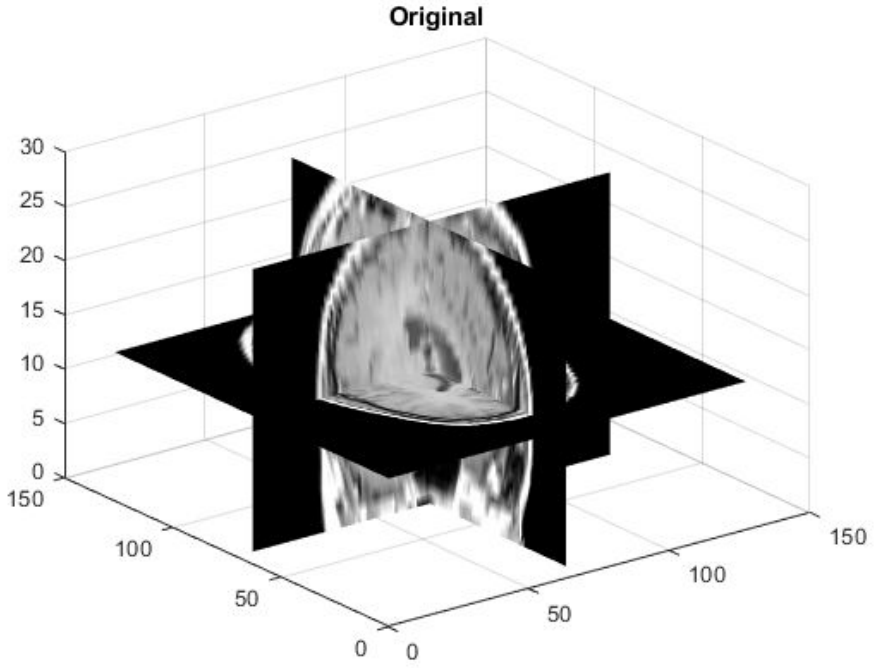


Practica 2: Escalado de imágenes con interpolación de Lanczos.

Johans Rodriguez Mulen

22 de noviembre de 2019

Ejercicios

Tablas y gráficas

Planificación y Conclusión

Ejercicios:

Ejercicio 1: Empieza por modificar el programa original para que obtenga y muestre el tiempo de ejecución de la llamada a la función que realiza el escalado completo de la imagen. Guarda esta versión con el nombre lanSeq.c.

En este primer ejercicio, creo las variables start, end y presicion para medir el tiempo, hago uso del omp_get_wtime() antes de la llamada a la función y después, y mido dicho tiempo, el resultado sería a end restarle start y lo guardé en otra variable, aunque sabemos que con una sola variable podía hacerse.

```
//Empiezo a tomar el valor del tiempo.
    double presicion, start, end;
    start = omp_get_wtime();
    err = resizeRGB(w,h,A,w2,h2,la);
    //Termino de tomar el valor del tiempo
    end = omp_get_wtime();
    presicion = end - start;
    printf("El tiempo de ejecución a la llamada es de %f segundos\n", presicion);
    if (err==0) {
      if (output!=NULL)write_ppm(output,w2,h2,A);
          }
          free(A);
  }
  return 0;
}
```

Ejercicio 2: Paraleliza la función resizeRGB. Guarda esta versión con el nombre lanRGB.c Presta especial atención a la variable "aux". Se trata de un array que resize2D utiliza para almacenar datos temporalmente. Cada llamada a resize2D sobreescribe el array entero, sin importar su valor antes de la llamada ni después de la misma.

En esta ocasión he usado sections, como vemos debajo:

```
//paralelización
    #pragma omp sections
      {
     #pragma omp section
      {
      resize2D(w,h,A,w2,h2,aux,la);   /* red */
      }
     #pragma omp section
      {
      resize2D(w,h,A+1,w2,h2,aux,la); /* green */
      }
     #pragma omp section
      {
      resize2D(w,h,A+2,w2,h2,aux,la); /* blue */
      }
      }
```

Ejercicio 3: Paraleliza la función resize2D. Guarda esta versión con el nombre lan2D.c Presta especial atención a la variable "v".

En este ejercicio he hecho dos paralelizaciones, a los dos bubles for externos, donde podemos ver que ambos casos la variable **v** es privada ya que al ejecutarse concurrentemente, puede influir en el valor de otro hilo si por el contrario no se especificara el private en esta directiva. Si fuese share, o no se especificara, corre el riesgo de corromperse el valor de dicha variable y no llegar de manera fiable el valor que realmente debería tener.

```
#pragma omp parallel for private(y,v)
            for (x=0;x<w;x++) {
                for (y=0;y<h;y++) {
                ……..................................
#pragma omp parallel for private(x,v)
    for (y=0;y<h2;y++) {
        resize1D(w,&mat_elem(B,0,y,w),w2,v,1,la);
```

Ejercicio 4: Paraleliza la función resize1D. Haz dos versiones diferentes: una paralelizando el bucle externo i (llámala lan1Di.c) y otra paralelizando el bucle interno j (lan1Dj.c), siempre que ambos bucles sean paralelizables. Comprueba que las paralelizaciones realizadas en los ejercicios anteriores funcionan correctamente. Si tomas algunos tiempos te dar´as cuenta de que hay una versión paralela que es muy ineficiente, siendo más lenta que el programa secuencial. Descarta esa versión para el ejercicio siguiente.

