UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Computación Paralela (CPA)

Práctica 2: Reconstrucción de imágenes

Curso 2021/22

Índice

Advertencia

La memoria y el código fuente a entregar en esta práctica deben ser el trabajo personal y original del correspondiente grupo (de un máximo de dos estudiantes, ambos pertenecientes al mismo grupo de prácticas). La copia de cualquier parte de la memoria o del código de cualquier otra fuente, supondrá una calificación de cero en el trabajo, al margen de las medidas disciplinarias que pudieran derivarse.

1. Introducción

En un escenario hipotético, se ha detectado un *ransomware* que se dedica a infectar móviles y modificar todas las fotos que encuentra, sustituyéndolas por imágenes aparentemente aleatorias.

Tras estudiar las imágenes ofuscadas, se ha observado que lo que hace el *ransomware* es intercambiar aleatoriamente bloques de filas y columnas de la imagen, aunque sin mover el primer bloque de filas ni el primer bloque de columnas. En la figura 1 se puede ver un ejemplo de cuál sería la transformación que realizaría el ransomware sobre una determinada imagen.

Teniendo en cuenta lo que se conoce del *ransomware*, se ha desarrollado un programa (fichero `restore.c`) que trata de recuperar una imagen ofuscada, recolocando primero los bloques de filas y luego los bloques de columnas. Los bloques de filas se van recolocando de arriba a abajo, buscando siempre que la fila superior de un bloque sea lo más parecida posible a la fila inferior del bloque anterior. De manera análoga, los bloques de columnas se recolocan de izquierda a derecha. No se tienen garantías de que este algoritmo reconstruya cualquier imagen, pero parece funcionar bien en la mayoría de los casos.

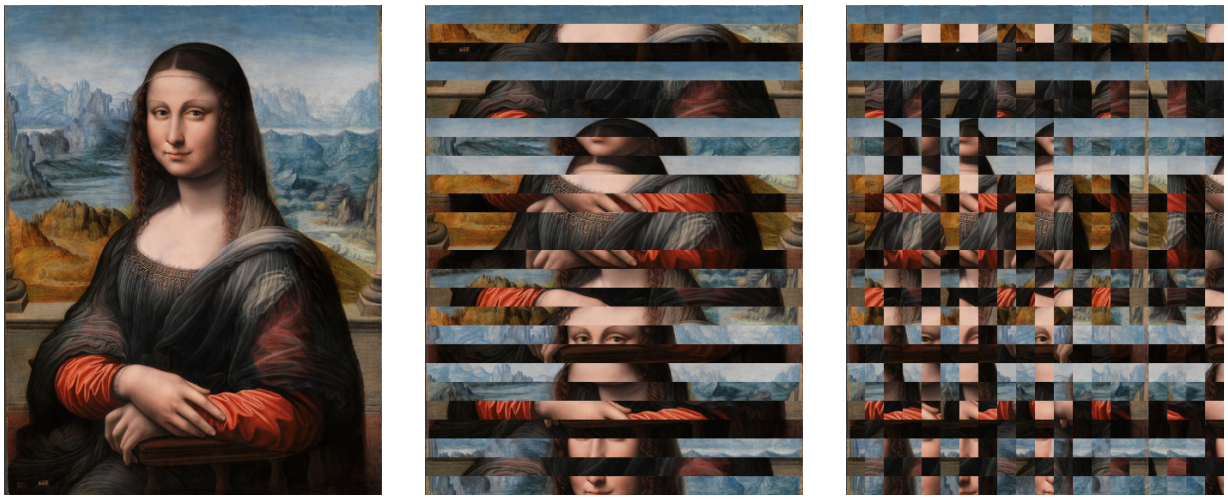

Figura 1: Transformación de una imagen por el *ransomware*: imagen original (izquierda), sobre la cual se barajan bloques de filas (centro), y finalmente se barajan bloques de columnas (derecha).

