# Práctica 1 Hilos

León Reyes Marcos
Morales Garcia Christian Arturo
Piñón Caballero Ángel Ramón

**Introducción**

Esta práctica consta de 2 incisos.

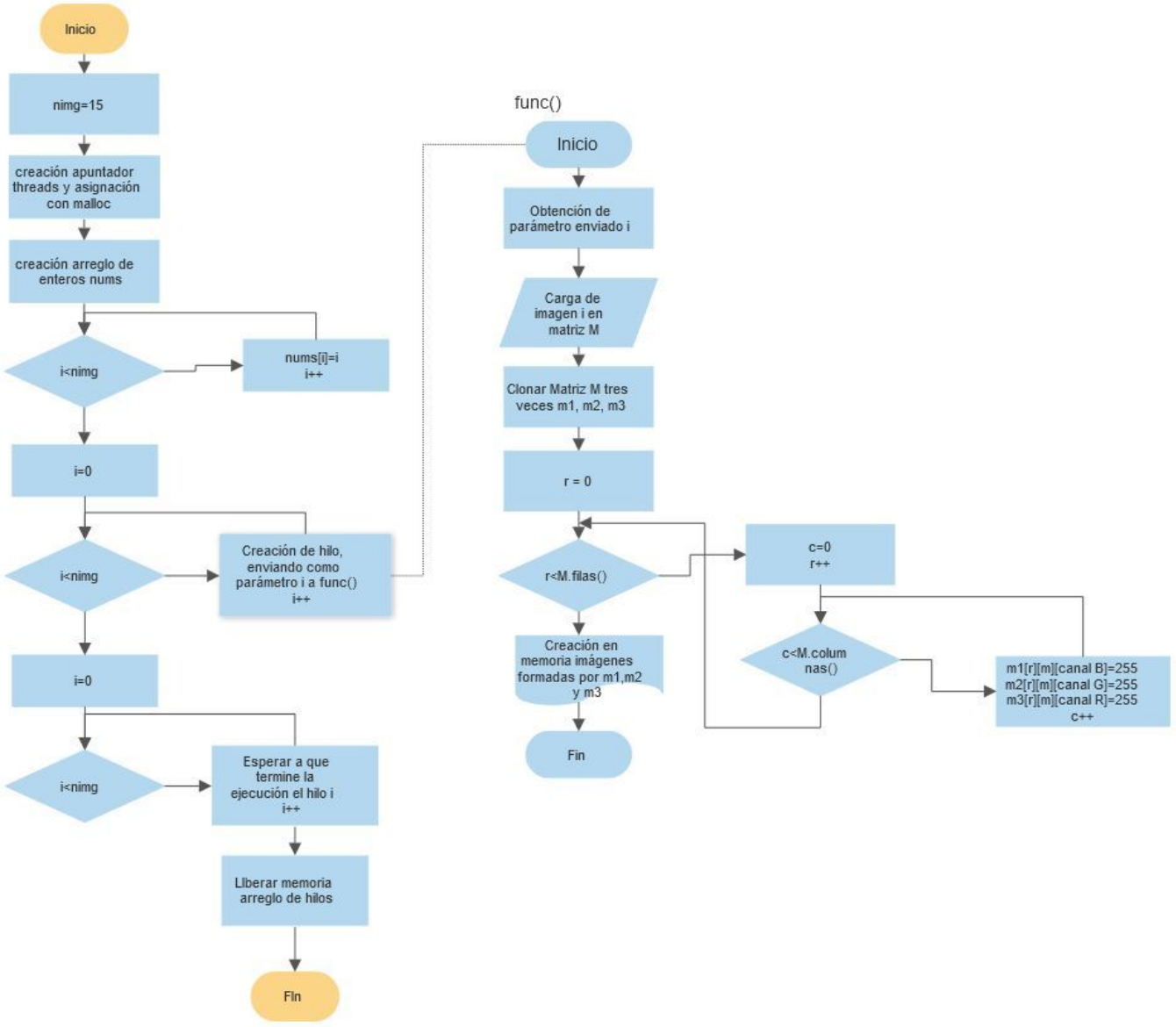
A) Dentro de un directorio copiar +10 imágenes, cada una de estas imágenes van a ser asignadas a un hilo. El hilo va a abrir la imagen como una matriz RGB y va agenerar 3 copias de dicha imagen, la primera copia va a asignar el valor R a 255, en la segunda copia va a asignar el valor G a 255 y en la tercera copia va a asignar el valor B a 255. Cada una de las copias modificadas serán guardadas nuevamente como una imagen, lo cual provocará que al final se tengan por cada imagen original 3 más una saturada en Red otra saturada en Green y otra saturada en Blue.
B) Generar la multiplicación de matrices donde A = 2000 *1500  B= 1500 * 2000 . Se van a ejecutar diferentes cantidades de hilos que resuelvan esta matriz, en cada ejecución se deben medir los tiempos y al final se debe generar una conclusión: que sucede cuando mas hilos ejecutan la matriz, vale la pena crear grandes cantidades de hilos para resolver el problema? La cantidad de hilos a ejecutar es 2,4,8,16,32

