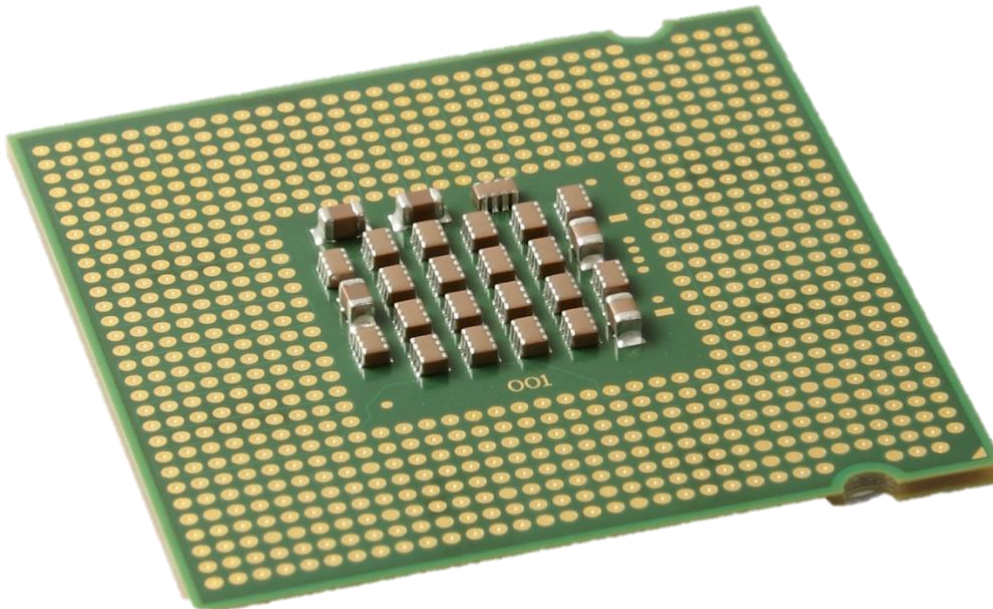


Memoria CPA

López Muñoz, Alejandro

Benites Aldaz, Dairon Andrés



Índice

Ejercicio 1.....	3
Ejercicio 2.....	5
Ejercicio 3.....	6
Ejercicio 4.....	7
Ejercicio 5.....	8
Ejercicio 6.....	10
Anexos	14

Ejercicio 0

```
int main(int argc, char *argv[]) {
    char option, *s, *in = "in.ppm", *out = "out.ppm";
    int i, w, h, bw=8, bh=8;
    Byte *a;

    for ( i = 1 ; i < argc ; i++ ) {
        if ( argv[i][0] != '-' ) { option = argv[i][0]; s = &argv[i][1]; }
        else { option = argv[i][1]; s = &argv[i][2]; }
        if ( option != '\0' )
            if ( *s == '\0' ) { i++; if ( i < argc ) s = &argv[i][0]; else s = ""; }
    }

    switch ( option ) {
        case 'i': in = s; break;           // input filename
        case 'o': out = s; break;         // output filename
        case 'w': bw = atoi(s); break;    // block width
        case 'h': bh = atoi(s); break;    // block height
        case 'b': bw = bh = atoi(s); break; // block size (both width and
height)
        default: fprintf(stderr, "ERROR: Unknown option %c.\n", option); return
1;
    }
}

a = read_ppm(in, &w, &h);
if ( a == NULL ) return 1;

if ( bw == 0 || w % bw != 0 ) {
    fprintf(stderr, "ERROR: Inexact number of vertical blocks ( %d / %d =
%.2f ).\n", w, bw, (float)w/bw);
    return 2;
}
if ( bh == 0 || h % bh != 0 ) {
    fprintf(stderr, "ERROR: Inexact number of horizontal blocks ( %d / %d =
%.2f ).\n", h, bh, (float)h/bh);
    return 3;
}
double t1, t2; //añadido
t1 = omp_get_wtime(); //añadido

process( w, h, a, bw, bh );

t2 = omp_get_wtime(); //añadido
printf("Para secuencial el tiempo de ejecucion es %f\n", t2-t1);

if ( out[0] != '\0' ) write_ppm(out, w, h, a);
free(a);
return 0;
}
```

