6CS005 Learning Journal - Semester 1 2020/21

Shakti Dewan - 2038581

Table of Contents

Table of Contents 1
1 Parallel and Distributed Systems 2
1.1 Answer of First Question 2
1.2 Answer of Second Question 2
1.3 Answer of Third Question 2
1.4 Answer of Fourth Question 3
1.5 Answer of Fifth Question 3
1.6 Answer of Sixth Question 6
2 Applications of Matrix Multiplication and Password Cracking using HPC-based CPU system 7
2.1 Single Thread Matrix Multiplication 7
2.2 Multithreaded Matrix Multiplication 14
2.3 Password cracking using POSIX Threads 19
3 Applications of Password Cracking and Image Blurring using HPC-based CUDA System 32
3.1 Password Cracking using CUDA 33
3.2 Image blur using multi dimension Gaussian matrices 38

1 Parallel and Distributed Systems

1.1 Answer of First Question

1. What are threads and what they are designed to solve? (2 marks)

   Answer:

   Thread is a basic unit of execution so each program may have a number of processes associated with it and each process can have a number of threads executing in it. Thread is a basic unit of CPU utilization. Thread are designed to reduce time for context switching. Thread usage created concurrency within a process.

1.2 Answer of Second Question

2. Name and describe two process scheduling policies. Which one is preferable and does the choice of policies have any influence on the behavior of Java threads? (2 marks)

   Answer:

   1. Co-operative scheduling

      in co-operative scheduling, the OS never initiates a context transfer from a running process to another process. For it to function, all functions must cooperate. It can hog the CPU if one program does not cooperate. Some examples are Macintosh OS version 8.0-9.2.2 and Windows 3.x OS.

   2. Pre-emptive Scheduling

      in pre-emptive scheduling, the OS determines how long a task should be performed until authorizing another task to use the OS. It forces programs to share CPU if they want to or not. Some examples are UNIX, Windows 95, Windows NT operating systems.

1.3 Answer of Third Question

3. Distinguish between Centralized and Distributed systems? (2 marks)

Answer:

Client/server architecture framework where a single or more than one client is directly linked to a central server is Centralized systems. This type of systems are widely used in many companies where clients send a request to a company server to receive the response. It's convenient to be physically secure. It is less costly for small systems to set up and can be update in short period. System may collapse there is disconnection on nodes. Data backup possibility is less and server maintenance is hard.

A distributed systems are groups of independent computers which appears to be computers as a single cohesive structure, consumer. Distributed system networked machine modules communicates and organizes their tasks by transferring messages only. It is possible to add computing power in small quantities increments. It allows multiple people to access and share database. It is easy to access also applies to hidden data. It can cause networking problem.

1.4 Answer of Fourth Question

4. Explain transparency in D S? (2 marks)

Answer:

Transparency refers hiding something. It is major issue for realizing the single image of the system which prepare system as simple to use as a single processor scheme. Classification of Transparency are access transparency, location transparency, migration transparency, replication transparency, concurrency transparency, failure transparency, performance transparency, Scaling transparency.

1.5 Answer of Fifth Question

5. The following three statements contain a flow dependency, an antidependency 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. (4 marks)

1. Flow Dependency:

B=A+C

B=C+D

C=B+D

2. Antidependency:

B=A+C

B=C+D

C=B+D

3. Output dependency:

B=A+C

B=C+D

C=B+D

4. Removing above dependency by making temporary variable:

Btemp=A+C

Ctemp=Btemp+D

B=Ctemp+D

Note: Show all the works in your report and produce a simple C code simulate the process of producing the C and B values. (2marks for solving dependencies and 2marks for the code)

```
#include <stdio.h>

int main()

{

        int A,B,C,D;

        A=2;

        B=4;

        C=6;

        D=8;


        int Btemp = B;

        int Ctemp = C;

        Btemp = A + C;

        B= Ctemp + D;

        C = B + D;

        printf("Output:\n");

        printf("%d %d %d %d", A, B, C, D);

}
```