En esta práctica se nos pide que trabajemos con 2 ejercicios en los cuales tendremos que notar algunas característica al momento de trabajar con hilos. Lo primero que tenemos que tener en cuenta es que a la hora de trabajar con aplicaciones que deben realizar operaciones complejas, lo ideal es tener varios flujos de ejecución lo cual se logra utilizando hilos, donde cada hilo representaría una tarea. Esto permite realizar una tarea sin necesidad de esperar a las otras, porque en un flujo normal las tareas se realizan de forma secuencial, es decir, las tareas se ejecutarán una después de la otra. Entonces, si se está ejecutando una tarea que tarda demasiado esto impide ejecutar otra hasta que el proceso termine. En esta ocasión se nos pone un ejercicio de multiplicación de matrices donde se nos pide que se introduzcan 2 matrices y un número de hilos, estos hilos se dividirán el trabajo para calcular el producto de las matrices.

Un punto a tener en cuenta a la hora de trabajar con los hilos es la sincronización esto para que no alteren datos que usan otros hilos ni interfieran con datos que están siendo usados. Todos los hilos comparten el mismo espacio de direcciones y otros recursos como pueden ser archivos abiertos y cualquier modificación de un recurso desde un hilo afecta al entorno del resto de los hilos del mismo proceso.

## Parte 1

## Diagrama de flujo



**Desarrollo**

Se incluyen las librerías de OpenCV así como la librería para trabajar con hilos**.**

```
#include<opencv2/core.hpp>
#include<opencv2/imgcodecs.hpp>
#include<opencv2/highgui.hpp>
#include<opencv2/imgproc.hpp>
#include<iostream>

#include<pthread.h>
```

Se especifica que se utilizará el espacio de nombres cv, el cual corresponde a la librería OpenCV. De la misma manera se especifica que se utilizará cout y endl del espacio de nombres std.

```
using namespace cv;
using std::cout;
using std::endl;
```

**main()**

Se define el número de hilos.

```
int nimg=15;
```

Se crea un arreglo de hilos de tamaño nimg, asignando memoria dinámica al apuntador threads.

```
pthread_t *threads;
threads = (pthread_t *)malloc(sizeof(pthread_t)*nimg);
```

Se crea un arreglo de enteros y se llena con los valores de i.

```
int nums[nimg];
for(int i=0;i<nimg;i++)nums[i]=i;
```

Se crean nimg hilos empleando la función pthread_create(), la cual recibe como parámetros el identificador del hilo, la función a ejecutar por el hilo (func), y la referencia al elemento i del arreglo nums.

```
for(int i=0;i<nimg;i++){
    pthread_create(&threads[i], NULL, func,(void*)&nums[i]);
}
```

Se ejecuta un ciclo para esperar a que los nimg hilos terminen su ejecución, recibiendo como parámetro el identificador del hilo i del arreglo threads.

```
for(int i=0; i<nimg; i++){
    pthread_join(threads[i], NULL);
}
```

**func()**

Se castea el apuntador de tipo void a int y se desreferencia el valor. Este valor es el número de imagen que modificará el hilo.

```
void * func(void * arg){

    int n=*((int*)arg);
```

Se generan 4 arreglos de caracteres, de los cuales el primero contiene el path de donde se obtiene la imágen principal y los tres siguientes  el path de donde se almacenarán las tres imágenes resultantes.

```
char pathmain[30]="./main/";
char pathb[30]="./blue/";
char pathg[30]="./green/";
char pathr[30]="./red/";
```

Se crean dos arreglos que servirán para construir tanto el nombre de la imagen que se va a modificar como el nombre de las tres nuevas imágenes.

```
char fname[20]="img", ext[5]=".jpg";
```