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*

distance

```
// Compute the difference between two (horizontal or vertical) lines of an
// image
// a1 and a2 are the two lines. Each of them is n pixels long
int distance( int n, Byte a1[], Byte a2[], int stride ) {
    int d,i,j, r,g,b;
    stride *= 3;
    d = 0;
    #pragma omp parallel for private(j,r,g,b) reduction(+:d)
    for ( i = 0 ; i < n ; i++ ) {
        j = i * stride;
        r = (int)a1[j] - a2[j]; if ( r < 0 ) r = -r; // Difference in red
        g = (int)a1[j+1] - a2[j+1]; if ( g < 0 ) g = -g; // Difference in green
        b = (int)a1[j+2] - a2[j+2]; if ( b < 0 ) b = -b; // Difference in blue
        d += r + g + b;
    }
    return d;
}
```

Hemos decidido paralelizar el bucle for, para ello ha sido necesario declarar “private” las variables *j, r,g,b* ya que en cada iteración se producen modificaciones de las variables, por tanto podrían ocurrir conflictos entre ejecuciones de distintos hilos.

Hemos declarado *reduction(+:d)* ya que necesitamos la suma de todas las iteraciones de los hilos.

swap

Hemos decidido paralelizar el bucle for, para ello ha sido necesario declarar “private” las variables *x,d* y *aux* ya que en cada iteración se producen modificaciones de las variables, por tanto podrían ocurrir conflictos entre ejecuciones de distintos hilos.

```
// Swap two rectangles (a1 and a2) of an image.
// rw and rh define the width and height of both rectangles
// w is the width of the complete image that contains the rectangles
void swap( Byte a1[],Byte a2[],int rw,int rh,int w ) {
    int x,y,d;
    Byte aux;

    if ( a1 != a2 ) {
        rw *= 3; w *= 3; // Each pixel is 3 bytes
        #pragma omp parallel for private(x,d,aux)
        for ( y = 0 ; y < rh ; y++ ) {
            // Swap row y of the two rectangles
            d = w * y;
            for ( x = 0 ; x < rw ; x++ ) {
                // Swap a single byte of the two rows
                aux = a1[d+x];
                a1[d+x] = a2[d+x];
                a2[d+x] = aux;
            }
        }
    }
}
```

Ejercicio 2

Obten una segunda versió paralela paralelizando el cospu de la funció process, dejando la funció swap paralelizada (pero no la funció distance). Observa que los bucles externos de la funció process no se pueden paralelizar, con lo que deberás paralelizar los bucles internos. Llama a esta versió restore2.c.

```
// Process image a, of width w and height h, considering horizontal blocks
// of bh rows and vertical blocks of bw columns */
void process( int w,int h,Byte a[], int bw,int bh ) {
    int x,y, x2,y2, mx,my,min, d;

    // Place each horizontal block to minimize difference with previous one
    for ( y = bh ; y < h ; y += bh ) {
        min = INT_MAX; my = y;
        // Blocks up to row y-1 are already placed
        // Find the block whose first row minimizes the difference with row y-1
        #pragma omp parallel for private(d)
        for ( y2 = y ; y2 < h ; y2 += bh ) {
            d = distance( w, &a[3*(y-1)*w], &a[3*y2*w], 1 );
            #pragma omp critical
            if ( d < min ) { min = d; my = y2; }
        }
        // Block starting at row my minimizes the difference
        // Place the block in its place by swapping it with the block starting
        // at row y
        swap( &a[3*y*w],&a[3*my*w],w,bh,w );
    }

    // Place each vertical block to minimize difference with previous one
    for ( x = bw ; x < w ; x += bw ) {
        // Blocks up to column x-1 are already placed
        // Find the block whose first column minimizes the difference with column x-1
        min = INT_MAX; mx = x;
        #pragma omp parallel for private(d)
        for ( x2 = x ; x2 < w ; x2 += bw ) {
            d = distance( h, &a[3*(x-1)], &a[3*x2], w );
            #pragma omp critical
            if ( d < min ) { min = d; mx = x2; }
        }
        // Block starting at column mx minimizes the difference
        // Place the block in its place by swapping it with the block starting
        // at column x
        swap( &a[3*x],&a[3*mx],bw,h,w );
    }
}
```

Hemos decidido paralelizar los dos bucles for [x2,y2], ya que los bucles externos no se pueden paralelizar.

Para ello ha sido necesario declarar “private” la variable *d* ya que en cada iteración se producen modificaciones de la variable, por tanto, podrían ocurrir conflictos entre ejecuciones de distintos hilos.

Ha sido necesario declarar una sección crítica en la actualización de la variable *min* y *mx*.

