Práctica 2:
Descifrando imágenes ocultas

Memoria

Carolina Alba Marugan Rubio
4º GIA

Índice

Introducción

Esta memoria recoge las respuestas a las preguntas que se hacen en el enunciado de la práctica 2. No se han incluido los enunciados. Para la entrega, se han dejado los ficheros realign1.c y realign2.c conforme a las implementaciones que se piden en los ejercicios 1 y 2. No obstante, si se quisieran ejecutar conforme a los ejercicios 4 y 5, habría que añadir las directivas que en las respuestas de esta memoria se indican (Ejercicio 4 y Ejercicio 5).

Deberemos tener en cuenta que para todas las versiones del código a paralelizar se he añadido la siguiente directiva que referencia la librería OpenMP en C: `#include <omp.h>`

Ejercicio 1

Para la realización del ejercicio 1 se ha modificado la parte 1 y 3 de la función realign() .

```
...
// Part 1. Find optimal offset of each line with respect to the previous line
#pragma omp parallel for private(off, d, dmin, bestoff)
  for ( y = 1 ; y < h ; y++ ) {
...
  // Part 3. Shift each line to its place, using auxiliary buffer v
  #pragma omp parallel private(v)
  {
  v = malloc( 3 * max * sizeof(Byte) ); // tiene que ser local para cada hilo
  if ( v == NULL )
    fprintf(stderr,"ERROR: Not enough memory for v\n");
  else {
    #pragma omp for private(y)
    for ( y = 1 ; y < h ; y++ ) {
      cyclic_shift( w, &a[3*y*w], voff[y], v );
    }
    free(v);
   }
  }// region paralela hasta aqui
  free(voff);
} // FIN REALIGN
```

Como cada hilo debe contar con un array diferente, y la variable "v" ha sido declarada al inicio de la función, debemos indicar que la variable v será privada en la directiva omp. La variable "y" del bucle for de la tercera parte es privada por defecto, aunque no hubiera sido necesario indicarlo, se ha decidido hacerlo. En cambio, para la primera parte, sí es necesario indicar que la variable off del bucle anidado es privada. En esta primera parte, "d" se va a modificar por cada hilo, por ello debe ser privada. Las variables dmin y bestoff, como únicamente van a ser usadas dentro del bucle y, las podemos declarar como privadas (estando inicializadas dentro del mismo bucle).

Ejercicio 2

En este caso se nos pide que utilicemos una única región paralela obteniendo el mismo resultado que en el ejercicio anterior. Se trata también de la función realign() y se han incluido las 3 partes de la función en la región paralela:

```
...
#pragma omp parallel private(v)
  {// Part 1. Find optimal offset of each line with respect to the previous line
    #pragma omp for private(y, off, d, dmin, bestoff)
      for ( y = 1 ; y < h ; y++ ) {

        // Find offset of line y that produces the minimum distance between lines y
and y-1
        dmin = distance( w, &a[3*(y-1)*w], &a[3*y*w], INT_MAX ); // offset=0
        bestoff = 0;
        for ( off = 1 ; off < w ; off++ ) {
          d  = distance( w-off, &a[3*(y-1)*w], &a[3*(y*w+off)], dmin );
          d += distance( off, &a[3*(y*w-off)], &a[3*y*w], dmin-d );
          // Update minimum distance and corresponding best offset
          if ( d < dmin ) { dmin = d; bestoff = off; }
        }
        voff[y] = bestoff; //distancia de una fila a otra lo menor posible
      }

    #pragma omp single
    {
      // Part 2. Convert offsets from relative to absolute and find maximum offset
of any line
      max = 0;
      voff[0] = 0;
      for ( y = 1 ; y < h ; y++ ) {
        voff[y] = ( voff[y-1] + voff[y] ) % w;
        d = voff[y] <= w / 2 ? voff[y] : w - voff[y];
        if ( d > max ) max = d;
      }
    }
    // Part 3. Shift each line to its place, using auxiliary buffer v
     v = malloc( 3 * max * sizeof(Byte) ); // tiene que ser local para cada hilo
     if ( v == NULL )
            fprintf(stderr,"ERROR: Not enough memory for v\n");
     else {
            #pragma omp for private(y)
            for ( y = 1 ; y < h ; y++ ) {
              cyclic_shift( w, &a[3*y*w], voff[y], v );
            }
            free(v);
     }
  }// region paralela hasta aqui
...
```

En cuanto al ámbito de las variables, no cabe destacar nada, dado que no cambian. Esta vez se ha indicado la declaración de "v" al inicio de la directiva de región paralela, por comodidad. Sí cabe mencionar que la parte 2 de la función ha sido necesario declararla con la directiva "single", dado que no es paralelizable, y ha de ser ejecutada por un único hilo. Además, el resto de hilos han de esperar en una barrera implícita al final de la segunda parte, pues el vector voff es modificado aquí y necesario para la tercera parte.

