MEMORIA PRÁCTICA 4 ARQO:

Javier Fraile Iglesias

Apartado 0:

Un equipo personal

La máquina donde se está ejecutando el comando es un procesador con nombre de modelo “Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz”.

Es un procesador a 6 cores y de 12 hilos, lo que quiere decir que este procesador cuenta con hiperthreading, ya que cada núcleo físico cuenta con más de un núcleo virtual o hilo.

El equipo de los laboratorios

El equipo de los laboratorios es un procesador con nombre de modelo “Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz”

Es un procesador de 12 cores con una frecuencia de 800.174 por core. Hay 8 cores físicos (cpu cores) y 16 cores virtuales (siblings).

Como tiene el doble de cores virtuales, se dice que tiene hiperthreading de 2 hilos.

El equipo del cluster

El cluster cuenta con dos procesadores, uno con amd y otro con intel, para obtener la información de cada uno de los procesadores, usamos el comando “qsub -q intel.q ó qsub -q amd.q” junto con el .sh “info_cluster.sh” que contiene el comando “cat /proc/cpuinfo”, y observamos lo siguiente:

Intel: Es un procesador con nombre de modelo “Intel(R) Xeon(R) CPU E5520 @ 2.27GHz”.

Cuenta con 16 procesadores, y cada uno de ellos cuenta con 4 cores y con 8 hilos o sibblings lo que quiere decir que dicho procesador permite hiperthreading

Amd: La otra cpu del cluster, la de Amd, es un procesador con nombre del modelo “Six-Core AMD Opteron(tm) Processor 2427”.

Este procesador cuenta con 12 procesadores, y cada uno cuenta con 6 cores y con 6 hilos o sibblings, lo que quiere decir que al tener el mismo número de cores que de hilos es que no permite hiperthreading.

Apartado 1:

1.1 => Si es posible utilizar más hilos que cores siempre y cuando la cpu tenga la capacidad de multithreading. La idea de los hilos es hacer creer a la máquina que se

pueden hacer varias tareas a la vez de modo que en vez de realizar una tarea por completo, se divide en varias subtareas y cada hilo se encarga de realizar una tarea en concreto. Teniendo en cuenta la arquitectura del cluster, que es la máquina en la que ejecutamos las tareas, si usamos para la ejecución el procesador de intel si tendría sentido utilizar más hilos porque cuenta con hiperthreading pero, en el caso de usar el procesador de amd, no tendría sentido porque no cuenta con hiperthreading.

1.2 => Cómo se ha indicado en el apartado 0, nuestro equipo personal, cuenta con 12 hilos, por lo que son estos los hilos que se deberían utilizar en la ejecución, en el caso de los equipos de los laboratorios, se debería usar 16 hilos en caso de ejecutarlo desde ellos y en el caso de cluster, si usamos la cpu de amd sería 8 hilos por cada procesador y si utilizamos el de intel serían 16 por cada procesador.

SALIDA TRAS EJECUTAR EL PROGRAMA omp2.c:

Inicio: a = 1, b = 2, c = 3

&a = 0x7fffec0d471c,x &b = 0x7fffec0d4718, &c = 0x7fffec0d4714

[Hilo 0]-1: a = 0, b = 2, c = 3

[Hilo 0] &a = 0x7fffec0d46d4, &b = 0x7fffec0d4718, &c = 0x7fffec0d46d8

[Hilo 0]-2: a = 15, b = 4, c = 3

[Hilo 2]-1: a = 0, b = 2, c = 3

[Hilo 2] &a = 0x7f1c49fdde04, &b = 0x7fffec0d4718, &c = 0x7f1c49fdde08

[Hilo 2]-2: a = 21, b = 6, c = 3

[Hilo 3]-1: a = 0, b = 2, c = 3

[Hilo 3] &a = 0x7f1c495dce04, &b = 0x7fffec0d4718, &c = 0x7f1c495dce08

[Hilo 3]-2: a = 27, b = 8, c = 3

[Hilo 1]-1: a = 32540, b = 2, c = 3

[Hilo 1] &a = 0x7f1c4a9dee04, &b = 0x7fffec0d4718, &c = 0x7f1c4a9dee08

[Hilo 1]-2: a = 1058851633, b = 10, c = 1058851603

Fin: a = 1, b = 10, c = 3

&a = 0x7fffec0d471c, &b = 0x7fffec0d4718, &c = 0x7fffec0d4714

1.4 => Cuando se declara una variable privada, no estarán inicializadas al comienzo de la región y no dejaran rastro al final, y esas variables tendrán una dirección de memoria distinta a la que tienen en el programa principal (antes de que se lance la región paralela) y también, diferente al resto de los hilos.