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.

```
// Place each horizontal block to minimize distance with previous one
for ( y = bh ; y < h ; y += bh ) {
    min = INT_MAX; my = y;
    int numitera;
    int identificadorh;
    int distanciamax;
    int distanciamin;
    #pragma omp parallel private (numitera, identificadorh, distanciamax, distanciamin)
    {
        numitera=0;
        identificadorh=omp_get_thread_num();
        distanciamax=0;
        distanciamin=INT_MAX;
        #pragma omp for private(d)
        for ( y2 = y ; y2 < h ; y2 += bh ) {
            numitera++;
            d = distance( w, &a[3*(y-1)*w], &a[3*y2*w], 1 );
            #pragma omp critical
            if ( d < min ) { min = d; my = y2; }
        }
        if ( d < distanciamin ) { distanciamin=d; }
        if ( d > distanciamax ) { distanciamax=d; }
    }
    if(y==bh) {
        printf("Hilo %d: %d iteraciones con distancias entre %d y %d.\n ", identificadorh, numitera, min, distanciamax);
    }
}
```

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?

No, tiene una carga equilibrada por tanto no hace falta una planificación diferente.

¿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).

Sí, tiene un desequilibrio de cargas por tanto no sería una buena opción la planificación por defecto.

El desequilibrio es debido a qué conforme va progresando el bucle de y2 hace menos iteraciones.

Por tanto, una de las alternativas puede ser cualquiera que no sea **STATIC por defecto**.

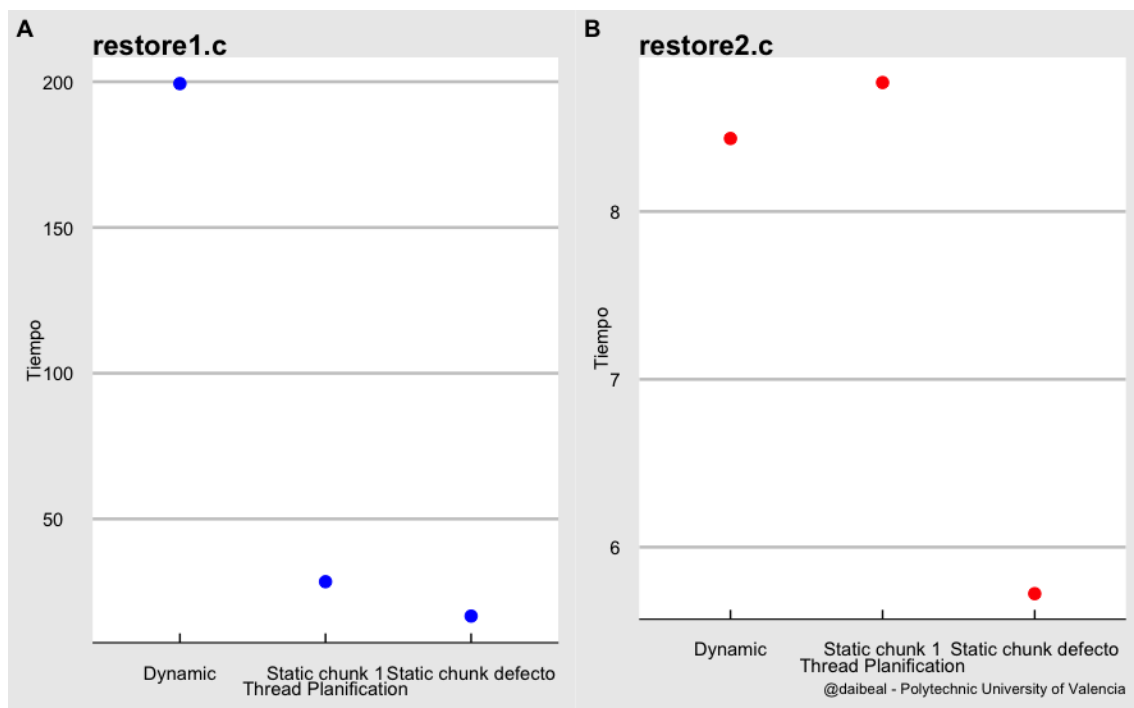
Ejemplo:

- STATIC CHUNK = 1
- GUIDED
- DYNAMIC

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 <i>chunk</i> por defecto	static con tamaño de <i>chunk</i> 1.	Dynamic con tamaño de <i>chunk</i> por defecto
restore1.c	17.974302 s	28.464924 s	199.36364536 s
restore2.c	5.7226924 s	8.768748 s	8.4352 s



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

Como es evidente la mejor planificación definida es la *Static con chunk por defecto* ya que la carga es equilibrada entre los distintos hilos de ejecución.

Los bucles paralelizados de las funciones:

- *distance*, *swap* y *process* tienen la carga equilibrada.