La primera paralelización se hace al "bucle for externo", se recomienda en muchas ocasiones paralelizar el externo antes que hacerlo con el interno, ya que suele dar mejores resultados y el tiempo de ejecución suele ser más corto.

```
#pragma omp parallel for private(fondo,j,s,xi,x)
      for (i=0;i<m;i++) {
            ……..............
          for (j=j1;j<j2;j++) {
            ……..............
            }
          vec_elem(v,i,inc) = s;
       }
```

La segunda paralelización se le hace al "bucle for" interno:

```
for (i=0;i<m;i++) {
       ……..............................
  #pragma omp parallel for private(x) reduction(+:s)
          for (j=j1;j<j2;j++) {
            ….
            }
          vec_elem(v,i,inc) = s;
```

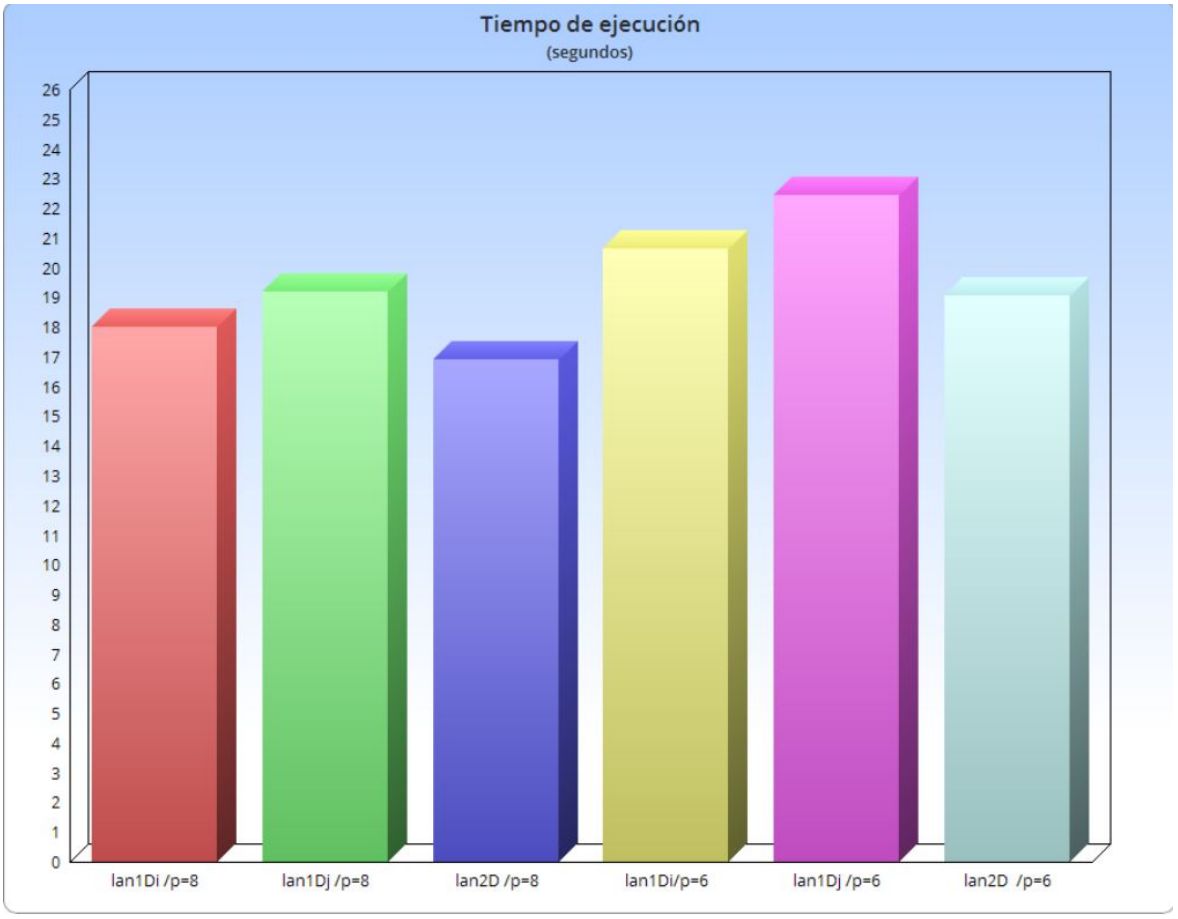
Ejercicio 5: Obtén tiempos de ejecución, speed-ups y eficiencias de las diferentes versiones paralelas con diferentes números de hilos.