1.5 => Como bien se ha comentado antes, las variables privadas no estarán inicializadas en la región paralela al inicio, lo que quiere decir, que cada hilo al recibirlo lo inicializará a otro valor, que por defecto será 0 aunque puede darse el caso de que la variable quede inicializada a otro valor.

1.6 => Cómo las variables privadas no dejan rastro, el valor de “a” fuera de la región paralela no mantendrá el valor de “a” al final de la región paralela, puesto que cada variable tendrá una dirección de memoria diferente y como en el caso del programa omp2.c el valor de “a” no cambia entre el inicio y fin, “a” finalizará con valor “1”.

1.7 => En el caso de las variables públicas (o shared en OpenMP), están consideradas como un recurso que es compartido observando que la dirección de memoria tanto fuera como en la región compartida para cada hilo, tiene el mismo valor, y si uno de los hilos realiza un cambio sobre dicha variable, el resto de hilos utilizarán dicho valor, observando que al final de la ejecución, el valor de “b” ha cambiado. En referencia a la ejecución, al ejecutar varios hilos a la vez, el valor inicial de “b” tienen el mismo valor, pero a medida que cada hilos se actualiza, el siguiente usa el valor actualizado y por tanto los resultados no son los mismos ya que cada hilo usa el valor que haya en cada momento.

Apartado 2:

2.1 => En la ejecución en serie, se puede observar que variando el tamaño M de los vectores, cuanto más grande sea, más tiempo de ejecución, porque el vector generado es de mayor tamaño, con lo cual tiene que dar más vueltas de bucle hasta finalizar la ejecución.

2.2 =>

- A pesar de que la concurrencia si que se realiza de manera correcta, porque está dentro de una región paralelizada donde los threads dados por “nproc” se distribuyen el trabajo del bucle, el resultado de la operación no siempre es correcto y varía un poco entre ejecuciones.
- Un posible caso, es que la variable sum que almacena el resultado, está siendo escrita de manera concurrente, y como cada hilo ejecuta el bucle en su momento y la variable suma no está protegida, se están dando condiciones de carrera sobre esa variable, variando el resultado en cada ejecución.

2.3 =>

- El problema si que se resuelve si se emplean ambas directivas, los cambios son los siguientes:

```
//Dentro del bucle (Caso de atomic)

#pragma omp atomic

            sum += A[k]*B[k];



//Dentro del bucle (Caso de critical)

#pragma omp critical

            sum += A[k]*B[k];
```

- Hemos empleado la primera opción de omp atomic, ya que para este caso el tiempo de ejecución es menor, debido a que no se bloquea la operación atómica que está a punto de suceder (suma).

  La sección crítica, por el contrario, tiene unos costes generales por el hecho de que un subproceso entre en la sección crítica, con lo cual para este caso es menos conveniente.

2.4 =>

```
      #pragma omp parallel for reduction(+:sum)

         for(k=0;k<M;k++)

         {

                  sum += A[k]*B[k];

         }
```

La reducción crea una copia “local” de cada hilo mientras este hace sus operaciones (en este caso la suma), evitando de esta manera la condición de carrera que antes se producía (2.1) en la variable suma; y posteriormente, se suman todas y se produce el resultado.

Es más eficiente que la atómica y más aún que la crítica, debido a que las anteriores dos tan solo podían acceder a la variable “sum” de una en una, y una vez dentro, modificarla; sin embargo, en la reducción, se permite concurrencia porque cada hilo tiene su propia variable “sum”.

Apartado 3:

Para la obtención de los tiempos, se ejecutó el script “multiplica.sh” para un tamaño de matriz de 1800 a 2600 y haciendo posteriormente una media de los datos obtenidos. Los datos se iban guardando en un fichero, y una vez realizadas todas las ejecuciones, se agruparon los datos en ficheros según el número de hilo usado en cada ejecución.