Output:

```
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW$ ./task1
Output:
2 14 22 8shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW$
```

1.6 Answer of Sixth Question

6. What output do the following 2 programs produce and why? (3 marks)

```
shakti@shakti-HP-Laptop-15-da0xxx: ~/Desktop/PasswordEncrypCrack
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$ cc -o 02 02.c -lcrypt -pthread
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$ ./02
500000
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$ ./02
500000
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$ cc -o 01 01.c -lcrypt -pthread
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$ ./01
175925
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$ ./01
235140
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$
```

Reason:

In program 01.c, same variable are manipulate by every thread at same time. Thread generates values which override by another thread values which leads to random value between 100000-500000.

In program 02.c, each thread runs sequentially until one thread does not finish another will not start and produce same result i.e 500000.

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

2.1 Single Thread Matrix Multiplication

Study the following algorithm that is written for multiplying two matrices A and Band storing the result in C.

```
int A[N][P], B[P][M], C[N][M];

for (int i = 0; i < N; i++)

    {

      for (int j = 0; j < M; j++)

        {

      C[i][j] = 0;

            for (int k = 0; k < P; k++)

            {

            C[i][j] = C[i][j] + A[i][k] * B[k][j];

            }

      }

    }
```

- The analysis of the algorithm's complexity. (1 mark)
  Answer:
  The complexity of this algorithm is $O(n^3)$ because it runs three loops.

- 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)

  Answer:

  Any three different ways to speed up the matrix multiplication given here are:

  1. Utilisation of cache memory
  2. Using CUDA
  3. Using Threads

- 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)

  Answer:

```
Paste your algorithm's pseudo code here

for(s=0;s<M;s++){

        for(p=0;p<P;p++){

            for(d=0;d<N;d++){

                c[s][d]=c[s][d]+a[s][p]*b[p][d];

            }

        }

    }
```

- 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

Answer: C program that implements matrix multiplication using above versions:

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

int main()
{
         int size;
        printf("\nEnter size of Matrix\n");
        scanf("%d", &size);
        int N[size][size], M[size][size], P[size][size];
        int s, p, d;
        //printf("\nEnter Firt Matrix\n");
        for(s=0; s<size; s++) {
         for(p=0; p<size; p++) {
          N[s][p]=rand() % 50;
           }
        }

        for(s=0; s<size; s++) {
         for(p=0; p<size; p++) {
          M[s][p]=rand() % 50;
         }
        }

 clock_t begin = clock();

        for(s=0; s<size; s++) {
         for(p=0; p<size; p++) {
          P[s][p]=0;
          for(d=0; d<size; d++) {
                P[s][p]=P[s][p] + N[s][d] * M[d][p];
           }
```

```
            }
          }
          printf("\nThe results is…\n");
          for(s=0; s<size; s++) {
           for(p=0; p<size; p++) {
            }
          }
          clock_t end = clock();
   double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
   printf("Time elapsed was :%lfs\n", time_spent);
   return 0;
}
```

Output:

```
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartA$ ./singleMatrix1

Enter size of Matrix
500

The results is…
Time elapsed was :0.481776s
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartA$
```

Answer: C program that implements matrix multiplication using improved versions:

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

int main()
{
        int size;
        printf("\nEnter size of Matrix\n");
        scanf("%d", &size);
        int N[size][size], M[size][size], P[size][size];
        int s, p, d;
        //printf("\nEnter Firt Matrix\n");
        for(s=0; s<size; s++) {
         for(p=0; p<size; p++) {
          N[s][p]=rand() % 50;
          }
        }

        for(s=0; s<size; s++) {
         for(p=0; p<size; p++) {
          M[s][p]=rand() % 50;
         }
        }

   clock_t begin = clock();

        for(s=0; s<size; s++) {
         for(p=0; p<size; p++) {
          for(d=0; d<size; d++) {
                P[s][p]=P[s][p] + N[s][d] * M[d][p];
           }
          }
        }
        printf("\nThe results is…\n");
        for(s=0; s<size; s++) {
         for(p=0; p<size; p++) {
```

```
            }
        }
        clock_t end = clock();
  double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
  printf("Time elapsed was :%lfs\n", time_spent);
  return 0;
}
```

