UNIVERSITY PARTNER

High Performance Computing

(6CS005)

Portfolio Report

University Id : 2040367

Student Name : Birat Jung Thapa

Student ID : NP03A170004

Cohort/Batch : 4

Submitted on : 26-12-2020

Table of Contents

Parallel and Distributed Systems

1. ?
   A thread is part of the process, operating within its own space of execution, and in one process there can be several threads. With the help of it, OS can do multiple tasks in parallel (depending upon the number of processors the machine consists). Threads allow CPU to execute many tasks of one process at the same time.

2. ?
   Two process scheduling policies are:
   Pre-emptive: This scheduler oversees how long a process runs for. If a process exceeds its time slice, the scheduler stops the process.
   Co-operative: Each process is never interrupted by the OS and the processes are in-charge of how long they run for. When a process feels like co-operating, it will surrender execution.
   Pre-emptive is preferable, Java runtime system's thread scheduling algorithm is also pre-emptive.

3. ?
   In centralized system, all calculations are done on one computer(system). In distributed system, the calculation is distributed to multiple computers. For example: When there is a large amount of data then it can be divided and sent to each part of computers to carry it out.

4. ?
   Transparency in Distributed system means one distributed system looks like a single computer by concealing distribution from the user and application programmer.

5. 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.
   Note: Show all the works in your report and produce a simple C code simulate the process of producing the C and B values.
   B=A+C
   B=C+D
   C=B+D

B=A+C is a flow dependency
C=B+D is an anti-dependency
B=C+D is an output dependency

6. ?

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.

Matrix Multiplication and Password Cracking

Single Thread Matrix Multiplication

- The complexity of above program is $O(n^3)$.
- Divide and Conquer could also speed the multiplication process but the better option would be Strassen's matrix multiplication.
- 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.

```
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.
```

- The output is:

```
Enter the 4 elements of first matrix: 1 2 3 4
Enter the 4 elements of second matrix: 5 6 7 8

The first matrix is

1       2
3       4
The second matrix is

5       6
7       8
After multiplication using

19      22
43      50
Process returned 0 (0x0)   execution time : 18.411 s
Press any key to continue.
```

Matrix Multiplication using Multithreading

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

#define MAT_SIZE 10
#define MAX_THREADS 100

/********
 Compile with   cc -o task2b 2040367_Task2_B.c -pthread

 ./task2b

********/


int i,j,k;            //Parameters For Rows And Columns
int matrix1[MAT_SIZE][MAT_SIZE]; //First Matrix
int matrix2[MAT_SIZE][MAT_SIZE]; //Second Matrix
int result [MAT_SIZE][MAT_SIZE]; //Multiplied Matrix

//Type Defining For Passing Function Argumnents
typedef struct parameters {
    int x,y;
}args;

//Function For Calculate Each Element in Result Matrix Used By Threads - - -//
void* mult(void* arg){

    args* p = arg;

    //Calculating Each Element in Result Matrix Using Passed Arguments
    for(int a=0;a<j;a++){
        result[p->x][p->y] += matrix1[p->x][a]*matrix2[a][p->y];
    }
    sleep(3);

    //End Of Thread
    pthread_exit(0);
}
```

```c
int main(){

    //Initializing All Defined Matrices By Zero - - - - - - - - - - - - - - - -//
    for(int x=0;x<10;x++){
        for(int y=0;y<10;y++){
            matrix1[x][y] = 0;
            matrix2[x][y] = 0;
            result[x][y] = 0;
        }
    }


    //Getting Matrix1 And Matrix2 Info from User - - - - - - - - - - - - - - - -//

    printf(" --- Defining Matrix 1 ---\n\n");

    // Getting Row And Column(Same As Row In Matrix2) Number For Matrix1
    printf("Enter number of rows for matrix 1: ");
    scanf("%d",&i);
    printf("Enter number of columns for matrix 1: ");
    scanf("%d",&j);

    printf("\n --- Initializing Matrix 1 ---\n\n");
    for(int x=0;x<i;x++){
        for(int y=0;y<j;y++){
            printf("Enter variable [%d,%d]: ",x+1,y+1);
            scanf("%d",&matrix1[x][y]);
        }
    }

    printf("\n --- Defining Matrix 2 ---\n\n");