Ejercicio 3

En este ejercicio se nos pregunta por la conveniencia de un tipo de planificación u otra según las condiciones que nos proponen.

En el caso de no tener la condición d < c en distance() y querer paralelizar el bucle off de la función realign(), la cuestión estaría en que el nº de veces que se ejecute el bucle en distance() depende del primer argumento. Distance() se llama dos veces en el bucle anidado de la primera parte de realign(). En la primera llamada, cuanto menor sea el off, más veces se ejecutará el bucle en distance(), en la segunda, cuanto más pequeño sea el valor de off, menos veces se ejecutará el bucle en distance(); y viceversa. Al hacerse dos llamadas, las iteraciones que no se hacen en la primera, se hacen en la segunda, para cualquier valor de off. Por tanto, el número de iteraciones es el mismo para todos los hilos. En este caso, para un mejor aprovechamiento de la distribución de bloques de memoria caché, podría ser mejor utilizar una planificación estática con tamaño de chunk por defecto.

En el caso de tener la condición d < c en distance() y querer paralelizar el bucle off de la función realign() se ha de tener en cuenta que cuanto menor sea la variable dmin (que sería compartida), menos iteraciones habrá en el bucle de la función distance() y menos trabajo tendría el hilo que se crea en realign() que lo ejecuta. Además, dada la condición del if(d < dmin), la variable dmin cada vez será menor. Por tanto, las primeras ejecuciones de los hilos tardarán más que las siguientes. En este caso, para evitar el alto desequilibrio de trabajo que ocurriría con la planificación estática, sería más apropiado usar una planificación dinámica con tamaño de chunk por defecto. Ya que, ésta puede suponer una mayor sobrecarga, dada la asignación durante la ejecución.

Ejercicio 4

Para este ejercicio se pide sacar tiempos de ejecución de las versiones realign1.c y realign2.c Únicamente modificaremos la planificación de los bucles de la parte 1 de la función realign(). Para utilizar 32 hilos, añadiremos en ambos programas la siguiente instrucción en el main()

…

```
int nh;
  omp_set_num_threads(32);
  #pragma omp parallel
  {
    nh = omp_get_num_threads();
  }
  t=omp_get_wtime();
```

…

Se ha realizado las siguientes modificaciones en los programas:

planificación en realign1.c

```
#pragma omp parallel for private(off, d, dmin, bestoff) schedule(static)
```

planificación en realign2.c

```
#pragma omp for private(y, off, d, dmin, bestoff) schedule(static)
```

planificación en realign1.c

```
#pragma omp parallel for private(off, d, dmin, bestoff) schedule(static,1)
```

planificación en realign2.c

```
#pragma omp for private(y, off, d, dmin, bestoff) schedule(static,1)
```

planificación en realign1.c

```
#pragma omp parallel for private(off, d, dmin, bestoff) schedule(dynamic)
```

planificación en realign2.c

```
#pragma omp for private(y, off, d, dmin, bestoff) schedule(dynamic)
```

Para la imagen small.ppm se han obtenido estos tiempos (se han hecho dos pruebas)

Realign1.c

Nº hilos	static	Static,1	dynamic
32	1.016878	1.005227	1.010087
32	1.009265	1.025637	1.008289

Realign2.c

Nº hilos	static	Static,1	dynamic
32	1.009986	1.012287	1.001909
32	1.004957	1.020542	1.007140

Las diferencias son realmente pequeñas, para una mejor comprobación, realizaremos lo mismo con la imagen large.ppm

Para la imagen large.ppm se han elaborado los archivos tiemposEjer4realign1.sh y tiemposEjer4realign2.sh que se lanzan con la orden sbatch del terminal. También se ha utilizado la directiva Schedule(runtime).

```
camaru@alumno.upv.es@kahan:~/cpa/pract2$ gcc -fopenmp -o realign1 realign1.c -lm
camaru@alumno.upv.es@kahan:~/cpa/pract2$ sbatch tiemposEjer4realign1.sh
Submitted batch job 36238
camaru@alumno.upv.es@kahan:~/cpa/pract2$ cat realign1.txt
Código paralelo con planificación estática y chunk=0 (reparto por bloques)
      Tiempo para 32 hilos: 2.646620
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1)
      Tiempo para 32 hilos: 2.459845
Código paralelo con planificación dinámica y chunk=1
       Tiempo para 32 hilos: 2.413660
```

