6CS005 Learning Journal - Semester 1 2019/20

Miraj Pudasaini 1928185

# Table of Contents

[Table of Contents 1](#_Toc527650969)

[1 POSIX Threads 2](#_Toc527650970)

[1.1 Password Cracking 2](#_Toc527650971)

[1.2 Image Processing 2](#_Toc527650972)

[1.3 Linear Regression 2](#_Toc527650973)

[2 CUDA 3](#_Toc527650974)

[2.1 Password Cracking 3](#_Toc527650975)

[2.2 Image Processing 3](#_Toc527650976)

[2.3 Linear Regression 3](#_Toc527650977)

[3 MPI 4](#_Toc527650978)

[3.1 Password Cracking 4](#_Toc527650979)

[3.2 Image Processing 4](#_Toc527650980)

[3.3 Linear Regression 4](#_Toc527650981)

[4 Verbose Repository Log 5](#_Toc527650982)

# POSIX Threads

## Password Cracking

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.

Paste your source code for a 3 initial password cracker here. The code should be neatly indented and lines should not wrap

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.

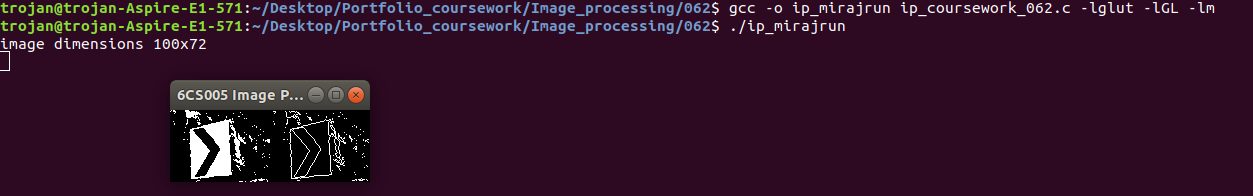
## Image Processing

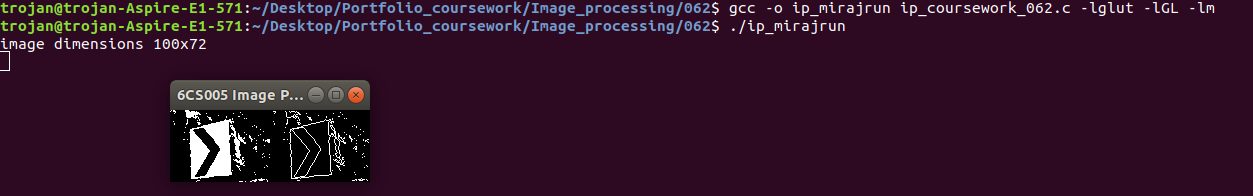
a)



This is the 100\*72 image provided in the 062 course work.

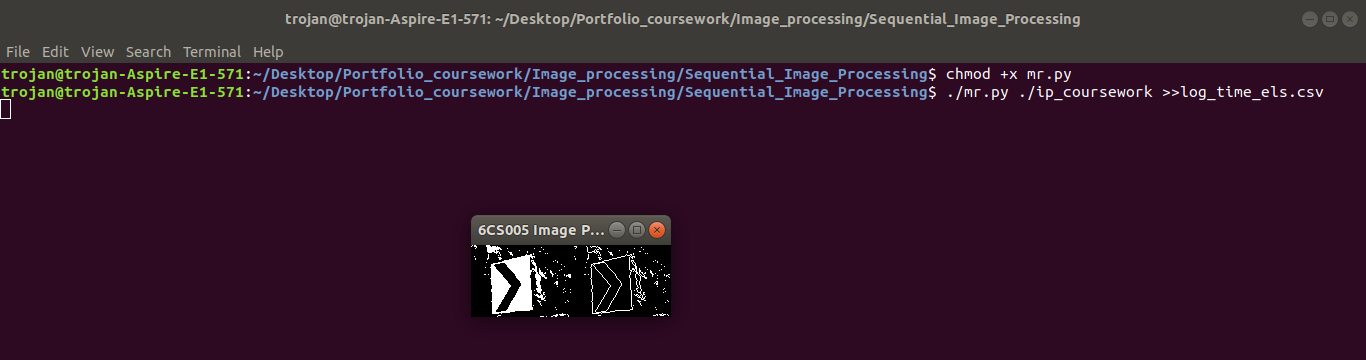
On the first simple manual run of the program provided on a linux bash terminal following output was observed



  
Illustration 1: Simple run of the provided program

By running the **sequential version** of the pogram for 10 times with a python script the mean running time was calculated.

For calculation of mean running time weighted measure performance measuring technique was followed which furhter more detail can be found here.([)](http://ece-research.unm.edu/jimp/611/slides/chap1_2.html)

  
Illustration 2: Sequential Program 10 runs

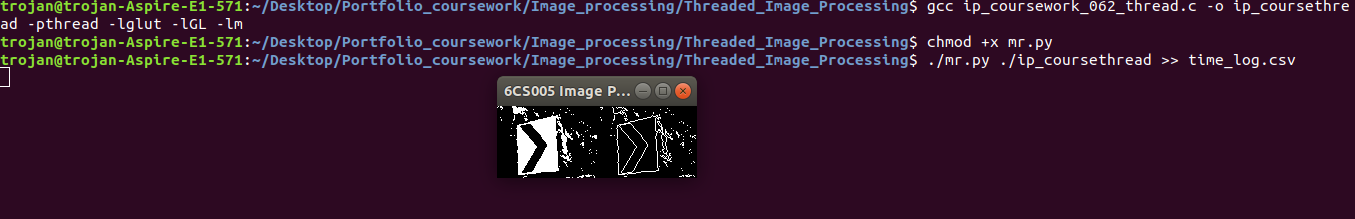
The mean running time was calculated from the *log\_time\_els.csv* file:

|  |  |
| --- | --- |
| **Iteration** | **Time Elapsed** |
| First Iteration | 128052ns/0.000128052s |
| Second Iteration | 129700ns/0.000129700s |
| Thrid Iteration | 140614ns/0.000140614s |
| Fourth Iteration | 195262ns/0.000195262s |
| Fifth Iteration | 124384ns/0.000124384s |
| Sixth Iteration | 124384ns/0.000124384s |
| Seventh Iteration | 86716ns/0.000086716s |
| Eight Iteration | 209098ns/0.000209098s |
| Ninth Iteration | 126886ns/0.000126886s |
| Final Iteration | 116201ns/0.000116201s |
| **Mean Running Time** | 0.000138324s |

b)

After running the previous version the sequential version of the program was modified to sub divide the process batch job for parallel processing and hence 8 equal **POSIX Thread** executed the sequential program, with the help of the *‘mr.py’* script this version was also ran for 10 times:

After analysing the results from the **‘***time\_log.csv****’*** *a mean running time was analysed as follow:*

  
Illustration 3: Threaded Version 10 run

|  |  |
| --- | --- |
| **Iteration** | **Time Elapsed** |
| First Iteration | 711637ns or 0.000711637s |
| Second Iteration | 2174209ns or 0.002174209s |
| Thrid Iteration | 677832ns or 0.000677832s |
| Fourth Iteration | 559941ns or 0.000559941s |
| Fifth Iteration | 559941ns or 0.000559941s |
| Sixth Iteration | 748271ns or 0.000748271s |
| Seventh Iteration | 894318ns or 0.000894318s |
| Eight Iteration | 510498ns or 0.000510498s |
| Ninth Iteration | 748473ns or 0.000748473s |
| Final Iteration | 517468ns or 0.000517468s |
| **Mean Running Time** | 754264.7ns or 0.000754265s |

*c)*

On observation,

M.R.T(mean running time) for the Sequential Version of the program MRTs= 0.000138324s

M.R.T(mean runnig time) for the Multi Threaded Parallel processed program MRTp=0.000754265s

Since the MRTp > MRTs

which means that on a batch execution with no pseudo parallelism (I.e on a single CPU) the execution time for the Multi Threaded program which used four thread execution took longer to execute than the normal one hence the use of parallelism seems to be unecessary for the execution time comparision. Instead of this, **Forking** the batch job(main process) into smaller processes or Using **Semaphores** might have helped on improving the optimization . *The reason for the threads to not reduce the execution time is estimated to be the small data* **throughput** *that was fetched*. It can be explained as if the time for creation and the exectution of the threads were larger than the exection of the edge detection algorithm .

Paste your source code for your multithread password cracker here

On a process level with a normal clocking speed CPU (3.5GHz) ignoring the *Wallclock time and Latency* of the CPU:

The process time for batch job execution of the threads alone can be calculated as

MRTt(Mean Run Time for Threads) = MRTp – MRTs= 0.000138324-0.000754265=0.000615941s

Here we can see that MRTt is highly greater than MRTs which means that creating and executing the threads were more costly than executing the edge detection sequentially hence it is discovered that for small through put use of multi threading could result in more costly executiong time hence proven more time consuming.

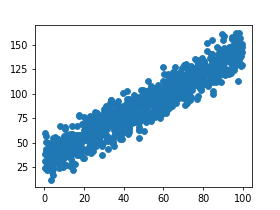
From the first table Sequential run for 10 time, if the data were to be plotted with X ordinate as ith iteration and Y absisa as the execution time it seems to follow a **sinosidual curve(sine curve)** (I.e the execution time of first few iteration were decreasing and suddenly from fourth iteration the execution time increased and then again decreased for two iterations reachead at a peak and finally on the final execution decreased to where it started which if plotted as a graph would result into a wave like structure).

This can be described as if the CPU on first iteration of execution had no blocked or ready execution on the batch job index hence with max clocking executed the program but following the piling up of the other iteration the clock speed might have been distributed and it took a bit longer to execute the algorithm following this cycle the formation of sinosidual curve.

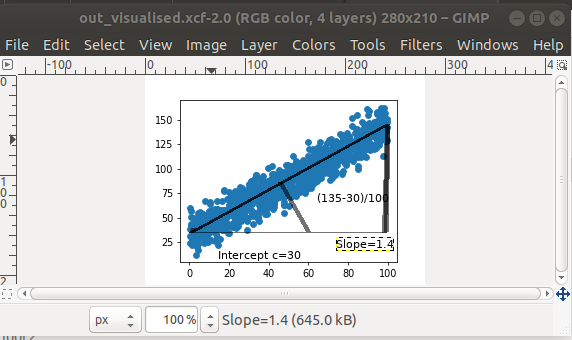
From the second table User level prallelism with POSIX’s 10 iteration, if the data were plotted in the graph it would follow a **Cosine(Co-sinosidual) curve (cose curve)** which means that the execution time were higher and then lower and were fluctuating heavily following the cosine curve cycle. Which is quite unusual and opposite comparing it to the sequential version.   
 This could be described as the CPU architecture for handeling the workload latency and creation and execution of kernel level thread space and thread memory initilization.

## Linear Regression

a) The data provided from the university was fetched with the source code of lr00.c and the absicass and y intercept were plotted with the help of numpy and matlibplot (python libraries) the source code of which is included on the py.py python file included on the zip protfolio.

  
Illustration 4: Scatter Plot of the given ordinate and y intersection

b)

  
Illustration 5: Slope and y intercept visualization

The above scattered plot visualized and from the figure above we can observe that   
 y-intercept is about 30 units and the slope of the visualization is about 1.4 units.

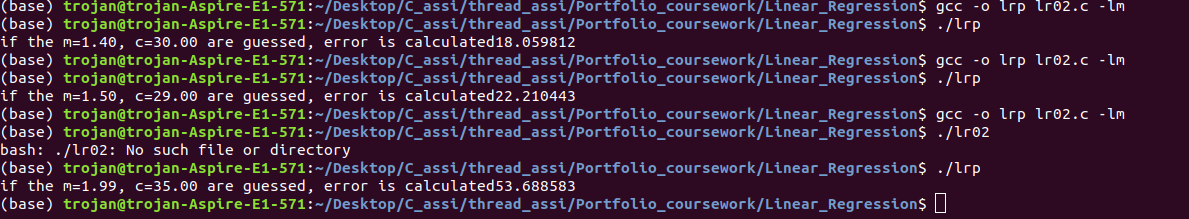
Here for optimium guess of m and c three times are as follows

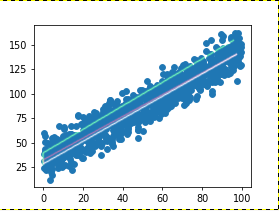
**M1 = 1.3 and C = 30**

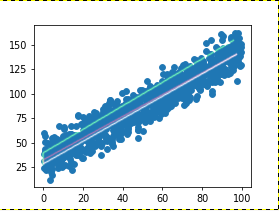
**M2=1.5 and C=29**

**M3=1.99 and C=35**

For another iteration the dataset provided was ran with the source code for finding the rms mean slope and c the hard coded values for m and C were replace and ran for three times for the three guesses individually. The output observed are as follows:

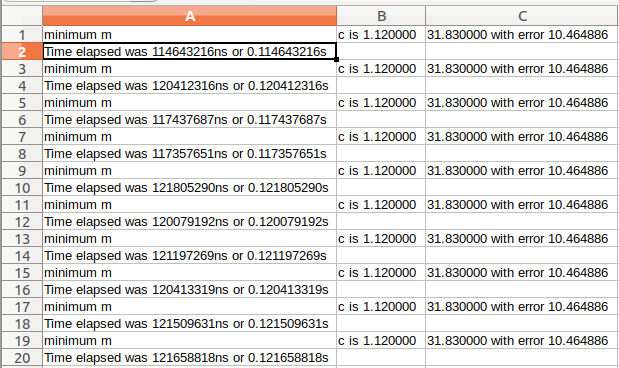
  
Illustration 6: Slope C and RMS for consiquitive guess

Illustration 7: A graph overlay of the guesses

  
Illustration 8: Observation of regression lines of M and C with the result

Here on the illustration of the line formed

as seen over the scatter plot the first guess gave the **red regression line** the second guess gave the **white regression line** and the sky blue **regression line** was as plotted by the third guess. As it can be seen that the ***rms error*** for the first run was 18.059812 the regression line was less disperresed the same was the case for the second regression line. However for the third guess the **rms error** was 53.688583 which led the regression line to be more inclined towards the outbound of the scatter plot. The scatter plot here shows a **higher corelation of linearity (high linear corelation).**



Here is table for ten times run of the sequential program along with the M and C.   
Mean running time for the program as calculated is:

MRTlrs= formula here.

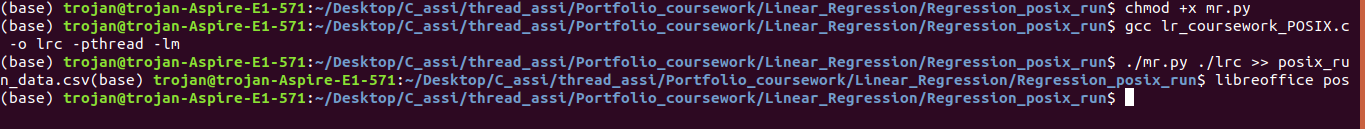
=0.119651439 second is the mean run time   
since mean runtime needs to be calculated as rate the harmonic mean run time would be:

MRTlrs Harmonic= n / ∑ [1/Xi]

hence,

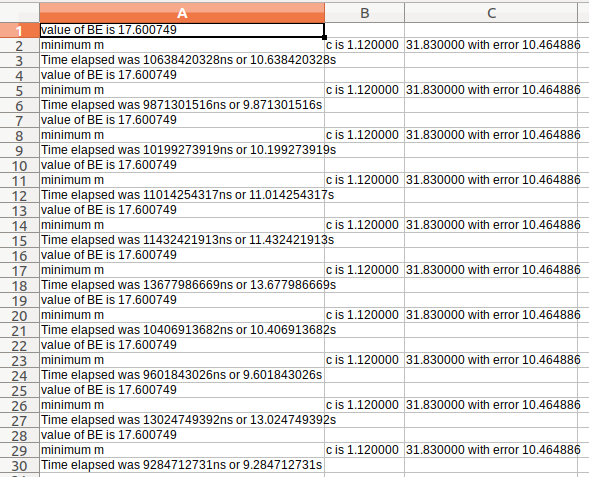
del(execution time)= 0.119607871 per program.

Running the posix Version 10 times.



Write a short analysis of the results.

Compiling and running the cuda version 10 times.



# CUDA

## Password Cracking

Paste your source code for your CUDA based password cracker here

Insert a table that shows running times for the original and CUDA versions.

Write a short analysis of the results

## Image Processing

Insert a table that shows running times for the original and CUDA versions.

Paste your source code for your CUDA based image processing.

Write a short analysis of the results

## Linear Regression

Insert a table that shows running times for the original and CUDA versions.

Write a short analysis of the results

# MPI

## Password Cracking

Insert a table that shows running times for the original and MPI versions.

Paste your source code for your MPI based password cracker here

Write a short analysis of the results

## Image Processing

Paste your source code for your MPI based image processor

Insert a table that shows running times for the original and MPI versions.

Write a short analysis of the results

## Linear Regression

Paste your source code for your MPI based linear regression

Paste your source code for your multithread linear regression program here.

Insert a table that shows running times for the original and MPI versions.

Write a short analysis of the results

# Verbose Repository Log

Paste your verbose format repository log here. With subversion this can be achieved by the following:

svn update

svn –v log > log.txt

gedit log.txt

Then select, copy and paste the text here