    // Getting Column Number For Matrix2
    printf("Number of rows for matrix 2 : %d\n",j);
    printf("Enter number of columns for matrix 2: ");
    scanf("%d",&k);

    printf("\n --- Initializing Matrix 2 ---\n\n");
    for(int x=0;x<j;x++){
        for(int y=0;y<k;y++){
            printf("Enter variable [%d,%d]: ",x+1,y+1);
            scanf("%d",&matrix2[x][y]);
        }
    }
```

```c
//Printing Matrices - - - - - - - - - - - - - - - - - - - - - - - - - - -//

printf("\n --- Matrix 1 ---\n\n");
for(int x=0;x<i;x++){
    for(int y=0;y<j;y++){
        printf("%5d",matrix1[x][y]);
    }
    printf("\n\n");
}

printf(" --- Matrix 2 ---\n\n");
for(int x=0;x<j;x++){
    for(int y=0;y<k;y++){
        printf("%5d",matrix2[x][y]);
    }
    printf("\n\n");
}


//Multiply Matrices Using Threads - - - - - - - - - - - - - - - - - - - - -//

//Defining Threads
pthread_t thread[MAX_THREADS];

//Counter For Thread Index
int thread_number = 0;

//Defining p For Passing Parameters To Function As Struct
args p[i*k];

//Start Timer
time_t start = time(NULL);

for(int x=0;x<i;x++){
    for(int y=0;y<k;y++){

        //Initializing Parameters For Passing To Function
        p[thread_number].x=x;
        p[thread_number].y=y;

        //Status For Checking Errors
        int status;
```

```c
            //Create Specific Thread For Each Element In Result Matrix
            status = pthread_create(&thread[thread_number], NULL, mult, (void *) &p[thread_number]);

            //Check For Error
            if(status!=0){
                printf("Error In Threads");
                exit(0);
            }

            thread_number++;
        }
    }


    //Wait For All Threads Done - - - - - - - - - - - - - - - - - - - - - - - //

    for(int z=0;z<(i*k);z++)
        pthread_join(thread[z],NULL );


    //Print Multiplied Matrix (Result) - - - - - - - - - - - - - - - - - - -//

    printf(" --- Multiplied Matrix ---\n\n");
    for(int x=0;x<i;x++){
        for(int y=0;y<k;y++){
            printf("%5d",result[x][y]);
        }
        printf("\n\n");
    }


    //Calculate Total Time Including 3 Soconds Sleep In Each Thread - - - -//

    printf(" ---> Time Elapsed : %.2f Sec\n\n", (double)(time(NULL) - start));


    //Total Threads Used In Process - - - - - - - - - - - - - - - - - - - -//

    printf(" ---> Used Threads : %d \n\n",thread_number);
    for(int z=0;z<thread_number;z++)
        printf(" - Thread %d ID : %d\n",z+1,(int)thread[z]);

    return 0;
}
```

Output:

