Práctica 2
Reconstrucción de Imágenes

MEMORIA DESCRIPTIVA DE LOS CÓDIGOS EMPLEADOS Y LOS RESULTADOS OBTENIDOS

Miguel Ángel Mateo Casalí
Virginia Palomares Hernández

Índice

Ejercicio 1

Modifica el código secuencial proporcionado de forma que se muestre por pantalla el tiempo empleado en la función encaja, que es la encargada de tratar de recuperar la imagen original.

Para poder utilizar las funciones de omp, se ha añadido al principio:

```
#include <omp.h>
```

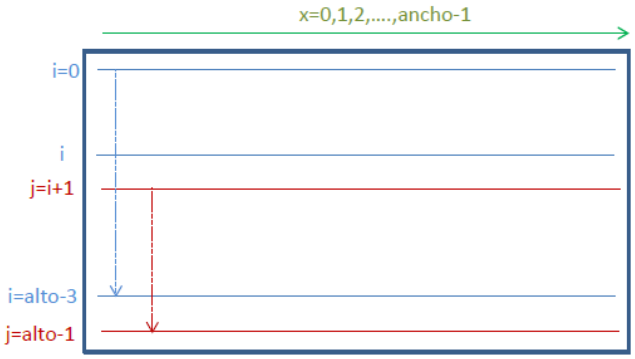
Para medir tiempos se ha utilizado la función “omp_get_wtime” en el programa principal. Esta función se ha añadido en la función main, antes y después de la llamada a la función encaja.

```
double t1,t2
t1=omp_get_wtime();
encaja(&ima);
t2=omp_get_wtime();
printf("Para secuencial el tiempo de ejecución es %f\n",t2-t1);
```

Ejercicio 2

Paraleliza la función encaja. Para cada uno de sus tres bucles (por separado), indica si se puede paralelizar o no y caso de que se pueda, paralelízalo.

Disponemos de los siguientes tres bucles:

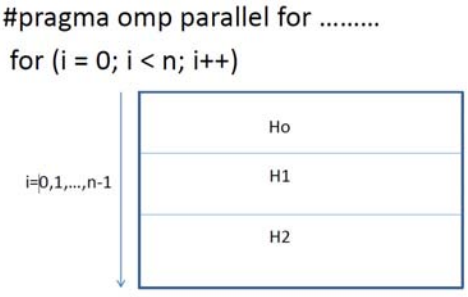


- **i**: recorre todas líneas de la imagen.
- **j**: para cada i, la variable j recorre todas las líneas comprendidas entre la i+1 y la última (ancho-1), para encontrar la más cercana (parecida) a la línea i. Una vez encontrada, se intercambia la fila i y la más cercana.
- **x**: recorre los pixeles de la línea j.

Considerando **distancia** como la suma de las diferencias entre los pixeles de las líneas comparadas (la i y la j).

La forma más eficiente consiste en paralelizar el bucle más externo, pues se produce una menor sobrecarga debida a la activación y desactivación de hilos, y también se reducen los tiempos de espera debidos a la sincronización implícita al final de la directiva.

Paralelización del bucle i



El bucle i, que es el bucle más externo no puede paralelizarse. Tiene que ejecutarse de forma secuencial. Esto es debido a que el algoritmo coloca en cada línea de la imagen, a partir de la segunda, aquella de todas las líneas que aún no se han colocado que más se parece a la línea anterior. Y para esto, calcula la distancia de cada línea con la última colocada (entendiendo la distancia como la suma de las diferencias de cada punto), escogiendo la que tiene una menor distancia e intercambiándola con la línea actual. En cada iteración del bucle i se coloca una línea en su lugar definitivo.

Paralelización del bucle j

Al llegar a la directiva “parallel for”, se crean los hilos. Las iteraciones del bucle se reparten entre los hilos. Al finalizar el bucle se sincronizan los hilos.

Se ha utilizado “private” para que las variables “x” y “distancia” no sean compartidas La variable j del dicho bucle, ya es privada por definición del “parallel for”.

Para conseguir la exclusión mutua a la variable compartida “distancia_minima” y evitar la condición de carrera, se ha utilizado la directiva “critical”. Cuando un hilo llega al segundo bloque “if” (la sección crítica), espera hasta que no haya otro hilo ejecutándola al mismo tiempo. Teniendo en cuenta que “distancia_minima” nunca se decrementa, se ha añadido el primer “if”, para entrar a la sección crítica con menor frecuencia. El segundo “if” sigue siendo necesario porque se ha leído “distancia_minima” en el primer “if”, que está fuera de la sección crítica.

```
void encaja(Imagen *ima)
{
        unsigned n, i, j, x, linea_minima = 0;
        long unsigned distancia, distancia_minima;
        const long unsigned grande = 1 + ima->ancho * 768ul;

        n = ima->alto - 2;
        for (i = 0; i < n; i++) {
                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;
                                }
                        }
                }
          intercambia_lineas(ima, i+1, linea_minima);
        }
}
```

Paralelización del bucle x

Se ha utilizado “reduction”. Cada hilo realiza una porción de la suma de “distancia”, y al final se combina en la suma total.

```
void encaja(Imagen *ima)
{
        unsigned n, i, j, x, linea_minima = 0;
        long unsigned distancia, distancia_minima;
        const long unsigned grande = 1 + ima->ancho * 768ul;

        n = ima->alto - 2;

        for (i = 0; i < n; i++) {
                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);
                }
}
```

¿Convendría modificar la planificación por defecto de OpenMP para alguno de ellos?

Todas las iteraciones de los bucles j y x tienen el mismo coste, por tanto no es de esperar que la planificación utilizada haga variar el tiempo de ejecución (dejamos de lado el efecto que la utilización de la memoria caché pueda tener en las prestaciones).

Sería válido decir también que la planificación “dynamic” podría tener un poco más de sobrecarga, por tener que hacer el reparto en tiempo de ejecución. Por lo tanto la opción “schedule(static)”.es preferible a “schedule(dynamic)”.

Como las iteraciones los bucles j y x hacen aproximadamente la misma cantidad de trabajo, al paralelizar el bucle tanto la opción “schedule(static)” como “schedule(static,1)” harían un reparto equilibrado de la carga.

Haz que todos los programas paralelos de este ejercicio muestren por pantalla el tiempo empleado en la función *encaja* y el número de hilos con el que se ejecutan.

Al "main" del ejercicio 1, se ha añadido la obtención del número de hilos con el que se paraleliza el programa, para poder indicarlo por pantalla.

```
int nh;
double t1,t2;
#pragma omp parallel
nh=omp_get_num_threads();
t1=omp_get_wtime();
encaja(&ima);
t2=omp_get_wtime();
printf("Para %d hilos el tiempo de ejecución es %f\n",nh,t2-t1);
```

Se ha creado un script para ejecutar el sistema de colas, para "encaja-e2-pj". Para comparar resultados, se han lanzado varias ejecuciones con el script.

```
#!/bin/sh
#PBS -l nodes=1,walltime=00:05:00
#PBS -q cpa
#PBS -d .
#PBS -o tiempos_e2_pJ.txt
./encaja-e1

OMP_NUM_THREADS=1 ./encaja-e2-pj
OMP_NUM_THREADS=2 ./encaja-e2-pj
OMP_NUM_THREADS=4 ./encaja-e2-pj
OMP_NUM_THREADS=8 ./encaja-e2-pj
OMP_NUM_THREADS=16 ./encaja-e2-pj
OMP_NUM_THREADS=32 ./encaja-e2-pj
```

De la misma forma se ha creado otro script para "encaja-e2-px".