Se crea un arreglo de caracteres y se almacena como una cadena el número n recibido como parámetro.

```
char num[5];
sprintf(num,"%d",n);
```

Se genera el path  completo de donde se almacena el recurso original y los tres paths donde se almacenarán las nuevas imágenes.

```
/*path recurso origen*/
strcat(fname,num);
strcat(fname,ext);
strcat(pathmain,fname);

/*path recurso destino*/
strcat(pathb,fname);
strcat(pathg,fname);
strcat(pathr,fname);
```

Se lee una imagen ubicada en pathmain, empleando el formato IMREAD_COLOR, formato BGR, y se almacena en el objeto de tipo Mat.
Nota: La librería OPENCV invierte la configuración RGB a BGR.

```
Mat img=imread(pathmain,IMREAD_COLOR);
```

En caso de no encontrar la imagen ubicada en pathmain se lanza una alerta y se termina la ejecución del programa.

```
if(img.empty()){
    cout<<"could not read the image: "<<fname<<endl;
    exit(1);
}
```

Se clona la img tres veces empleando la función clone().

```
Mat img1 = img.clone();
Mat img2 = img.clone();
Mat img3 = img.clone();
```

Se generan dos ciclos for, el primero va a iterar sobre las filas (row) de la matriz almacenada en el objeto img. El segundo ciclo iterará sobre las columnas (col) de la matriz almacenada en el objeto img.
Se emplea un Vec3b o vector de tres bytes, el cual se recibe como argumento en la función at(). Con esto se puede acceder al pixel en la posición (row, col) y en el canal 0, 1 o 2. El canal 0 corresponde al Blue, el 1 al Green y el 2 al Red.
Finalmente se satura el valor del canal 0, 1, y 2, dependiendo de la imagen. Para esto se lleva el valor a 255.

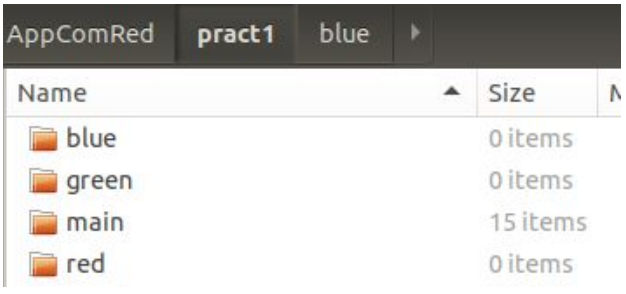
```
for (int row = 0; row < img1.rows; row++){
    for (int col = 0; col < img1.cols; col++){
        /*dependiento de la imagen se satura el
        img1.at<Vec3b>(row, col)[0] = 255;
        img2.at<Vec3b>(row, col)[1] = 255;
        img3.at<Vec3b>(row, col)[2] = 255;
    }
}
```

Se escriben las imágenes(contenidas en el objeto img1, img2 y img3) en la ruta especificada por el primer parámetro de la función imwrite().

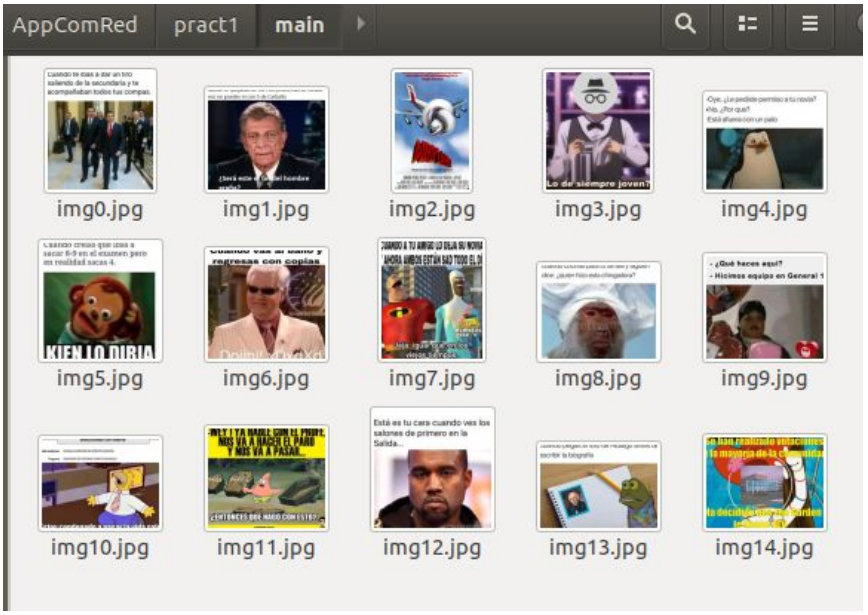
```
imwrite(pathb,img1);
imwrite(pathg,img2);
imwrite(pathr,img3);
```

**Pruebas**

Como se observa, la carpeta main tiene 15 imágenes y las otras se encuentran vacías.