```
--- Matrix 1 ---

   5    4    6

   8    8    6

   2    1    3

--- Matrix 2 ---

   4    4    5    6    6

   8    5    2    2    3

   0    2    2    3    5

--- Multiplied Matrix ---

  52   52   45   56   72

  96   84   68   82  102

  16   19   18   23   30

---> Time Elapsed : 3.00 Sec

---> Used Threads : 15

- Thread 1 ID : 506509056
- Thread 2 ID : 498116352
- Thread 3 ID : 489723648
- Thread 4 ID : 481330944
- Thread 5 ID : 472938240
- Thread 6 ID : 464545536
- Thread 7 ID : 456152832
- Thread 8 ID : 447760128
- Thread 9 ID : 439367424
- Thread 10 ID : 430974720
- Thread 11 ID : 422582016
- Thread 12 ID : 414189312
- Thread 13 ID : 405796608
- Thread 14 ID : 397403904
- Thread 15 ID : 389011200
com59@herald-OptiPlex-3050:~/6cs005_PortfolioS1_19_20_2040367_Birat_Jung_Thapa$
```

The program uses each thread per element in resultant matrix. So, for a matrix of size 1024*1024, 1048576 threads are used.

Password Cracking using POSIX thread

1. .

Number of Program runs	Time in nano seconds	Time in seconds
1	126000873620.00	126.000873620
2	126636175726.00	126.636175726
3	126226767587.00	126.226767587
4	126308696476.00	126.308696476
5	126079736292.00	126.079736292
6	126721747567.00	126.721747567
7	126903683830.00	126.903683830
8	126358508917.00	126.358508917
9	126025959560.00	126.025959560
10	126596760633.00	126.596760633
Average		126.3858910208

2. .
   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
   = 126.3858910208 * 26
   = 3,286.0331665408 seconds

   Converting to minutes = 3,286.0331665408 / 60
   = 54.76721944234667 minutes

   Therefore, estimated time is 55 minutes.

3. .

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

/***********************************************************************
*******

  Compile with:
    cc -o task2c3 2040367_Task2_C_3.c -lcrypt

    ./task2c3 > task2c3.txt
***********************************************************************
******/


int n_passwords = 1;

char *encrypted_passwords[] = {
 "$6$AS$iHk20J571/kPUXRfcKd6.73Vzpkp.Uygop5a0Oq4m23.UBzYt1llV6g27zKF.BTqsVtp.MX/y9usB3EPnTY7E0
};


void substr(char *dest, char *src, int start, int length){
  memcpy(dest, src + start, length);
  *(dest + length) = '\0';
}


void crack(char *salt_and_encrypted){
  int b, i, r, a;     // 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(b='A'; b<='Z'; b++){
    for(i='A'; i<='Z'; i++){
     for(r='A'; r<='Z'; r++){
      for(a=0; a<=99; a++){
        sprintf(plain, "%c%c%c%02d", b, i, r, a);
```

```
          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 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);
}


int main(int argc, char *argv[]){
  int i;
struct timespec start, finish;
  long long int time_elapsed;

  clock_gettime(CLOCK_MONOTONIC, &start);


  for(i=0;i<n_passwords;i<i++) {
    crack(encrypted_passwords[i]);
  }

 clock_gettime(CLOCK_MONOTONIC, &finish);
  time_difference(&start, &finish, &time_elapsed);
  printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed,
         (time_elapsed/1.0e9));

  return 0;
}
```

Output:

```
#122310   BVB09  $6$AS$iHk20J571/kPUXRfcKd6.73Vzpkp.Uygop5a0Oq4m23.UBzYt1llV6g27zKF.BTqsVtp.MX/y9usB3EPnTY7E0
```

4. .

Number of Program runs	Time in nano seconds	Time in seconds
1	3288492659305.00	3288.4926593050
2	3280508108454.00	3280.5081084540
3	3286985627056.00	3286.9856270560
4	3282365845262.00	3282.3658452620
5	3289597435812.00	3289.5974358120
6	3281562475100.00	3281.5624751000
7	3283536357519.00	3283.5363575190
8	328845826851254.00	3288.4582685125
9	3285512760316.00	3285.5127603160
10	3289363482565.00	3289.3634825650
Average	35841375160264.30	3285.6383019902

The actual time is: 3285.6383019902 seconds which converted is 54.76063836650333 minutes. The estimated time was 54.76721944234667 which is which is a bit more than the estimated time but still in the same timeframe.

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

5. .

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

/******************************************************************************
*******

  Compile with:
    cc -o task2c5 2040367_Task2_C_5.c -lcrypt -lrt -pthread

    ./task2c5 > task2c5.txt
******************************************************************************
******/


int n_passwords = 1;

char *encrypted_passwords[] = {

"$6$AS$M3J7Zk1gFfAYLJ/sujt3irgzlIngxqtP1RcYOiWnBWm/r/3fQ6wfvKIBxvKxdChMDiRueRbdMqxHDtt1tFhBy"
};



void substr(char *dest, char *src, int start, int length){
  memcpy(dest, src + start, length);
  *(dest + length) = '\0';
}


void run()
{
  int i;
pthread_t thread_1, thread_2;

    void *kernel_function_1();
    void *kernel_function_2();
for(i=0;i<n_passwords;i<i++) {
```

```c
    pthread_create(&thread_1, NULL,kernel_function_1, encrypted_passwords[i]);
    pthread_create(&thread_2, NULL,kernel_function_2, encrypted_passwords[i]);

    pthread_join(thread_1, NULL);
    pthread_join(thread_2, NULL);
  }
}

void *kernel_function_1(void *salt_and_encrypted){
  int b, i, a;     // Loop counters
  char salt[7];    // String used in hahttps://www.youtube.com/watch?v=L8yJjIGleMwshing the password. Need space
  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(b='A'; b<='M'; b++){
    for(i='A'; i<='Z'; i++){
      for(a=0; a<=99; a++){
        sprintf(plain, "%c%c%02d", b,i,a);
        enc = (char *) crypt(plain, salt);
        count++;
        if(strcmp(salt_and_encrypted, enc) == 0){
          printf("#%-8d%s %s\n", count, plain, enc);
        }
      }
    }
  }
  printf("%d solutions explored\n", count);
}


void *kernel_function_2(void *salt_and_encrypted){
  int b, i, a;     // Loop counters
  char salt[7];    // String used in hahttps://www.youtube.com/watch?v=L8yJjIGleMwshing the password. Need space
  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);
```

```c
  for(b='N'; b<='Z'; b++){
    for(i='A'; i<='Z'; i++){
      for(a=0; a<=99; a++){
        sprintf(plain, "%c%c%02d", b,i,a);
        enc = (char *) crypt(plain, salt);
        count++;
        if(strcmp(salt_and_encrypted, enc) == 0){
          printf("#%-8d%s %s\n", count, plain, enc);
        }
      }
    }
  }
  printf("%d solutions explored\n", count);
}

//Calculating time

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);
}
int main(int argc, char *argv[])
{

    struct timespec start, finish;
    long long int time_elapsed;

    clock_gettime(CLOCK_MONOTONIC, &start);

            run();

    clock_gettime(CLOCK_MONOTONIC, &finish);
      time_difference(&start, &finish, &time_elapsed);
      printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed,
                                         (time_elapsed/1.0e9));
  return 0;

  return 0;
}
```

```
com59@herald-OptiPlex-3050:~/6cs005_PortfolioS1_19_20_2040367_Birat_Jung_Thapa$ ./mr.py ./task2c5 | grep Time
Time elapsed was 63451680561ns or 63.451680561s
Time elapsed was 63051267376ns or 63.051267376s
Time elapsed was 63068918934ns or 63.068918934s
Time elapsed was 63094323747ns or 63.094323747s
Time elapsed was 63060856432ns or 63.060856432s
Time elapsed was 63079669960ns or 63.079669960s
Time elapsed was 63112758771ns or 63.112758771s
Time elapsed was 63188655284ns or 63.188655284s
Time elapsed was 63072563625ns or 63.072563625s
Time elapsed was 63045828425ns or 63.045828425s
com59@herald-OptiPlex-3050:~/6cs005_PortfolioS1_19_20_2040367_Birat_Jung_Thapa$
```

6. .

Number of times program ran	Time taken by original program	Time taken by multithread version
1	126.000873620	63.45168056
2	126.636175726	63.05126738
3	126.226767587	63.06891893
4	126.308696476	63.09432375
5	126.079736292	63.06085643
6	126.721747567	63.07966996
7	126.903683830	63.11275877
8	126.358508917	63.18868828
9	126.025959560	63.07256363
10	126.596760633	63.04582843
Average (seconds)	126.3858910208	63.12265561

Here the single thread version took 126.3858910208 seconds while the multithread version took 63.12265561 seconds. 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.

CUDA

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

Password Cracking using CUDA

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


//__global__ --> GPU function which can be launched by many blocks and threads
//__device__ --> GPU function or variables
//__host__ --> CPU function or variables



/***********************************************************************
*******
 pass = cxbdwy2745

  nvcc -o task3a 2040367_Task3_A.cu

  ./task3a >task3a.txt
***********************************************************************
******/


char *encrypted_passwords[]{
    "cxbdwy2745"
};

int i =0;

int n_passwords = 1;

char *encrypted_passwords[]{
    "AN4019"
};
void substr(char *dest, char *src, int start, int length){
  memcpy(dest, src + start, length);
  *(dest + length) = '\0';
}
```

```
__device__ char* CudaCrypt(char* rawPassword){

    char * newPassword = (char *) malloc(sizeof(char) * 11);

    newPassword[0] = rawPassword[0] + 2;
    newPassword[1] = rawPassword[0] - 2;
    newPassword[2] = rawPassword[0] + 1;
    newPassword[3] = rawPassword[1] + 3;
    newPassword[4] = rawPassword[1] - 3;
    newPassword[5] = rawPassword[1] - 1;
