Trabajo Práctico 2
Sincronización de procesos
Detección de Bordes

Tecnología Digital II

El trabajo práctico debe realizarse en grupos de tres personas. Tienen dos semanas para realizarlo.
La fecha de entrega límite es el viernes 14 de abril hasta las 23:59.
Se solicita no realizar consultas del trabajo práctico por los foros públicos. Limitar las preguntas al foro privado creado para tal fin.

Procesos y Threads

La creación de procesos se hace por medio de primitivas del sistema operativo como `fork`. Esta operación permite crear una copia nueva del proceso original, con la misma memoria que el proceso padre pero en un espacio de memoria diferente. Para que ambos procesos, padre e hijo compartan una parte de su memoria, esta debe ser compartida por medio de otros llamados al sistema.
Con el fin de simplificar toda esta operatoria, evitando llamados al sistema y reduciendo los costos de crear nuevos procesos, existen los *threads*. Estos ejecutan en el mismo espacio de memoria que el proceso padre, pero sobre un contexto de ejecución diferente.
En este trabajo práctico utilizaremos *threads* para resolver un filtro de detección de bordes sobre imágenes. Los *threads* se crean de forma similar a utilizar `fork` y se espera por su terminación de forma similar a utilizar `wait`. Ambos llamados en términos de *threads* son respectivamente `pthread_create` y `pthread_join`. Estos llamados forman parte de la `libC` e implementan una interfaz para generar *threads* contra el sistema operativo.

A continuación se describen los parámetros de estas funciones:

- ```
  int pthread_create(pthread_t* thread,
                     const pthread_attr_t* attr,
                     void* (*routine)(void *),
                     void* arg);
  ```
  `thread`: Puntero al lugar donde se guardará el PID del thread creado.
  `attr`: Atributos del thread que será creado (`NULL` indica atributos por defecto).
  `routine`: Puntero a la rutina que será ejecutada en el nuevo thread.
  `arg`: Argumentos pasados al thread a crear (`NULL` indica sin parámetros).

- `int pthread_join(pthread_t thread, void **status)`
  `thread`: Puntero al PID del thread por el que se quedará esperando.
  `status`: Valor de retorno del thread por el que se esperaba (`NULL` indica que debe ser ignorado).

Creando *threads* de esta forma, vamos a poder utilizar la misma memoria de datos entre todos contextos de ejecución.

En los ejercicios que siguen, declararemos un arreglo en memoria y múltiples *threads* podrán acceder a este leyendo y modificando la memoria compartida. Es fundamental resolver la sincronización entre los *threads* para lograr un resultado correcto.
  ```

Imágenes

Las imágenes se representan digitalmente como un arreglo de pixeles. Cada pixel está compuesto por componentes de color, cada uno con un valor que indica su brillo. Los componentes de color del modelo de síntesis aditiva son el rojo (`r`), verde (`g`) y azul (`b`).

Ahora, en una imagen con pixeles de 24 bits y componentes de 8 bits, cada componente de color ocupa un 1 byte, es decir 8 bits, pudiendo valer entre 0 y 255. Para operar en memoria las imágenes con pixeles de 24 bits son más complejas de representar, ya que cada pixel no tendrá un tamaño potencia de dos. Para simplificar esto, optamos por usar imágenes de 32 bits, con un componente de transparencia. Las imágenes que utilizaremos estarán representadas como, `r`, `g`, `b` y `a`, donde el último componente corresponde al valor de transparencia del pixel, que para fines prácticos deberá ser siempre 255 (es decir, sin transparencia).

Para simplificar aun más el manejo de imágenes, la cátedra provee una biblioteca de funciones para leer y escribir archivos de tipo `bmp` (*bitmap*). Si bien la biblioteca provista esta limitada a algunos tipos específicos de archivos `bmp`, es suficiente para el desarrollo del presente trabajo práctico. Además, al usar una biblioteca desarrollada por la cátedra, es posible entender y explorar el código que se utiliza para acceder a los archivos de imágenes.

Algoritmo

El algoritmo de detección de bordes que implementaremos consta de tres etapas, en la primera se calcula el brillo de cada uno de los pixeles, en la segunda se detectan los bordes, calculando la diferencia de cambio de brillo entre pixeles consecutivos, y por último se procesa una mezcla entre los pixeles de la imagen original y los bordes detectados.

A continuación se describe cada uno de estos procesos:

- **Paso 1**: Calculo de Brillo (`step1_brightness`)

  Obtener el brillo de un pixel consiste en realizar un promedio de los valores de los componentes de color de cada pixel. En particular se suele aplicar un promedio ponderado, priorizando el color verde, ya que el humano logra distinguir mejor las diferencias entre tonalidades del color verde. La formula aplicada en este caso es:

  $$\texttt{brightness[i][j]} = (\ \texttt{img[i][j].r} + 2 \cdot \texttt{img[i][j].g} + \texttt{img[i][j].b}) \ / \ 4.$$

  Donde `img` es el arreglo de pixeles de la imagen original y `brightness` es un nuevo arreglo donde almacenar un byte por el brillo de cada pixel.

- **Paso 2**: Calculo de Bordes (`step2_edges`)

  Existen múltiples formas de implementar filtros de detección de bordes. En este caso, utilizaremos el filtro *Sobel*, que consta de dos operadores, uno detecta el gradiente horizontal y el otro el gradiente vertical. Es decir, la diferencia de cambio de brillo desde un pixel al siguiente.
  Las operaciones realizadas son las siguientes:

  $$\begin{aligned}\texttt{dh} = abs(\ &(-1) \cdot \texttt{b[i-1][j-1]} + (0) \cdot \texttt{b[i][j-1]} + (+1) \cdot \texttt{b[i+1][j-1]} + \\ &(-2) \cdot \texttt{b[i-1][j+0]} + (0) \cdot \texttt{b[i][j+0]} + (+2) \cdot \texttt{b[i+1][j+0]} + \\ &(-1) \cdot \texttt{b[i-1][j+1]} + (0) \cdot \texttt{b[i][j+1]} + (+1) \cdot \texttt{b[i+1][j+1]}\ ) \ / \ 8\end{aligned}$$

  $$\begin{aligned}\texttt{dv} = abs(\ &(-1) \cdot \texttt{b[i-1][j-1]} + (-2) \cdot \texttt{b[i][j-1]} + (-1) \cdot \texttt{b[i+1][j-1]} + \\ &(+0) \cdot \texttt{b[i-1][j+0]} + (+0) \cdot \texttt{b[i][j+0]} + (+0) \cdot \texttt{b[i+1][j+0]} + \\ &(+1) \cdot \texttt{b[i-1][j+1]} + (+2) \cdot \texttt{b[i][j+1]} + (+1) \cdot \texttt{b[i+1][j+1]}\ ) \ / \ 8\end{aligned}$$

  $$\texttt{edges[i][j]} = saturate(\ \texttt{dh} + \texttt{dv}\ )$$

  Donde para pixeles consecutivos se los multiplica por un coeficiente y luego se calcula el valor absoluto de su diferencia. En el ejemplo `b` es la matriz de brillo de cada pixel y `edges` corresponde

a la matriz resultado. En cada valor de la matriz resultado se almacena la suma saturada del calculo de los dos gradientes, tanto horizontal como vertical. La suma saturada es tal que al sumar no se supere el máximo del componente, en este caso 255 ya que su tamaño es de un byte.

El resultado de este proceso será una matriz con datos de 0 a 255, donde 255 corresponde al máximo gradiente posible y 0 a que no existe gradiente, no hay cambio entre los pixeles consecutivos.

- **Paso 3**: Mezcla de imágenes (`step3_merge`)

  El último paso corresponde a utilizar la matriz de bordes como razón de ponderación para cada uno de los pixeles de la imagen original.
  La operación aplica para cada pixel la siguiente formula:

  $$value = saturate(\texttt{edges[i][j]} \cdot 10)$$
  $$\texttt{result[i][j].r} = (\ value \cdot \texttt{img[i][j].r}\ ) \ / \ 255$$
  $$\texttt{result[i][j].g} = (\ value \cdot \texttt{img[i][j].g}\ ) \ / \ 255$$
  $$\texttt{result[i][j].b} = (\ value \cdot \texttt{img[i][j].b}\ ) \ / \ 255$$
  $$\texttt{result[i][j].a} = 255$$

  Donde *value* corresponde al borde detectado para cada pixel multiplicado por 10 y saturado. Esta operación se realiza para ponderar aun más la detección del borde. En filtros de imágenes más inteligentes se utiliza un proceso más complejo de normalización de los datos. Los pixeles de la imagen original tomados de `img` son ponderados según el *value* calculado. El resultado se guarda dentro de `result`, para luego se almacenado como la nueva imagen. Notar que el componente de transparencia se completa de forma fija con el valor 255.

Las funciones que realizan cada uno de estos procesos son las siguientes:

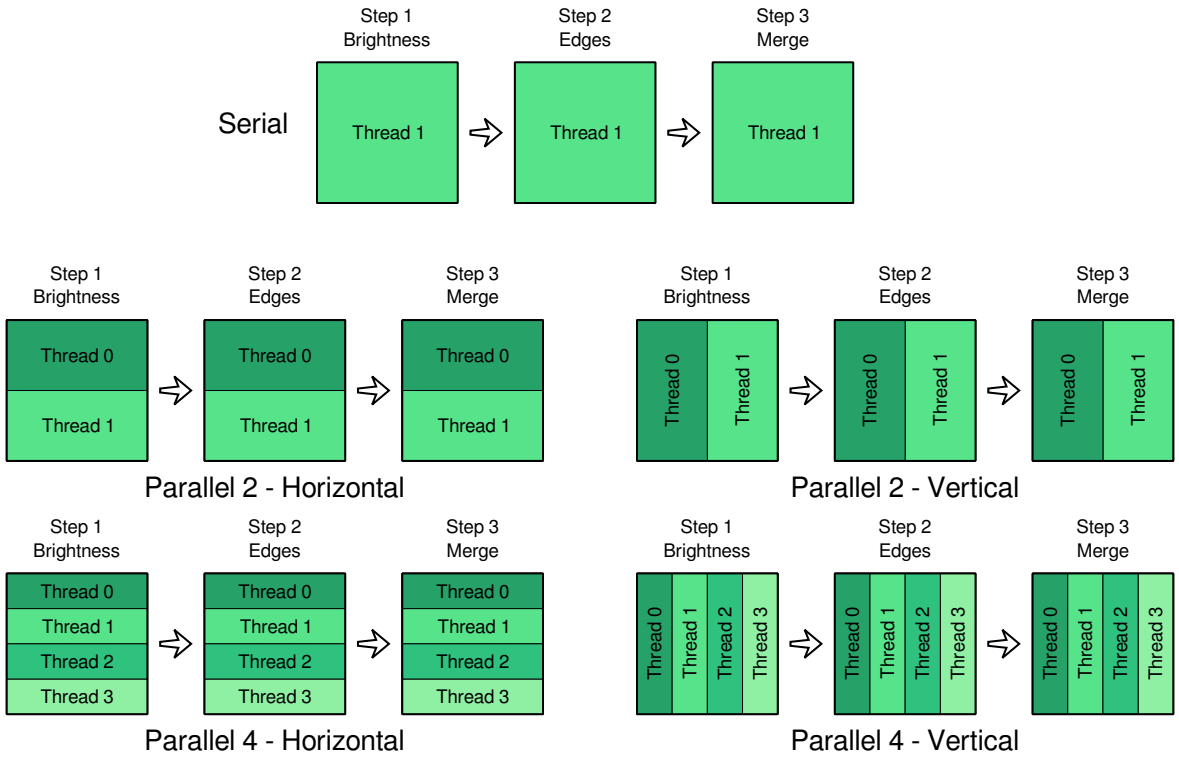
- `step1_brightness(width, height, image, brightness, start_width, end_width, start_height, end_height)`
- `step2_edges(width, height, brightness, edges, start_width, end_width, start_height, end_height)`
- `step3_merge(width, height, image, edges, result, start_width, end_width, start_height, end_height)`

Donde:

`width`: Ancho en pixeles de la imagen.
`height`: Alto en pixeles de la imagen.
`image`: Puntero a la matriz de pixeles de la imagen original.
`brightness`: Puntero a la matriz de brillo por pixel.
`edges`: Puntero a la matriz de bordes detectados.
`result`: Puntero a la matriz resultado.
`start_width`: Primer pixel a recorrer en ancho.
`end_width`: Último pixel a recorrer en ancho.
`start_height`: Primer pixel a recorrer en alto.
`end_height`: Último pixel a recorrer en alto.

Enunciado

Los ejercicios calcularán el procedimiento mencionado en el apartado anterior utilizando diferentes formas de paralelización. En la figura se ilustran las diferentes formas de paralelización a resolver.



Vamos a partir del algoritmo serial implementado en `EdgeDetect_S.c` para estudiar el funcionamiento de las distintas funciones. Se espera que identifiquen el funcionamiento de estas funciones y las reutilicen para el código de sus ejercicios.

Además como ejemplo de uno de los casos paralelos se provee el archivo `EdgeDetect_P2H.c` que contiene la implementación del algoritmo paralelizado utilizando dos *threads* de forma horizontal.

Tener en cuenta que al paralelizar y requerir en la operación de celdas lindantes de la matriz, los *threads* van a necesitar compartir datos en los limites del área de la matriz que estén procesando.

Ejercicio 1

Estudiar las funciones provistas por la cátedra. Probando su funcionamiento con ejemplos simples e incluso alterando parte del código provisto. Implementar la solución con dos procesos de forma vertical, es decir, cada *thread* tomará la mitad de las columnas y recorrerá la mátriz por toda las filas. Completar la solución en el archivo: `EdgeDetect_P2V.c`.

Ejercicio 2

Utilizando de base las soluciones horizontal y vertical para dos *threads*, extrapolar estás soluciones e implementar el algoritmo de detección de bordes utilizando 4 *threads* de forma horizontal y vertical. Tener en cuenta que los procedimientos en los *threads* centrales van a tener dos caras lindantes, procesos a derecha e izquierda o arriba y abajo según corresponda. Esto implica que estos procesos van a recorrer al menos una columna o fila de más. Completar las soluciones en los archivos: `EdgeDetect_P4H.c` y `EdgeDetect_P4V.c` respectivamente.

Ejercicio 3

Considerando todas las implementaciones realizadas. Realizar una tabla de tiempos, midiendo para cada implementación cuanto demora.

	duration
EdgeDetect	
EdgeDetect_P2H	
EdgeDetect_P2V	
EdgeDetect_P4H	
EdgeDetect_P4V	

Las mediciones las pueden realizar utilizando el comando `time` en linux. Tener en cuenta, tomar varias muestras de cada experimento, sobre todo de los experimentos que demoran menos tiempo total. Además, para realizar una comparación de algoritmos, se recomienda calcular el porcentaje de relación entre el tiempo de ejecución del caso serial contra cada uno de los casos paralelos.

Además presentar valores promedio de las muestras o el valor mínimo obtenido como representante del mejor tiempo de ejecución. Recordar que todas las muestras deben ser tomadas en una misma computadora y que no pueden estar otros procesos corriendo simultáneamente. Esto podría generar ruido en las mediciones. Además deben comentar cualquier acceso a entrada/salida, como imprimir en pantalla, pudiendo así medir solo el tiempo que demora en ejecutar.

Al ser un algoritmo de imágenes, gran parte del tiempo se utilizará para cargar en memoria la imagen y guardarla luego en disco. Para evitar problemas de *buffers*, pueden correr varias veces el programa, ignorando las primeras muestras, hasta que la imagen este en memoria. Sin embargo, los tiempos medidos incluirán el tiempo de carga de la imagen, que lo consideraremos como un componente serial que no es paralelizado.

Interpretación del código

Los programas incluyen instrucciones y tipos de datos que no vimos en clase. Parte del objetivo del trabajo es que investiguen de qué se trata cada elemento del código que no conozcan.

Entregable

La entrega debe constar de los archivos .c que realizaron, la tabla solicitada y los gráficos resultantes. El código que les damos ya hecho no debe ser alterado.

Por último, el código que realicen debe estar comentado. Deben explicar qué aspecto del problema resuelve cada parte del código y cómo la resuelve. No explicar instrucción por instrucción.