Realign1.c

Nº hilos	static	Static,1	dynamic
32	2.646620	2.459845	2.413660

```
camaru@alumno.upv.es@kahan:~/cpa/pract2$ cat realign2.txt
Código paralelo con planificación estática y chunk=0 (reparto por bloques)
      Tiempo para 32 hilos: 2.659811
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1)
      Tiempo para 32 hilos: 2.381790
Código paralelo con planificación dinámica y chunk=1
       Tiempo para 32 hilos: 2.522748
```

Realign2.c

Nº hilos	static	Static,1	dynamic
32	2.659811	2.381790	2.522748

Las diferencias continúan siendo pequeñas. Aunque ahora ya vemos una diferencia más clara entre la planificación estática de bloques, y la planificación estática y dinámica circular. Son mejores estas dos últimas.

Ejercicio 5

En el ejercicio anterior habíamos configurado el numero de hilos desde el programa main. En este ejercicio lo haremos desde los archivos tiemposEjer5realign1.sh y tiemposEjer5realign2.sh

Por tanto, quitaremos del main la función omp_set_num_threads() y lanzamos la orden sbatch desde el terminal, guardando el resultado en los archivos realign1.txt y realign2.txt

Realign1.c

```
camaru@alumno.upv.es@kahan:~/cpa/pract2$ cat realign1.txt
Código paralelo con planificación estática y chunk=0 (reparto por bloques) y 2 hilos
      Tiempo para 2 hilos: 33.761677
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1) y 2 hilos
      Tiempo para 2 hilos: 31.098589
Código paralelo con planificación dinámica y chunk=1 y 2 hilos
      Tiempo para 2 hilos: 30.881627
Código paralelo con planificación estática y chunk=0 (reparto por bloques) y 4 hilos
      Tiempo para 4 hilos: 17.029899
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1) y 4 hilos
      Tiempo para 4 hilos: 15.872219
Código paralelo con planificación dinámica y chunk=1 y 4 hilos
      Tiempo para 4 hilos: 15.466585
Código paralelo con planificación estática y chunk=0 (reparto por bloques) y 8 hilos
      Tiempo para 8 hilos: 8.671352
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1) y 8 hilos
      Tiempo para 8 hilos: 8.236933
Código paralelo con planificación dinámica y chunk=1 y 8 hilos
      Tiempo para 8 hilos: 7.760039
Código paralelo con planificación estática y chunk=0 (reparto por bloques) y 16 hilos
      Tiempo para 16 hilos: 4.949004
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1) y 16 hilos
      Tiempo para 16 hilos: 4.997327
Código paralelo con planificación dinámica y chunk=1 y 16 hilos
      Tiempo para 16 hilos: 4.597174
Código paralelo con planificación estática y chunk=0 (reparto por bloques) y 32 hilos
      Tiempo para 32 hilos: 3.100222
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1) y 32 hilos
      Tiempo para 32 hilos: 2.916598
Código paralelo con planificación dinámica y chunk=1 y 32 hilos
       Tiempo para 32 hilos: 2.477674
```

Realign1.c (segundos)

Nº hilos	static	Static,1	dynamic
2	33.761677	31.098589	30.881627
4	17.029899	15.872219	15.466585
8	8.671352	8.236933	7.760039
16	4.949004	4.997327	4.597174
32	3.100222	2.916598	2.477674

Para obtener el Speed-up obtenemos primero el tiempo de ejecución en secuencial y aplicamos la fórmula S(n,p)=t(n)/t(n,p). Para obtener la eficiencia aplicamos la fórmula E(n,p)=S(n,p)/p. El tiempo en secuencial para la imagen large.ppm (la utilizada en las medidas anteriores) es: Tiempo para 1 hilos: 54.670800

Realign1.