Mide los tiempos de ejecución y calcula los índices de prestaciones para cada versión paralela realizada.

Utiliza para ello una cualquiera de las imágenes almacenadas en la carpeta otras, que son imágenes que requieren un mayor coste computacional en su reconstrucción.

Nota: Se aconseja probar cada nueva versión desarrollada con una de las imágenes pequeñas para comprobar que es correcta. Después resultaría conveniente utilizar la opción -t del programa, cuando sólo interesan los tiempos, ya que evita la escritura en disco de la imagen de salida.

Tiempos de ejecución para la paralelización del bucle j (e2-pj):

Para secuencial el tiempo de ejecución es 15.305868
Para 1 hilos el tiempo de ejecución es 13.840646
Para 2 hilos el tiempo de ejecución es 7.051230
Para 4 hilos el tiempo de ejecución es 3.516938
Para 8 hilos el tiempo de ejecución es 2.047917
Para 16 hilos el tiempo de ejecución es 1.196425
Para 32 hilos el tiempo de ejecución es 0.734447

Tiempos de ejecución para la paralelización del bucle x (e2-px):

Para secuencial el tiempo de ejecución es 15.314217
Para 1 hilos el tiempo de ejecución es 13.885485
Para 2 hilos el tiempo de ejecución es 8.145111
Para 4 hilos el tiempo de ejecución es 5.679996
Para 8 hilos el tiempo de ejecución es 5.236254
Para 16 hilos el tiempo de ejecución es 8.098419
Para 32 hilos el tiempo de ejecución es 12.616663

El **speedup** indica la ganancia de velocidad que consigue el algoritmo paralelo con respecto a un algoritmo secuencial. Siendo **t(n)** el mejor algoritmo secuencial conocido o el algoritmo paralelo ejecutado en 1 procesador

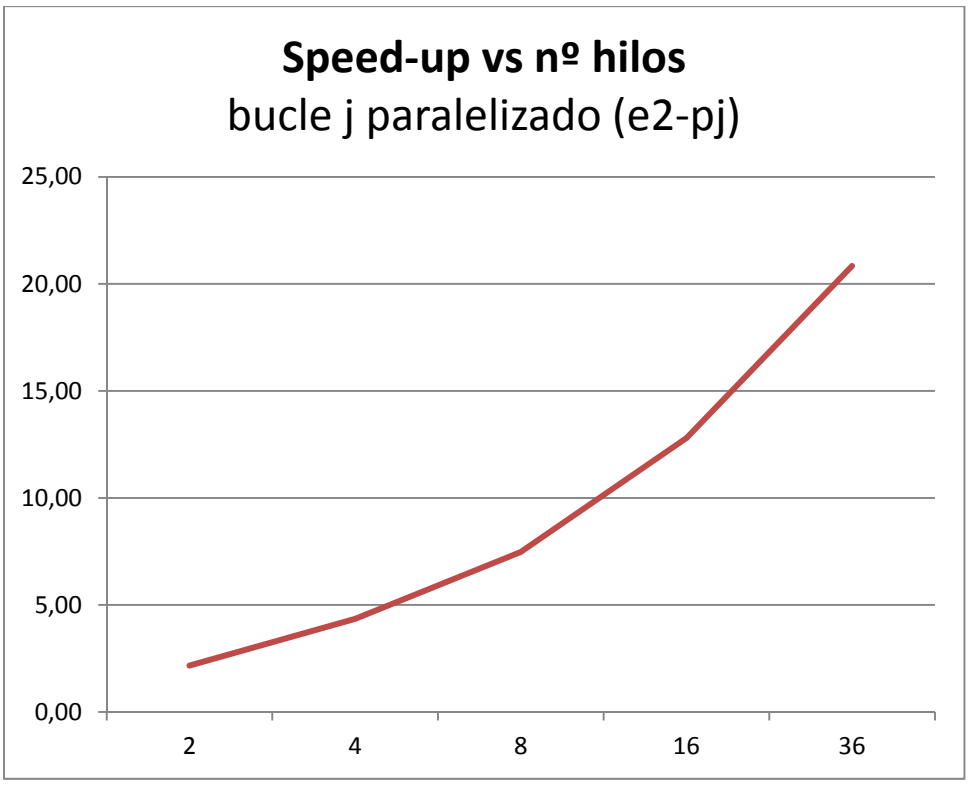
$$\mathbf{S(n,p) = \frac{t(n)}{t(n,p)}}$$

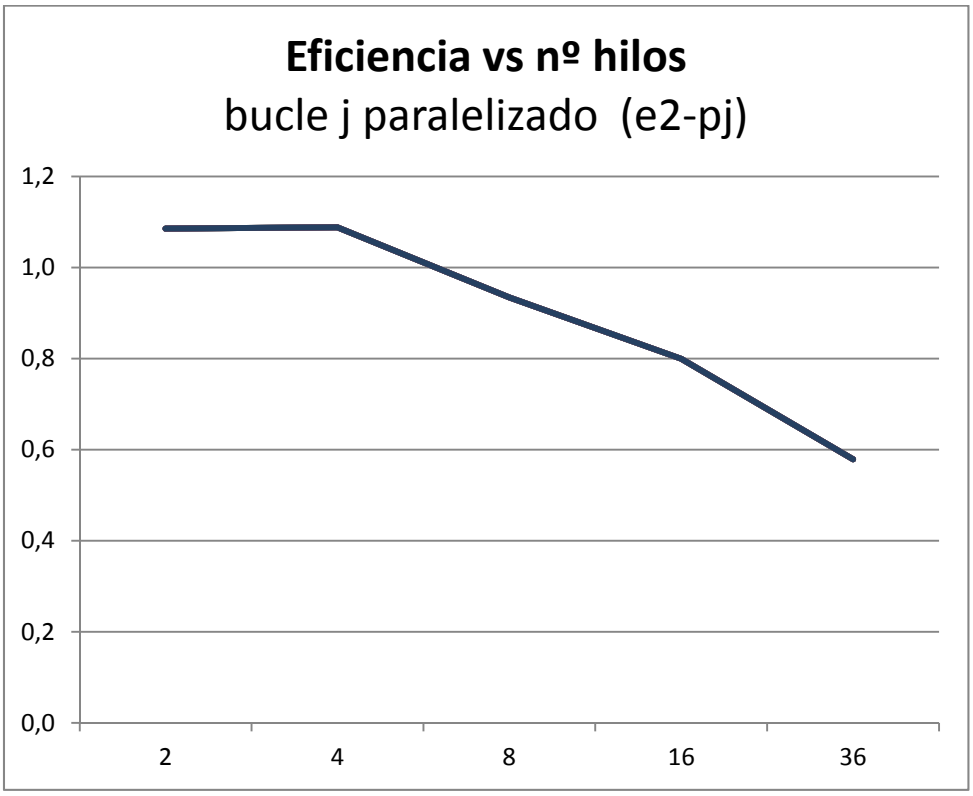
.

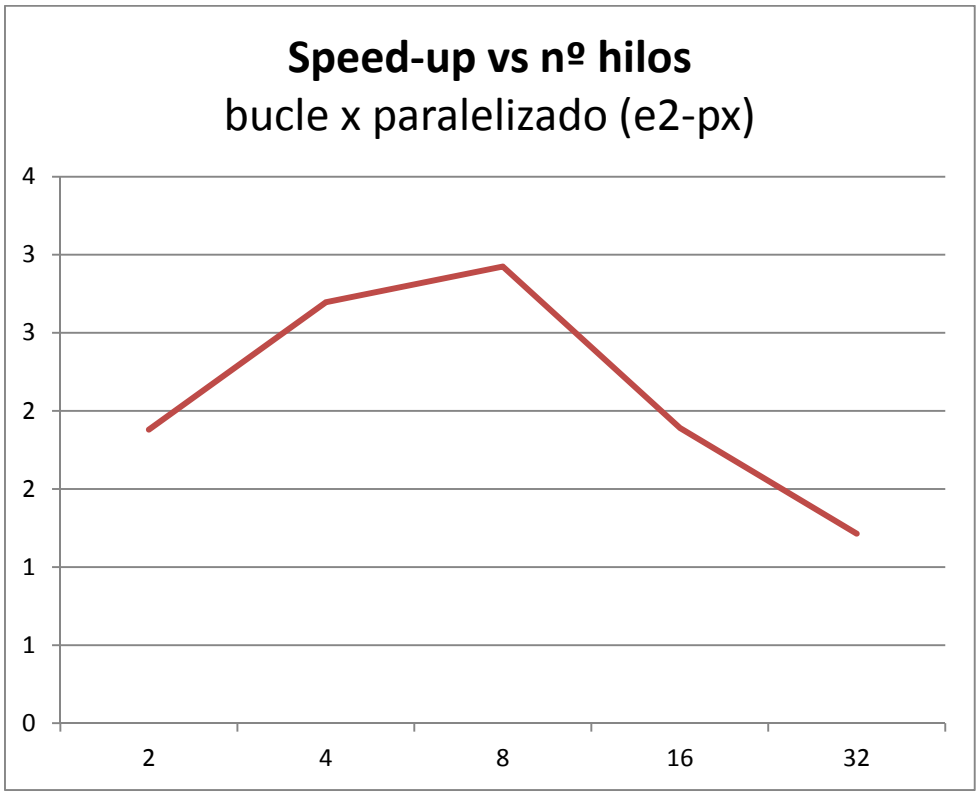
La **eficiencia** mide el grado de aprovechamiento que un algoritmo paralelo hace de un computador paralelo

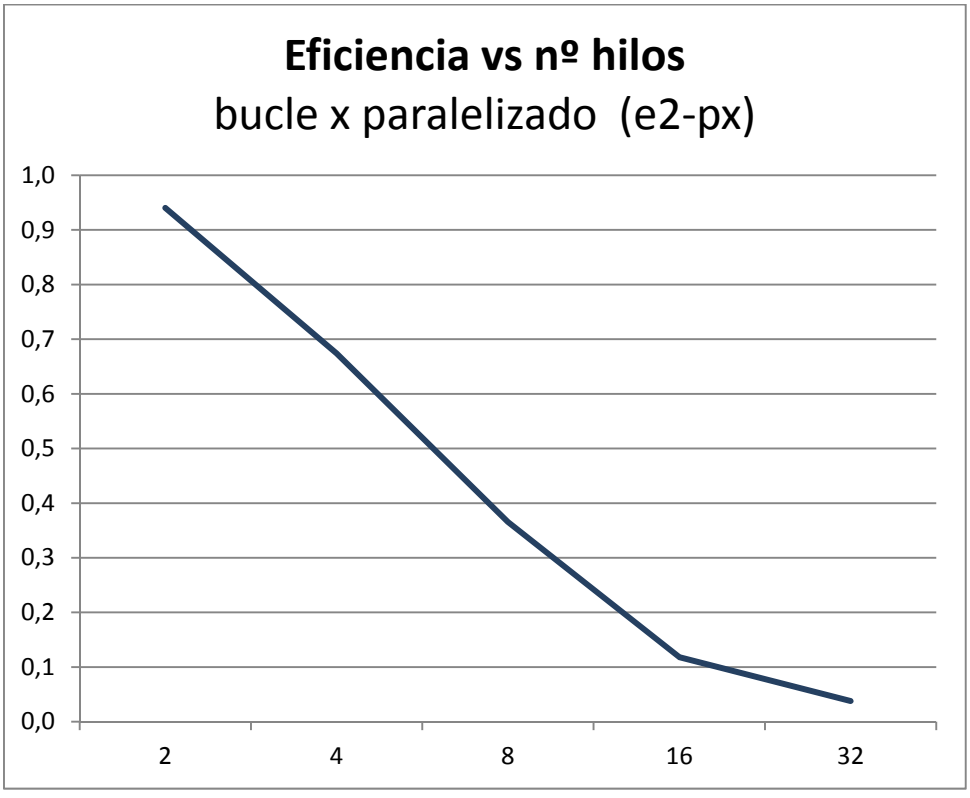
$$\mathbf{E(n,p) = \frac{S(n,p)}{p}}$$

Tomando ts = tiempo secuencial






Speed-up vs nº hilos
bucle x paralelizado (e2-px)
4
3
3
2
2
1
1
0
2
4
8
16
32


Eficiencia vs nº hilos
bucle x paralelizado (e2-px)
1,0
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0,0
2
4
8
16
32

Ejercicio 3

Crea una nueva versión secuencial modificando el bucle x para que el bucle finalice en cuanto el cálculo parcial de la distancia actual sea mayor que la distancia mínima calculada hasta ahora (y por tanto la línea puede ser descartada sin completar el cálculo de su distancia).

Para esta nueva versión secuencial, más eficiente que la secuencial original, se ha añadido la condición “distancia < distancia_minima”.

```
void encaja(Imagen *ima)
{
	unsigned n, i, j, x, linea_minima = 0;
	long unsigned distancia, distancia_minima;
	const long unsigned grande = 1 + ima->ancho * 768ul;

	n = ima->alto - 2;
	for (i = 0; i < n; i++) {
		distancia_minima = grande;
		for (j = i + 1; j < ima->alto; j++) {
			distancia = 0;
			for (x = 0; x < ima->ancho && distancia<distancia_minima ; 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);
	}
}
```

Ejercicio 4

Partiendo del código del ejercicio 3, paraleliza (por separado) cada uno de los bucles que pueda ser paralelizado. Obtén tiempos de ejecución e índices de prestaciones de la mejor versión paralela realizada en este ejercicio (comparando con la nueva versión secuencial).

