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

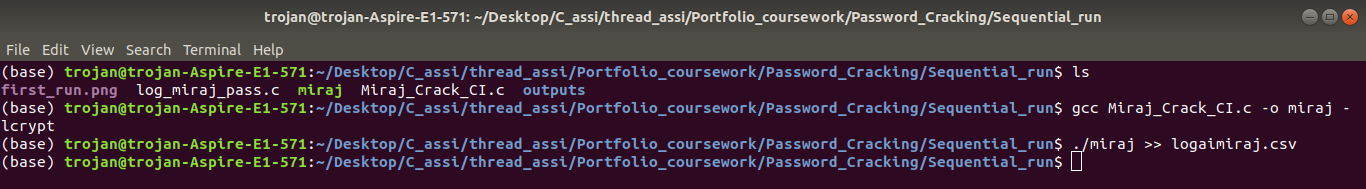


Figure Running password cracking algorithm

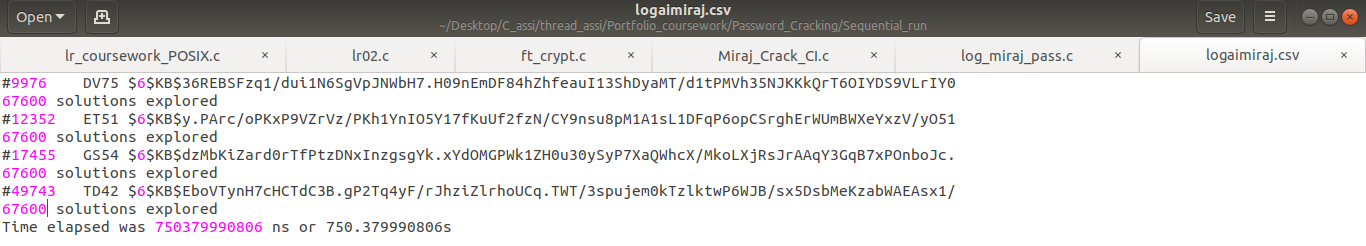
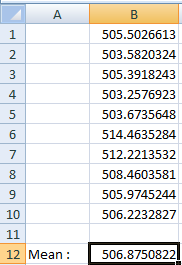


Figure Cracked passwords

**Mean running time for the sequential program.**

The program was run ten times manually and the mean running time for was calculated as follows:



The mean execution time of the algorithm to brute force the passwords of two initials and two digits came around 8.4 mins in calculations  
as there were four passwords in one single algorithm the mean execution time per one password could be estimated around 2.1 mins per password.

**Execution time as the variables like length of password increases (A Short Estimation).**

As seen on above exploration of the algorithm the mathematical expressions (using **principles of permutation and combination)** for cracking the two initial two numerical passwords is as illustrated on table one:

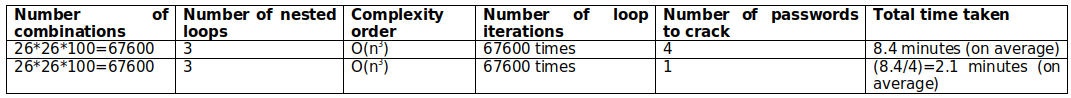
****

Figure Estimation Table one

On adding one letter to the combination the following increase of tested combination goes through the algorithm. Accordingly the **time complexity** of the algorithm increases by exponent of order of one which in compare to that of two initial.

Since,

N3 clutter of loopresulted on 2.1 minutes of execution time per password (for two initials)  
 N4 clutter of loop for three initial shall result on

Here for the increased number of alphabet result in combination nA = 67600\*26=1757600 combinations check

Execution of 1757600 iteration= 218.4minutes (for four password) = 218.4/4 = 54.6minutes for one single password

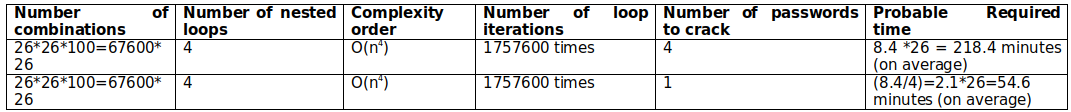


Figure Estimation of table two

Hence an estimation of **218.4 mins** is approximated for the all of four three initial password to get cracked provided with the same resources for processing speed and without adding any optimization techniques on the source code is made for answering this section.