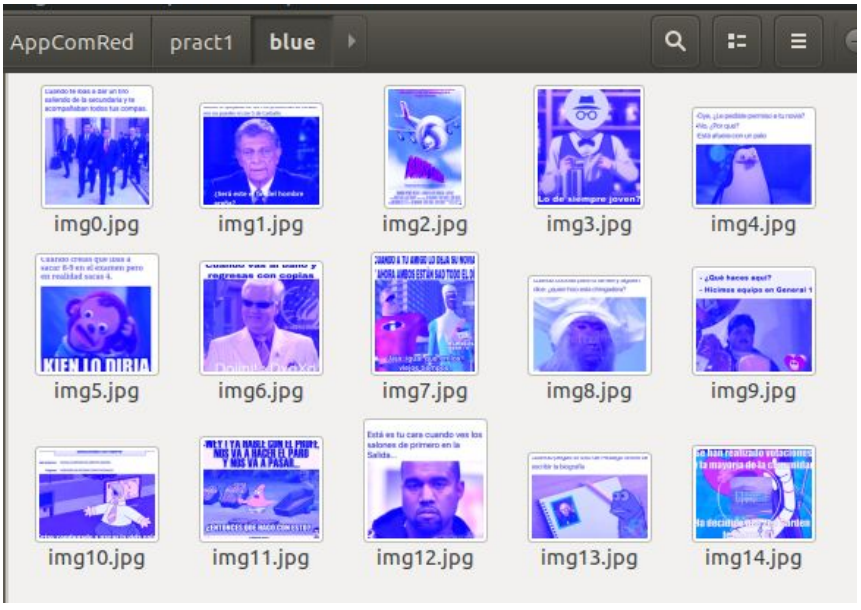
Carpeta main:




Compilación y ejecución:

```
marcos@desktop:~$ cd AppComRed/
marcos@desktop:~/AppComRed$ cd pract1/
marcos@desktop:~/AppComRed/pract1$ g++ `pkg-config --cflags opencv` main.cpp `pkg-config --libs opencv` -lpthread
marcos@desktop:~/AppComRed/pract1$ ./a.out
marcos@desktop:~/AppComRed/pract1$
```