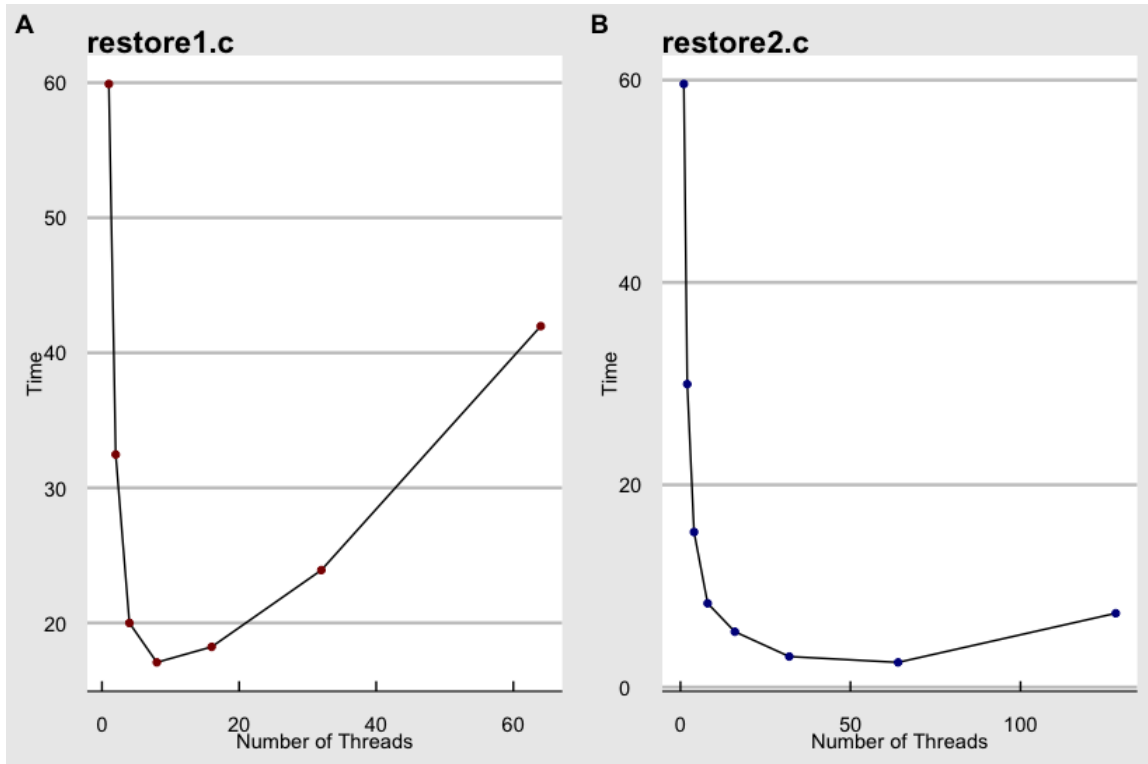
Ejercicio 5

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.

N_Threads	restore1.c	restore2.c
1	59.908	59.617
2	32.468	29.951
4	19.994	15.334
8	17.077	8.288
16	18.226	5.481
32	23.911	3.030
64	41.974	2.453



restore1.c	restore2.c
<i>N_Threads</i> <i>Time</i>	<i>N_Threads</i> <i>Time</i>
Min. : 1.00 Min. :17.08	Min. : 1.00 Min. : 2.453
1st Qu.: 3.00 1st Qu.:19.11	1st Qu.: 3.00 1st Qu.: 4.255
Median : 8.00 Median :23.91	Median : 8.00 Median : 8.289
Mean :18.14 Mean :30.51	Mean :18.14 Mean :17.737
3rd Qu.:24.00 3rd Qu.:37.22	3rd Qu.:24.00 3rd Qu.:22.643
Max. :64.00 Max. :59.91	Max. :64.00 Max. :59.617

Consideramos que el número máximo de hilos es 64 debido que a partir de este número de hilos **restore2.c** empieza a ser menos eficiente.

Speedup

El speedup indica la ganancia de velocidad que consigue el algoritmo paralelo con respecto a un algoritmo secuencial.

$$S(n, p) = \frac{t(n)}{t(n, p)}$$

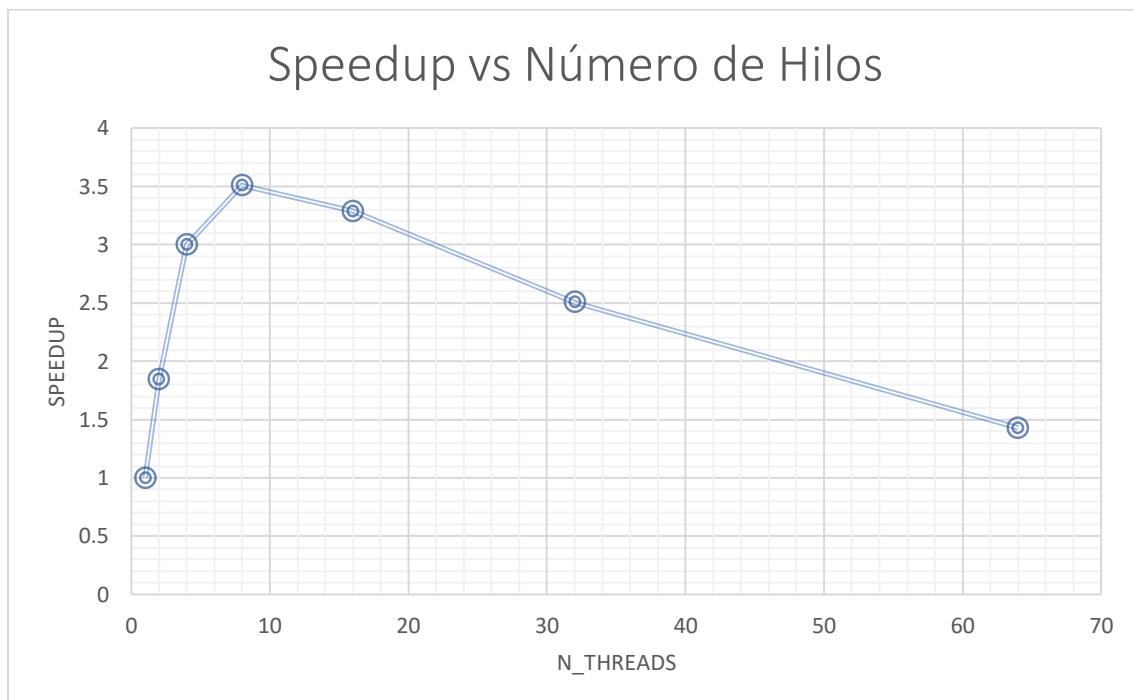
Siendo $t(n)$ el mejor algoritmo secuencial conocido o el algoritmo paralelo ejecutado en 1 procesador