Sería interesante disponer de una versión paralela de este código para poder regenerar las imágenes con mayor celeridad. En ese contexto, en este trabajo se pide que paralelices el código de diferentes maneras, y que realices ejecuciones para poder comparar las prestaciones obtenidas por las diferentes versiones.

Antes de empezar

Como hiciste en la práctica 1, empieza por crear una carpeta en la unidad `W`, por ejemplo `W/cpa/prac2/`, donde dejar los ficheros de código fuente e imágenes de esta práctica.

2. Código de partida

La función `process` es la que se encarga de realizar el procesamiento completo de la imagen. La primera parte de la función (bucle `y`) recoloca los bloques horizontales de la imagen, considerando bloques de `bh` filas cada uno. La segunda parte (bucle `x`) hace lo mismo con los bloques verticales, de `bw` columnas cada uno.

Centrándonos en la primera parte de la función `process`, cada iteración del bucle `y` parte de que los bloques que cubren desde la fila 0 a la fila `y-1` de la imagen han sido ya colocados, y busca colocar el siguiente bloque. Para ello, recorre, mediante el bucle `y2`, los bloques que quedan por colocar. Para cada uno de estos bloques, calcula la diferencia (mediante la función `distance`) entre la última fila colocada (fila `y-1`) y la primera fila del bloque (fila `y2`), con el fin de encontrar el bloque que minimice esa diferencia. Una vez encontrado dicho bloque, se coloca en su lugar (a partir de la fila `y`), intercambiándolo con el bloque que ahora ocupa su sitio (mediante la función `swap`).

La segunda parte de la función es totalmente análoga, pero trabaja con bloques verticales en vez de horizontales.

Las funciones `distance` y `swap` son bastante sencillas. La primera obtiene la diferencia entre dos líneas horizontales o verticales de la imagen. Para ello, recorre simultáneamente ambas líneas (píxel 0 de ambas líneas, píxel 1, píxel 2, etc), calculando para cada píxel la diferencia entre las dos líneas y acumulando el resultado. La función `swap` intercambia dos zonas rectangulares de la imagen, que pueden corresponder a dos bloques horizontales o dos bloques verticales.

El programa se utiliza de la siguiente manera:

```
$ ./restore -i imagen_entrada.ppm -o imagen_salida.ppm -b tamaño_de_bloque
```

Las opciones que se pueden proporcionar son:

- `-i imagen_entrada.ppm`: especifica el nombre del fichero que contiene la imagen que se desea restaurar. Debe ser una imagen en formato PPM de tipo P6. Por defecto, se toma `"in.ppm"` como nombre del fichero de entrada.
- `-o imagen_salida.ppm`: indica el fichero donde se guardará la imagen restaurada. Se puede usar `""` como nombre de fichero para no generar la imagen de salida (por ejemplo, cuando sólo queramos medir tiempos). Por defecto, se toma `"out.ppm"` como nombre del fichero de salida.
- `-w anchura_de_bloque`: indica la anchura de los bloques verticales de la imagen (número de columnas por bloque). El valor por defecto es 8.
- `-h altura_de_bloque`: indica la altura de los bloques horizontales de la imagen (número de filas por bloque). El valor por defecto es 8.
- `-b tamaño_de_bloque`: indica un valor común tanto para la altura como para la anchura de los bloques.

Hay que destacar que se debe indicar (mediante la opción `-b`, o bien mediante las opciones `-w` y `-h`) el tamaño de los bloques horizontales y verticales usados para desordenar la imagen. Supondremos que este tamaño es conocido.

Tienes disponibles dos imágenes para que pruebes los programas a desarrollar. La imagen `peque.ppm` es pequeña y utiliza un tamaño de bloque 8, que es rápido de procesar. Utilízala para comprobar que tus versiones funcionan. Para ello, como se hizo en la práctica 1, ejecuta el programa secuencial y guarda el fichero de salida, por ejemplo con el nombre `ref.ppm`, de manera que luego puedas comparar, mediante la orden `cmp`, dicho fichero con el producido por el programa paralelo. Lógicamente, ambos deberían ser idénticos.