Tiempo de ejecución en segundos

Version/# hilos	1	2	3	4
serie	39.561	39.548	41.129	39.534
paralela-bucle1	42.817	32.005	25.742	24.494
paralela-bucle2	39.678	22.661	15.306	12.202
paralela-bucle3	39.594	22.243	15.489	11.767

Aceleración (SpeedUp)

Version/# hilos	1	2	3	4
serie	1	1	1	1
paralela-bucle1	0.925	1.218	1.555	1.581
paralela-bucle2	0.997	1.739	2.618	3.265
paralela-bucle3	0.998	1.785	2.562	3.381

3.1 =>

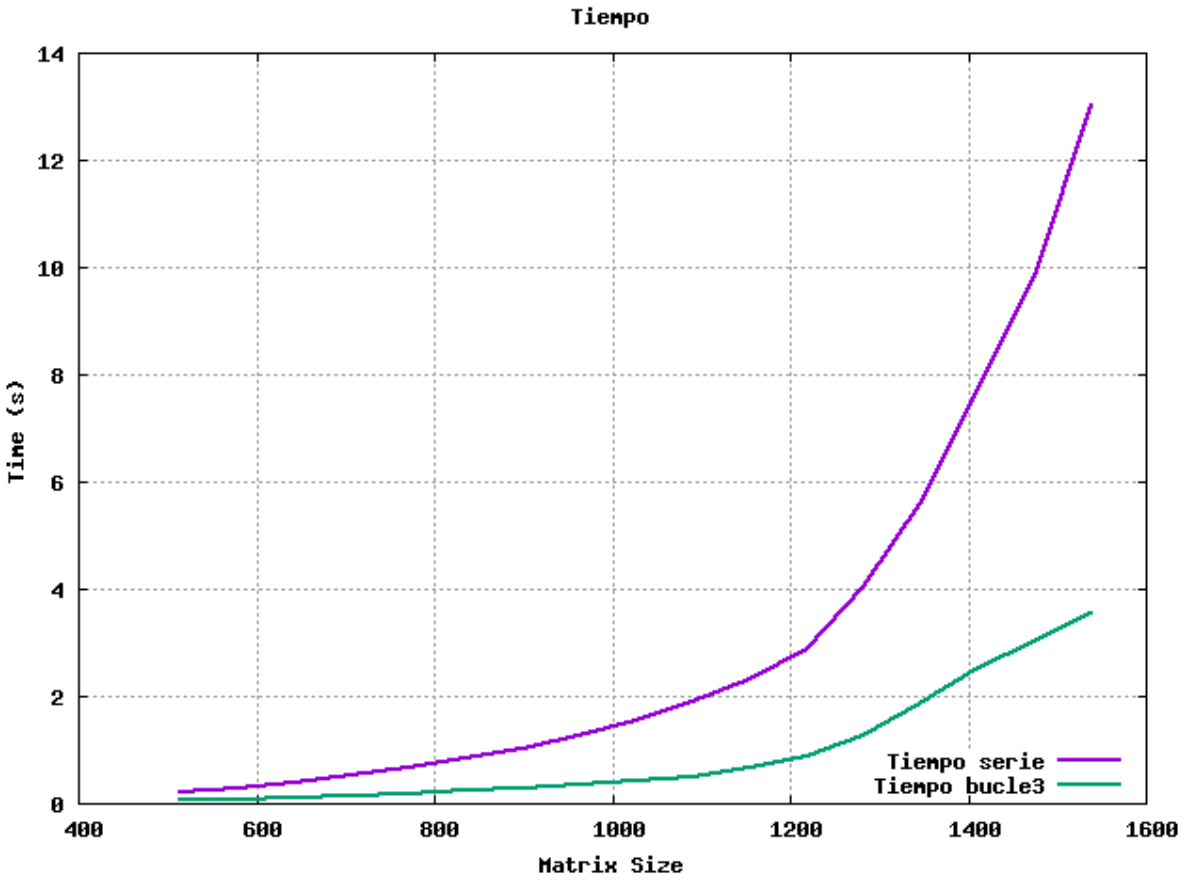
La versión que obtiene un peor rendimiento es la del bucle1 o paralelización en el bucle interno, hablando en términos general, independientemente del número de hilos utilizados, esto se debe a que solo se paraleliza un único bucle mientras que los restantes se ejecutan en orden secuencial sumado a que por cada vuelta de cada bucle externo, cuando se llega al bucle interno se deben inicializar la región paralela, etc… Por otro lado, la versión que obtiene un mejor rendimiento es la del bucle 3 o paralelización en el bucle más externo, debido a que al paralelizar todos los bucles, se paraleliza todo el trabajo y solo se prepara la región una única vez al principio.

