mSE0xfLqIawh1hJe6wqsv4yAvijb9.5Pzi.3He9RXuSR8F6D6brdsSuTANEkbvQ7J/
        Time taken to complete run 3 was = 8.27 seconds
#1813    AS12 $6$AS$RpBqfxKaD5wax.zvXKR/mSE0xfLqIawh1hJe6wqsv4yAvijb9.5Pzi.3He9RXuSR8F6D6brdsSuTANEkbvQ7J/
        Time taken to complete run 4 was = 7.57 seconds
#1813    AS12 $6$AS$RpBqfxKaD5wax.zvXKR/mSE0xfLqIawh1hJe6wqsv4yAvijb9.5Pzi.3He9RXuSR8F6D6brdsSuTANEkbvQ7J/
        Time taken to complete run 5 was = 7.46 seconds
#1813    AS12 $6$AS$RpBqfxKaD5wax.zvXKR/mSE0xfLqIawh1hJe6wqsv4yAvijb9.5Pzi.3He9RXuSR8F6D6brdsSuTANEkbvQ7J/
        Time taken to complete run 6 was = 7.20 seconds
#1813    AS12 $6$AS$RpBqfxKaD5wax.zvXKR/mSE0xfLqIawh1hJe6wqsv4yAvijb9.5Pzi.3He9RXuSR8F6D6brdsSuTANEkbvQ7J/
        Time taken to complete run 7 was = 7.06 seconds
#1813    AS12 $6$AS$RpBqfxKaD5wax.zvXKR/mSE0xfLqIawh1hJe6wqsv4yAvijb9.5Pzi.3He9RXuSR8F6D6brdsSuTANEkbvQ7J/
        Time taken to complete run 8 was = 7.18 seconds
#1813    AS12 $6$AS$RpBqfxKaD5wax.zvXKR/mSE0xfLqIawh1hJe6wqsv4yAvijb9.5Pzi.3He9RXuSR8F6D6brdsSuTANEkbvQ7J/
        Time taken to complete run 9 was = 7.36 seconds
#1813    AS12 $6$AS$RpBqfxKaD5wax.zvXKR/mSE0xfLqIawh1hJe6wqsv4yAvijb9.5Pzi.3He9RXuSR8F6D6brdsSuTANEkbvQ7J/
        Time taken to complete run 10 was = 7.56 seconds
        The average time taken to complete was = 7.55 seconds
trueamanojyaku@LAPTOP-KGQRCR3M:/mnt/c/Users/truea$
```

Answer of Second Question

In your learning journal make an estimate of how long it would take to run on the same computer if the number of initials were increased to 3. Include your working in your answer.

Given the results obtained in question 1, the program was able to process roughly 240 password permutations per second, whilst running on a 2 core i3-6006h @ 2.00ghz processor.

When the number of initials in the password are increased to three, this has the direct effect of increasing the number of possible combinations to 1,757,600.

If you then extrapolate this out at 240 permutations per second, this gives a total possible run time of 7,323.3 seconds or 122.05 minutes, for a single run.

Answer of Third Question

Modify the program to crack the three-initials-two-digits password given in the three_initials variable. An example password is HPC19.

Please see file included in attached zip directory.

I have amended the given crack program to now crack a password that contains three uppercase initials. Rather than use the given password of HPC19 I have used ABC12. The reason for this is that HPC19 represents the 512420th possible permutation, my given hardware would take over 30 minutes to reach that point. Alternatively, ABC12 is the 2813th possible permutation, allowing for faster testing.

Filename: **1725412_Task2_C_3.c**

Compile command: **cc 1725412_Task2_C_3.c -o 1725412_Task2_C_3 -lcrypt**

Execute command: **./1725412_Task2_C_3**

```
trueamanojyaku@LAPTOP-KGQRCR3M:/mnt/c/Users/truea$ ./1725412_Task2_C_3
#2813    ABC12 $6$AS$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6.
