Actividad 3: Paralelización

Erik García Cruz, A01732440

Escuela de Ingeniería y Ciencias, Instituto Tecnológico y de Estudios Superiores de Monterrey

TE2004B.501: Diseño de sistemas embebidos avanzados (Gpo 501)

Profesor: Emmanuel Torres

18 de noviembre, 2025

Liga al GitHub

Liga al Git: https://github.com/Erik-G-C/Act_3_Paralalizacion.git

Descripción

Este proyecto busca optimizar el procesamiento de un lote de imágenes BMP mediante la implementación de paralelismo utilizando la librería OpenMP en lenguaje C, aprovechando la arquitectura multinúcleo del procesador. Mi laptop cuenta con 14 núcleos.

Elementos a emplear:

Elemento	Descripción
Qué se va a hacer	Procesar 30 imágenes BMP de 24 bits (True Color) con cuatro transformaciones distintas: Escala de grises Inversión vertical Inversión horizontal Blurring 57x57).
Herramientas Empleadas	Lenguaje C (código fuente) VSC como editor de código GCC/MinGW (compilador) OpenMP (librería para paralelismo) MSYS2 UCRT64 (entorno de ejecución).
Elementos de Garantía	El uso de la directiva #pragma omp parallel garantiza una distribución de la carga de trabajo entre los hilos.

Elementos empleados en la solución

Metodología

Diseño de la Solución Propuesta

El diseño utiliza un modelo de Paralelismo de Tareas (Task Parallelism).

- **Paralelismo de Lote**: Se utiliza #pragma omp parallel seguido de #pragma omp sections en mian_v2.c. Esto crea un equipo de 30 hilos (NUM_THREADS 30) y asigna explícitamente una tarea (procesar_imagen()) a cada sección.
- **Asignación de Carga**: La distribución es estática: cada sección se ejecuta una sola vez y solo por uno de los hilos del equipo. La carga de trabajo (30 imágenes) coincide con el número de hilos solicitados (30).

Entradas, Salidas y Condiciones de Operación a emplear:

Elemento	Descripción
Entradas (Input)	30 archivos de imagen en formato BMP de 24 bits
Salidas (Output)	120 archivos de imagen procesados (4 por imagen de entrada) guardados en `imagenes_salidas` (carpeta creada automáticamente).
Condiciones de Operación	**30 hilos** (`NUM_THREADS 30`), entorno Windows (MSYS2) y el compilador GCC con `-fopenmp`. La condición fundamental es que las 30 imágenes sean **BMP de 24 bits**.

Entradas, salidas y condiciones de operación de la solución

Manejo de memoria

Uso de `malloc()` para la Lectura de Imágenes

En funciones como `imagenes_ByN.c`, `blurring_57x57.c` y las de inversión, el uso de **`malloc`** es esencial para reservar espacio en la RAM basado en el tamaño dinámico de la imagen (calculado a partir del *header* BMP):

Reserva para el Header Completo (`fullHeader`): Se reserva el espacio para el encabezado de archivo y de información (los primeros `dataOffset` bytes), lo que permite reescribir la metadata original en el archivo de salida sin corromper la estructura BMP.

```
48      unsigned char *fullHeader = malloc(dataOffset);
49      fseek(fin, 0, SEEK_SET);
50      fread(fullHeader, 1, dataOffset, fin);
```

Uso de malloc en código

Reserva para los Píxeles de Entrada (img o inData): Se calcula el tamaño total en bytes del arreglo de píxeles (rowSize * absHeight), que incluye el padding (relleno de bytes) necesario para la alineación de filas.

```
28      size_t pixelArraySize = (size_t)rowSize * absH;
29
30      unsigned char *fullHeader = malloc(dataOffset);
31      fseek(fin, 0, SEEK_SET);
32      fread(fullHeader, 1, dataOffset, fin);
```

Uso de malloc en código

Reserva para los Píxeles de Salida (out o outData): Se reserva una segunda porción de memoria del mismo tamaño para almacenar la imagen resultante después de la transformación (escala de grises, blurring, etc.).

```
39      unsigned char *outData = malloc(pixelArraySize);
40      memset(outData, 0, pixelArraySize);
```

Uso de malloc en código

Guardado de Datos de Imagen

El proceso de guardar los datos en el disco se realiza en dos pasos fundamentales en cada función:

1. **Escribir Header**: Se escribe el header previamente cargado (fullHeader) en el nuevo archivo de salida.

```
87      fwrite(fullHeader, 1, dataOffset, fout);
```

Proceso de salida

2. **Escribir Píxeles:** Se escribe el bloque de píxeles procesados (`outData`) inmediatamente después del *header*.

```
88      fwrite(outData, 1, pixelArraySize, fout);
```

Proceso de salida

3. Liberación de recursos (free()) Para evitar *Memory Leaks* (fugas de memoria) y garantizar que la RAM esté disponible para las siguientes imágenes, es indispensable liberar la memoria con **`free()`** al final de cada función de procesamiento:

```
91      free(inData);
92      free(outData);
93      free(fullHeader);
```

Proceso de salida

Resultados

Elemento	Descripción
Resultados Esperados	Debido al uso de 30 hilos, y asumiendo una carga de trabajo homogénea por imagen, se esperaría una aceleración considerable. Sin embargo, en un sistema con P < 30 núcleos (por ejemplo, 4 u 8), se espera una **degradación del rendimiento** (*speedup* negativo) debido a la **sobrecarga (overhead)** por el exceso de hilos (30), lo

	que resulta en una intensa conmutación de contexto por parte del sistema operativo.
Resultados Obtenidos (Datos del Experimento Previo)	El programa falló la validación inicialmente para las imágenes 01 a 26 y solo procesó exitosamente las imágenes **27 a 30**, registrando un tiempo total de **75.801 segundos**. Esto fue debido a que las imágenes no eran de 24 bits, después de cambiarlas se logró completar las 30 imágenes correctamente con un tiempo total de **572.723 segundos**
Condiciones del Experimento	Se confirma que el modelo `#pragma omp sections` crea y ejecuta las 30 tareas.

Resultados del experimento

Como vemos inicialmente solo se procesaron salieron correctamente las imágenes 27, 28, 29 y 30 debido a que las anteriores 26 no eran de 24 bits. Esto es producto del convertidor de imágenes jpg a BMP que utilice, pues en las primeras 26 utilicé una página diferente mientras que en las últimas utilice freeconverter (https://www.freeconvert.com/es/bmp-converter) la cuál sí brindó las imágenes en bmp de forma correcta. Para solucionarlo simplemente volví a convertir las imágenes de jpg a bmp utilizando esta segunda página. Este me permitió medir el tiempo total de ejecución que fue de 572.723 segundos, lo que equivale a 9.545 minutos (tiempo largo posiblemente debido a que me contraba utilizando Spotify y YouTube en el navegador de forma simultánea a la operación de las imágenes).

Conclusiones

Elementos Clave para la Obtención de Resultados

1. **Modelo de Paralelismo (`#pragma omp sections`):** Este modelo es adecuado para un número fijo de tareas, pero la necesidad de **escribir manualmente las 30**

secciones lo hace menos escalable y propenso a errores que el bucle `parallel for`.

2. **Configuración de Hilos (`NUM_THREADS 30`):** La elección de 30 hilos para 30 tareas en una máquina típica de 4 u 8 núcleos resulta en una **ineficiencia significativa** (sobre-suscripción). Esto aumenta el *overhead* de OpenMP y del sistema operativo, resultando en un tiempo de ejecución mayor del necesario, independientemente de la falla de validación de las imágenes. Sin embargo gracias a que cuento con 14 núcleos no sufrí percance alguno.
3. **Manejo de Memoria:** La correcta gestión de la memoria (uso de `malloc` y `free` para los buffers de la imagen y el *header* en cada función) es crucial. El paralelismo de tareas aumenta el consumo de memoria RAM total, ya que varios hilos reservan memoria de manera concurrente.

Modificaciones de Acuerdo a los Datos del Experimento

Primeramente verificar que las imágenes estén correctas. También podría considerarse el trabajo secuencial con parallel for debido a que hacer las 30 llamadas simultáneas lleva el procesador a tope, no sufrí percance alguno, pero sí tenía miedo de alguna pantalla azul por parte de Windows o algún congelamiento