    newPassword[6] = rawPassword[2] + 2;
    newPassword[7] = rawPassword[2] - 2;
    newPassword[8] = rawPassword[3] + 4;
    newPassword[9] = rawPassword[3] - 4;
    newPassword[10] = '\0';

    for(int i =0; i<10; i++){
        if(i >= 0 && i < 6){ //checking all lower case letter limits
            if(newPassword[i] > 122){
                newPassword[i] = (newPassword[i] - 122) + 97;
            }else if(newPassword[i] < 97){
                newPassword[i] = (97 - newPassword[i]) + 97;
            }
        }else{ //checking number section
            if(newPassword[i] > 57){
                newPassword[i] = (newPassword[i] - 57) + 48;
            }else if(newPassword[i] < 48){
                newPassword[i] = (48 - newPassword[i]) + 48;
            }
        }
    }
    return newPassword;
}

__global__ void crack(char * alphabet, char * numbers, char *salt_and_encrypted){


char genRawPass[4];

genRawPass[0] = alphabet[blockIdx.x];
genRawPass[1] = alphabet[blockIdx.y];

genRawPass[2] = numbers[threadIdx.x];
genRawPass[3] = numbers[threadIdx.y];
```

```
//firstLetter - 'a' - 'z' (26 characters)
//secondLetter - 'a' - 'z' (26 characters)
//firstNum - '0' - '9' (10 characters)
//secondNum - '0' - '9' (10 characters)

//Idx --> gives current index of the block or thread

printf("%c %c %c %c = %s\n", genRawPass[0],genRawPass[1],genRawPass[2],genRawPass[3], CudaCrypt(genRawPass));
  int x, y, z;     // Loop counters
  char salt[7];    // String used in hashing the password. Need space for \0 // incase you have modified the salt value, then should modifiy the number accordingly
  char plain[7];   // The combination of letters currently being checked // Please modifiy the number when you enlarge the encrypted password.
  char *enc;       // Pointer to the encrypted password
  int count=0;

  substr(salt, salt_and_encrypted, 0, 6);

  for(x='A'; 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);
        count++;
        if(strcmp(salt_and_encrypted, enc) == 0){
        printf("#%-8d%s %s\n", count, plain, enc);
        //return;   //uncomment this line if you want to speed-up the running time, program will find you the cracked password only without exploring all possibilites
        }
      }
    }
  }
}


void run(){
void *crack();
    for(int i=0;i<n_passwords;i++)
    {
            crack(encrypted_passwords[i]);
    }
}

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);
}


int main(int argc, char ** argv){
    struct  timespec start, finish;
    long long int time_elapsed;
    clock_gettime(CLOCK_MONOTONIC, &start);


char cpuAlphabet[26] = {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z'};
char cpuNumbers[26] = {'0','1','2','3','4','5','6','7','8','9'};

char * gpuAlphabet;
cudaMalloc( (void**) &gpuAlphabet, sizeof(char) * 26);
cudaMemcpy(gpuAlphabet, cpuAlphabet, sizeof(char) * 26, cudaMemcpyHostToDevice);

char * gpuNumbers;
cudaMalloc( (void**) &gpuNumbers, sizeof(char) * 26);
cudaMemcpy(gpuNumbers, cpuNumbers, sizeof(char) * 26, cudaMemcpyHostToDevice);

crack<<< dim3(26,26,1), dim3(10,10,1) >>>( gpuAlphabet, gpuNumbers );
cudaThreadSynchronize();

run();

    clock_gettime(CLOCK_MONOTONIC, &finish);
    time_difference(&start, &finish, &time_elapsed);
    printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed, (time_elapsed/1.0e9));

return 0;
}
```