OpenMP no permite la paralelización directa de un bucle “for” a menos que esté delimitado su inicio, final e incremento.

Como en el bucle x puede terminar antes (cuando se ha dado cuenta de que el cálculo parcial de la distancia actual ya es mayor que la distancia mínima), el bucle no es posible paralelizar con “parallel for”.

En realidad, es el incluir la comprobación de distancia dentro de la condición del bucle lo que no permite que OpenMP pueda paralelizar el bucle. Si se quitara esa parte de la condición, la función seguiría siendo correcta, pero hay que tener en cuenta que el tiempo de ejecución del programa se incrementaría mucho.

Una forma de hacer la versión paralela es realizar un reparto explícito del bucle entre los hilos. Para ello se usan funciones de OpenMP que permitan obtener el número de hilos y ejecutar en paralelo el bucle, pero haciendo que cada hilo realice solo una parte de él. En este caso, se ha repartido el bucle de forma cíclica entre los hilos.

Es importante conservar la condición de salida del bucle cuando se ve que el cálculo parcial de la distancia actual ya es mayor que la distancia mínima. De esta manera, cuando cualquier hilo descubra que la distancia es superior a la mínima, tarde o temprano, todos finalizarían el bucle.

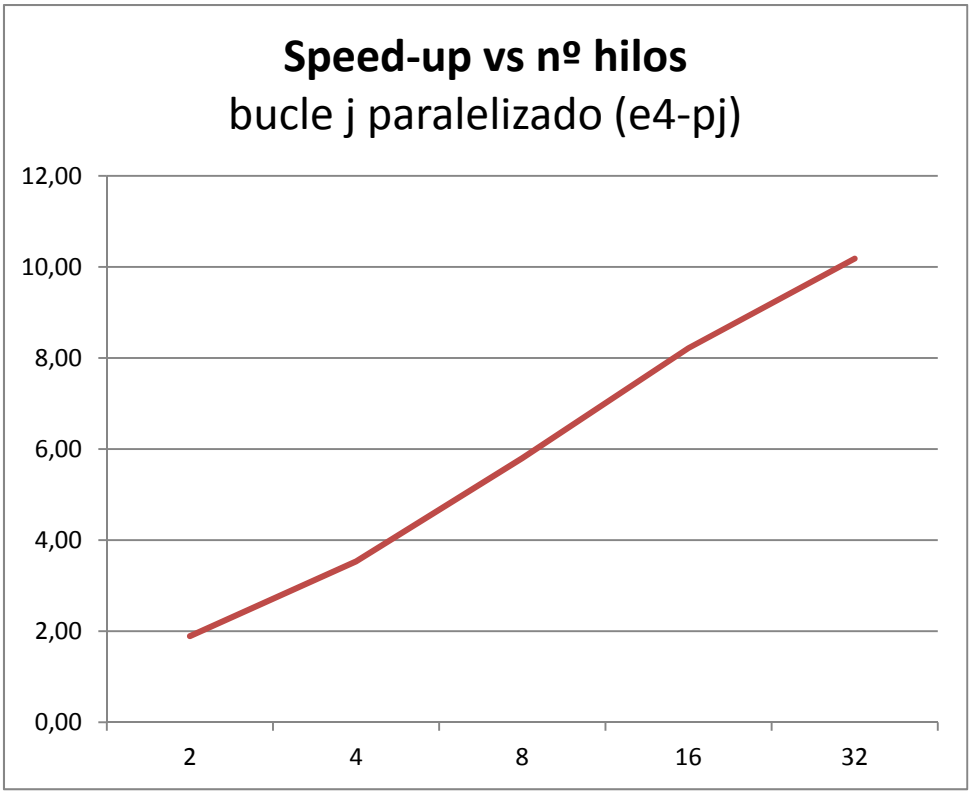
Paralelización del bucle j

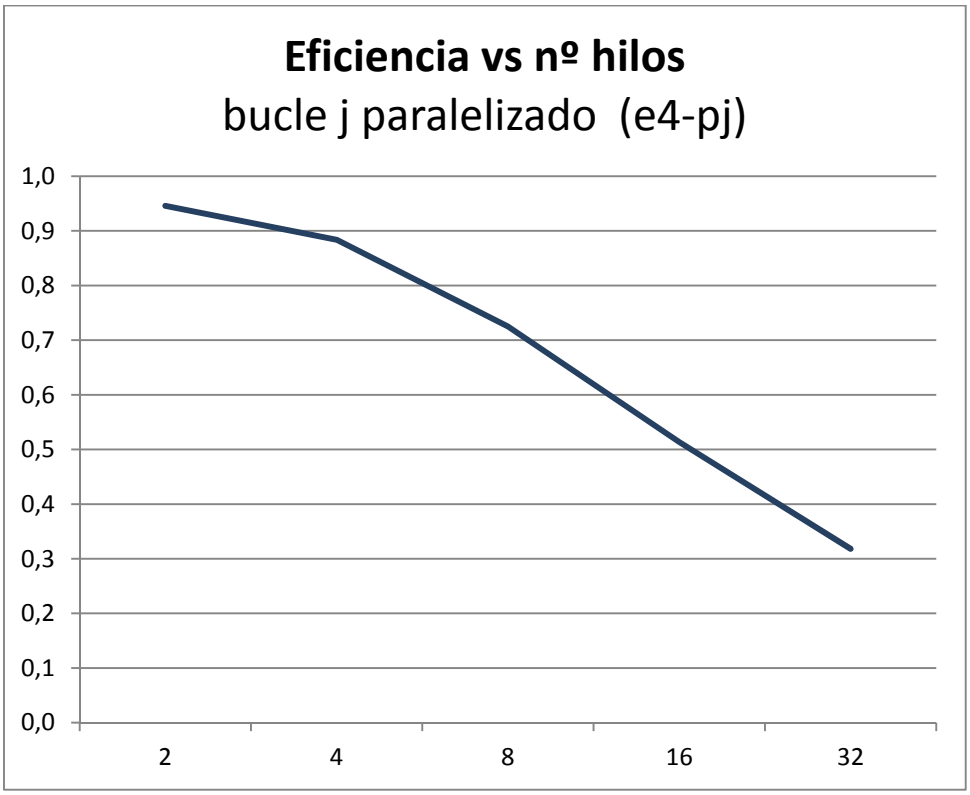
```
void encaja(Imagen *ima)
{

  unsigned n, i, j, x, linea_minima = 0;
  long unsigned distancia, distancia_minima;
  const long unsigned grande = 1 + ima->ancho * 768ul;

  n = ima->alto - 2;
  for (i = 0; i < n; i++) {

          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<distancia_minima; 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);
  }
}
```

Tiempos de ejecución para la paralelización del bucle j (e4-pj):