Output:

```
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartA$ ./singleMatrix2

Enter size of Matrix
500

The results is…
Time elapsed was :0.480238s
```

- 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)
  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.
  Answer:

	Original algorithm	Improved algorithm
Size of matrix	500	500
Time elapsed was	0.481776s	0.480238

Algorithm such as Strassen Algorithm have time complexity is O($n^{2.8}$) which is related to improved algorithm. The average time would be reduced to the potential use of it, so the average time would be around 9.685. As we know, Multithread execute on number of threads. The time for matrix number lookups would be minimized by the successful use of cache memory.

2.2 Multithreaded Matrix Multiplication

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

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

#define MAX_SIZE 1024

typedef struct parameters
{
  int s, p;
} args;

int matrix1[MAX_SIZE][MAX_SIZE];
int matrix2[MAX_SIZE][MAX_SIZE];
int result[MAX_SIZE][MAX_SIZE];

int maxThread;

void *multiply(void *arg)
{
  args *parm = arg;

  for (int b = 0; b < MAX_SIZE; b++)
  {
    result[parm->s][parm->p] += matrix1[parm->s][b] * matrix2[b][parm->p];
  }
}

void initializeMatrix()
{
  for (int l = 0; l < MAX_SIZE; l++)
  {
    for (int m = 0; m < MAX_SIZE; m++)
    {
      matrix1[l][m] = rand() % 10;
      matrix2[l][m] = rand() % 10;
```

```
      result[l][m] = 0;
    }
  }
}

void startThread()
{

  int s = 0, p = 0;

  while (s < MAX_SIZE)
  {
    p = 0;
    while (p < MAX_SIZE)
    {
      pthread_t threads[maxThread];
      args parm[maxThread];

      for (int l = 0; l < maxThread; l++)
      {

        if (p >= MAX_SIZE)
        {
          break;
        }

        parm[l].s = s;
        parm[l].p = p;
        pthread_create(&threads[l], NULL, multiply, (void *)&parm[l]);

        pthread_join(threads[l], NULL);

        p++;
      }
    }
    s++;
  }
}
```

```c
int main()
{
  initializeMatrix();
  printf("Enter no. of threads:");
  scanf("%d", &maxThread);
  clock_t begin = clock();

  startThread();

  clock_t end = clock();
  double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
  printf("Time :%lf \n", time_spent);
}
```

Output:

```
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ gcc -o matrix  MatrixMultiThread.c  -lpthread
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ ./matrix
Enter no. of threads:50
Time :34.082248
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ ./matrix
Enter no. of threads:100
Time :31.099819
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ ./matrix
Enter no. of threads:150
Time :33.114493
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ ./matrix
Enter no. of threads:200
Time :33.787409
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ ./matrix
Enter no. of threads:250
Time :33.648982
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ ./matrix
Enter no. of threads:300
Time :34.712882
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ ./matrix
Enter no. of threads:350
Time :32.190562
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ ./matrix
Enter no. of threads:400
Time :37.118419
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ ./matrix
Enter no. of threads:450
Time :32.955536
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$ ./matrix
Enter no. of threads:500
Time :62.075556
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/HPC-CW/Task2-PartB$
```

- 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.

  Table:

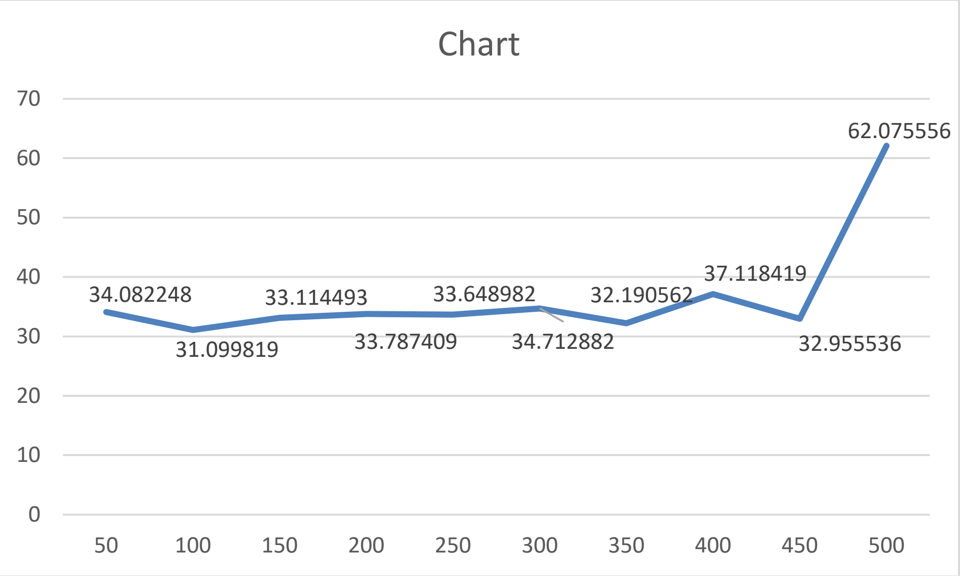
  Original program:

	Original algorithm
Size of matrix	500
Time elapsed was	0.481776s

Multithread version:

Number of threads	Time(sec)
50	34.082248
100	31.099819
150	33.114493
200	33.787409
250	33.648982
300	34.712882
350	32.190562
400	37.118419
450	32.955536
500	62.075556
Total	364.785906
Mean	36.4785906

Chart:



- Insert an explanation of the results presented in the above table.

The mean time of multi-threaded matrix multiplication was 36.4785906.

Typically, the use of the multithread improves the program's performance, but often if the thread number exceeds thread number needed, the performance ma y be lowered as the excess number of threads searches for operations. If the thread number is less than the thread number needed, then it will take longer to co mplete the task. At 500 thread numbers, the minimum time taken by a program is 62.07s, and at range (50-450) thread numbers, the limit is 32s.

We can evaluate from that table the optimal solution can be obtained at the use of 1 to 450 thread numbers.

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

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

int numberOfPasswords = 1;

char *passwordsEncrypted[] = {
"$6$AS$3geXjAVSaQZLWLz2NUF.JwY0GsFB6JHaXukmrX9z2Q08jy5RdJtvMyI0rVdCyf56qCfRH7h/fB4KAptcOPJm01"
};


void substr(char *dest, char *src, int start, int length){
  memcpy(dest, src + start, length);
  *(dest + length) = '\0';
}
void startCracking(char *salt_and_encrypted){
  int a, b, c;
  char salt[7];
  char plain[7];
  char *enc;
  int count = 0;
  substr(salt, salt_and_encrypted, 0, 6);
```

```
for(a='A'; a<='Z'; a++){
    for(b='A'; b<='Z'; b++){
      for(c=0; c<=99; c++){
        sprintf(plain, "%c%c%02d", a,b,c);
        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);
}
```

```c
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<numberOfPasswords;i<i++)
        {
                startCracking(passwordsEncrypted[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;
}
```

- Insert a table of 10 running times and the mean running time.

```
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$ cc -o CrackAZ99 CrackAZ99.c -lcrypt
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$ ./CrackAZ99 > pass.txt
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$ chmod a+x mr.py
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$ python3 ./mr.py ./CrackAZ99| grep Time
Time elapsed was 236055347326ns or 236.055347326s
Time elapsed was 193257553510ns or 193.257553510s
Time elapsed was 190250406032ns or 190.250406032s
Time elapsed was 186311705741ns or 186.311705741s
Time elapsed was 186360928245ns or 186.360928245s
Time elapsed was 187302280628ns or 187.302280628s
Time elapsed was 193265191047ns or 193.265191047s
Time elapsed was 207719815314ns or 207.719815314s
Time elapsed was 204894921669ns or 204.894921669s
Time elapsed was 203752859102ns or 203.752859102s
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack$
```