3.2 =>

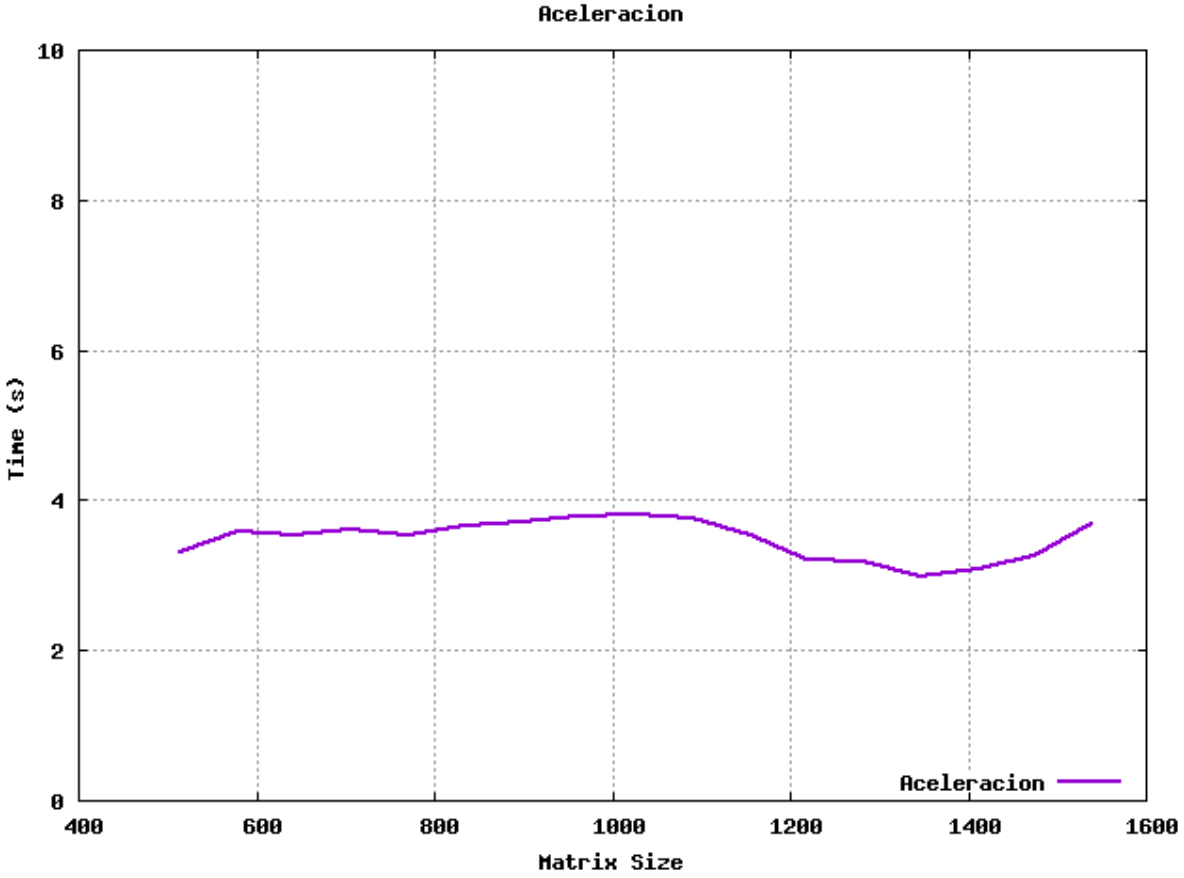
Viendo los resultados obtenidos, creemos que es mejor la paralelización de grano grueso, por el simple hecho de no tener que estar inicializando y preparando la

región paralela por cada vuelta de bucle en el caso de la paralelización en grano fino frente a una única vez como ocurre en la paralelización de grano grueso, pero todo depende del tipo de sentencia o programa que se quiera paralelizar.

3.3 =>



Como se puede apreciar, para tamaños pequeños de matriz, la paralelización no es más efectiva porque son pocas vueltas de bucle las que se producen; sin embargo, según vamos aumentando se ve como cada vez más, va siendo más efectivo paralelizar y dividir el número de iteraciones entre los hilos. Es por eso que aumentado el tamaño, cada vez se separan más la de serie (con un crecimiento exponencial), de la paralelizada (que mantiene un crecimiento más lento y lineal).