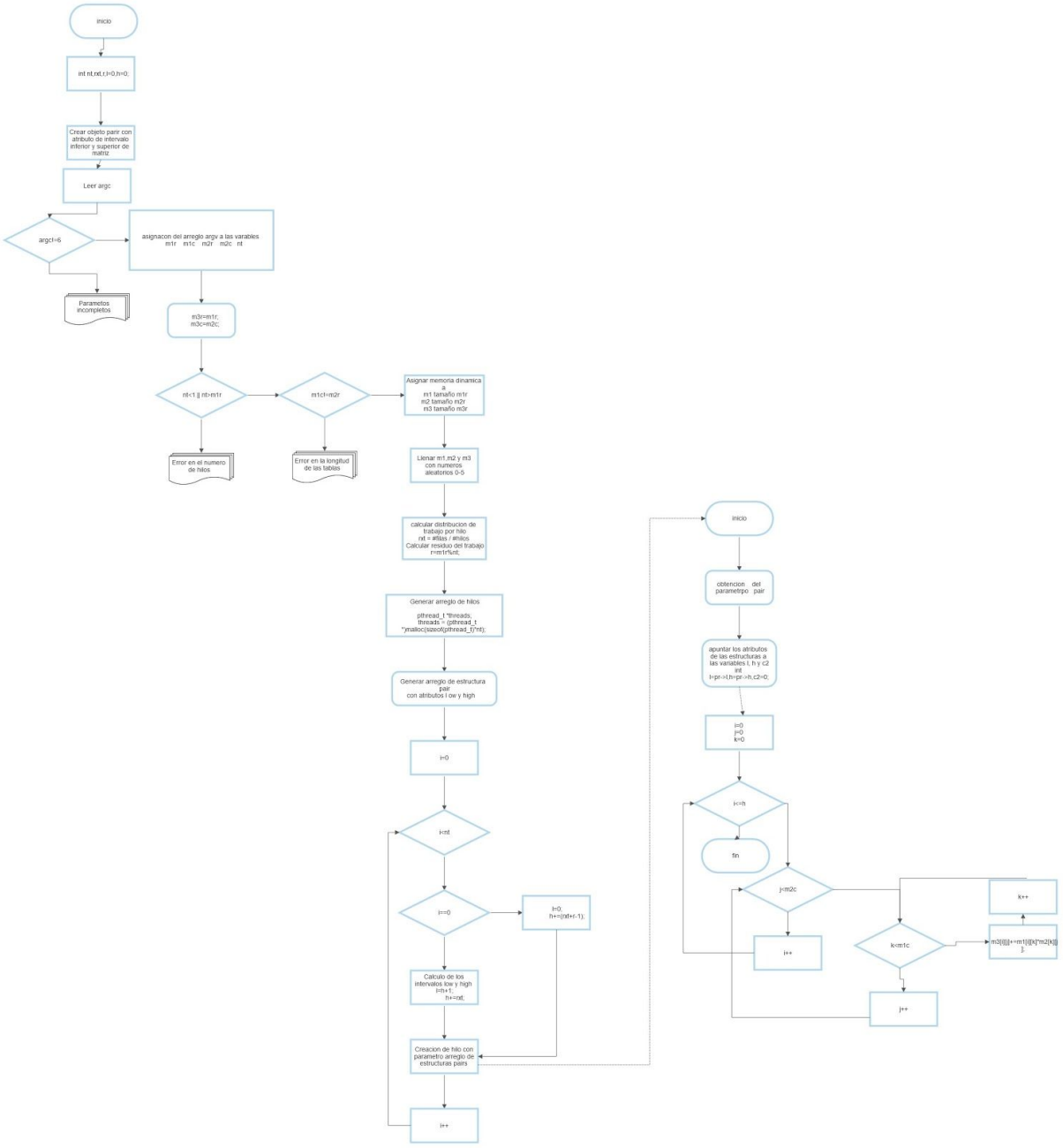
Carpeta blue:



Carpeta Green:


Carpeta Red:

# Parte 2
# Diagrama de flujo



**Desarrollo**

Inclusión de las librerías que permite trabajar con los hilos

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
```

Se crean de manera global tres doble apuntadores a enteros, los cuales servirán para trabajar con la matriz 1, matriz 2  y la matriz 2. Al declararlos de manera global todos los hilos tiene acceso a ellos.

```c
int **m1,**m2,**m3;
```

Se crean de manera global seis variables, las cuales servirán para almacenar la longitud de las matrices 1,2 y 3 (filas y columnas).

```c
int m1r,m1c,m2r,m2c,m3r,m3c;
```

Se crea un struct nombrado pair, el cual contiene dos enteros que representan el índice bajo (l) y alto (h). Estos índices le indicarán a los hilos sobre cuáles filas de la matriz va a operar.

```c
typedef struct {
    int l;
    int h;
}pair;
```

**main()**

Se valida que el número de parámetros obtenidos desde la terminal sean los indicados, en caso contrario se termina el programa.

```c
if(argc!=6){
    printf("parametros incompletos\n");
    exit(0);
}
```

Se toman los parámetros recibidos desde la terminal, se castean a enteros y se asignan para formar las las dos matrices a multiplicar.

Aprovechando la propiedad de que una matriz de m*n por una de n * k da como resultado una matriz de m *  k , se retoman esos valores de las matrices a multiplicar para crear las longitudes  de la matriz resultado.

```
m1r=atoi(argv[1]);
m1c=atoi(argv[2]);
m2r=atoi(argv[3]);
m2c=atoi(argv[4]);
m3r=m1r;
m3c=m2c;
```

De la misma terminal se recibe el número de hilos a utilizar.

```
nt=atoi(argv[5]);
```

Se valida que el número de hilos no sea cero o mayor al número de filas de la primera matriz. En caso contrario se termina el programa.

```
if(nt<1 || nt>m1r){
    printf("error en el # de hilos");
    exit(0);
}else if(m1c!=m2r){
    printf("error en la longitud de las tablas");
    exit(0);
}
```

Se asigna memoria dinámica a los dobles punteros definidos de manera global. En realidad se crea un arreglo de apuntadores de tipo entero, teniendo como longitud el número de filas de cada matriz.

```
  m1 = (int **)malloc(sizeof(int*)*m1r);
  m2 = (int **)malloc(sizeof(int*)*m2r);
  m3 = (int **)malloc(sizeof(int*)*m3r);
```

Se asigna memoria dinámica a cada apuntador contenido en el arreglo de doble apuntador, el cual conforma en su conjunto a la matriz m1. Además se asigna un valor aleatorio entre 0 y 5 a la posición i,j de la matriz.

```
for(int i=0;i<m1r;i++){
    m1[i]=(int*)malloc(sizeof(int)*m1c);
    for(int j=0;j<m1c;j++){
        int rr=rand()%6;
        m1[i][j]=rr;
    }
}
```

Para la matriz m2 ocurre lo mismo que con la matriz m1.

```
for(int i=0;i<m2r;i++){
    m2[i]=(int*)malloc(sizeof(int)*m2c);
    for(int j=0;j<m2c;j++){
        int rr=rand()%6;
        m2[i][j]=rr;
    }
}
```

Para la matriz m3 ocurre lo mismo que con la m1, salvo que en lugar de asignar valores aleatorios en cada posición se asignan 0's.

```
for(int i=0;i<m3r;i++){
    m3[i]=(int*)malloc(sizeof(int)*m3c);
    for(int j=0;j<m3c;j++){
        m3[i][j]=0;
    }
}
```

Se determina el número de filas que recorrerá cada hilo. Para esto se divide el número de filas de la matriz m1 entre el número de hilos. Para abarcar un posible residuo, se aplica la operación módulo. Con eso se tiene contemplado si el número de filas es igual al número de hilos o no.

```
rxt=m1r/nt;
r=m1r%nt;
```

Se crea un arreglo de hilos de tamaño nt (número de hilos), asignando memoria dinámica al apuntador threads.

```
pthread_t *threads;
threads = (pthread_t *)malloc(sizeof(pthread_t)*nt);
```

Se crea un arreglo de pair de tamaño nt (número de hilos), asignando memoria dinámica al apuntador pairs.

```
pair * pairs;
pairs = (pair*)malloc(sizeof(pair)*nt);
```

Se tiene un ciclo for, el cual va a iterar el número de hilos. Dentro de él se crean los hilos, empleando la función pthread_create(). Esta función recibe como parámetro el identificador del hilo i, la función a ser ejecutada por el hilo y la dirección del par en la posición i.
Antes de llamar a la función pthread_create, se asignan los valores l y h a cada par,los cuales indican el rango de filas que va a trabajar el hilo. Para ir actualizando los rangos, en cada iteración se van incrementando el número de filas que le corresponde a cada hilo. En la primera iteración se agrega el residuo en caso de que el número de hilos no concuerde con el número de filas de la primera matriz.

```
for(int i=0;i<nt;i++){

    if(i==0){
        l=0;
        h+=(rxt+r-1);
    }else{
        l=h+1;
        h+=rxt;
    }
    pairs[i].l=l;
    pairs[i].h=h;
    pthread_create(&threads[i], NULL, function,(void*)&pairs[i]);
}
```

Se ejecuta un ciclo para esperar a que los nt hilos terminen su ejecución, recibiendo como parámetro el identificador del hilo i del arreglo threads.

```
for(int i=0; i<nt; i++){
    pthread_join(threads[i], NULL);
}
```

Impresión de la matriz resultado m3

```
for(int i=0;i<m3r;i++){
    for(int j=0;j<m3c;j++){
        printf("%d ",m3[i][j]);
    }
    printf("\n");
}
```

Liberación de la memoria dinámica apuntada por threads.

```
free(threads);
```

**funct()**

Se castea el apuntador parametro a tipo pair y se asigna al apuntador pr.

