UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Computación Paralela (CPA)

Práctica 2: Descifrando imágenes ocultas

Curso 2022/23

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, la policía anda siguiendo la pista a un grupo de delincuentes que intercambia información a través de imágenes ofuscadas, es decir, imágenes que a primera vista parecen solo ruido aleatorio pero en realidad son alteraciones de imágenes válidas. Un análisis concienzudo de las imágenes ha revelado que el proceso usado por los delincuentes consiste en, dada una imagen original, alterarla desplazando a la derecha/izquierda cada una de sus líneas, siendo aleatorio el número de píxeles de desplazamiento que se aplican a cada línea, y teniendo en cuenta que la primera línea no se altera. En la figura 1 se puede ver un ejemplo de cuál sería la transformación que se realizaría sobre una determinada imagen.

Teniendo en cuenta lo anterior, se ha desarrollado un programa (fichero `realign.c`) que trata de recuperar una imagen ofuscada, realineando las filas de manera que la diferencia entre dos filas consecutivas sea lo menor posible. No se tienen garantías de que este algoritmo reconstruya cualquier imagen, pero parece funcionar bien en la mayoría de los casos.

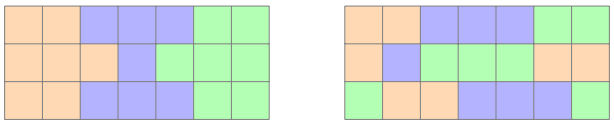

Figura 1: Imagen original de 7×3 píxeles (izquierda) e imagen transformada (derecha). La línea 0 permanece en su lugar, la línea 1 se ha desplazado 2 píxeles a la izquierda y la línea 2 un píxel a la derecha. Obsérvese que es un desplazamiento cíclico (los píxeles que salen por un extremo de la línea entran por el extremo opuesto).

El tiempo que se tarda en descifrar una imagen es considerable, especialmente en el caso de imágenes de gran tamaño y ese tiempo puede ser muy importante si hay que adelantarse a las acciones de los delincuentes. Por ello, la policía está interesada en versiones paralelas del programa `realign.c` que puedan acelerar el proceso.

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 `realign` es la que se encarga de realizar el procesamiento completo de la imagen. La primera parte de la función (primer bucle `y`) averigua cuántos píxeles habría que desplazar cada línea respecto a la línea anterior, para lograr que la diferencia de color entre ambas líneas sea lo menor posible. El resultado de esta fase es el vector `voff`, que nos dice que cada línea `y` debería desplazarse `voff[y]` píxeles a la izquierda respecto a la línea `y-1`.

Examinemos esta primera parte más en detalle. Para cada línea `y` se empieza por invocar la función `distance` para calcular la diferencia respecto a la línea anterior (`y-1`) sin aplicar ningún desplazamiento (o sea, un desplazamiento de cero píxeles). A continuación se examina, mediante un bucle anidado, todos los demás desplazamientos posibles (variable `off`), calculando para cada uno la diferencia entre la línea actual desplazada y la línea anterior y actualizando en su caso el valor de la variable `bestoff`, que contendrá el mejor desplazamiento hasta el momento. La diferencia entre la línea actual desplazada y la anterior se calcula como la suma de dos diferencias, como se ilustra en la Figura 2, donde se considera que la línea `y` está desplazada 4 posiciones respecto a la línea anterior.

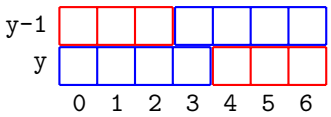

Figura 2: Diferencia entre las líneas `y` e `y-1`, considerando la línea `y` desplazada 4 posiciones respecto a la línea `y-1`. La diferencia será la suma de la diferencia entre los dos segmentos rojos más la diferencia entre los dos segmentos azules.

Al calcular la distancia entre dos líneas se introduce una optimización importante. Dadas las líneas `y` e `y-1`, queremos calcular el desplazamiento que produce la menor distancia entre las líneas. Por tanto, al

calcular una determinada distancia se abandonará el cálculo tan pronto como se alcance o supere el mínimo actual (`dmin`). Comentaremos esto al hablar de la función `distance` un poco más adelante.

La segunda parte de la función (segundo bucle `y`) modifica el vector `voff` para que sus valores correspondan a desplazamientos absolutos, en vez de desplazamientos respecto a la línea anterior. Además, calcula el máximo desplazamiento que se aplica a cualquier línea, lo que nos sirve para saber el tamaño del buffer auxiliar `v`, necesario para la siguiente parte de la función.

La tercera parte de la función `realign` se limita a aplicar el desplazamiento correspondiente a cada línea, usando para ello el buffer auxiliar `v`, que previamente reserva.

Por otra parte, la función `distance` calcula la diferencia en color entre dos segmentos de líneas. Como hemos comentado, queremos poder abandonar el cálculo si la distancia alcanza un determinado valor. Por esta razón, el bucle de esta función incorpora la condición `d<c`, que hará que se deje de iterar si deja de cumplirse.

Finalmente, la función `cyclic_shift` aplica un desplazamiento cíclico a una línea de la imagen, utilizando para ello un array auxiliar.

El programa se utiliza de la siguiente manera:

```
$ ./realign imagen_entrada.ppm imagen_salida.ppm
```

donde:

- El primer argumento (`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.
- El segundo argumento (`imagen_salida.ppm`) indica el fichero donde se guardará la imagen restaurada. Si se omite este argumento, la imagen no se guardará en ningún fichero (por ejemplo, cuando sólo queramos medir tiempos).

Tienes disponibles dos imágenes para que pruebes los programas a desarrollar. La imagen `small.ppm` es pequeña y rápida 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 `large.ppm`, que es mayor y supone un mayor coste. Para ahorrar espacio en disco, la imagen `large.ppm` se encuentra en la carpeta `/scratch/cpa/` de kahan. Para usarla, simplemente habrá que especificar la ruta `/scratch/cpa/large.ppm` como nombre del fichero de entrada del programa `realign`. 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/large.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 `realign`). Aunque sea un código secuencial, utiliza la función adecuada de OpenMP para medir tiempos. Guarda esta versión con el nombre `realign0.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, en la función `realign`, el bucle interno de la primera parte (bucle `off`) y el bucle de la tercera parte. El bucle de la segunda parte no se puede paralelizar, puesto que para calcular `voff[y]` se debe haber calculado previamente `voff[y-1]`. En el bucle de la parte 3 ten en cuenta que, dado que la función `cyclic_shift` utiliza el array `v` como buffer auxiliar, cada hilo deberá contar con un array `v` diferente (cada hilo debe hacer su `malloc` y su correspondiente `free`). Esta versión debe llamarse `realign1.c`.

Ejercicio 2: Obtén una segunda versión paralela paralelizando la función `realign` mediante una sola región paralela que contenga las tres partes de la función. En esta ocasión, queremos paralelizar el bucle externo de la parte 1, además del bucle de la parte 3. De nuevo, ten en cuenta que el bucle de la parte 2 no se puede paralelizar. Esta versión debe llamarse `realign2.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.

- Supongamos que en el bucle de la función `distance` eliminásemos la condición `d<c`, de manera que el bucle siempre se completara hasta llegar a `n`. Considera la paralelización del bucle `off` de `realign` y las siguientes planificaciones:
  - `static` con tamaño de *chunk* por defecto.
  - `static` con tamaño de *chunk* 1.
  - `dynamic` con tamaño de *chunk* por defecto.

  ¿Crees que alguna planificación produciría un mejor equilibrio de carga que otra? ¿De ser así, cuáles serían mejores/peores? ¿Por qué?
- Responde a la misma pregunta, pero considerando ahora la función `distance` original, que sí incluye la condición `d<c` en el bucle.

En la pregunta anterior se incide en cómo afecta la planificación del bucle al equilibrio de carga. Sin embargo, la planificación afecta también a otros aspectos, 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.

Por eso, en general no resulta fácil saber qué planificación es la más adecuada para un bucle, y en general hay que recurrir a la experimentación, como se pide en el siguiente ejercicio.

Ejercicio 4: En este ejercicio se pide evaluar la planificación desde un punto de vista experimental. Consideraremos solo la planificación de los bucles de la parte 1 de la función `realign`, dado que esa parte es, con diferencia, la que más contribuye al tiempo de ejecución total.

Utilizando el sistema de colas del cluster `kahan`, saca tiempos de ejecución de las dos versiones paralelas realizadas, usando 32 hilos y distintas planificaciones para el bucle que se paraleliza en la parte 1 de `realign`. En concreto, usa las planificaciones mencionadas en el ejercicio anterior.

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

Ejercicio 5: Utilizando el sistema de colas 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 indicando cuál es la mejor versión paralela o si tienen un comportamiento similar, y trata de justificar los resultados que obtengas.

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. Una versión paralela interesante (aunque poco eficiente)

Ejercicio 6: En este ejercicio se pide hacer una versión (en realidad dos) basada en la paralelización de las funciones `distance` y `cyclic_shift`. Se trata de una versión que no será competitiva, desde el punto de vista de las prestaciones, con las anteriores, pero sí es interesante desde el punto de vista de la programación.

Respecto a la paralelización de la función `cyclic_shift`, observa que no todos los bucles se pueden hacer en paralelo.

En cuanto a la paralelización de `distance`, debe mantenerse la condición `d<c` que hace que el número de iteraciones del bucle sea a priori desconocido. Eso hace que el programa de partida sea más rápido, aunque complica la paralelización.

Otro aspecto a tener en cuenta es si la variable `d` de la función `distance` debe ser compartida o no. Ambas opciones son posibles, aunque en uno de los casos habrá que añadir algo más para garantizar que la paralelización sea correcta. Además, cada opción tiene ventajas/desventajas desde el punto de vista de las prestaciones. Hay que tener en cuenta que en cada iteración se compara el valor actual de `d` con el valor de `c`, para dejar de iterar si deja de cumplirse que `d<c`.

Implementa las versiones `realign3a.c` y `realign3b.c`, que corresponderán a la variable `d` compartida y no compartida, respectivamente. Comenta ambas versiones, explicando el posible uso de directivas de sincronización y justificando los bucles que no se puedan paralelizar en la función `cyclic_shift`. Discute también qué ventajas o desventajas tendría la compartición o no de la variable `d`, desde el punto de vista de las prestaciones. No es necesario ejecutar las versiones para tomar tiempos.

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 ni tampoco ficheros de imágenes** (ocupan mucho y no son necesarios). El archivo debe estar comprimido en formato `.tgz` o `.zip`.

  Comprueba que todos los ficheros compilan correctamente y 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).
- El ejercicio 6 supone un 20 % de la nota del trabajo (0.3 puntos). Puedes optar por no entregarlo, pero en ese caso tu nota máxima será 1.2 puntos.
- 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.