Como se puede observar, la aceleración se mantiene lineal, y esto se debe que a partir de cierto tamaño de matriz no resulta efectivo paralelizar respecto a la versión de serie.

Apartado 4:

4.1 =>

En el fichero pi_serie.c se utilizan 100.000.000 de rectángulos, valor indicado en la variable “n”, siendo entonces, el valor de “h” = 1.0 / (double) n = 1x10^-8.

4.2 => Ejecutando los programas 1 a 1 obtenemos los siguientes datos:

Tiempo de ejecución en segundos

Version	Tiempo (s)
serie	2.670457
paralela1	2.170836
paralela2	2.553531
paralela3	0.177719

paralela4	0.311051
paralela5	0.181348
paralela6	2.117073
paralela7	0.300786

Aceleración(SpeedUp)

Version	Aceleracion
serie	1
paralela1	1.23
paralela2	1.045
paralela3	15.027
paralela4	8.5868
paralela5	14.73
paralela6	1.2611
paralela7	8.8779

FALTAN DISCUTIR LOS RESULTADOS

4.3 =>

Al declarar una variable como privada, ésta se tendrá que inicializar dentro de la región, y en el caso del programa, con el valor por defecto que tuviera antes de la región por el hecho de estar declarada como firstprivate. Además por el hecho de ser un puntero, tendrá que reservar memoria para cada elemento del puntero, en este caso, n. Y creemos que no tiene sentido declarar sum como privada si posteriormente quieres utilizar los resultados aplicados sobre la misma, en las regiones paralelas.

4.4 =>

Respecto a las versión 5 y la 1, se diferencian en el caĺculo del número pi, la versión calcula dicho valor dentro de la región paralela a partir de la sentencia “critical” y la versión 1 guarda los valores en un array y fuera de la región paralela calcula el valor de pi.

Respecto a las versiones 3 y 1, se diferencian en que la versión 3 hace uso del padding para la manera de guardar los valores de las sumas.

El término “False Sharong” es una situación que se produce cuando los threads comparten un bloque caché que contiene diferentes variables pese a que no estén

accediendo a las mismas variables. Las versiones que se ven afectadas son la 1,2,3 y 6.

En la versión 3, se obtiene el tamaño de la línea de la caché para poder calcular el padding y de esta forma el tamaño del bloque es mucho mayor.

4.5 =>

La directiva “critical” lo que indica es que la parte del código especificado solo se ejecute por un subproceso a la vez. Esto provoca que el rendimiento sea algo peor porque al ser una sección del código que solo se podrá acceder por un hilo simultáneamente, se irá creando una cola de acceso a dicha región del código.

4.6 =>

el programa pi_par6.c es de los que peor rendimiento ofrece, porque para cada región paralela que se crea, dentro, se indica “#pragma omp for” lo que significa que el trabajo de las iteraciones del bucle for, se reparte entre los diferentes hilos pero a su vez otros hilos están realizando tareas en sus respectivas regiones.

4.7 =>

El mejor rendimiento lo obtenemos en el programa “pi_par3.c”, en este programa se hace uso del padding provocando que el tamaño de los bloques sea mayor y por tanto cada bloque no contenga el array entero.

Apartado 5:

0 =>

Hemos realizado la ejecución con la imagen con resolución 8k, y se generaron 3 ficheros:

1. 8k_grey.jpg: Lo primero que realiza es un bucle que recorre el ancho (width) y otro interno recorriendo el alto (height) de la imagen, con lo cual se va pixel a pixel, abarcando las columnas primero. Y se obtiene llamando a la función “getRGB”, el codigo de color rgb (red, green, blue), y multiplicando por los valores que aparecen ahi, se transforma el color al pixel en su escala de grises. Esto se guarda en una matriz con la imagen y se manda a imprimir con stb1_write_jpg”.

2. 8k_grad.jpg: Se encarga de utilizar los borders, para rodearlos y pintarlos de blancomediante un algoritmo que detecta los bordes, guardando la imagen en una matriz de píxeles (edges) e imprimiendo.
3. 8k_grad_denoised.jpg: Que parte de los edges y los rellena y hace más nítidos los bordes de la imagen.