Cuando ya veas que tus códigos funcionan, utiliza para medir tiempos la imagen `grande.ppm`, que es mayor y utiliza un tamaño de bloque 2, lo que supone un mayor coste.

Para ahorrar espacio en disco, la imagen `grande.ppm` se encuentra en la carpeta `/scratch/cpa/` de kahan. Para usarla, simplemente habrá que especificar la ruta `/scratch/cpa/grande.ppm` como nombre del fichero de entrada del programa `restore`. Sin embargo, si quieres visualizar la imagen, puedes copiarla a la unidad `W`, por ejemplo a la carpeta `W/cpa/prac2`, previamente creada, y visualizarla desde allí (luego puedes eliminar la imagen). Puedes copiar la imagen con la orden:

```
cp /scratch/cpa/grande.ppm ~/W/cpa/prac2/
```

Puedes hacer lo mismo para visualizar la imagen restaurada y descubrir cuál es la imagen oculta.

3. Desarrollo de versiones paralelas

Empieza por modificar el código original para que muestre por pantalla el tiempo de cálculo del programa (el tiempo de la función `process`). Aunque sea un código secuencial, utiliza la función adecuada de OpenMP para medir tiempos. Guarda esta versión con el nombre `restore0.c`. Servirá para poder comparar, más adelante, el tiempo del programa secuencial con el de los programas paralelos.

En los siguientes ejercicios se pide que desarrolles dos versiones paralelas del programa. Ambas versiones deberán mostrar, además del tiempo de ejecución, el número de hilos con que se ejecutan. Comenta brevemente en la memoria las elecciones que hayas tomado sobre el ámbito de las variables, así como el posible uso de directivas de sincronización.

Ejercicio 1: Obtén una primera versión paralela del programa paralelizando las funciones que utiliza la función `process`: `distance` y `swap`. Llama a esta versión `restore1.c`.

Ejercicio 2: Obten una segunda versión paralela paralelizando el cuerpo de la función `process`, dejando la función `swap` paralelizada (pero no la función `distance`). Observa que los bucles externos de la función `process` no se pueden paralelizar, con lo que deberás paralelizar los bucles internos. Llama a esta versión `restore2.c`.

4. Estudio de prestaciones

Ejercicio 3: En este ejercicio no se pide que modifiques ningún código ni que realices ninguna ejecución, sino que respondas de forma razonada a lo que se pregunta.

- En la paralelización del bucle `y2` de `process`, ¿crees que con alguna planificación se obtendría un mejor equilibrio de carga que con otra? ¿Sí/no? ¿Cuáles? ¿Por qué?
- ¿Y en el bucle `y` de `process`? (como se ha dicho, ese bucle no se puede paralelizar, pero para este ejercicio teórico supongamos que sí se puede).

Aparte del equilibrio de carga, la planificación elegida puede afectar también a las prestaciones del programa por otras causas, como son:

- Sobrecarga de la propia planificación. Las planificaciones dinámicas (`dynamic` o `guided`) pueden suponer una mayor sobrecarga, puesto que la asignación se va haciendo durante la ejecución del programa.
- Aprovechamiento de la memoria caché. Generalmente, el aprovechamiento de la memoria caché es peor si se usa un tamaño de *chunk* de 1 y mejora al usar tamaños de *chunk* más grandes.

En el siguiente ejercicio se pide que evalúes distintas planificaciones y compruebes cómo afectan a las prestaciones.

Ejercicio 4: Utilizando los nodos de cálculo del cluster `kahan`, saca tiempos de ejecución de las dos versiones paralelas realizadas, usando 16 hilos y las siguientes planificaciones:

- `static` con tamaño de *chunk* por defecto.
- `static` con tamaño de *chunk* 1.
- `dynamic` con tamaño de *chunk* por defecto.

Analiza cuál es la mejor planificación en cada versión paralela, indicando a qué puede deberse.

Ejercicio 5: Utilizando los nodos de cálculo del cluster `kahan`, saca tiempos de ejecución de las dos versiones paralelas realizadas, variando el número de hilos y eligiendo en cada versión la planificación con la que se hayan obtenido mejores resultados en el ejercicio anterior. Para limitar el número de ejecuciones, se recomienda usar potencias de 2 para los valores del número de hilos (2, 4, 8...), llegando hasta el número de hilos que consideres adecuado (justifica por qué eliges ese número máximo de hilos).

Muestra tablas y gráficas para tiempos, speed-ups y eficiencias para las versiones paralelas que has realizado. Utiliza esas tablas y gráficas para comparar las prestaciones de las dos versiones. En vista de los resultados, extrae conclusiones sobre cuál es la mejor versión paralela, o bien si no hay diferencia significativa, y razona a qué crees que se debe.

Explica cómo has lanzado las ejecuciones en el cluster, indicando cómo estableces el número de hilos y la planificación y adjuntando alguno de los ficheros de trabajo utilizados.

5. Mostrando información de cada hilo

Ejercicio 6: En este ejercicio hay que hacer que cada hilo muestre información sobre las iteraciones que le ha tocado procesar de un bucle paralelizado. En concreto, partimos de la versión paralela del ejercicio 2, donde, en cada iteración del bucle `y`, se reparten las iteraciones del bucle `y2` entre los hilos. En principio habría que hacer que, en cada iteración del bucle `y`, cada hilo muestre un mensaje con su identificador, cuántas iteraciones ha procesado del bucle `y2` (en esa iteración del bucle `y`) y cuáles han sido la menor y mayor distancias que ha encontrado en esas iteraciones. Sin embargo, para evitar que

salgan demasiados mensajes por pantalla, haz que solo se muestren los mensajes correspondientes a la primera iteración del bucle `y`.

A modo de ejemplo, la salida sería algo similar a lo siguiente:

```
Hilo 2: 225 iteraciones con distancias entre 193367 y 971757.
Hilo 3: 224 iteraciones con distancias entre 158870 y 975028.
Hilo 1: 225 iteraciones con distancias entre 115248 y 973264.
Hilo 0: 225 iteraciones con distancias entre 163521 y 969807.
```

donde, por ejemplo, podemos ver que, en la primera iteración del bucle `y`, el hilo 0 ha hecho 225 iteraciones del bucle `y2`, calculando distancias con valores entre 163521 (mínimo) y 969807 (máximo).

Guarda esta versión con el nombre `restore3.c`. Comenta y justifica en la memoria las modificaciones realizadas en el código para esta versión.

6. Entrega

Hay **dos tareas** en PoliformaT para la entrega de esta práctica:

- En una de las tareas debes subir un fichero **en formato PDF** con la memoria de la práctica. No se admitirán otros formatos.
- En la otra tarea debes subir un único archivo comprimido con los ficheros de código fuente de las distintas versiones que hayas desarrollado, junto con alguno de los ficheros de trabajo utilizados para lanzar las ejecuciones. **No incluyas los ejecutables resultantes de la compilación** (ocupan mucho y no son necesarios porque se pueden generar a partir del fuente). El archivo debe estar comprimido en formato `.tgz` o `.zip`.

  Comprueba que todos los ficheros compilen correctamente y que tienen el nombre que se especifica en este boletín.

A la hora de realizar la entrega de la práctica, hay que tener en cuenta las siguientes recomendaciones:

- Hay que entregar una memoria descriptiva de los códigos empleados y los resultados obtenidos. Procura que la memoria tenga un tamaño razonable (ni un par de páginas, ni varias decenas).
- Los ejercicios 1-5 son obligatorios. El ejercicio 6 es opcional, aunque deberá entregarse si se desea poder optar al 100 % de la nota.
- No incluyas el código fuente completo de los programas en la memoria. Sí puedes incluir, si así lo deseas, las porciones de código que hayas modificado.
- Pon especial cuidado en preparar una buena memoria del trabajo. Se trata de entregar una memoria. No queremos un libro, pero sí que tenga una estructura y que tenga algo de narrativa y no una mera exposición de resultados.