Nº hilos	Speed-up static	Speed-up Static,1	Speed-up dynamic
2	1,619315296	1,757983296	1,770334186
4	3,210283279	3,444433321	3,534768664
8	6,30476078	6,637276277	7,045170778
16	11,04682882	10,94000853	11,89226251
32	17,63447908	18,74471559	22,0653726

Nº hilos	Eficiencia static	Eficiencia Static,1	Eficiencia dynamic
2	0,809657648	0,878991648	0,885167093
4	0,80257082	0,86110833	0,883692166
8	0,788095098	0,829659535	0,880646347
16	0,690426801	0,683750533	0,743266407
32	0,551077471	0,585772362	0,689542894

realign2.c

```
camaru@alumno.upv.es@kahan:~/cpa/pract2$ cat realign2.txt
Código paralelo con planificación estática y chunk=0 (reparto por bloques) y 2 hilos
      Tiempo para 2 hilos: 33.703075
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1) y 2 hilos
      Tiempo para 2 hilos: 31.057476
Código paralelo con planificación dinámica y chunk=1 y 2 hilos
      Tiempo para 2 hilos: 30.845506
Código paralelo con planificación estática y chunk=0 (reparto por bloques) y 4 hilos
      Tiempo para 4 hilos: 17.010730
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1) y 4 hilos
      Tiempo para 4 hilos: 15.850499
Código paralelo con planificación dinámica y chunk=1 y 4 hilos
      Tiempo para 4 hilos: 15.450472
Código paralelo con planificación estática y chunk=0 (reparto por bloques) y 8 hilos
      Tiempo para 8 hilos: 8.673621
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1) y 8 hilos
      Tiempo para 8 hilos: 8.230914
Código paralelo con planificación dinámica y chunk=1 y 8 hilos
      Tiempo para 8 hilos: 7.750386
Código paralelo con planificación estática y chunk=0 (reparto por bloques) y 16 hilos
      Tiempo para 16 hilos: 5.161443
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1) y 16 hilos
      Tiempo para 16 hilos: 4.966567
Código paralelo con planificación dinámica y chunk=1 y 16 hilos
      Tiempo para 16 hilos: 4.598581
Código paralelo con planificación estática y chunk=0 (reparto por bloques) y 32 hilos
      Tiempo para 32 hilos: 2.982405
Código paralelo con planificación estática y chunk=1 (reparto cíclico de 1 en 1) y 32 hilos
      Tiempo para 32 hilos: 2.613291
Código paralelo con planificación dinámica y chunk=1 y 32 hilos
       Tiempo para 32 hilos: 2.342080
```

realign2.c (segundos)

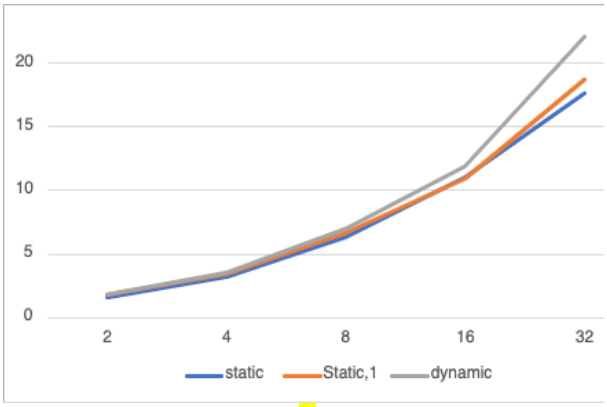
Nº hilos	static	Static,1	dynamic
2	33.703075	31.057476	30.845506
4	17.010730	15.850499	15.450472
8	8.673621	8.230914	7.750386
16	5.161443	4.966567	4.598581
32	2.982405	2.613291	2.342080

realign2

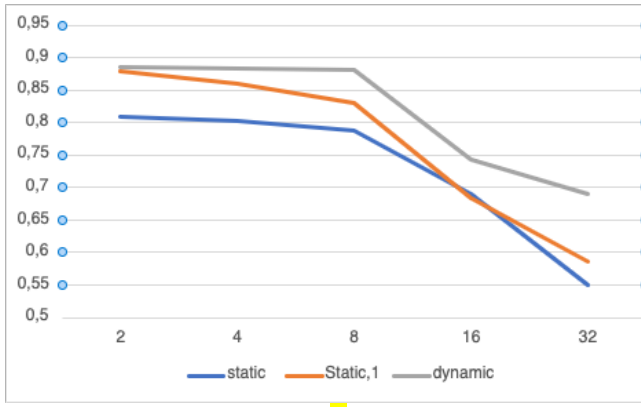
Nº hilos	Speed-up static	Speed-up Static,1	Speed-up dynamic
2	1,622130918	1,760310464	1,7724073
4	3,213900873	3,449153241	3,538455006
8	6,303111469	6,642129902	7,053945442
16	10,59215417	11,00776452	11,8886239
32	18,33111197	20,92028787	23,34284055

Nº hilos	Eficiencia static	Eficiencia Static,1	Eficiencia dynamic
2	0,811065459	0,880155232	0,88620365
4	0,803475218	0,86228831	0,884613752
8	0,787888934	0,830266238	0,88174318
16	0,662009636	0,687985282	0,743038994
32	0,572847249	0,653758996	0,729463767

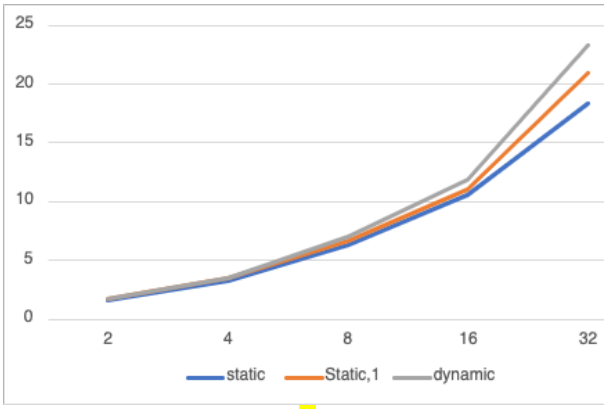
A continuación, se muestran las gráficas que describen los datos de las tablas mostradas anteriormente, para una mejor clarificación de los valores del Speed-Up y la Eficiencia , en función del número de hilos empleado y el tipo de planificación empleada.



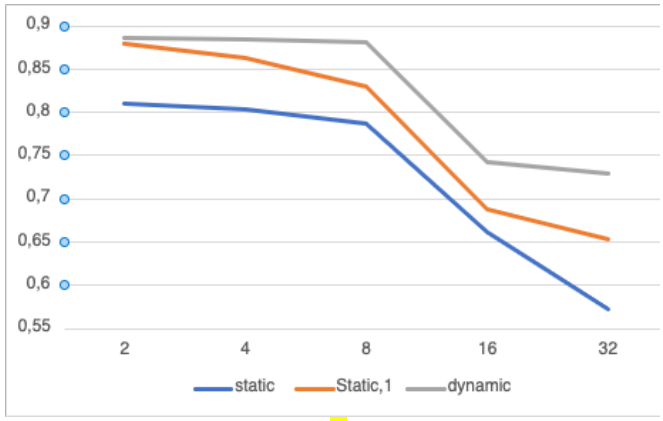
SPEED-UP REALIGN1.C



EFICIENCIA REALIGN1.C



SPEED-UP REALIGN2.C



EFICIENCIA REALIGN2.C

Se confirma lo explicado en ejercicios anteriores, que la planificación dinámica es la que ofrece un mejor desempeño. Se puede ver en los valores de la eficiencia y el Speed-Up. Cabe mencionar también que, para todas las planificaciones, a mayor número de hilos, menor es la eficiencia pero mayor es el Speed-Up. No se ha mostrado la función si de código secuencial se tratara porque siempre se ejecuta con un único hilo, pero el Speed-Up siempre sería 1 y la eficiencia sería menor que en paralelo.

Ejercicio 6

En este ejercicio se han modificado las funciones cyclic_shift() y distance() , manteniendo el resto de funciones como en la versión realign0.c , es decir, sin paralelizar (ya que no podemos llamar a una función paralela dentro de otra función paralela).

Se han realizado dos versiones. En ambas, la función cyclic_shift() permanece invariante. Es la función distance() la que cambia, en concreto, el tratamiento de la variable d.

Para la primera versión realign3a.c, la función distance() queda:

```
  ...
  int i, e;
  int volatile d;
  n *= 3; // 3 bytes per pixel (red, green, blue)
  d = 0;
  #pragma omp parallel private(i, e)
  {
    int nh = omp_get_num_threads();
    int id = omp_get_thread_num();
    for ( i = id ; i < n && d < c ; i+= nh ) {
      e = (int)a1[i] - a2[i];
      if ( e >= 0 ) {
        #pragma omp atomic
          d += e;
         // yo creo que como siempre se va a tener que ejecutar una condición u
otra, hacerlas todas atomic es menos eficiente
      }         // que con la reducción, pero checkea
      else{
        #pragma omp atomic
          d -= e;
      }
    }
  }
  return d;
...
```

Es muy interesante esta parte, dado que se ha implementado el bucle de una manera particular, indicando expresamente cuál será el siguiente valor de la "i" ejecutado por el mismo hilo, en función del número de hilos que se declaren. Evitando la directiva "#pragma omp for", que nos impediría compilar el programa debido a la condición d < c (que depende del valor leído en los vectores, e impide saber a priori el número de iteraciones que tendría el bucle). Como no hemos puesto la directiva "omp for" al compilador, hemos de indicar "i" como privada. También "e" porque su valor cambia para cada hilo. Como se nos ha pedido que tratemos "d" como compartida, cada vez que modifiquemos su valor deberemos hacerlo mediante una operación atómica (también hubiera sido posible con la directiva critical, pero por motivos de eficiencia se ha decidido usar atomic). Además, se ha declarado la variable d como volatile para que cuando se modifique su valor los hilos sean notificados y se pare antes la ejecución (cuando se cumpla la condición), haciendo más rápida la ejecución.

La función cyclic_shift() queda:

```
...
    // array v
    if ( p <= n / 2 ) { // right to left
```

```
      #pragma omp parallel for
      for ( i = 0 ; i < p ; i++ ) v[i] = a[i]; // antidependencia de a[i] con a[i-
p]
      for ( i = p ; i < n ; i++ ) a[i-p] = a[i];
      #pragma omp parallel for
      for ( i = 0 ; i < p  ; i++ ) a[d+i] = v[i]; // dependencia salida de a[i-p]
con a[d+i] y dependencia de flujo de v[i] con el primer v[i]
    } else { // left to right
      #pragma omp parallel for
      for ( i = 0 ; i < d ; i++ ) v[i] = a[p+i];
      for ( i = p-1 ; i >= 0 ; i-- ) a[i+d] = a[i];
      #pragma omp parallel for
      for ( i = 0 ; i < d  ; i++ ) a[i] = v[i];
    }
. . .
```

Se ha de comentar que aquellos bucles que editan la misma matriz que leen, no pueden ser paralelizados: `for ( i = p ; i < n ; i++ ) a[i-p] = a[i];` **y** `for ( i = p-1 ; i >= 0 ; i-- ) a[i+d] = a[i];`

Si nos hubiéramos planteado ver los bucles como tareas, y cada bucle se ejecutara en secuencial, pero entre ellos en paralelo (evitando así tantas activaciones y sincronizaciones de hilos, que hacemos cada vez que creamos una región paralela); veríamos que no es posible para ninguno, pues existen para todos ellos al menos una condición de Bernstein. En este caso se ha indicado sobre el código las dependencias observadas (de flujo, de salida y antidependencias).

Para la segunda versión, realign3b.c, la función cyclic_shift() permanece invariante y la función distance() queda:

```
...
  int i, e;
  int volatile d;
  n *= 3; // 3 bytes per pixel (red, green, blue)
  d = 0;
  #pragma omp parallel private(i, e) reduction(+:d)
  {
    int nh = omp_get_num_threads();
    int id = omp_get_thread_num();
    for ( i = 0 ; i < n && d < c ; i++ ) {
      e = (int)a1[i] - a2[i];
      if ( e >= 0 ) d += e; else d -= e;
    }
  }
...
```

Es similar a la versión anterior, es necesario declarar "i" y "e" como privadas. Pero utilizamos una reducción de suma para "d" (ya que en este caso no debe ser compartida), y así recogemos los cálculos parciales que han realizado los hilos.

Conclusiones

Para la elaboración de esta práctica, he desarrollado primero los ejercicios de código (1,2 y 6), comprobando con la imagen ref.ppm y el comando cmp que las imágenes descifradas eran correctas. Después, he analizado el comportamiento de la función distance() para dar respuesta al ejercicio 3, consultando información en las diapositivas de la asignatura acerca de las planificaciones. Por último, he realizado la toma de tiempos con cada tipo de planificación y diferente número de hilos. He intentado reflejar en este documento todos los pasos seguidos, para que el lector pueda entenderlo sin necesidad de ejecutar archivos o consultar otros.

Se han visto diferentes aspectos de la programación paralela utilizando la librería OpenMP de C. Tales como la paralelización de bucles, el uso de regiones paralelas, secciones críticas, operaciones atómicas y tipos de planificación, entre otros.

Bibliografía

https://stackoverflow.com/questions/10850155/whats-the-difference-between-static-and-dynamic-schedule-in-openmp - static vs dynamic

https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf -

https://stackoverflow.com/questions/246127/why-is-volatile-needed-in-c - volatile

https://learn.microsoft.com/es-es/cpp/cpp/volatile-cpp?view=msvc-170 - volatile