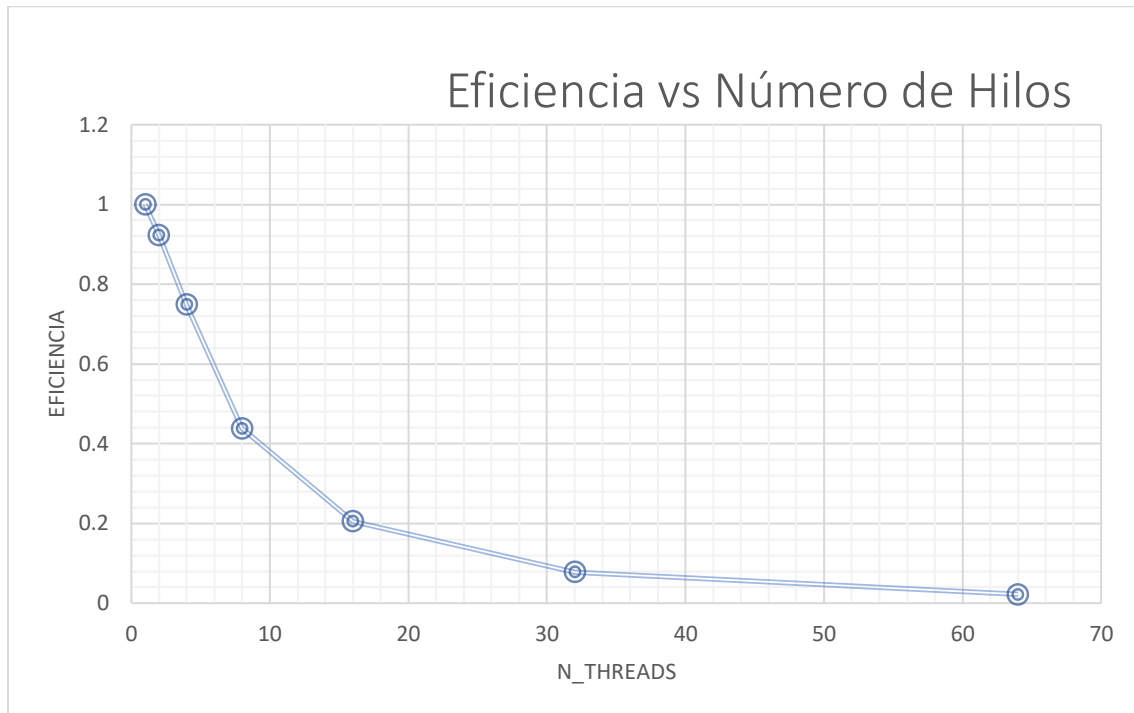
$$E(n, p) = \frac{S(n, p)}{p}$$

Eficiencia

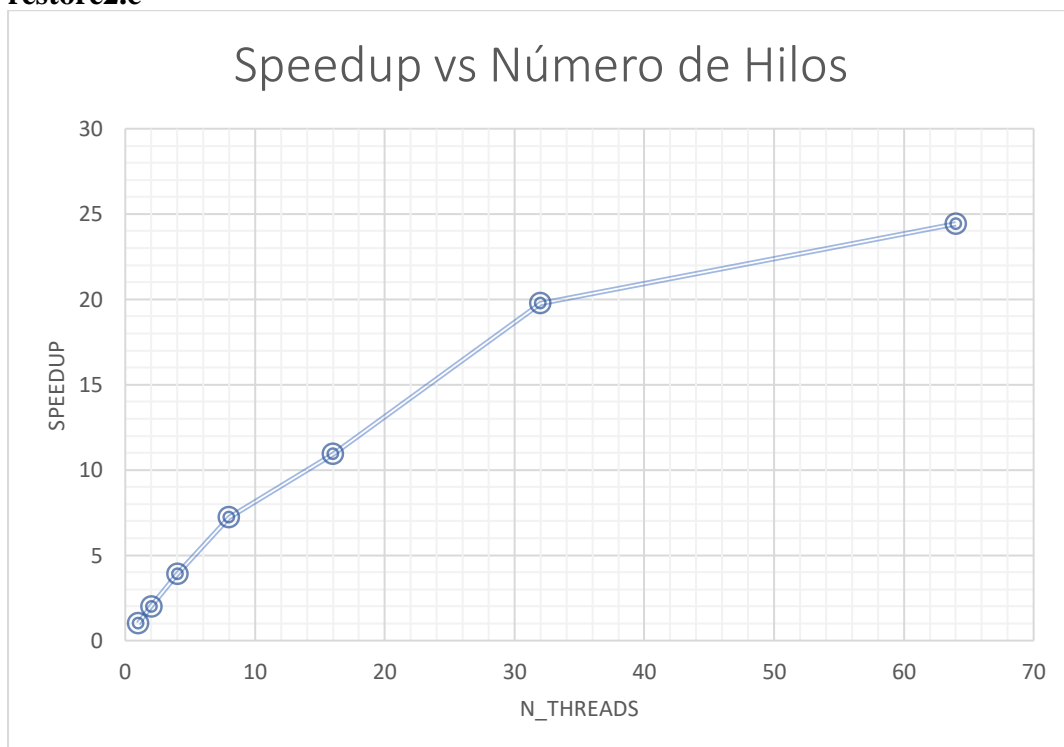
La eficiencia mide el grado de aprovechamiento que un algoritmo paralelo hace de un computador paralelo.

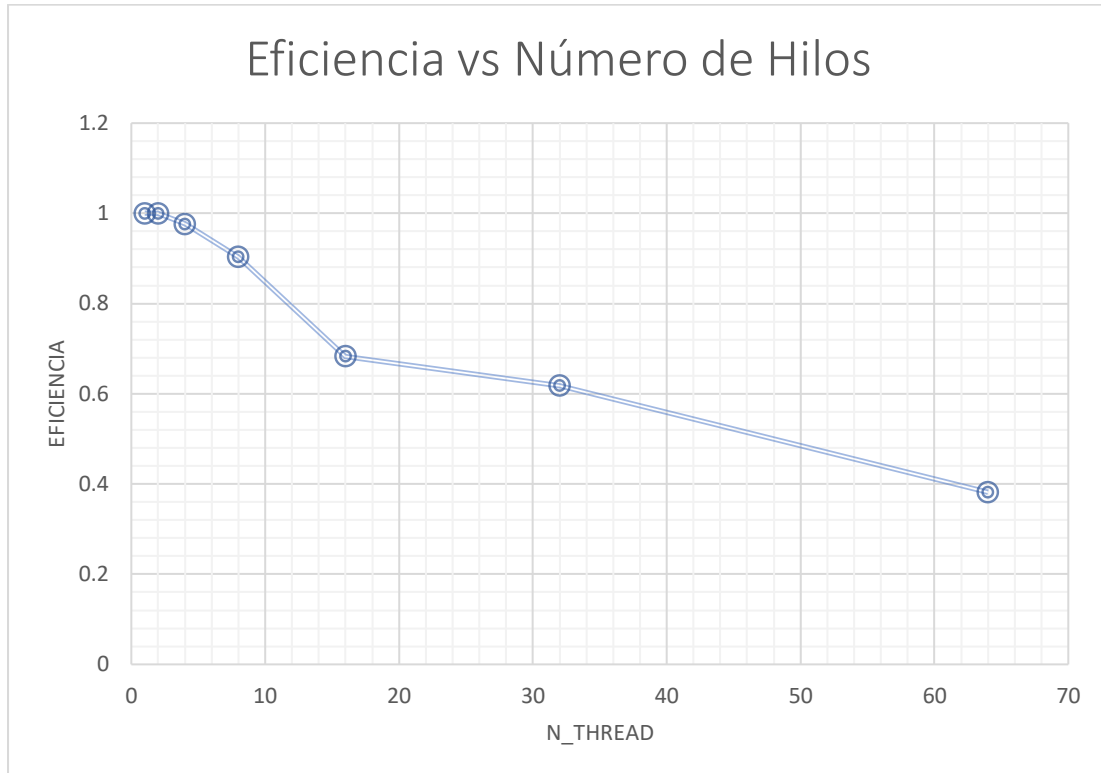
restore1.c





restore2.c





Considerarnos que la versión paralela de **restore2.c** aprovecha mejor la cantidad de hilos asignado y esto es debido a que cada vez que se llama al función de distance se instancian los hilos y se reparten las cargas sin embargo al hacerlo desde fuera se asignan menos veces y por tanto tiene menor coste.

Ejemplo de orden bash

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --time=5:00
#SBATCH --partition=cpa

OMP_NUM_THREADS=1 OMP_SCHEDULE=static ./restore2 -i grande.ppm -o grande_salida1.ppm -b 2
OMP_NUM_THREADS=2 OMP_SCHEDULE=static ./restore2 -i grande.ppm -o grande_salida1.ppm -b 2
OMP_NUM_THREADS=4 OMP_SCHEDULE=static ./restore2 -i grande.ppm -o grande_salida1.ppm -b 2
OMP_NUM_THREADS=8 OMP_SCHEDULE=static ./restore2 -i grande.ppm -o grande_salida1.ppm -b 2
OMP_NUM_THREADS=16 OMP_SCHEDULE=static ./restore2 -i grande.ppm -o grande_salida1.ppm -b 2
OMP_NUM_THREADS=32 OMP_SCHEDULE=static ./restore2 -i grande.ppm -o grande_salida1.ppm -b 2
OMP_NUM_THREADS=64 OMP_SCHEDULE=static ./restore2 -i grande.ppm -o grande_salida1.ppm -b 2
```

Nota: Adjuntamos órdenes de trabajo (.sh) en el fichero.*

OMP_SCHEDULE = static
OMP_SCHEDULE = static, 1
OMP_SCHEDULE = dynamic

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`.