1 =>

Como ya se comentó en el ejercicio 3, paralelizar el bucle más externo, o también conocido como paralelización de grano grueso, es una mejor solución porque solo se prepararía la región paralela una única vez y los hilos se dividirán todo el trabajo frente a paralelizar el bucle más interno que solo se paraleliza dicho bucle y se tendría que estar inicializando la región por cada vuelta de bucle anterior.

a)

Si se pasan menos argumentos, que cores, no pasa nada, es decir, no ocurren errores ni nada por el estilo, la ejecución se realiza correctamente..

b)

No sería la opción más adecuada por los gastos de memoria que supondría a la hora de reservar memoria por cada hilo. El programa se encarga de generar 3 imágenes a partir de una original, siendo estas, “grey”, “grad” y “grad denoised” de modo que cada hilo, reservará para la imagen “grey” el tamaño original que tuviera la imagen, el tamaño de la imagen “grey” menos 2 pixeles de altura y anchura para el caso de la imagen “grad” y el tamaño de la imagen “grad” restando 2 pixeles de altura y anchura también para el caso de la imagen “grad denoised”.

2 =>

b):

Las imágenes se crean recorriendo el alto (height) o columnas primero en vez de recorrer primero todo el ancho (width) o filas, y como ya se vio en esta práctica y en la anterior, esta manera es menos eficiente a la hora de acceder a los datos de la caché. Por lo que se ha implementado una copia del fichero “EdgeDetectorSerie.c” de modo que recorran todas las filas primero apra cada columna, llamado “EdgeDetectoA2.c”.

3 =>

Partiendo del código original, cremos 3 ficheros, donde en cada uno, se paraleliza un bucle for con los que se generan las imágenes, para el de “grey” es el fichero que termina con “edgeDetectorA3B1.c”, para el “grad” el fichero ““edgeDetectorA3B2.c” y para “grad denoise” “edgeDetectorA3B3.c”. Tras realizar la ejecución con estos programas, observamos que la version paralelizada que obtiene un mejor resultado en lo referente a tiempos es la del fichero “edgeDetectorA3B3.c”, en la que se han paraleizado los bucles for de la parte del código que se encarga de obtener la imagen “Denoising”.

4=>

El fichero con el que hemos comparado a la versión serie, ha sido el que nos ha ofrecido mejores resultados en las pruebas anteriores, que ha sido el fichero “edgeDetectorA3B3.c”.

Los resultados obtenidos tras la ejecución sobre las imágenes que son ofrecidas por los profesores son los siguientes:

Resolución	Tiempo de ejecución del programa serie (s)	Tiempo de ejecución con los cambios previos	Aceleración (SpeedUp)	Tasa de Fps
SD	0.079114	0.022651	3.49274	44.14816
HD	0.322653	0.13737	2.34878	7.27961
FHD	0.76117	0.263520	2.88847	3.794778
UHD-4k	3.066959	0.834808	3.673849	1.19788
UHD-8k	12.803694	4.876882	2.625376	0.205049

La tasa de FPS se calculó dividiendo 1/T y la aceleración = Tserie/Tcamios.

5=>

Añadiendo al comando “gcc la bandera -O3” los resultados obtenidos son los siguientes:

Resolución	Tiempo de ejecución del programa serie (s)	Tiempo de ejecución con los cambios previos	Aceleración (SpeedUp)	Tasa de Fps
SD	0.029003	0.011705	2.477829	85.433585
HD	0.103874	0.035403	2.934045	28.246194

FHD	0.235530	0.085533	2.753674	11.69139
UHD-4k	0.986294	0.319783	3.08426	3.12712
UHD-8k	4.453183	1.847228	2.410738	0.541352

La bandera -O3 es el nivel más alto de optimización posible, activando optimizaciones que son caras en términos de tiempo de compilación y uso de memoria. La bandera -O3 implementa el desenrollado de bucles provocando que aumente la huella de la caché provocando una reducción del rendimiento en el acceso a ella, pese a que según los resultados obtenidos, no se ha producido una reducción del rendimiento si no una mejora. Según hemos investigado, la flag -O3 realiza una vectorización de bucles que supondría un aumento de la velocidad significativo, partiendo de esto y de los resultados obtenidos, creemos que si se aplica una vectorización de bucles pero el desenrollado de bucles no está activado.

- Aclaraciones del ejercicio:

  Se han incluido ficheros .out donde poder observar los resultados de tiempos para los apartados 2,3, 4 y 5.