Antes de pasar a los gráficos, aquí tenemos la tabla con las anotaciones respectivas a los experimentos hechos, y hemos visto que cambia el tiempo de ejecución y se ahorra o se gastan recursos y prestaciones.

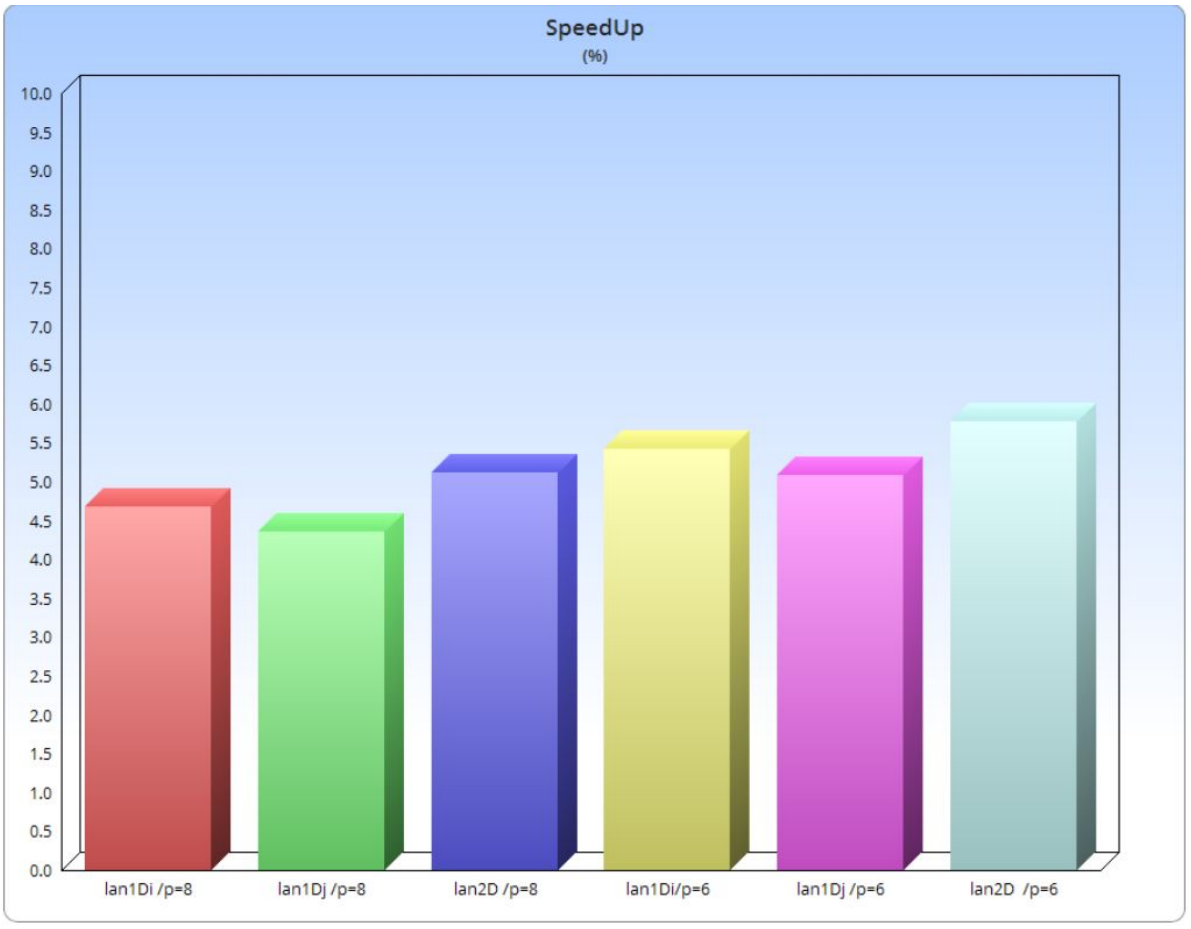
Tiempo de ejec. Secuencial: 97.67

	T. de ejecución	SpeedUp	Eficiencia	
lan1Di	20.65	4.68	0.78	
lan1Dj	22.45	4.35	0.72	
lan2D	19.07	5.12	0.85	
lan1Di	18.01	5.42	0.67	
lan1Dj	19.21	5.08	0.63	
lan2D	16.91	5.77	0.72	

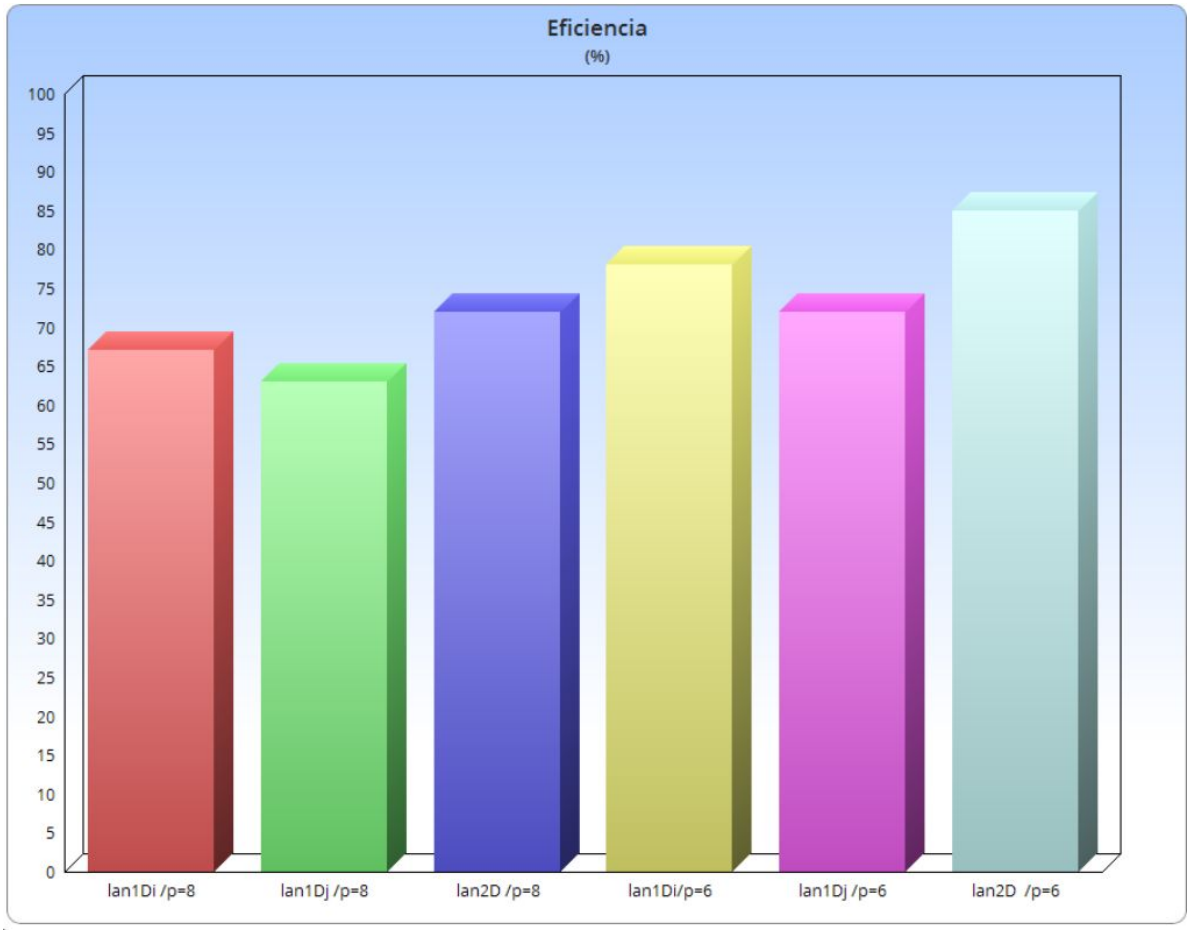
Empezamos por el tiempo de ejecución de los ficheros lan1Di, lan1Dj, y las2D, con 6 y 8 procesos respectivamente:



Luego vemos el SpeedUp, que no es más que la ganancia de velocidad que consigue un algoritmo paralelo con respecto a uno secuencial...



¿Cuán eficientes han sido el haber aplicado las distintas formas de paralelizar? Debajo dejo una gráfica con información sobre dicha Eficiencia.



Ejercicio 6: En la paralelización del bucle externo de resize1D ¿tendría sentido cambiar la planificación por defecto? Justifica tu respuesta. Prueba diferentes planificaciones y comprueba experimentalmente si el algoritmo se comporta de la forma esperada.

La planificación con dynamic y chunk “1” me dio mejores resultados, por ejemplo con el “lan2D” que subiendo el chunk al 3, por ejemplo, con “1” los hilos van pidiendo según van terminando y no se producen esperas innecesarias. Y resulta más útil a veces el static chunk=1 .

En conclusión, estas pruebas muestran que paralelizando correctamente podemos llegar a obtener una buena ganancia de tiempo frente a los algoritmos secuenciales, que si usamos la cantidad de procesos adecuados sea preferiblemente de manera dinámica, o de forma estática, evitaremos pérdida de recursos prestaciones y colas innecesarias.