Output:

```
# z z 9 9 = cxbdwy2745
```

Number of times program ran	Time taken by Original program	Time taken by Multithread version	Time taken by CUDA version
1	126.000873620	63.45168056	0.230222187
2	126.636175726	63.05126738	0.239700361
3	126.226767587	63.06891893	0.243830382
4	126.308696476	63.09432375	0.223726209
5	126.079736292	63.06085643	0.233408816
6	126.721747567	63.07966996	0.23461098
7	126.903683830	63.11275877	0.241078292
8	126.358508917	63.18868828	0.236716777
9	126.025959560	63.07256363	0.236223031
10	126.596760633	63.04582843	0.241125265
Average (seconds)	126.3858910208	63.12265561	0.23606423

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.

Gaussian Blur

```
#include <stdio.h>
#include <stdio.h>
#include <stdlib.h>

#include "lodepng.h"

/********
Compile with nvcc 2040367_Task3_B.cu lodepng.cpp -o task3b

            ./task3b
*********/

__global__ void blur_image(unsigned char * gpu_imageOuput, unsigned char * gpu_imageInput,int width,int height){

    int counter=0;

    int idx = blockDim.x * blockIdx.x + threadIdx.x;


    int i=blockIdx.x;
    int j=threadIdx.x;


    float t_r=0;
    float t_g=0;
    float t_b=0;
    float t_a=0;
    float s=1;

    if(i+1 && j-1){

        // int pos= idx/2-2;

        int pos=blockDim.x * (blockIdx.x+1) + threadIdx.x-1;
        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
```

```
        counter++;



    }

    if(j+1){

        // int pos= idx/2-2;

        int pos=blockDim.x * (blockIdx.x) + threadIdx.x+1;

        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
        t_a += s*gpu_imageInput[3+pixel];

        counter++;
    }

    if(i+1 && j+1){

        // int pos= idx/2+1;

        int pos=blockDim.x * (blockIdx.x+1) + threadIdx.x+1;


        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
```

```
        counter++;


    }

    if(i+1){
        // int pos= idx+1;

        int pos=blockDim.x * (blockIdx.x+1) + threadIdx.x;

        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
        t_a += s*gpu_imageInput[3+pixel];

        counter++;



    }

    if(j-1){

        // int pos= idx*2-2;
        int pos=blockDim.x * (blockIdx.x) + threadIdx.x-1;

        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
        t_a += s*gpu_imageInput[3+pixel];
```

```
        counter++;



    }

    if(i-1){

        // int pos= idx-1;
        int pos=blockDim.x * (blockIdx.x-1) + threadIdx.x;

        int pixel = pos*4;

        // t_r=s*gpu_imageInput[idx*4];
        // t_g=s*gpu_imageInput[idx*4+1];
        // t_b=s*gpu_imageInput[idx*4+2];
        // t_a=s*gpu_imageInput[idx*4+3];

        t_r += s*gpu_imageInput[pixel];
        t_g += s*gpu_imageInput[1+pixel];
        t_b += s*gpu_imageInput[2+pixel];
        t_a += s*gpu_imageInput[3+pixel];

        counter++;


    }

    int current_pixel=idx*4;

    gpu_imageOuput[current_pixel]=(int)t_r/counter;
    gpu_imageOuput[1+current_pixel]=(int)t_g/counter;
    gpu_imageOuput[2+current_pixel]=(int)t_b/counter;
    gpu_imageOuput[3+current_pixel]=gpu_imageInput[3+current_pixel];


}

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);
}

int main(int argc, char **argv){
struct  timespec start, finish;
    long long int time_elapsed;
    clock_gettime(CLOCK_MONOTONIC, &start);

    unsigned int error;
    unsigned int encError;
    unsigned char* image;
    unsigned int width;
    unsigned int height;
    const char* filename = "image.png";
    const char* newFileName = "blur.png";

    error = lodepng_decode32_file(&image, &width, &height, filename);
    if(error){
        printf("error %u: %s\n", error, lodepng_error_text(error));
    }

    const int ARRAY_SIZE = width*height*4;
    const int ARRAY_BYTES = ARRAY_SIZE * sizeof(unsigned char);

    unsigned char host_imageInput[ARRAY_SIZE * 4];
    unsigned char host_imageOutput[ARRAY_SIZE * 4];

    for (int i = 0; i < ARRAY_SIZE; i++) {
        host_imageInput[i] = image[i];
    }

    // declare GPU memory pointers
    unsigned char * d_in;
    unsigned char * d_out;

    // allocate GPU memory
    cudaMalloc((void**) &d_in, ARRAY_BYTES);
    cudaMalloc((void**) &d_out, ARRAY_BYTES);
```

```
    cudaMemcpy(d_in, host_imageInput, ARRAY_BYTES, cudaMemcpyHostToDevice);

    // launch the kernel
    blur_image<<<height, width>>>(d_out, d_in,width,height);


    // copy back the result array to the CPU
    cudaMemcpy(host_imageOutput, d_out, ARRAY_BYTES, cudaMemcpyDeviceToHost);

    encError = lodepng_encode32_file(newFileName, host_imageOutput, width, height);
    if(encError){
        printf("error %u: %s\n", error, lodepng_error_text(encError));
    }

    //free(image);
    //free(host_imageInput);
    cudaFree(d_in);
    cudaFree(d_out);

    //free(image);
    //free(host_imageInput);
    cudaFree(d_in);
    cudaFree(d_out);
    clock_gettime(CLOCK_MONOTONIC, &finish);
    time_difference(&start, &finish, &time_elapsed);
    printf("Time elapsed was %lldns or %0.9lfs\n", time_elapsed, (time_elapsed/1.0e9));

    return 0;
}
```

Image:

Result: