Trabajo Práctico 2
Sincronización de procesos
Blur Effect

Tecnología Digital II

El trabajo práctico debe realizarse en grupos de tres personas. Tienen tres semanas para realizar la totalidad de los ejercicios.
La fecha de entrega límite es el 12 de septiembre hasta las 23:59.
Se solicita no realizar consultas del trabajo práctico por canales públicos. Para realizar consultas específicas de su trabajo práctico pueden crear un canal de slack e incluir a todos los docentes y compañeros de grupo. Tengan presente crear el canal de slack con su nombre de grupo, a fin de evitar confusiones.

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 effecto *blur* o *Gaussian blur* es un efecto gráfico utilizado para reducir el ruido y detalles en una imagen. También se utiliza para reducir *aliasing* o efecto Moire en imágenes compuestas por pixeles.

En este trabajo práctico vamos a implementar un simplificación de este algoritmo para una matriz de 3x3.

Calculo un paso del Efecto Blur (`step_blur`)

Consiste en aplicar para todos los pixeles de la imagen un promedio ponderado entre todos los pixeles vecinos. Esta operación se realiza independientemente para cada color, pero dejando el valor de transparencia en 255.

Para cada pixel y cada componente de color la operación es la siguiente:

$$\begin{aligned} \texttt{r[i][j]} = (\ &(1)\cdot\texttt{b[i-1][j-1]} + (2)\cdot\texttt{b[i][j-1]} + (1)\cdot\texttt{b[i+1][j-1]} + \\ &(2)\cdot\texttt{b[i-1][j+0]} + (4)\cdot\texttt{b[i][j+0]} + (2)\cdot\texttt{b[i+1][j+0]} + \\ &(1)\cdot\texttt{b[i-1][j+1]} + (2)\cdot\texttt{b[i][j+1]} + (1)\cdot\texttt{b[i+1][j+1]}\ ) \ /\ 16 \end{aligned}$$

Donde para `r` es la matriz resultado y `b` es la matriz original de la imagen

Este filtro se aplicará para todos los pixeles exceptuando los bordes.

La función que realizan este proceso es la siguiente:

- `step_blur(width, height, image, result,`
  `start_width, end_width, start_height, end_height)`

Donde:

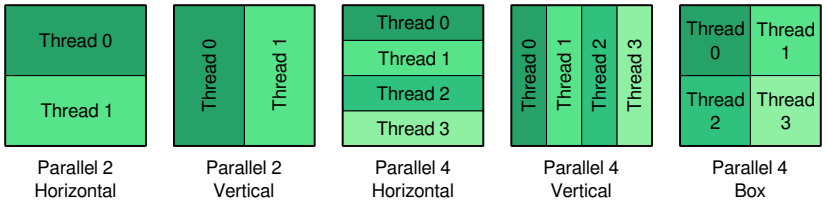
`width`: Ancho en pixeles de la imagen.
`height`: Alto en pixeles de la imagen.
`image`: Puntero a la matriz de pixeles de la imagen original.
`result`: Puntero a la matriz resultado.
`start_width`: Primer pixel a recorrer en ancho.
`end_width`: Último pixel a recorrer en ancho.
`start_height`: Primer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

Calculo del Efecto Blur

Este procedimiento será calculado repetidas veces sobre el resultado de cada iteración. La cantidad de veces que se repetirá está dada por un parámetro tomado desde la consola. El procedimiento para poder iterar una y otra vez, sin necesidad de copiar toda la matriz será ir intercambiando los punteros entre la matriz de la imagen y la matriz resultado.

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. Tener en cuenta que cada etapa del efecto Blur será paralelizada de forma independiente, ya que se debe tener la matriz resultado completa antes de continuar con la siguiente etapa.



Partiendo del algoritmo serial implementado en `Blur_S.c`, se estudiará el funcionamiento del algoritmo básico. Se espera que identifiquen el funcionamiento de las funciones y las reutilicen para el código de sus ejercicios.

Además como ejemplo de uno de los casos paralelos se provee el archivo `Blur_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 matriz por toda las filas. Completar la solución en el archivo: `Blur_P2V.c`.

Ejercicio 2

Utilizando de base las soluciones horizontal y vertical para dos *threads*, extrapolar estás soluciones e implementar el algoritmo 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: `Blur_P4H.c` y `Blur_P4V.c` respectivamente.

Ejercicio 3

Utilizando de base las soluciones anteriores, implementar el último esquema de particionado denominado `box`. Completar la solución en el archivo: `Blur_P4B.c`.

Ejercicio 4

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

	image small			image big		
	1 step	5 step	10 step	1 step	5 step	10 step
Blur						
Blur_P2H						
Blur_P2V						
Blur_P4H						
Blur_P4V						
Blur_P4B						

Las mediciones las pueden realizar utilizando el comando `time` en linux, tomando el tiempo indicado como `real`. Tomar varias muestras de cada experimento, sobre todo de los experimentos que demoran

menos tiempo total. Al menos deben tomar 10 muestras por cada experimento y realizar un promedio o mediana de los datos. 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.

Como datos de entrada se recomienda utilizar dos imágenes, una de tamaño reducido y otra de gran tamaño. Además, como cantidad de pasos se recomienda al menos utilizar los valores 1, 5 y 10.

En base a los datos presentados en la tabla se pide que realicen una pequeña reflexión respondiendo la pregunta: ¿Resulta más eficiente resolver el problema en paralelo?

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.