        Time taken to complete run 1 was = 11.05 seconds
#2813    ABC12 $6$AS$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6.
        Time taken to complete run 2 was = 11.27 seconds
#2813    ABC12 $6$AS$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6.
        Time taken to complete run 3 was = 10.95 seconds
#2813    ABC12 $6$AS$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6.
        Time taken to complete run 4 was = 11.15 seconds
#2813    ABC12 $6$AS$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6.
        Time taken to complete run 5 was = 11.21 seconds
#2813    ABC12 $6$AS$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6.
        Time taken to complete run 6 was = 11.36 seconds
#2813    ABC12 $6$AS$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6.
        Time taken to complete run 7 was = 11.31 seconds
#2813    ABC12 $6$AS$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6.
        Time taken to complete run 8 was = 11.48 seconds
#2813    ABC12 $6$AS$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6.
        Time taken to complete run 9 was = 11.73 seconds
#2813    ABC12 $6$AS$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6.
        Time taken to complete run 10 was = 11.41 seconds
        The average time taken to complete was = 11.29 seconds
trueamanojyaku@LAPTOP-KGQRCR3M:/mnt/c/Users/truea$
```

Answer of Forth Question

Write a short paragraph to compare the running time of your three_initials program with your earlier estimate. If your estimate was wrong explain, why you think that is.

The average running time is slightly better than the previous two initial program, a potential 4% speed increase from 240 permutations per second to 250 permutations per second. This could be down to background processes also being run on the system. Furthermore, dedicated testing would be required to enable a firmer conclusion to be drawn.

Answer of Fifth Question

Modify the original version of the program to run on 2 threads. It does not need to do anything fancy, just follow the following algorithm.

I have created a multi-threaded password decryption program that works with '**n'** number of threads dynamically. In addition, the same program allows for decryption of passwords of differing length. It achieves this by using a dictionary file, passed in on the command line. I have provided the dictionary files to be used with this program. They are;

Filename: **1725412_Task2_C_5.c**

Compile command: **cc 1725412_Task2_C_5.c -pthread -lcrypt -o Task2_C_5**

Execute command: **./Task2_C_5 'hash to be cracked' 'dictionary file'**

Example execute command:

./Task2_C_5 \$6\$AS\$.Lb98VbQhR0UpKS47MHfBeBfaUv2xOVdxq5melSrsQmrwf.zwvsHJXQrWnyO62J.fsiKymRLZ9vOXs2nwl4L6. AAA00.txt

Answer of the Sixth Question

Compare the results of the mean running time of the original program with the mean running time of the multithread version.

The multi-threaded version of the cracking program performs faster than the single threaded version.

3 Applications of Password Cracking and Image Blurring using HPC-based CUDA system

Part A: Password cracking using CUDA

Answer of First Question
Crack passwords using CUDA

Filename: **1725412_Task3_A.cu**

Compile command: **nvcc 1725412_Task3_A.cu -o Task3_A**

Execute command: **./Task3_A 'hash to be cracked' 'dictionary file'**

Example execute command:

./Task3_A cxbdwy2744 aa00.txt

The code provided does not fully work. There is an error in copying over the created dictionary file from host to device.

Answer of Second Question
Comparison, analysis and evaluation of CUDA version for password cracking (compare with normal and "pthread" version)

If the program had been able to be fully working then the assumption would be that it would outperform the multithreading version by a large degree.

Part B: Image blur using multi dimension gaussian matrices (multi-pixel processing)

Answer of First Question
Applying gaussian blur with 3x3 matrix using CUDA

Filename: **1725412_Task3_B.cu**

Compile command: **nvcc 1725412_Task3_B.cu lodepng.cpp -o Task3_B**

Execute command: **./Task3_B**

The image filename to have the blur applied to is hard coded into the file and can be changed in there.

Answer of Second Question

Applying gaussian blur with multiple dimension matrices using CUDA

Not attempted

Appendix A