```
C restore1.c C restore3.c X C restore2.c
home > alumno.upv.es > allomuo > W > CPA > Pract2 > material > C restore3.c
98
99 for ( y = bh ; y < h ; y += bh ) {
100     min = INT_MAX; my = y;
101     int numitera;
102     int identificadorh;
103     int distanciamax;
104     int distanciamin;
105     #pragma omp parallel private (numitera, identificadorh, distanciamax, distanciamin)
106     {
107         numitera=0;
108         identificadorh=omp_get_thread_num();
109         distanciamax=0;
110         distanciamin=INT_MAX;
111         #pragma omp for private(d) schedule(runtime)
112         for ( y2 = y ; y2 < h ; y2 += bh ) {
113             numitera++;
114             d = distance( w, &a[3*(y-1)*w], &a[3*y2*w], 1 );
115             #pragma omp critical
116             if ( d < min ) { min = d; my = y2; }
117             if ( d < distanciamin ) { distanciamin=d; }
118             if ( d > distanciamax ) { distanciamax=d; }
119         }
120     }
121     if(y==bh) {
122         printf("Hilo %d: %d iteraciones con distancias entre %d y %d.\n ",identificadorh, numitera, distanciamin, distanciamax);
123     }
}
```

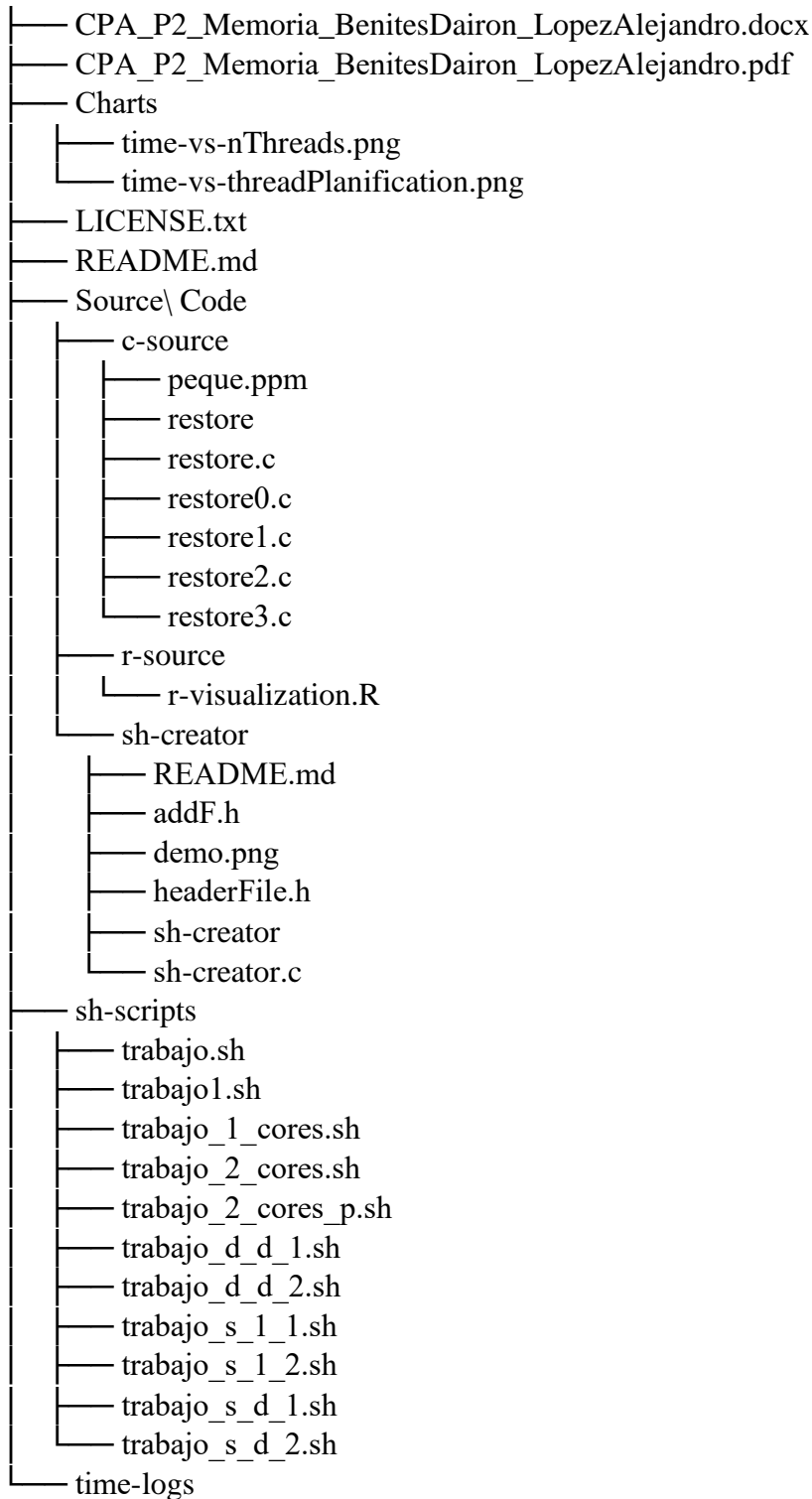
```
allomuo@alumno.upv.es@kahan:~/cpa/pract2$ cat slurm-19863.out
Hilo 0: 392 iteraciones con distancias entre 115248 y 975028.
Hilo 3: 164 iteraciones con distancias entre 221022 y 970948.
Hilo 1: 169 iteraciones con distancias entre 182140 y 971325.
Hilo 2: 174 iteraciones con distancias entre 163521 y 973973.
Para 4 hilos el tiempo de ejecucion es 19.426584
allomuo@alumno.upv.es@kahan:~/cpa/pract2$
```

Hemos declarado privadas las variables **numitera**, **identificadorh**, **distanciamax**, **distanciamin**.

Hemos incluido una condición **y==bh** para imprimir las líneas correspondientes a la primera iteración.

Anexos

Estructura de archivos



- └─ example-time-log0.csv
- └─ example-time-log1.csv
- └─ example-time-log2.csv