Table:

Time elpased	Time in nanosec	Time in sec
	236055347326.00	236.06
	193257553510.00	193.26
	190250406032.00	190.25
	186311705741.00	186.31
	186360928245.00	186.36
	187302280628.00	187.30
	193265191047.00	193.27
	207719815314.00	207.72
	204894921669.00	204.89
	203752859102.00	203.75
SUM	1989171008614.00	1989.17
MEAN	361667456111.64	198.92

- 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.

The average time obtain from two characters and two integers was 198.92 seconds. It will obviously the run time will be exceed enormously. Assume that, two initial have two loops whereas as three initial have three loops then the run time may take 20 times more than two initial. Run time may be 198.92*20 i.e 3978.4.

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

Output:

```
1263913 #1263913 SSD12 $6$AS$uQfmhr0I9ChtjndeTewPVuOgb5LDUmHQzpLPpG7IGWcIj/k.12vvheH/TFO8s1H/yQMncQ7O4NljfcST1FeMc0
1757601 1757600 solutions explored
1757602 Time elapsed was 4873186175760ns or 4873.186175760s
```

```
Activities   Terminal ▾                                  दिसम्बर 24  10:
                                 shakti@shakti-HP-Laptop-15-da0xxx: ~
shakti@shakti-HP-Laptop-15-da0xxx:~$ cd Desktop
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop$ cd PasswordEncrypCrack1
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$ cc -o 3initials 3initials.c -lcrypt
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$ ^C
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$ cc -o 3initials 3initials.c -lcrypt
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$ python3 ./mr.py ./3initials |grep Time
Time elapsed was 5184697591180ns or 5184.697591180s
Time elapsed was 4211265223650ns or 4211.265223650s
Time elapsed was 3702883532664ns or 3702.883532664s
Time elapsed was 3655948391437ns or 3655.948391437s
```

```
Time elapsed was 3508520285800ns or 3508.520285800s
Time elapsed was 3521491737299ns or 3521.491737299s
Time elapsed was 3787463483219ns or 3787.463483219s
Time elapsed was 3662762748069ns or 3662.762748069s
Time elapsed was 3669358925272ns or 3669.358925272s
Time elapsed was 3674296581350ns or 3674.296581350s
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$
```

Table:

Time elpased	Time in nanosec	Time in sec
	5184697591180.00	5184.70
	4211265223650.00	4211.27
	3702883532664.00	3702.88
	3655948391437.00	3655.95
	3508520285800.00	3508.52
	3521491737299.00	3521.49
	3787463483219.00	3787.46
	3662762748069.00	3662.76
	3669358925272.00	3669.36
	3674296581350.00	3674.30
SUM	38578688499940.00	38578.69
MEAN	7014306999989.09	3857.87

Answer: The average time of 3 initial (three character and two integer) was 3857 seconds which is approximately close to 3978.4. I have passed only one encrypted password in both 2 initial and 3 initials programs because it will long and difficult to run whole day. In two initial, mean time was 198.92 seconds whereas 3 initial took 3857.87 seconds to crack the password (SSD12).

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

int numberOfPasswords = 1;

char *encryptedPasswords[] = {
"$6$AS$uQfmhr0I9ChtjndeTewPVuOgb5LDUmHQzpLPpG7IGWcIj/k.12vvheH/TFO8s1H/yQMncQ7O4NljfcST1FeMc0"
};

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

void startingCrack(char *salt_and_encrypted){
  int a, b, c,d;
  char salt[7];
  char plain[7];
  char *enc;
  int count = 0;

  substr(salt, salt_and_encrypted, 0, 6);

  for(a='A'; a<='Z'; a++){
    for(b='A'; b<='Z'; b++){
      for(c='A'; c<='Z'; c++){
       for(d=0; d<=99; d++){
        sprintf(plain, "%c%c%c%02d", a,b,c,d);
        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<numberOfPasswords;i<i++)
        {
                startingCrack(encryptedPasswords[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;
}
```

- Write a paragraph that compares the original results with those of your multithread password cracker.