```
void * function(void * parametro){

    pair *pr = (pair*)parametro;
```

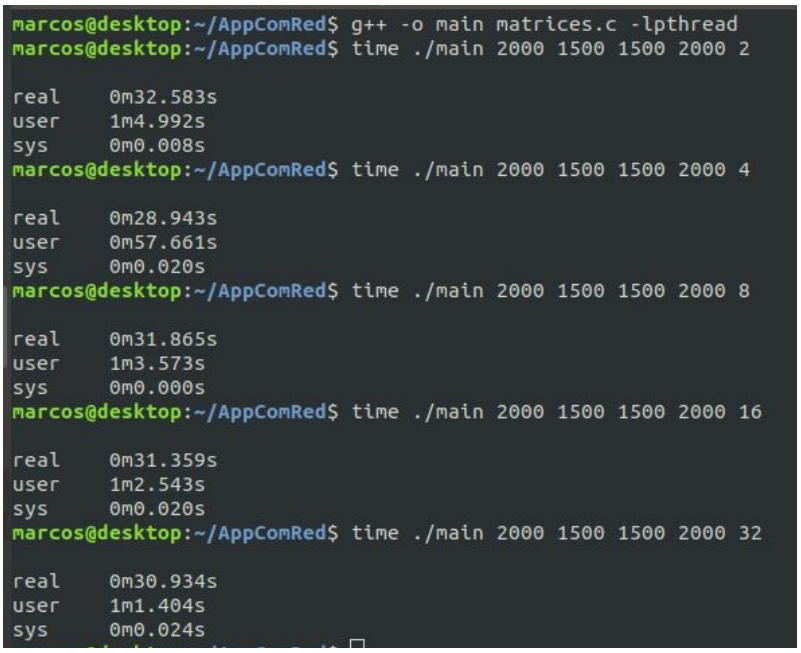
Se almacenan los valores l y h en variables enteras.

```
int l=pr->l,h=pr->h;
```

Se realiza la multiplicación de matrices. El primer ciclo for va a iterar sobre el rango de filas (l a h) que el hilo va a multiplicar. El segundo ciclo for va a iterar sobre las columnas de la segunda matriz m2. el tercer ciclo for va a iterar sobre el número de columnas de la matriz uno. Con lo anterior se logra que cada elemento de la columna j matriz m1 se multiplique y se sume por los los elementos de la fila i matriz m2, y esto a su vez se almacene en la posición i,j de la matriz resultado m3.

```
for(int i=l;i<=h;i++){
    for(int j=0;j<m2c;j++){
        for(int k=0;k<m1c;k++){
            m3[i][j]+=m1[i][k]*m2[k][j];
        }
    }
}
```

**Pruebas**




Como se observa en la imagen, los tiempos de ejecución son similares sin importar el número de hilos que se emplea, con lo cual se concluye en un mayor número de hilos no contribuye a tener un mayor tiempo de ejecución. Esto se debe a que los hilos no están trabajando de manera paralela, sino concurrente, lo cual indica que todos los hilos ocupan el mismo procesador y esperan a que el planificador los ponga en ejecución. Además, todos los hilos están realizando la misma tarea.

**Concusiones**

Piñon Caballero Angel Ramon

Como lo vimos a lo largo de la práctica podemos utilizar hilos para dividir tareas y que estas se puedan realizar concurrentemente, esto para que no sea necesario que una tarea termine para poder realizar la otra, es por esto que es importante conocer el funcionamiento de estos para el desarrollo de tareas multihilos. Al realizar las pruebas pudimos observar que si bien este caso los tiempos de ejecución no fueron extremadamente diferentes, en un programa que requiera un mayor tiempo de ejecución el reducirlo un poco puede ser benefisioso.

Sin embargo tambien debemos saber que el manejo de hilos no solo tiene sus cosas buenas ya que para el correcto funcionamiento es necesario que las tareas no deban realizarse en un orden especifico o si esto se desea es necesario utilizar algun metodo de sincronización ya que si esto no se hace podemos obtener una respuesta diferente a lo que esperamos al cruzarse las tareas que realizan los hilos.

Marcos León Reyes

El desarrollo de la presente práctica arrojó conclusiones interesantes, la primera de ella es que la librería opencv se trabaja mejor en python que en c++, no solo por la característica de las listas presentes en python sino por la facilidad para instalar la librería.

Por otro lado, en cuestión de los hilos se concluye que solo serán de utilidad cuando se encuentren haciendo tareas distintas, como por ejemplo, que uno de los hilos se encuentre bloqueado esperando la llegada de un recurso, mientras que otro hilo está realizando operaciones matemáticas.

Morales Garcia Christian Arturo

Lo primero que puede notar es que si bien los hilos tienen algunas ventajas que resultan muy atractivas a la hora de ejecutar programas, tienen su grado de complejidad ya que requieren que se les preste atención a pequeños detalles ya que como llegan a compartir algunos recursos pueden llegar a interferir en los cálculos de otros hilos por lo que es importante estar consciente de cuando es bueno usar los hilos yc cuando no.