Memoria

Práctica 2: Reconstrucción de imágenes

Felix Gerschau
Arouna Ali Perveen

Indice

Introducción

En esta práctica se propone el problema de ordenar una imagen cuyas líneas horizontales, salvo la primera, están desordenadas. Para esto se nos proporciona un algoritmo secuencial que tenemos que paralelizar mediante OpenMP. Todas las medidas en esta memoria se han hecho con la imagen 'crc.ppm' de la carpeta 'otras'.

```
Funcion encaja

  Para i = 0, 1, 2 ... alto-3
    distancia_mínima <- valor_muy_grande
    Para j = i+1, i+2 ... alto-1
      distancia <- 0
      Para x = 0, 1, 2 ... ancho-1
        distancia <- distancia + diferencia(pixel(x,i),pixel(x,j))
      Fin para
        Si distancia < distancia_mínima entonces
          distancia_mínima <- distancia
            línea_mínima <- j
        Fin si
    Fin para
    Intercambia_líneas(i+1,línea_mínima)
  Fin para
Fin funcion
```

Algoritmo secuencial

El bucle i recorre todas las líneas de la imagen, mientras que el bucle j recorre las líneas i+1 hasta la última para encontrar la línea que más se parece a i (la más cercana). El bucle x recorre los píxeles de la línea j y suma la distancia entre los píxeles de las líneas que se están comparando. La línea que más se parece a la línea i y será intercambiada con la línea en la posición i+1.

Ejercicio 1

En este ejercicio hemos medido el tiempo de ejecución de la función encaja. Para eso tomamos el tiempo antes y después de la llamada a la función mediante omp_get_wtime y restamos el primer resultado del segundo para obtener el tiempo de ejecución.

```
[feger@kahan material]$ more encaja.sh.o141082
Tiempo de encaja: 151.485268
```

La ejecución secuencial de la función 'encaja' ha tardado aproximadamente 2 minutos y 21 segundos.

```
[feger@kahan material]$ more encaja.sh
#!/bin/sh
#PBS -l nodes=1,walltime=00:05:00
#PBS -q cpa
#PBS -d .
OMP_NUM_THREADS=1 ./encaja
```

El script para ejecutar el programa con un solo hilo en el cluster.

Ejercicio 2

En el bucle más externo se intercambia la línea i+1 con la línea más parecida a *i*, que se calcula en los otros bucles *j* y *x*. Los otros hilos siempre trabajan con la línea *'i'*, suponiendo que ésta se encuentra en el sitio correcto. Por lo tanto el bucle *i* no es paralelizable, ya que una paralelización no puede garantizar que los bucles *j* y *x* hacen referencia a una línea *i* que está en el sitio correcto.

```
for (i = 0; i < n; i++) {
  /* Buscamos la linea que mas se parece a la i y la ponemos en i+1 */
  distancia_minima = grande;
  #pragma omp parallel for private (x,distancia)
  for (j = i + 1; j < ima->alto; j++) {
    distancia = 0;
    for (x = 0; x < ima->ancho; x++)
      distancia += diferencia(&A(x, i), &A(x, j));
    if (distancia < distancia_minima) {
      #pragma omp critical
      if(distancia<distancia_minima){
              distancia_minima = distancia;
              linea_minima = j;
      }
    }
  }
}
```

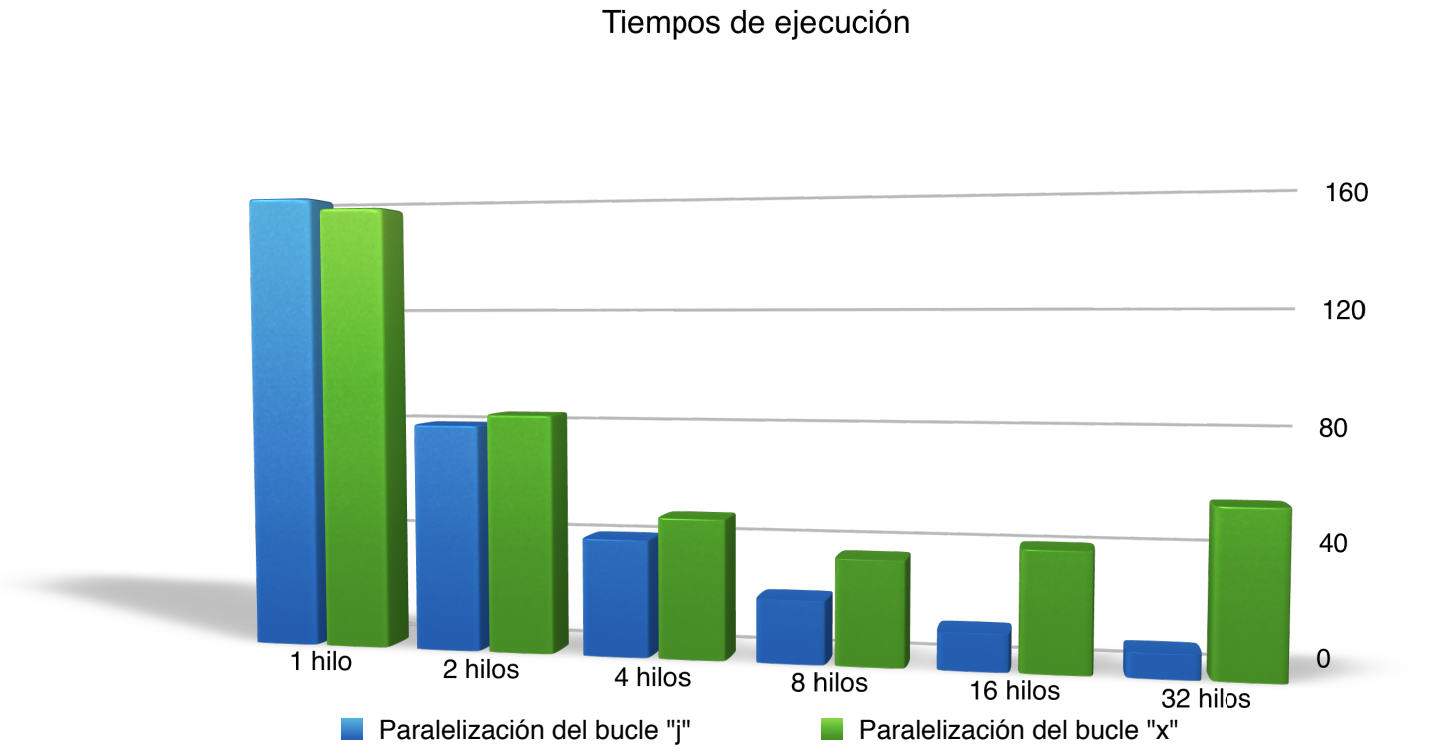
```
Numero de hilos: 1
Tiempo de encaja: 159.182330
Numero de hilos: 2
Tiempo de encaja: 80.288850
Numero de hilos: 4
Tiempo de encaja: 41.829592
Numero de hilos: 8
Tiempo de encaja: 21.842446
Numero de hilos: 16
Tiempo de encaja: 12.978302
Numero de hilos: 32
Tiempo de encaja: 7.678419
```

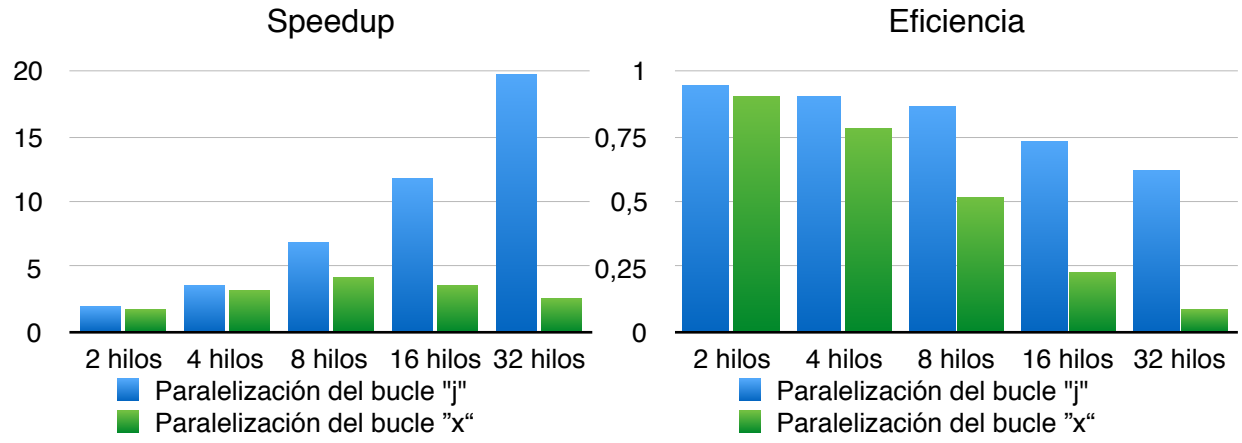
Para paralelizar el bucle *j* hemos usado la cláusula *omp parallel for*, declarando las variables *x* y *distancia* como privadas. Las variables *distancia_minima* y *línea_minima* son públicas. La parte del código que modifica dichas variables públicas tiene que protegerse para evitar condiciones de carrera. A la izquierda, y en el gráfico más abajo, se puede observar como mejora el rendimiento en relación con la cantidad de hilos empleados.

```
for (i = 0; i < n; i++) {
  /* Buscamos la linea que mas se parece a la i y la ponemos en i+1 */
  distancia_minima = grande;
  for (j = i + 1; j < ima->alto; j++) {
    distancia = 0;
    #pragma omp parallel for reduction(+:distancia)
    for (x = 0; x < ima->ancho; x++)
      distancia += diferencia(&A(x, i), &A(x, j));
    if (distancia < distancia_minima) {
      distancia_minima = distancia;
      linea_minima = j;
    }
  }
  intercambia_lineas(ima, i+1, linea_minima);
}
```

```
Numero de hilos: 1
Tiempo de encaja: 157.160602
Numero de hilos: 2
Tiempo de encaja: 84.105372
Numero de hilos: 4
Tiempo de encaja: 48.664101
Numero de hilos: 8
Tiempo de encaja: 36.807991
Numero de hilos: 16
Tiempo de encaja: 42.110502
Numero de hilos: 32
Tiempo de encaja: 57.939304
```

El bucle *x* también se paraleliza con una cláusula *omp parallel for*. Añadimos un *reduction* sobre la suma de la variable *distancia*. Sin embargo esta paralelización es poco eficiente y el tiempo de ejecución aumenta cuando se usan muchos hilos. A partir de la ejecución con 16 hilos, el coste de la creación de los hilos supera la carga del bucle y por lo tanto el tiempo de ejecución aumenta.





Comparando las dos opciones en las gráficas se ve claramente que la paralelización del bucle *j* es más eficiente y más rápida. Sin embargo, se ve que en la ejecución con pocos hilos no hay mucha diferencia entre las dos paralelizaciones.

```
#!/bin/sh
#PBS -l nodes=1,walltime=00:10:00
#PBS -q cpa
#PBS -d .
for i in 1 2 4 8 16 32
do
        echo "Numero de hilos: $i"
        OMP_NUM_THREADS=$i ./pencaja2 -t
done
```

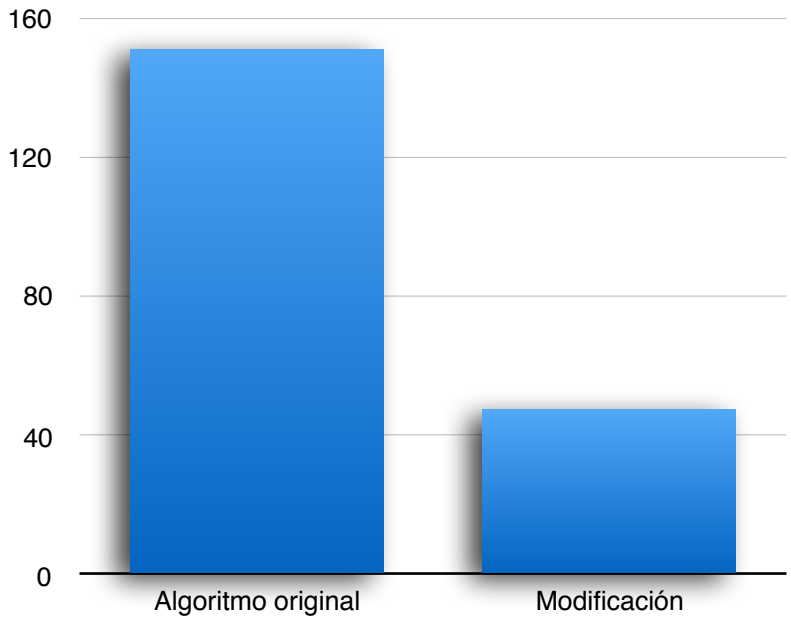
Script para la ejecución de ”encaja-e2-pj.c“ en el cluster Kahan.

Ejercicio 3

Hemos añadido al bucle *x* una condición adicional que comprueba si la variable *distancia* ha superado *distancia_minima*.

```
for (x = 0; x < ima->ancho && distancia_minima>distancia; x++){
  distancia += diferencia(&A(x, i), &A(x, j));
  }
```

```
[aram@kahan material]$ more ejercicio3.sh.o142450
Tiempo de encaja: 47.560462
```



La modificación propuesta es aproximadamente tres veces más rápida.

Ejercicio 4

```
for (i = 0; i < n; i++) {
  /* Buscamos la linea que mas se parece a la i y la ponemos en i+1 */
  distancia_minima = grande;
  #pragma omp parallel for private (x,distancia)
  for (j = i + 1; j < ima->alto; j++) {
    distancia = 0;
    for (x = 0; x < ima->ancho&& distancia_minima>distancia; x++)
      distancia += diferencia(&A(x, i), &A(x, j));
    if (distancia < distancia_minima) {
      #pragma omp critical
      if(distancia<distancia_minima){
              distancia_minima = distancia;
              linea_minima = j;
      }
    }
  }

  intercambia_lineas(ima, i+1, linea_minima);
}
```

```
Numero de hilos: 1
Tiempo de encaja: 50.471905
Numero de hilos: 2
Tiempo de encaja: 25.336213
Numero de hilos: 4
Tiempo de encaja: 12.843942
Numero de hilos: 8
Tiempo de encaja: 7.587500
Numero de hilos: 16
Tiempo de encaja: 4.471240
Numero de hilos: 32
Tiempo de encaja: 2.846159
```

El bucle *j* se puede paralelizar igual que en el ejercicio 2. Se observa una mejora del tiempo de ejecución global pero esto no afecta mucho el comportamiento respeto a la cantidad de hilos usados. El speedup y la eficiencia son ligeramente peor cuando se ejecuta con muchos hilos.