Para secuencial el tiempo de ejecución es 2.707536
Para 1 hilos el tiempo de ejecución es 2.805785
Para 2 hilos el tiempo de ejecución es 1.431381
Para 4 hilos el tiempo de ejecución es 0.766028
Para 8 hilos el tiempo de ejecución es 0.466874
Para 16 hilos el tiempo de ejecución es 0.329478
Para 32 hilos el tiempo de ejecución es 0.265946

Tomando ts = tiempo secuencial





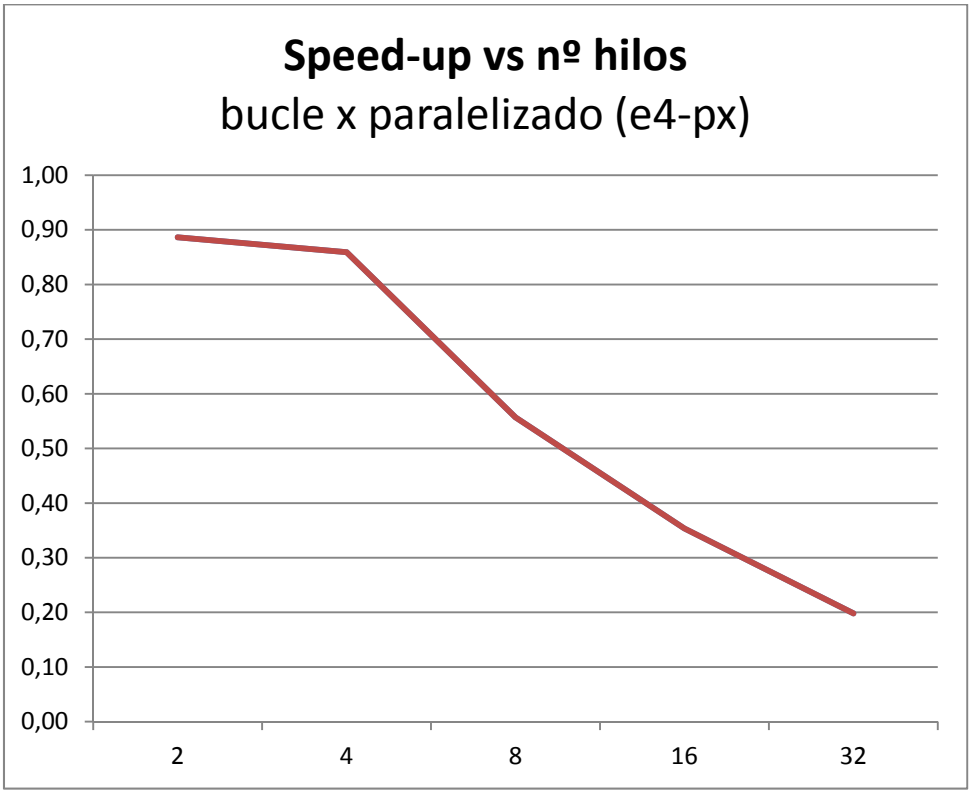
Paralelización del bucle x

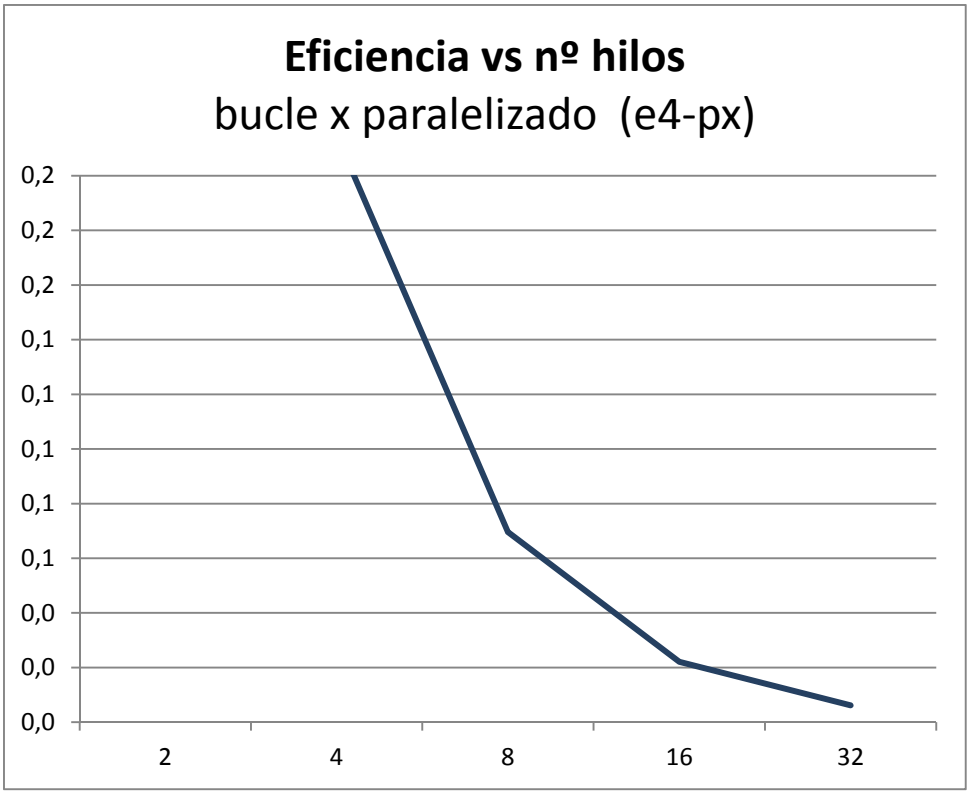
```
void encaja(Imagen *ima)
{
  unsigned n, i, j, x, linea_minima = 0;
  long unsigned distancia, distancia_minima;
  const long unsigned grande = 1 + ima->ancho * 768ul;
  int yo, nh;
  n = ima->alto - 2;
  for (i = 0; i < n; i++) {
