6CS005 Learning Journal - Semester 1 2019/20

Name: Sangay Sherpa

Student Number: 1928082

Table of Contents

Table of Contents 1

1 POSIX Threads 2

1.1 Password Cracking 2

1.2 Image Processing 7

1.3 Linear Regression 10

2 CUDA 12

2.1 Password Cracking 15

2.2 Image Processing 15

2.3 Linear Regression 15

3 MPI 16

3.1 Password Cracking 16

3.2 Image Processing 16

3.3 Linear Regression 16

4 Verbose Repository Log 17

1 POSIX Threads

1.1 Password Cracking

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

⇨

No. of Run Time	Results (in Seconds)
1	585.203727044s
2	569.738277546s
3	570.767224421s
4	570.144376664s
5	569.837971214s
6	569.003456403s
7	567.514470663s
8	565.767019692s
9	566.962488523s
10	570.485336305s
Total run time	5705.424348475s
Total mean time: 570.5424348475s	

⇨ As shown in table, the normal time taken to run the program would take 565 – 585 seconds after running 10 times and the mean running time is 570.5424348475 seconds.

b. 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.

⇨ If the number of initials were increased to 3, it will take more time. Since, there are 26 letters in the alphabet from A-Z. If the number of initials increased to 3 then it will take multiple of 26 of previous mean running time. The total mean time of 2 initial alphabet was 570.5424348475 seconds. The estimated time for the 3 initial passwords will be 26 times of 570.5424348475 seconds i.e. 14834.103306035 seconds. The time is increased when the program needs to run loop to check encrypted password.

c. Modify the program to crack the three-initials-two-digits password given in the three initials variable. An example password is JSB99.
⇨

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <crypt.h>
#include <time.h>

/*********************************************************************
  Demonstrates how to crack an encrypted password using a simple
  "brute force" algorithm. Works on passwords that consist only of 3 uppercase
  letters and a 2 digit integer. Your personalised data set is included in the
  code.

  Compile with:
    cc -o ThreeVariablesSha ThreeVariablesSha.c -lcrypt

  If you want to analyse the results then use the redirection operator to send
  output to a file that you can view using an editor or the less utility:

    ./ThreeVariablesSha > ThreeVariablesSha_results.txt

  Dr Kevan Buckley, University of Wolverhampton, 2018
*********************************************************************/
int n_passwords = 4;

char* encrypted_passwords[] = {
    "$6$KB$aZ4U4eAlfjO3blbNlW5gDR28n/8YHZyZx3siefWCTfF5Os3Hvw25E1JEpkQiQqMwv4V7IBkc6gE8iCAjKVy8n0",
    "$6$KB$Lrw700jMejJHdZkZHz5l1bqk20qvPwGg/j3w2fBVQOD66v87INBv8j7o.8Aakw/cQyc9WsOFGUdSrtgIquKPo/",
    "$6$KB$gAT.Jh07RuAtAjfrdTym.Shunp/0XE9N5zKr6GwTeQXckwDXWtgFOYO3ZnaLWe4zvhdrf9rS02dWDicoobkMF1",
    "$6$KB$NzjpAuGWI8//zrjSOvxu.bBwcqMSM9pygMuMMmIGOCwHJzK6acJ3OIJi588mCPfNHiO6QikSXkHcWZcVBJES0/"
```

```
};

/**
 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';
}
a
/**
 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 x, y, j, 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 (x = 'A'; x <= 'Z'; x++) {
        for (y = 'A'; y <= 'Z'; y++) {
            for (j = 'A'; j <= 'Z'; j++) {
                for (z = 0; z <= 99; z++) {
                    sprintf(plain, "%c%c%c%02d", x, y, j, 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);
                    }
                }
            }
        }
    }
```

```
77.     printf("%d solutions explored\n", count);
78. }
79. // Calculate the difference between two times. Returns zero on
80. // success and the time difference through an argument. It will
81. // be unsuccessful if the start time is after the end time.
82.
83. int time_difference(struct timespec* start,
84.     struct timespec* finish,
85.     long long int* difference)
86. {
87.     long long int ds = finish->tv_sec - start->tv_sec;
88.     long long int dn = finish->tv_nsec - start->tv_nsec;
89.
90.     if (dn < 0) {
91.         ds--;
92.         dn += 1000000000;
93.     }
94.     *difference = ds * 1000000000 + dn;
95.     return !(*difference > 0);
96. }
97.
98. int main(int argc, char* argv[])
99. {
100.           int i;
101.           struct timespec start, finish;
102.           long long int time_elapsed;
103.
104.           clock_gettime(CLOCK_MONOTONIC, &start);
105.
106.           for (i = 0; i < n_passwords; i < i++) {
107.               crack(encrypted_passwords[i]);
108.           }
109.           clock_gettime(CLOCK_MONOTONIC, &finish);
110.           time_difference(&start, &finish, &time_elapsed);
111.           printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed,
112.               (time_elapsed / 1.0e9));
113.
114.           return 0;
115.       }
```

```
Time elapsed was 14680517158138ns or 14680.517158138s
```

d. Write a short paragraph to compare the running time of your three initials program with your earlier estimate. If your estimate was wrong explained why you think that is.

⇨ To compare the running time of three initials program with my earlier estimate was 14834.103306035 seconds and the actual running time was 14680.517158138 seconds. The difference between them was only 153.586147897 seconds. The actual running time was close to the estimated running time. Hence, the running time of three initials password cracking is correct.

e. 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.
Record the system time a nano second timer
Launch thread_1 that calls kernel_function_1 that can search for passwords starting from A-M
Launch thread_2 that calls kernel_function_2 that can search for passwords starting from N-Z
Wait for thread_1 to finish
Wait for thread_2 to finish

⇨

```
#7379    CV78 $6$KB$6SsUGf4Cq7/Oooym9WWQN3VKeo2lynKV9gXVyEG4HvYy1UFRx.XAye89TLp/
OTcW7cGpf9UlU0F.cK/S9CfZn1
33800 solutions explored
33800 solutions explored
#14984   FT83 $6$KB$.bVspZYQofaBc4KhsjlqZSxu4R7r7mH7.Q/uCYlJ.
3nRV2x5Jz.TKYX6Aa97sUZhTjmN3rett7GrCFr3aO3uR/
33800 solutions explored
33800 solutions explored
#19948   HR47 $6$KB$wO5OXgJxqVzltwW29G/XJuG5ZszGOhwhvpEBnFM/
ThGmTNiNhnUTBS8lnNWs6SwP6XpUwzKXVTv90S2XEWwy./
33800 solutions explored
33800 solutions explored
#13077   SA76 $6$KB$JXfY4sYXD0vnuRY3KIvezu913aIc1TJ0CE0oVTWDrsRSXk4/k.RnljKOR3/
Vw7wxAOnEsxllD01WU/MJP4IIj1
33800 solutions explored
33800 solutions explored
Time elapsed was 283167527310ns or 283.167527310s
```

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

⇨ The mean running time of the original program took 570.5424348475 seconds and the mean running tome of the multithread version took 283.167527310 seconds. Whereas, the multi-threading took half of the time to the original password cracking because the original password cracking had only one thread whereas multi-thread had two thread for password cracking. Since, multi-thread runs parallel and execute the same time due to which the running time was halved compared to the original password cracking.

1.2 Image Processing

a. Run the program and capture the resulting image to put in your learning journal.

⇨

b. Implement a version of the edge detector that can process 4 pixels in parallel. It should do this by using a striding technique, i.e. each thread processes every fourth pixel as shown in the small image below. The first thread processes the blue/purple pixels, the second processes the cyan ones, the third thread processes the yellow ones and the fourth thread processes the red ones.

⇨ Image processing using poxis thread

No. of run time	Results (in seconds)
1	0.000065402s

2	0.000066869s
3	0.000068282s
4	0.000065556s
5	0.000061137s
6	0.000064373s
7	0.000068915s
8	0.000063581s
9	0.000071779s
10	0.000068504s
Total run time	0.000664398s
Total mean time: 0.0000664398s	

⇨ Image processing using Multi Thread

No. of run time	Results (in seconds)
1	0.000506253s
2	0.000280587s
3	0.000190617s
4	0.000218836s

5	0.000222390s
6	0.000256847s
7	0.000269417s
8	0.000282680s
9	0.000193568s
10	0.001276379s
Total run time	0.003416987s
Total mean time: 0.0003416987s	

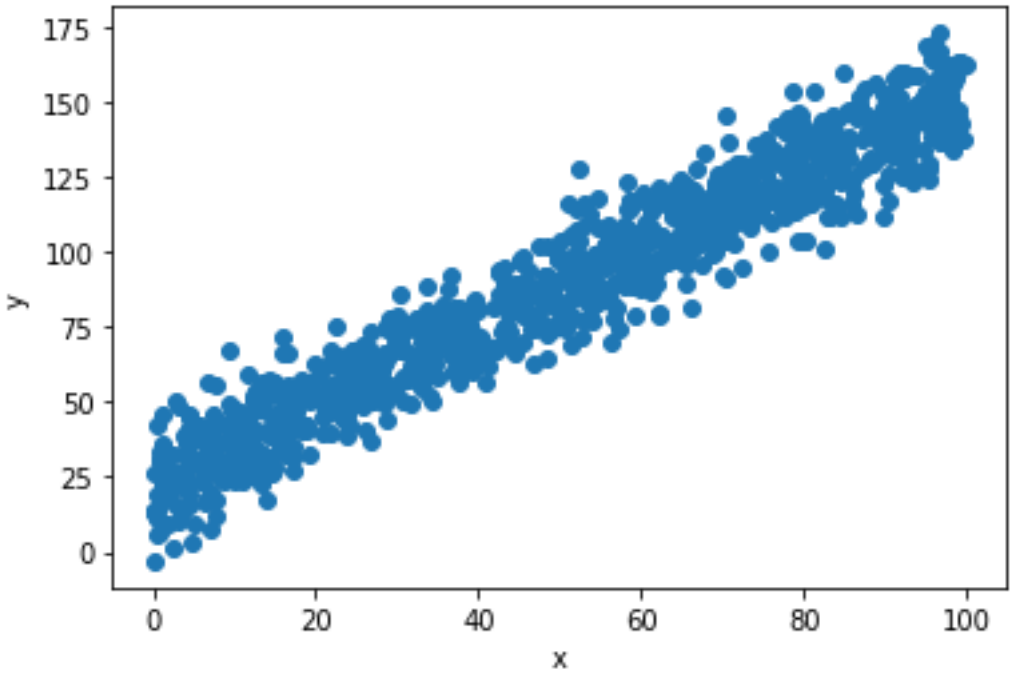
c. Compare the relative running times of the original edge detection program, with the multithread one.

⇨ The original edge detection program took 0000664398 seconds and multi thread edge detection took 0.0003416987 seconds. Since, multi thread edge detection took little more time compare to original program because multi thread is not suitable and necessary for small program to detect the edge of image and display the image.

1.3 Linear Regression

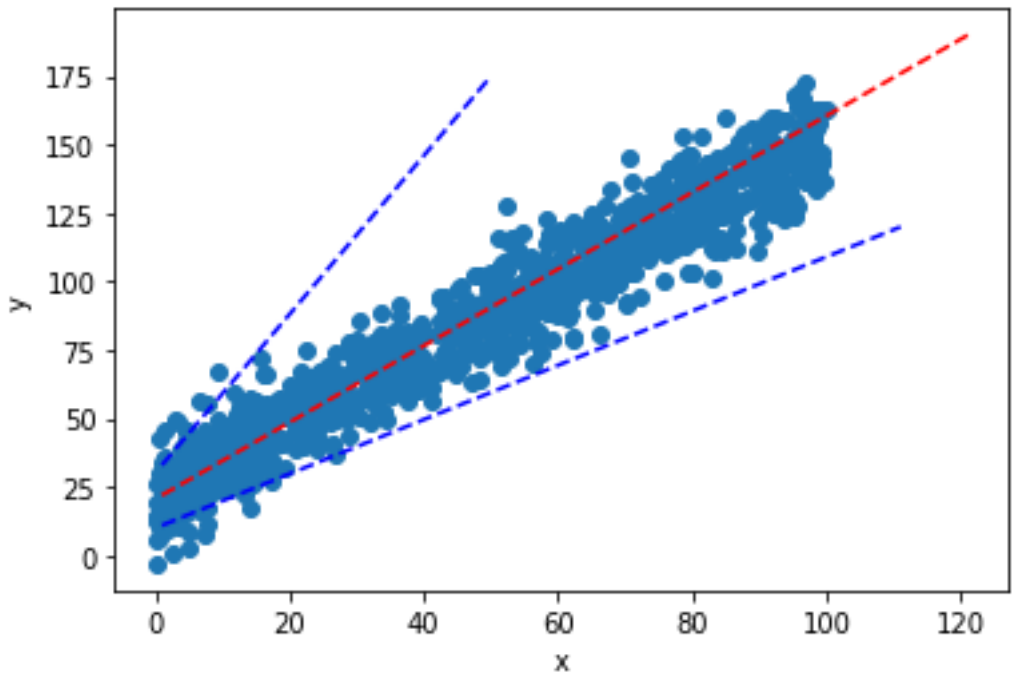
a. Do a scatter plot of your dataset and put it in your portfolio.

⇨



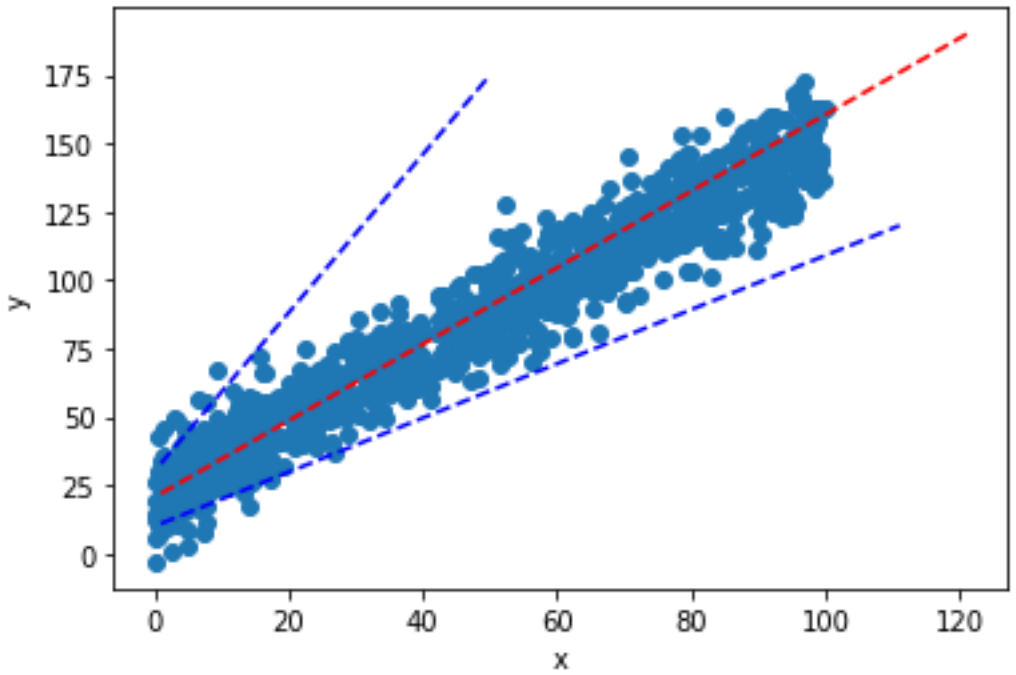
b. Have three guesses at the optimum values for m and c and plot them.

⇨



c. Run the program to see what solution it finds. Overlay the line that was found by the program on to a dataset scatter plot and comment on the solution.

⇨



d. Remove any extraneous printf statements from the program and find its mean running time.

⇨

No. of Run Time	Results (in Seconds)
1	0.116638709s
2	0.116209649s
3	0.115953620s
4	0.115676172s
5	0.117864340s
6	0.117109029s
7	0.117935084s
8	0.115637000s
9	0.117667039s
10	0.117810390s
Total run time	1.168501032s
Total mean time: 0.1168501032s	

⇨ The total run time is 1.168501032s and mean time is 0.1168501032s.

⇨

e. During each iteration of the program, eight values for m and c are evaluated to measure the resulting error. Create a modified version of the program that performs each of the evaluations on a different thread.

⇨

No. of Run Time	Results (in Seconds)
1	2.701784681s
2	1.372279003s
3	2.848234634s
4	3.400179074s
5	1.113095478s
6	2.438337627s
7	1.087493762s
8	1.453895336s
9	2.234917962s
10	1.344484552s
Total Run time	19.994702109s
Mean running time (in sec): 1.9994702109s	

⇨ The total run time is 19.994702109s and mean run time is 1.9994702109s.

f. Calculate the mean running time of the multithread version of the program and use to make comments on the relative performance of the 2 versions.

⇨ The original version took 0.1168501032s and the multithread version took 1.9994702109s. In this case, multithread version took more time than original version because original linear regression is comparatively small program and the multithread is not suitable for small programs so multithread linear regression takes more time.

2 CUDA

2.1 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

2.2 Image Processing

```
Paste your source code for your CUDA based image processing.
```

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

Write a short analysis of the results

2.3 Linear Regression

```
Paste your source code for your CUDA based linear regression
```

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

Write a short analysis of the results

3 MPI

3.1 Password Cracking

```
Paste your source code for your MPI based password cracker here
```

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

Write a short analysis of the results

3.2 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

3.3 Linear Regression

```
Paste your source code for your MPI based linear regression
```

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

Write a short analysis of the results

4 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
```