```
shakti@shakti-HP-Laptop-15-da0xxx: ~/Desktop/PasswordEncrypCrack1
shakti@shakti-HP-Laptop-15-da0xxx:~$ cd Desktop
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop$ cd PasswordEncrypCrack1
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$ cc -o Th CrackThread.c -lcrypt -pthread
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$ chmod a+x mr.py
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$ cc -o Thread Thread.c -lcrypt -pthread
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$ chmod a+x mr.py
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$ python3 ./mr.py ./Thread |grep Time
Time elapsed was 179021323212ns or 179.021323212s
Time elapsed was 195117175859ns or 195.117175859s
Time elapsed was 192443832358ns or 192.443832358s
Time elapsed was 208629487557ns or 208.629487557s
Time elapsed was 226678899451ns or 226.678899451s
Time elapsed was 192228155126ns or 192.228155126s
Time elapsed was 191228342716ns or 191.228342716s
Time elapsed was 240673275354ns or 240.673275354s
Time elapsed was 190490762483ns or 190.490762483s
Time elapsed was 178894339372ns or 178.894339372s
shakti@shakti-HP-Laptop-15-da0xxx:~/Desktop/PasswordEncrypCrack1$
```

Table:

Time elapsed	Time in nanosec	Time in sec
Time elapsed was	179021323212.00	179.02
Time elapsed was	195117175859.00	195.12
Time elapsed was	192443832358.00	192.44
Time elapsed was	208629487557.00	208.63
Time elapsed was	226678899451.00	226.68
Time elapsed was	192228155126.00	192.23
Time elapsed was	191228342716.00	191.23
Time elapsed was	240673275354.00	240.67
Time elapsed was	190490762483.00	190.49
Time elapsed was	178894339372.00	178.89
SUM	1995405593488.00	1995.41
MEAN	362801016997.82	199.54

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

int noOfPasswords = 1;

char *encryptedPasswords[] = {

"$6$AS$3geXjAVSaQZLWLz2NUF.JwY0GsFB6JHaXukmrX9z2Q08jy5RdJtvMyI0rVdCyf56qCfRH7h/fB4KAptcOPJm01"
};


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


void crackwithThread()
{
  int a;
pthread_t th1, th2;

    void *kernel_function_1();
    void *kernel_function_2();
for(a=0;a<noOfPasswords;a<a++) {


    pthread_create(&th1, NULL,kernel_function_1, encryptedPasswords[a]);
    pthread_create(&th2, NULL,kernel_function_2, encryptedPasswords[a]);

    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
 }
}
```

```
void *kernel_function_1(char *salt_and_encrypted){
  int s, a, n;
  char salt[7];
  char plain[7];
  char *enc;
  int count = 0;

  substr(salt, salt_and_encrypted, 0, 6);

  for(s='A'; s<='M'; s++){
    for(a='A'; a<='Z'; a++){
      for(n=0; n<=99; n++){
        enc = (char *) crypt(plain, salt);
        count++;
      }
    }
  }
  printf("%d solutions explored\n", count);
}

  for(x='N'; x<='Z'; x++){
    for(y='A'; y<='Z'; y++){
      for(z=0; z<=99; z++){
        enc = (char *) crypt(plain, salt);
        count++;
      }
    }
  }
  printf("%d solutions explored\n", count);
}
```

```
void *kernel_function_2(char *salt_and_encrypted){
  int x, y, z;      // 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(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);
        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[])
{

        struct timespec start, finish;
        long long int time_elapsed;

        clock_gettime(CLOCK_MONOTONIC, &start);

                crackwithThread();

        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;
}
```

The average time took by multithread version is 199.54 which took more time than 2 initial (original version) that is 198.2. Execution of the program cannot be faster if we use more threads. Threads will perform better on a specific tasks. Assuming that, this multithread version is suitable to execute which took more runtime.