**Note:**The mathematical explanation for the algorithm’s execution time are calculations neglecting the constants like syntax and variable memory initialization so in case of loosely handling memory management in C the execution time may vary depending upon the constants.

**Source code for three initial:**

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <crypt.h>  #include <sys/stat.h>  #include <time.h>  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  this program will decrypt passwords which is combination of 3 initials and 2 digits.  an extra outer for loop has been added to iterate entire loop yet another 26 times. the loop control variable 'w'  has been used to designate the leftmost initials of the password.    a password hash has been generated using SHA512 algorithm as follows:  password : ZZZ99    hash value : $6$KB$.mzYlKZH98ERJvUQ4W2HLVk4bQuv.E/AjFR0SYIXgrutOdtW27ojLaBXhgZGvkYqA6PPEQw5y2omHyD4j1u4B/  the hash value has been assigned to the string variable called "encrypted\_password" below:    the program will print the password and hash value when match is found.  since there are 4 loops the order of complexity of the program n^4  total number of loop executions in this case will be  26\*26\*26\*100=1757600 times.    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  Char \*encrypted\_password= "$6$KB$.mzYlKZH98ERJvUQ4W2HLVk4bQuv.E/AjFR0SYIXgrutOdtW27ojLaBXhgZGvkYqA6PPEQw5y2omHyD4j1u4B/";  int time\_difference(struct timespec \*start, struct timespec \*finish, long long int \*difference) {  long long int ds = finish->tv\_sec - start->tv\_sec;  long long int dn = finish->tv\_nsec - start->tv\_nsec;  if(dn < 0 ) {  ds--;  dn += 1000000000;  }  \*difference = ds \* 1000000000 + dn;  return !(\*difference > 0);  }  /\*\*  Required by lack of standard function in C.  \*/  void substr(char \*dest, char \*src, int start, int length){  memcpy(dest, src + start, length);  \*(dest + length) = '\0';  }  /\*\*  This function can crack the kind of password explained above. All combinations  that are tried are displayed and when the password is found, #, is put at the  start of the line. Note that one of the most time consuming operations that  it performs is the output of intermediate results, so performance experiments  for this kind of program should not include this. i.e. comment out the printfs.  \*/  void crack(char \*salt\_and\_encrypted){  int w,x, y, z; // Loop counters  char salt[7]; // String used in hashing the password. Need space for \0  char plain[7]; // The combination of letters currently being checked  char \*enc; // Pointer to the encrypted password  int count = 0; // The number of combinations explored so far  substr(salt, salt\_and\_encrypted, 0, 6);  for(w='A'; w<='Z'; w++){  for(x='A'; x<='Z'; x++){  for(y='A'; y<='Z'; y++){  for(z=0; z<=99; z++){  sprintf(plain, "%c%c%c%02d",w, x, y, z);  enc = (char \*) crypt(plain, salt);  count++;  if(strcmp(salt\_and\_encrypted, enc) == 0){  printf("#%-8d%s %s\n", count, plain, enc);  } else {  //printf(" %-8d%s %s\n", count, plain, enc);  }  }  }  }  }  printf("%d solutions explored\n", count);  }  int main(int argc, char \*argv[]){  struct timespec start, finish;  long long int time\_elapsed;  int i;  clock\_gettime(CLOCK\_MONOTONIC, &start);  crack(encrypted\_password);    clock\_gettime(CLOCK\_MONOTONIC, &finish);  time\_difference(&start, &finish, &time\_elapsed);  printf("Time elapsed was %lld ns or %0.9lfs\n", time\_elapsed,  (time\_elapsed/1.0e9));  return 0;  } |

Explain your results of running your 3 initial password cracker with relation to your earlier hypothesis.

**Analysis of Estimated Time vs Actual Time**



The code inside the pccw\_3initials.c was compiled and ran on the PC made available at the college lab since the resource on my personal PC were not enough execute the algorithm with such time complexity.



This was the output as observed from the execution. To compare; the program for brute forcing three initial took approximately **344.719884220s** in conversion can be said that it took 55.4 minutes. Comparing it to the estimated time.   
 Te=54.6minutes  
 (Real observed time) Tr=55.4minutes   
Considering the extra execution time as change in execution time from mathematical explanation as del(time).

Del(time)= Tr-Te = 55.4 -54.6=0.8minutes

A small amount of del(time) can be explained with respect to the constants discussed earlier since we didn’t included the memory processing of them in the estimation. Theoretically speaking we didn’t made estimation of execution of all the flushing and reinitiating of the loop variables however it still do not explain the rate of change ‘del(time)’ being 3%.   
For explanatory reasoning’s I have hypothesised that this has happened due to the shared memory architecture on the core processors. At the instance while running this algorithm the CPU scheduling for ruining the batch job of this particular execution might have gone through various state transfers inside the CPU since there are almost 100’s of kernel process instances running on the same shared memory architecture which made this batch job of this process to halt and execute several time which might have created an exponential chain effect to occur such delay.

**Multithreaded Version of two initials (Using posix and optimized Mutex for posix threads)**

For the purpose of using clutter free loops and fetching the thread with desired chunk of and for **load balancing** between two threads a simple data optimization technique has been used which is called **Domain Decomposition.** The domain decomposed divides the data into desirable chunk.



Here the 67600 combination is divided into two equal chunk as Domain Decomposition.

**Mutex for thread synchronization**

Mutex are inbuilt kernel functions provided on UNIX based operating Systems to restrict the posix thread from collusion while being executed on the same CPU. Although POSIX threads are an exercise of parallel processing but if we examined that with respect to processor execution of the whole algorithm batch job we will observe that the process executes each ready stated execution one at a time since both the thread will have origin (PID) of same execution cycle the CPU will not recognize our threads as two different process this technique of parallel process of data to compiler before execution is called **User Level Threading**. Hence in order for both threads to work simultaneously without being confused with each other and wasting time to decide which process should execute first we use a Scheduling and Exclusion technique called MUTEX (Mutual Exclusion).

There are other such techniques like ***Forking, Using Semaphores on Mutex, Using Futex*** instead however we only deal with user level parallelism in this part since we are using CUDA for further analysis and demonstration of pseudo parallelism.  
General Principle for mutex to work is the use of MUTEX locking variables which is demonstrated as follows:

Source code:

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include <pthread.h>  #include <crypt.h>  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Compile and run with:  cc -o pccw\_multithreadVersion pccw\_multithreadVersion.c -pthread -lcrypt  ./pccw\_multithreadVersion    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* a mutex lock has been defined with global scope so that it could be used  by both of the threads \*/  pthread\_mutex\_t mutex\_lock;  int n\_passwords = 4;  /\* variable for counting solutions examined by two threads \*/  int solution\_counter;      /\* those variables are made global so that each thread can access them \*/  char \*saltAndEncryptedText;  char salt[7];  char \*enc;  /\*\*\*\*\* 4 hash value of password supplied \*\*\*\*\*\*\*/  char \*encrypted\_passwords[] = {  "$6$KB$PUtGvfAAHdh8/92SCkEAjarFqQI6k3e.3e9t8lZ1MN/wY5qvIG.RchUF4VzV6uKJDZL1up/s95NWn82Fz/FQR.",  "$6$KB$u98pcr1UEAEX25XrSrB2Eakm846e83r/lrp2REKCG5o/rphTou9l.3GiEFzchEHZAN9hoAYBlT/sMhr4RxFeL/",  "$6$KB$7JS8Bt9NYczSqQEpWl/8IMZDD4yIs2kYqUYud.83hoyiSpS6ZFF8SeIFm1P5KZvkA1ytLT8WeNG4l/.yaHiVf1",  "$6$KB$uZeo3hLi9vsKvvY.xhFwqRrLyBCB/v/q9vc8QGCaaCnhU8AkqigCnXBc/cNTM.bBPwSV2.6pNBwAf9p4jLsMC1"  };  int time\_difference(struct timespec \*start,  struct timespec \*finish,  long long int \*difference) {  long long int ds = finish->tv\_sec - start->tv\_sec;  long long int dn = finish->tv\_nsec - start->tv\_nsec;  if(dn < 0 ) {  ds--;  dn += 1000000000;  }  \*difference = ds \* 1000000000 + dn;  return !(\*difference > 0);  }  /\* A function to extract substring from a src string to dest string starting  from index 'start' upto index 'length' \*/  void substr(char \*dest, char \*src, int start, int length){  memcpy(dest, src + start, length);  \*(dest + length) = '\0';  }  /\* the thread function to decrypt hash and compared it against the hash value supplied  the function makes comparsion brute-forcely. the function check for password starting form initial  letter 'A' to 'M' \*/  void \*crackA2M(){  int x, y, z; // Loop counters  char plain[7];  for(x='A'; x<='M'; x++){  for(y='A'; y<='Z'; y++){  for(z=0; z<=99; z++){  sprintf(plain, "%c%c%02d", x, y, z);  enc = (char \*) crypt(plain, salt);  /\* locking critical section of the program \*/  pthread\_mutex\_lock(&mutex\_lock);    solution\_counter++;    /\* releasing lock from critical section of the program \*/  pthread\_mutex\_unlock(&mutex\_lock);    if(strcmp(saltAndEncryptedText, enc) == 0){  printf("#%-8d%s %s (computed by thread 1 : ID %ld)\n", solution\_counter, plain, enc,pthread\_self());  } else {  //printf("%-8d%s %s (computed by thread %ld)\n", count, plain, enc,pthread\_self());  }  }  }  }  pthread\_exit(NULL);  }  /\*\*\* the purpose of this function is same as that of 'crackA2M' except that it checks for passwords  starting from 'N' to 'Z' \*\*/  void \*crackN2Z(){  int x, y, z; // Loop counters  char plain[7];  for(x='N'; x<='Z'; x++){  for(y='A'; y<='Z'; y++){  for(z=0; z<=99; z++){  sprintf(plain, "%c%c%02d", x, y, z);  enc = (char \*) crypt(plain, salt);    /\* locking critical section of the program \*/  pthread\_mutex\_lock(&mutex\_lock);    solution\_counter++;    /\* releasing lock from critical section of the program \*/  pthread\_mutex\_unlock(&mutex\_lock);    if(strcmp(saltAndEncryptedText, enc) == 0){  printf("#%-8d%s %s (computed by thread 2 : %ld)\n", solution\_counter, plain, enc,pthread\_self());  } else {  //printf("%-8d%s %s (computed by thread %ld)\n", count, plain, enc,pthread\_self());  }  }  }  }  pthread\_exit(NULL);  }  /\* a function to implement 2 threads \*/  void crack(char \*salt\_and\_encrypted){  solution\_counter=0;  saltAndEncryptedText=salt\_and\_encrypted;  pthread\_t t1, t2;  substr(salt, salt\_and\_encrypted, 0, 6);  if(pthread\_create(&t1, NULL, crackA2M, NULL)!=0){  printf("Sorry, Thread could not be created !\n");  exit(EXIT\_FAILURE);  }    if(pthread\_create(&t2, NULL, crackN2Z, NULL)!=0){  printf("Sorry, Thread could not be created !\n");  exit(EXIT\_FAILURE);  }  pthread\_join(t1, NULL);  pthread\_join(t2, NULL);  printf("%d solutions explored\n",solution\_counter); //shows total number of comparsion performed  }    int main(int argc, char \*argv[]){    struct timespec start, finish;  long long int time\_elapsed;  int i;  clock\_gettime(CLOCK\_MONOTONIC, &start);  /\*\* initialization of mutex lock with error checking \*\*/  if(pthread\_mutex\_init(&mutex\_lock, NULL)!=0) {  printf("problem with initialising mutex...\nProgram terminating...");  exit(EXIT\_FAILURE);  }  for(i=0;i<n\_passwords;i<i++) {  crack(encrypted\_passwords[i]);  }  /\* destroying of the mutex lock at the end \*/  pthread\_mutex\_destroy(&mutex\_lock);  clock\_gettime(CLOCK\_MONOTONIC, &finish);  time\_difference(&start, &finish, &time\_elapsed);  printf("Time elapsed was %lld ns or %0.9lfs\n\n", time\_elapsed,  (time\_elapsed/1.0e9));  return 0;  } |

**Analysis of mean execution time for Sequential Run and Parallel Runs**

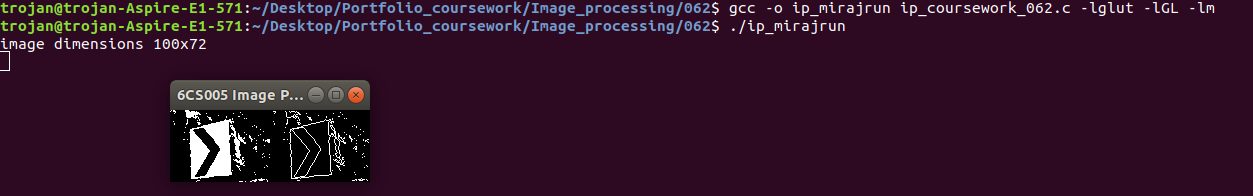
## Image Processing

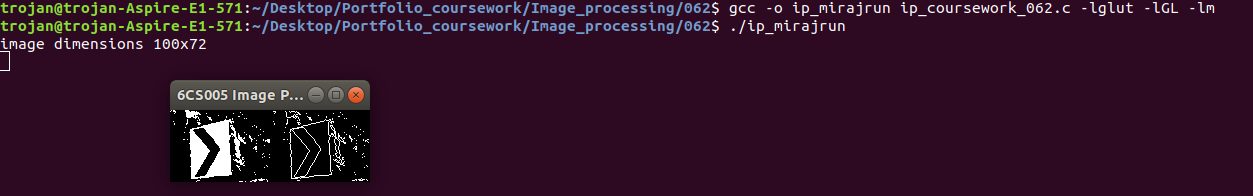
a)



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

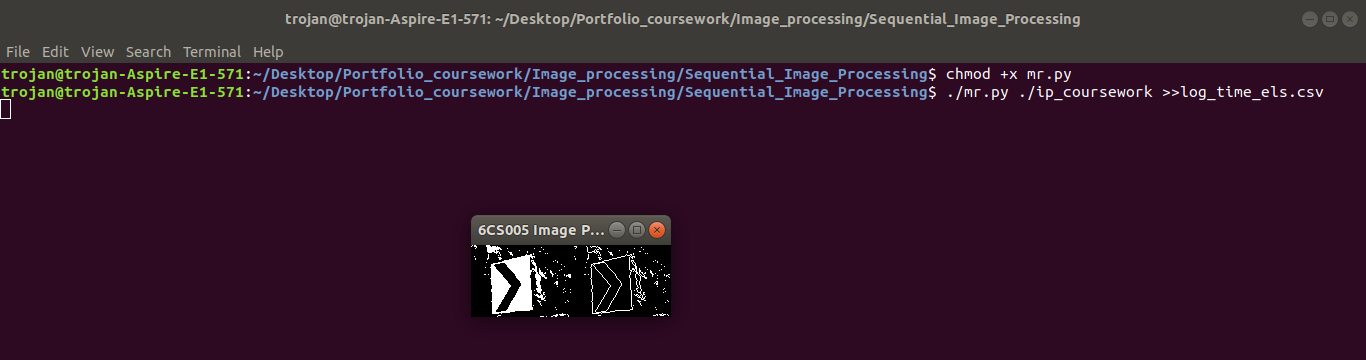
On the first simple manual run of the program provided on a linux bash terminal following output was observed



  
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

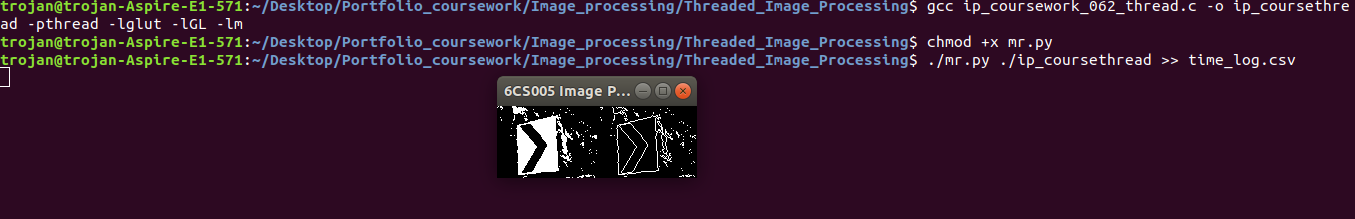
The mean running time was calculated from the *log\_time\_els.csv* file:

|  |  |
| --- | --- |
| **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.

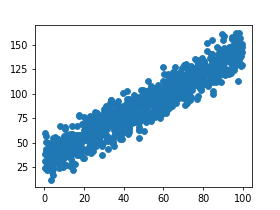
From the first table Sequential run for 10 time, if the data were to be plotted with X ordinate as ith iteration and Y absisa as the execution time it seems to follow a **sinosidual curve(sine curve)** (I.e the execution time of first few iteration were decreasing and suddenly from fourth iteration the 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.

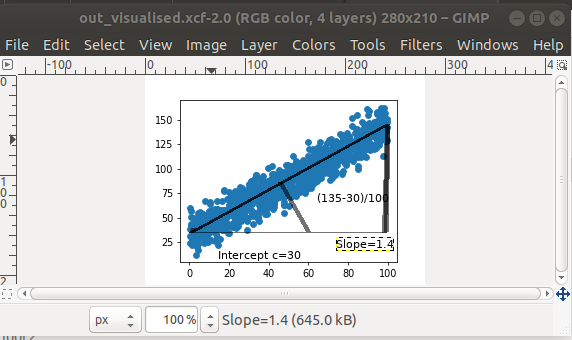
From the second table User level prallelism with POSIX’s 10 iteration, if the data were plotted in the graph it would follow a **Cosine(Co-sinosidual) curve (cose curve)** which means that the execution time were higher and then lower and were fluctuating heavily following the cosine curve cycle. Which 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.

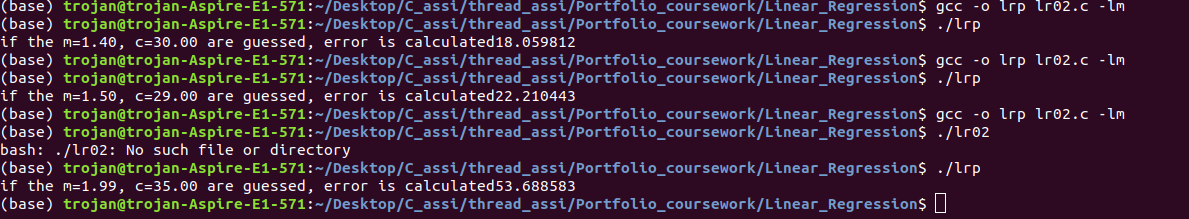
Here for optimium guess of m and c three times are as follows

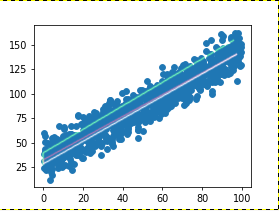
**M1 = 1.3 and C = 30**

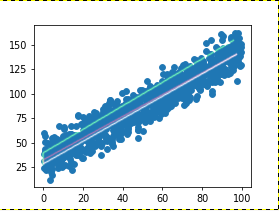
**M2=1.5 and C=29**

**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:

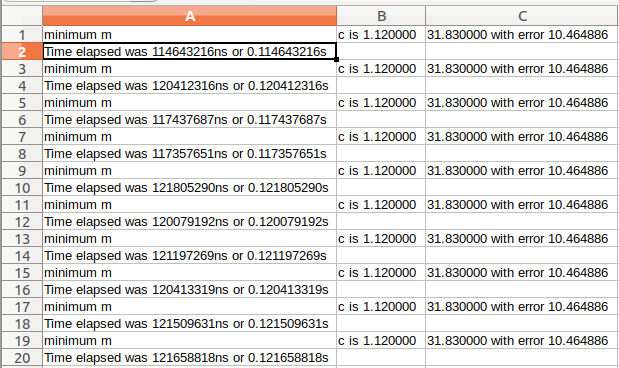
  
Illustration 6: Slope C and RMS for consiquitive guess

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:

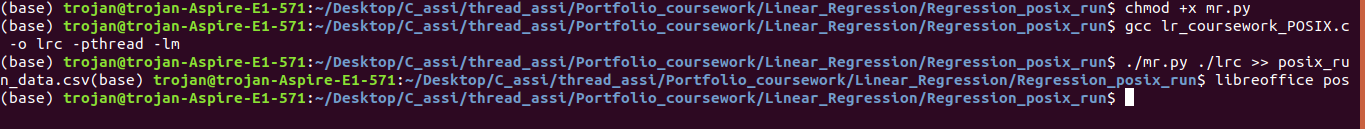
MRTlrs= **∑ xi/ n**

=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.

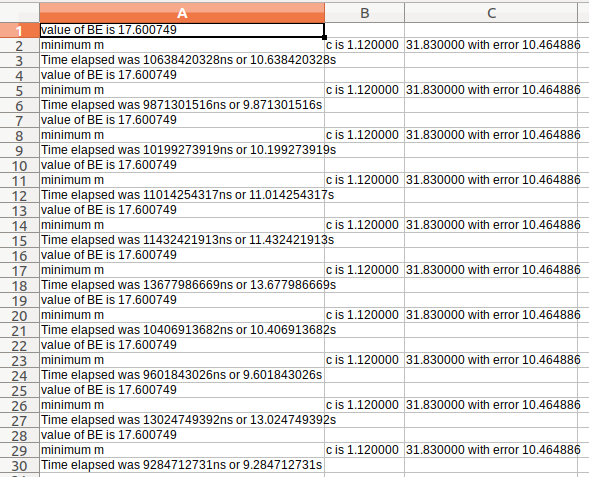


Figure Compiling and running the posix version 10 times

Figure 6Output of ten runs

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

MRTlrp= **∑ xi/ n**

= 3.2716375805 second is the mean run time   
since mean runtime needs to be calculated as rate the harmonic mean run time would be:

MRTlrp Harmonic= n / ∑ [1/Xi]

hence,

del(execution time)= 3.271607871 second per program.

**Comparisons table**

|  |  |
| --- | --- |
| **Sequential RUN** | **Posix RUN** |
| **Mean Execution time (Average Execution)=** 0.119651439s | **Mean Execution Time (Average Execution)=** 3.2716375805s |
| **Rate Of execution per program=** 0.119607871s | **Rate of Execution Per program=** 3.271607871s |

Here analysis of the Mean Execution time and rate of execution per program shows that the time taken for the execution of sequential version was less than that compare to Posix run. This shows a similar behaviour of batch job execution by as analysed and explained on the Image Processing hence for optimization another replacement technique is suggested.

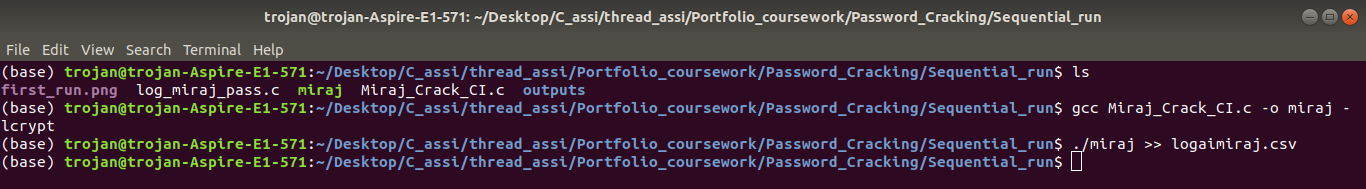
# CUDA

## 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