              distancia_minima = grande;
              for (j = i + 1; j < ima->alto; j++) {
                    distancia = 0;


                    #pragma omp parallel private (yo,x) reduction (+:distancia)
                    {
                       yo=omp_get_thread_num(); // Numero de hilo
                       nh=omp_get_num_threads(); //Numero de hilos

                       for (x = yo; x < ima->ancho && distancia<distancia_minima ; x+=nh)
                                distancia += diferencia(&A(x, i), &A(x, j));

                       if (distancia < distancia_minima) {
                                distancia_minima = distancia;
                                linea_minima = j;
                       } / /if
                    } // pragma
                 } // for j

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

Tiempos de ejecución para la paralelización del bucle x (e4-px):

Para secuencial el tiempo de ejecución es 2.715169
Para 1 hilos el tiempo de ejecución es 3.172687
Para 2 hilos el tiempo de ejecución es 3.063676
Para 4 hilos el tiempo de ejecución es 3.161159
Para 8 hilos el tiempo de ejecución es 4.875417
Para 16 hilos el tiempo de ejecución es 7.677833
Para 32 hilos el tiempo de ejecución es 13.711357

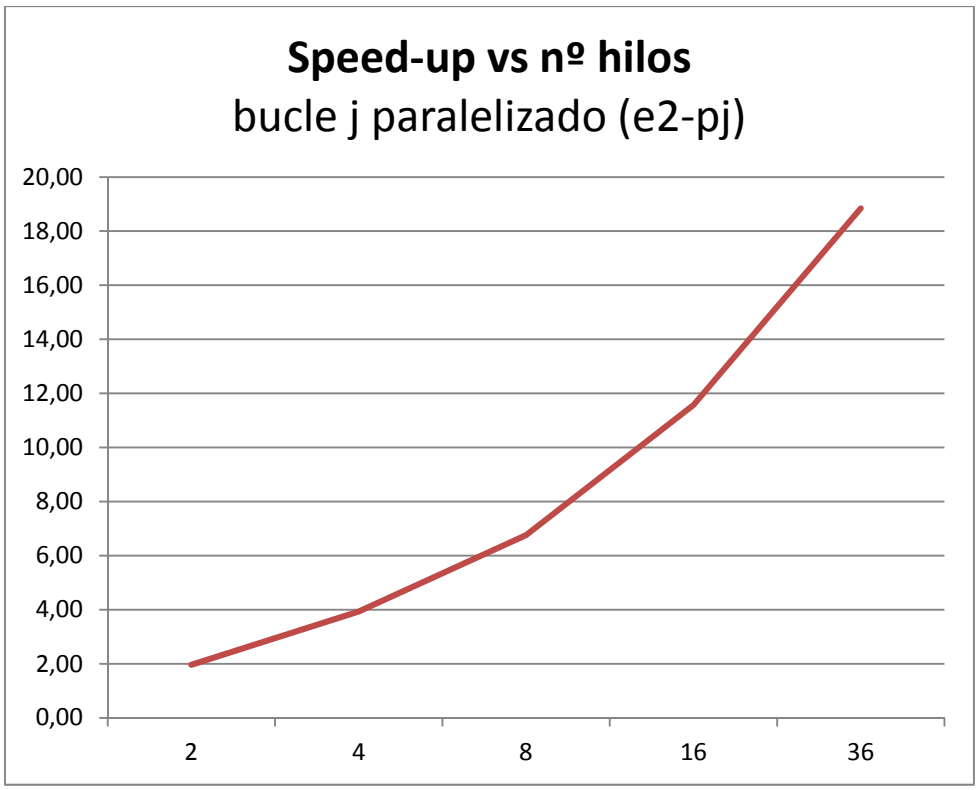
Tomando ts = tiempo secuencial

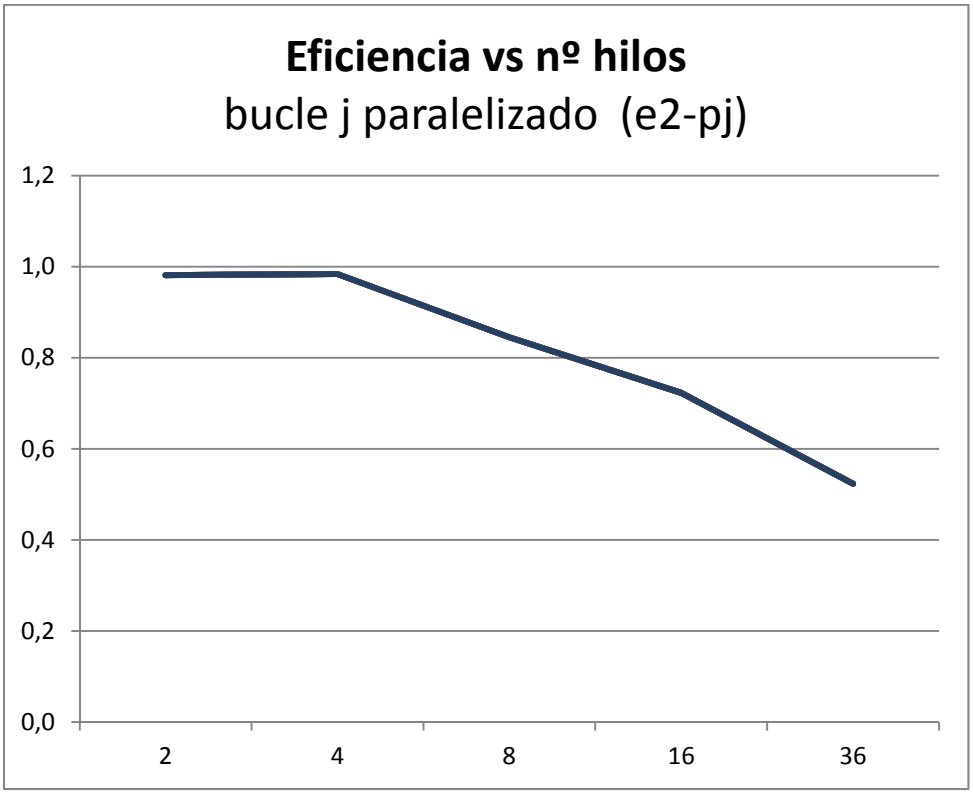


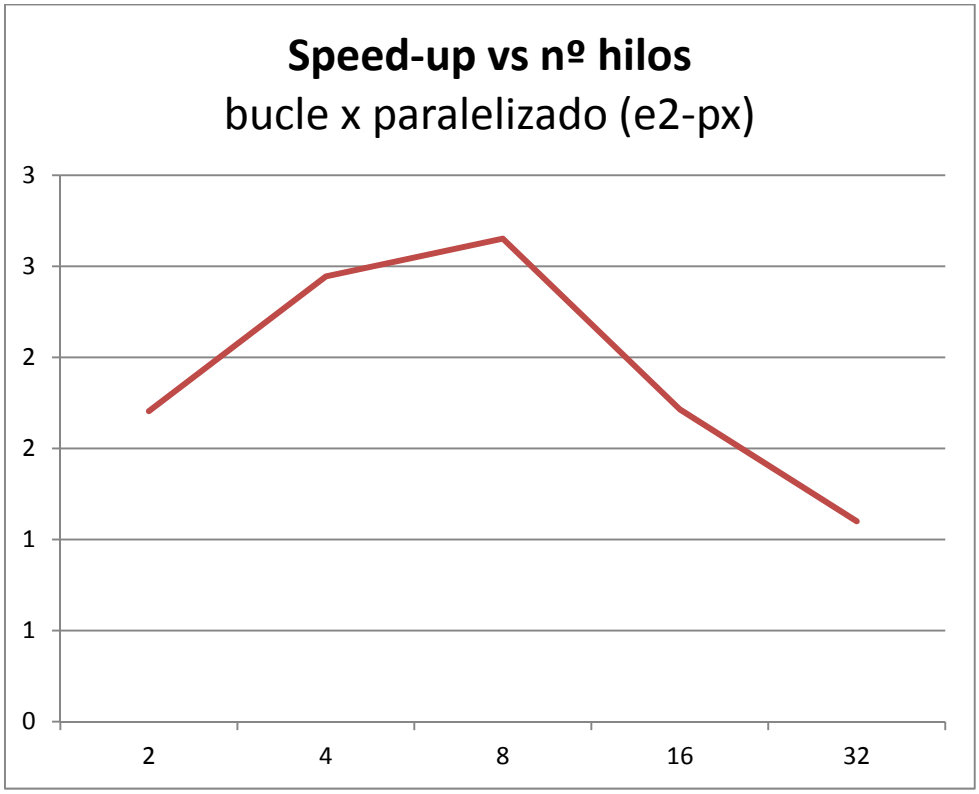


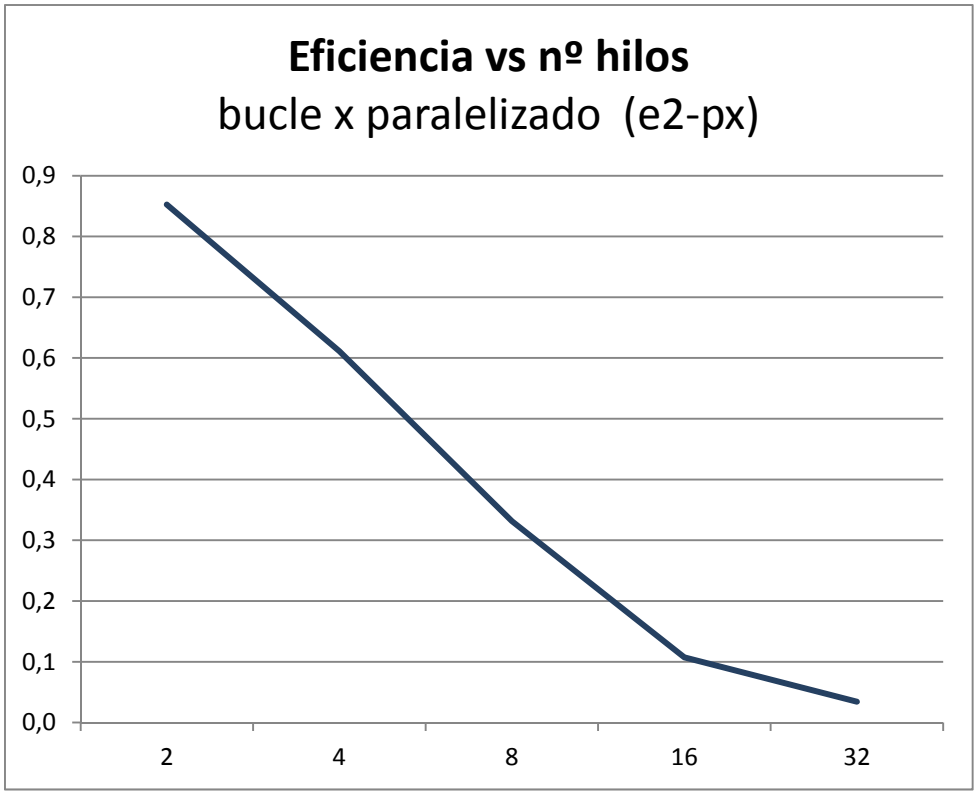
Anexo: Resultados con un hilo

Ejercicio 2. Resultados obtenidos con 1 hilo.

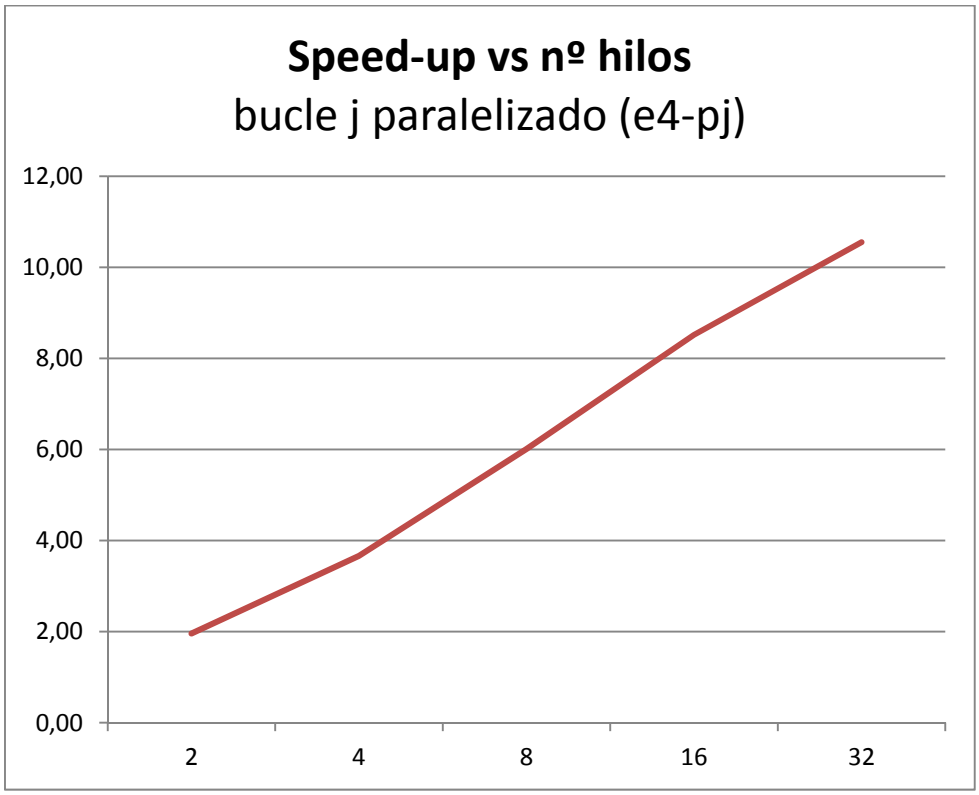


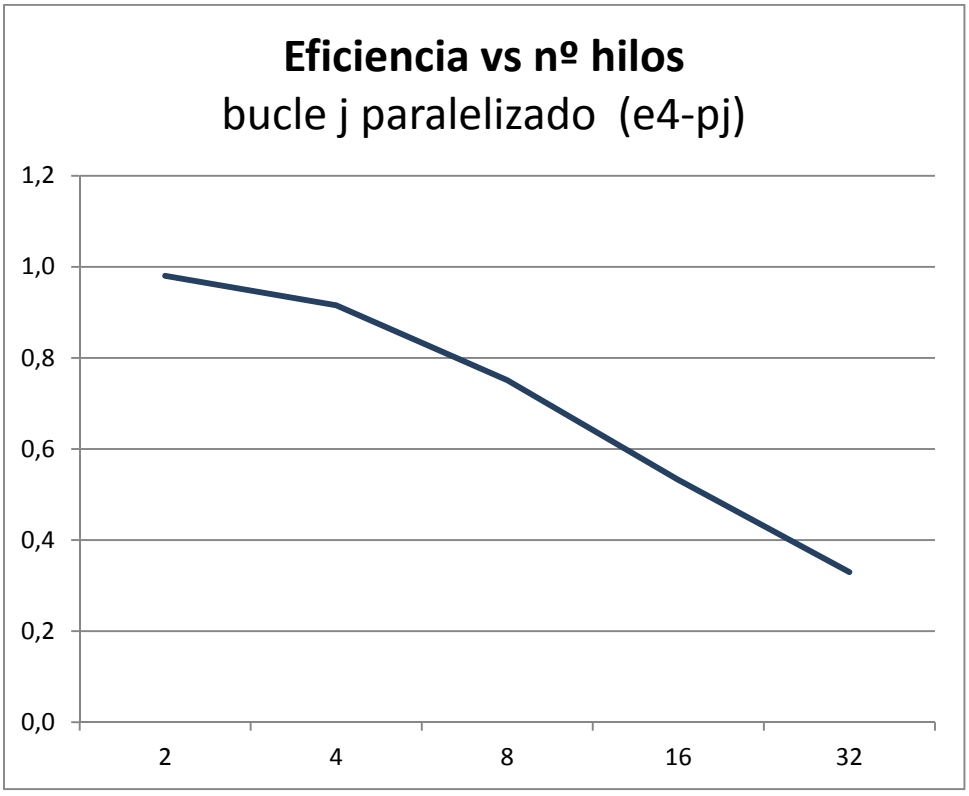


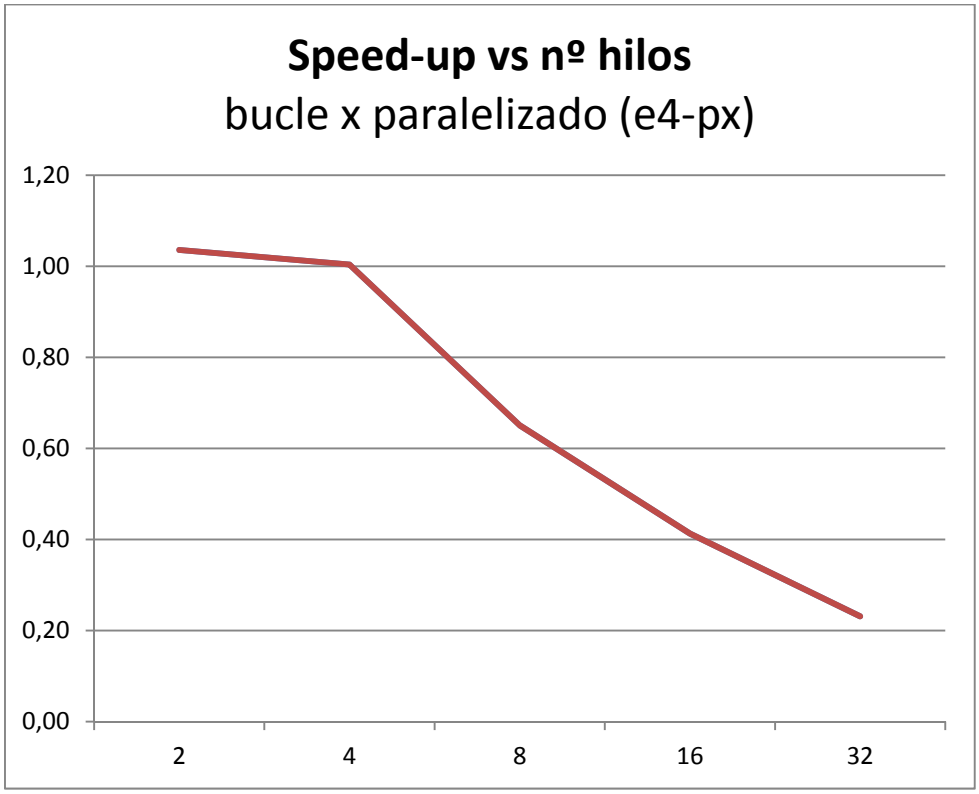

Speed-up vs nº hilos
bucle x paralelizado (e2-px)
3
3
2
2
1
1
0
2
4
8
16
32

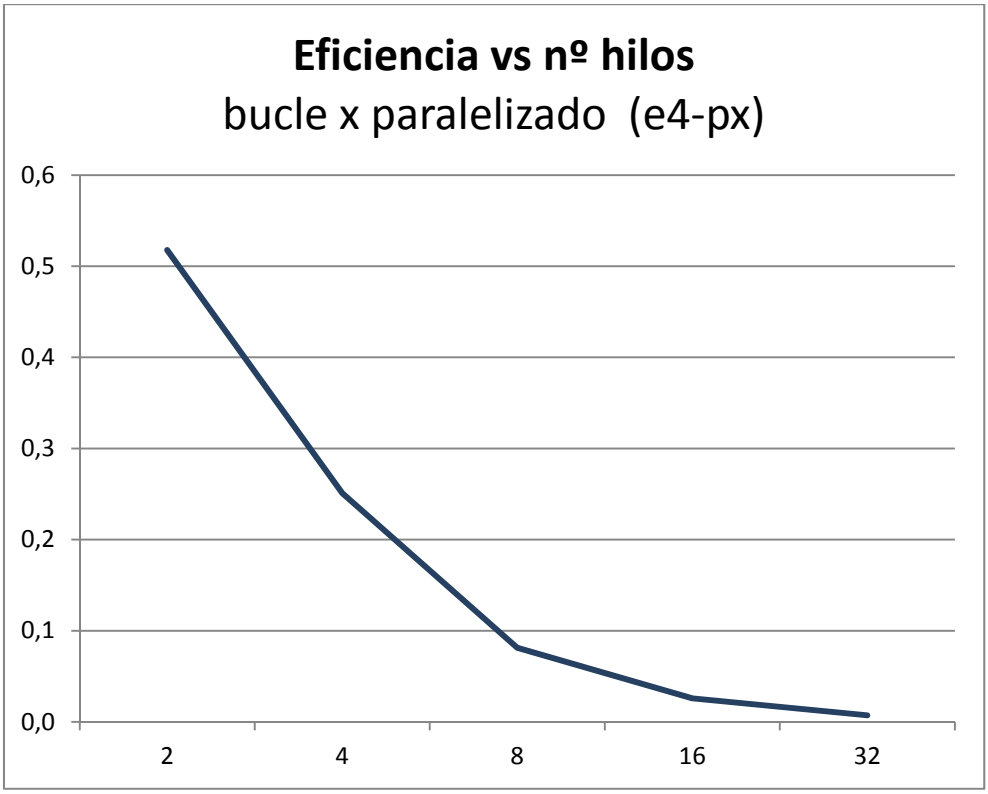

Eficiencia vs nº hilos
bucle x paralelizado (e2-px)
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0,0
2
4
8
16
32

Ejercicio 4. Resultados obtenidos con 1 hilo.






Speed-up vs nº hilos
bucle x paralelizado (e4-px)
1,20
1,00
0,80
0,60
0,40
0,20
0,00
2
4
8
16
32


Eficiencia vs nº hilos
bucle x paralelizado (e4-px)
0,6
0,5
0,4
0,3
0,2
0,1
0,0
2
4
8
16
32