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

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

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

__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 *decrypted)
{

  char genRawPass[4];

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

  genRawPass[2] = numbers[threadIdx.x];
  genRawPass[3] = numbers[threadIdx.y];
  int i;
  bool m = true;

  char *encryp = CudaCrypt(genRawPass);
```

```
for (i = 0; i < 11; i++)
  {
    if (encryp[i] != decrypted[i])
    {
      m = false;
      break;
    }
  }
  if (m)
  {
    printf("%s\n", genRawPass);
  }
}

int main(int argc, char **argv)
{
  char *decryptedPassword = argv[1];
  char *gpuDecryptedPass;

  cudaMalloc((void **)&gpuDecryptedPass, sizeof(char) * 11);
  cudaMemcpy(gpuDecryptedPass, decryptedPassword, sizeof(char) * 11, cudaMemcpyHostToDevice);
  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, gpuDecryptedPass);
  cudaThreadSynchronize();
  return 0;
}
```

Output:

```
com136@herald-OptiPlex-3050:~/Cuda$ cc -o sh cudaCrypt.c
com136@herald-OptiPlex-3050:~/Cuda$ ./sh sd12
uqtgac3162
com136@herald-OptiPlex-3050:~/Cuda$ ^C
com136@herald-OptiPlex-3050:~/Cuda$ nvcc -o shakti PasswordCrack.cu
com136@herald-OptiPlex-3050:~/Cuda$ ./shakti uqtgac3162
sd12
Time elapsed was 191342024ns or 0.191342024s
com136@herald-OptiPlex-3050:~/Cuda$ ./mr.py ./shakti uqtgac3162 |grep Time
Time elapsed was 181894767ns or 0.181894767s
Time elapsed was 159473366ns or 0.159473366s
Time elapsed was 161184289ns or 0.161184289s
Time elapsed was 161419618ns or 0.161419618s
Time elapsed was 165978566ns or 0.165978566s
Time elapsed was 161064426ns or 0.161064426s
Time elapsed was 163678248ns or 0.163678248s
Time elapsed was 158353102ns or 0.158353102s
Time elapsed was 164700512ns or 0.164700512s
Time elapsed was 163569755ns or 0.163569755s
com136@herald-OptiPlex-3050:~/Cuda$
```

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

TABLE:

Time elapsed	Time in nanosec	Time in sec
Time elapsed was	181894767.00	0.18
Time elapsed was	159473366.00	0.16
Time elapsed was	161184289.00	0.16
Time elapsed was	161419618.00	0.16
Time elapsed was	165978566.00	0.17
Time elapsed was	161064426.00	0.16
Time elapsed was	163678248.00	0.16
Time elapsed was	158353102.00	0.16
Time elapsed was	164700512.00	0.16
Time elapsed was	163569755.00	0.16
SUM	1641316649.00	1.64
MEAN	298421208.91	0.16

- Write a short analysis of the results

The average time of password cracking with CUDA is 0.16 whereas 2 initial is 198.92 and multithread is 199.54. The run time with CUDA is much lot faster than both of above. CUDA works parallel which helps in faster execution. It enables to speed up the execution by harnessing the power of GPUs for the parallelizable part of the computation.

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
  cuda_blur.cu

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

#include "lodepng.h"



__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[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 && 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];
        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++;


    }
```

```
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[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[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 main(int argc, char **argv){

        unsigned int error;
        unsigned int encError;
        unsigned char* image;
        unsigned int width;
        unsigned int height;
        const char* filename = "Me.png";
        const char* newFileName = "pic.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);
```

```
        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));
        }

        cudaFree(d_in);
        cudaFree(d_out);

        return 0;
}
```

```
com176@herald-OptiPlex-3050:~/HPC$ nvcc -o pic cuda_blur.cu lodepng.cpp
com176@herald-OptiPlex-3050:~/HPC$ ./img
com176@herald-OptiPlex-3050:~/HPC$
```

Input Image:

output Image: