# 6CS005 Learning Journal - Semester 1 2019/20

Akash Chanara, 2040368

## Contents

# 1 Parallel and Distributed Systems

## 1.1 Answer of First Question

A thread is part of the process, operating within its own execution space, and there can be several threads in one process. OS can do multiple tasks in parallel with the help of it (depending upon the number of processors the machine consists). Threads enable the CPU to perform many tasks at the same time in one process.

## 1.1 Answer of Second Question

Two process scheduling policies are:

Pre-emptive: This scheduler monitors how long a process will run. The scheduler will stop the process if a process goes beyond its time slice.

Co-operative: Each process is responsible for how long it lasts. When a process feels like cooperation, execution will be abandoned.

Pre-emptive is preferred; the thread scheduling algorithm of the Java runtime system is also pre-emptive.

## 1.2 Answer of Third Question

All calculations are done on one computer in a centralized scheme (system). The calculation is distributed to multiple computers within the distributed system. For example: If there is a large amount of information, it can be split and sent to each computer component to perform it.

## 1.3 Answer of Fourth Question

Transparency in the distributed system means that, by concealing distribution from the user and application programmer, one distributed system looks like a single computer.

## 1.4 Answer of Fifth Question

Include your code using a text file in the submitted zipped file under name Task1.5

B=A+C is a flow dependency

C=B+D is an anti-dependency

B=C+D is an output dependency

## 1.5 Answer of Sixth Question

Program 1 answer: 349151

Program 2 answer: 500000

Program 1 and 2 both use thread function to perform thread count for the given unassigned integer [N]. Program 1 tuns an extra loop.

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

### 2.1 Single Thread Matrix Multiplication

- The analysis of the algorithm’s complexity. (1 mark)

  The complexity of above program is O($n^3$).

- 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 marks)

  Divide and Conquer could also speed the multiplication process but the better option would be Strassen’s matrix multiplication.

- Write your improved algorithms as pseudo-codes using any editor. Also, provide reasoning as to why you think the suggested algorithm is an improvement over the given algorithm. (1 marks)

```
begin
  If n = threshold then compute
        C = a * b is a conventional matrix.
  Else
        Partition a into four sub matrices a11, a12, a21, a22.
        Partition b into four sub matrices b11, b12, b21, b22.
        Strassen (n/2, a11 + a22, b11 + b22, d1)
        Strassen (n/2, a21 + a22, b11, d2)
        Strassen (n/2, a11, b12 – b22, d3)
        Strassen (n/2, a22, b21 – b11, d4)
        Strassen (n/2, a11 + a12, b22, d5)
        Strassen (n/2, a21 – a11, b11 + b22, d6)
        Strassen (n/2, a12 – a22, b21 + b22, d7)

        C = d1+d4-d5+d7    d3+d5
        d2+d4          d1+d3-d2-d6

  end if
  return (C)
end.
```

Since the above program does 8 multiplications for matrices of size N/2 * N/2 and 4 additions, Strassen's multiplication performs it in 7 multiplications, so it is faster.

- Write a C program that implements matrix multiplication using both the loop as given above and the improved versions that you have written. (1marks)

Include your code using a text file in the submitted zipped file under name Task2.1

- 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). (1 marks)

  The improved program performs the matrix multiplication in 10.063 seconds.

  Insert a paragraph that hypothesises how long it would take to run the original and improved algorithms. Include your calculations.

  Explain your results of running time.

  The time complexity of original program is O($n^3$) and the improved program is O($n^{2.80}$). So, the time taken by the improved version is less than the original version.

## 2.2 Multithreaded Matrix Multiplication

- Include your code using a text file in the submitted zipped file under name Task2.2

- Insert a table that has columns containing running times for the original program and your multithread version. Mean running times should be included at the bottom of the columns.
- Insert an explanation of the results presented in the above table.

| Number of programs runs | Original Program | Multithread version |
|---|---|---|
| 1 | 10.063 | 3.0 |
| 2 | 10.400 | 3.0 |
| 3 | 10.381 | 3.0 |
| 4 | 10.411 | 3.0 |
| 5 | 10.298 | 3.0 |
| 6 | 10.402 | 3.0 |
| 7 | 10.366 | 3.0 |
| 8 | 10.252 | 3.0 |
| 9 | 10.498 | 3.0 |
| 10 | 10.512 | 3.0 |
| Average (seconds) | 10.3583 | 3.0 |

The original program was run on CodeBlocks whereas multithread version was run on Ubuntu terminal. Multithread version is obviously very fast compared to the original program because it breaks down the program and each thread perform each multiplication of the matrix.

## 2.3 Password cracking using POSIX Threads

- Include your code using a text file in the submitted zipped file under name Task2.3.1, Task2.3.3, Task2.3.5
- Insert a table of 10 running times and the mean running time.
- Insert a paragraph that hypothesises how long it would take to run if the number of initials were to be increased to 3. Include your calculations.
- Explain your results of running your 3 initial password cracker with relation to your earlier hypothesis.
- Write a paragraph that compares the original results with those of your multithread password cracker.

| Number of Program runs | Time in nano seconds | Time in seconds |
|---|---|---|
| 1 | 126338942492.00 | 126.338942492 |
| 2 | 126653558762.00 | 126.653558762 |
| 3 | 126622886091.00 | 126.622886091 |
| 4 | 126230990570.00 | 126.230990570 |
| 5 | 126533256007.00 | 126.533256007 |
| 6 | 126301290324.00 | 126.301290324 |
| 7 | 127114770627.00 | 127.114770627 |
| 8 | 126889819803.00 | 126.889819803 |
| 9 | 126142905876.00 | 126.142905876 |
| 10 | 126315538400.00 | 126.315538400 |
| Average | 126514395895.20 | 126.514395895 |

Since the above program cracks two initials, three initials can be cracked using the same code but adding just one loop for alphabets. Since alphabets consists of 26 characters and the loop goes through those 26 characters, the estimated time can be:

Estimated Time = Original time * 26

= 126.514395895 * 26

= 3,289.37429327seconds

Converting to minutes = 3,289.37429327/ 60

= 54.82290488783333 minutes

Therefore, estimated time is 55 minutes.

| Number of Program runs | Time in nano seconds | Time in seconds |
|---|---|---|
| 1 | 3288222652622.00 | 3288.222652622 |
| 2 | 3280521542264.00 | 3280.521542264 |
| 3 | 3286626232322.00 | 3286.626232322 |
| 4 | 3282346742455.00 | 3282.346742455 |
| 5 | 3281556841522.00 | 3281.556841522 |
| 6 | 3281251521515.00 | 3281.251521515 |
| 7 | 3283562652352.00 | 3283.562652352 |
| 8 | 3288426284122.00 | 3288.426284122 |
| 9 | 3285586246265.00 | 3285.586246265 |
| 10 | 3289265662662.00 | 3289.265662662 |
| Average | 3284736637810.10 | 3284.736637810 |

The actual time is:  3284.736637810 seconds which converted is 54.74561063016667minutes. The estimated time was 54.82290488783333 which is which is a bit more than the estimated time.

This could be due to the background processes running while the two-initial program was running.

| Number of times program ran | Time taken by original program | Time taken by multithread version |
|---|---|---|
| 1 | 126.338942492 | 63.07447578 |
| 2 | 126.653558762 | 63.02594458 |
| 3 | 126.622886091 | 63.21775411 |
| 4 | 126.230990570 | 63.04287490 |
| 5 | 126.533256007 | 63.07007623 |
| 6 | 126.301290324 | 63.04756700 |
| 7 | 127.114770627 | 63.19036198 |
| 8 | 126.889819803 | 63.40375928 |
| 9 | 126.142905876 | 66.16176498 |
| 10 | 126.315538400 | 69.06917650 |
| Average (seconds) | 126.514395895 | 64.03037553 |

Here the single thread version took 126.514395895 seconds while the multithread version took 64.03037553seconds. This is because in multithread, there are two threads while the other has one thread. That is why the multithread is faster than the single thread version.

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

### 3.1 Password Cracking using CUDA

- Include your code using a text file in the submitted zipped file under name Task3.1
- Insert a table that shows running times for the original and CUDA versions.
- Write a short analysis of the results

| Number of times program ran | Time taken by Original program | Time taken by Multithread version | Time taken by CUDA version |
|---|---|---|---|
| 1 | 126.338942492 | 63.07447578 | 0.580939718 |
| 2 | 126.653558762 | 63.02594458 | 0.595920426 |
| 3 | 126.622886091 | 63.21775411 | 0.583720913 |
| 4 | 126.230990570 | 63.04287490 | 0.583794837 |
| 5 | 126.533256007 | 63.07007623 | 0.582852925 |
| 6 | 126.301290324 | 63.04756700 | 0.580954998 |
| 7 | 127.114770627 | 63.19036198 | 0.572143096 |
| 8 | 126.889819803 | 63.40375928 | 0.581389062 |
| 9 | 126.142905876 | 66.16176498 | 0.578276899 |
| 10 | 126.315538400 | 69.06917650 | 0.575746898 |
| Average (seconds) | 126.514395895 | 64.03037553 | 0.581573977 |

As we can see, CUDA version of password cracking is very faster than the original and multithread version. The difference between those is that CUDA runs on GPU and POSIX runs on CPU. Since GPU has many CUDA cores, it will be faster than the original program. CUDA is also a parallel computing platform and can process huge number of data parallelly.

## 3.2 Image blur using multi dimension Gaussian matrices

- Include your code using a text file in the submitted zipped file under name Task3.2
- Insert a table that shows running times for the original and CUDA versions.
- Write a short analysis of the results

| Number of Program runs | Original Program | CUDA version |
|---|---|---|
| 1 | 0.061681437 | 0.118552626 |
| 2 | 0.039438951 | 0.092801256 |
| 3 | 0.062531324 | 0.09549946 |
| 4 | 0.039430455 | 0.010654483 |
| 5 | 0.039219192 | 0.093559856 |
| 6 | 0.039161119 | 0.096120374 |
| 7 | 0.039048638 | 0.097841633 |
| 8 | 0.039421445 | 0.09442725 |
| 9 | 0.038957069 | 0.94293611 |

| | | |
|---|---|---|
| 10 | 0.039210008 | 0.093710725 |
| Average | 0.043809964 | 0.173610377 |

We can see from the table above that the original program running on the CPU is faster than the CUDA version. This might be because the CPU is more powerful than the machine's CPU.