Ejercicio 0:

Para este ejercicio, hemos incluído la salida de ejecutar el comando “cat /proc/cpuinfo” en un fichero. Esta práctica la hemos realizado en los ordenadores del laboratorio 14 de la EPS, que tienen 2 procesadores, con 2 cores y 2 siblings. Dado que el número de cores y de siblings es igual, estas máquinas no permiten hyperthreading.
Ambos procesadores tienen una frecuencia de 1600 MHz.

Ejercicio 1:
SALIDA DE OMP2

omp2.output

```
Inicio: a = 1,   b = 2,  c = 3
        &a = 0x7ffd29128f04,x   &b = 0x7ffd29128f08,    &c = 0x7ffd29128f0c

[Hilo 0]-1: a = 0,       b = 2,  c = 3
[Hilo 0]         &a = 0x7ffd29128ed8,    &b = 0x7ffd29128f08,    &c = 0x7ffd29128ed4
[Hilo 0]-2: a = 15,      b = 4,  c = 3
[Hilo 2]-1: a = 0,       b = 2,  c = 3
[Hilo 2]         &a = 0x7f51dd572e58,    &b = 0x7ffd29128f08,    &c = 0x7f51dd572e54
[Hilo 2]-2: a = 21,      b = 6,  c = 3
[Hilo 3]-1: a = 0,       b = 2,  c = 3
[Hilo 3]         &a = 0x7f51dcd71e58,    &b = 0x7ffd29128f08,    &c = 0x7f51dcd71e54
[Hilo 3]-2: a = 27,      b = 8,  c = 3
[Hilo 1]-1: a = -569106140,      b = 8,  c = 3
[Hilo 1]         &a = 0x7f51ddd73e58,    &b = 0x7ffd29128f08,    &c = 0x7f51ddd73e54
[Hilo 1]-2: a = -1578480335,     b = 10,         c = -1578480365

Fin: a = 1,      b = 10,         c = 3
        &a = 0x7ffd29128f04,    &b = 0x7ffd29128f08,    &c = 0x7ffd29128f0c
```

1. ¿Cómo se comporta OpenMP cuando declaramos una variable privada?

El comportamiento de OpenMP al declarar una variable privada depende de cómo haya sido declarada. En el ejemplo omp2 proporcionado, tenemos los siguientes dos casos:

1-La variable se declara en una cláusula private()

Al hacer esto, la variable no se inicializa previamente, quedando dentro del hilo, una "copia" de la variable a la que solo éste puede acceder pero con un valor inicial aleatorio dentro de la región paralela

2-La variable se declara en una cláusula firstprivate()

Al hacer esto, la variable se inicializa con el valor que ésta tenía antes de la región paralela y además sigue comportándose como una variable privada (cada hilo tiene su propia versión de esa variable)

2. ¿Qué ocurre con el valor de una variable privada al comenzar a ejecutarse la región paralela?

Como antes hemos comentado, la variable declarada con private() obtiene un valor aleatorio en la región paralela. En nuestro caso, podemos observar este fenómeno en que para los distintos hilos, a tiene valores 0, 0, 0 y -1578480335. En principio, podríamos pensar que los tres ceros de los hilos 0, 2 y 3 son un valor consistente que OpenMP da, sin embargo, con el valor de a en el hilo 1 vemos que esto no es así (de hecho, si ejecutamos varias veces, se corrobora esta observación). Además, los valores de a para los distintos hilos no tienen nada que ver con el valor inicial de ésta (a=1). Por el contrario, al declarar la variable firstprivate c, ésta se inicializa siempre con el valor 3, que es el que tenía antes de comenzar la región paralela.

3. ¿Qué ocurre con el valor de una variable privada al finalizar la región paralela?

Al finalizar la región paralela, todas las variables privadas retoman el valor que guardaban antes de que la región comenzara, independientemente de cómo fueran inicializadas . Esto es porque, dentro de cada hilo, se le asigna una nueva dirección a cada variable privada y por tanto, las modificaciones se hacen sobre esa dirección, que es distinta de la “verdadera” dirección de la variable en el hilo maestro. Es por esto que, al terminar la región paralela, la variable retoma su dirección original y, por tanto, su valor original (de antes de comenzar la región paralela)

4. ¿Ocurre lo mismo con las variables públicas?

Con las variables públicas, podemos observar que en la región paralela conservan el valor que tenian antes de la misma al inicializarse y que, además, son compartidas por todos los hilos (e.d. si un hilo accede a b para modificarlo, en este caso, todos los hilos accederán a ese valor, que estará previamente modificado o no, en ese mismo instante) . No obstante, cuando un hilo acceda a la variable compartida para leer únicamente, sin haber escrito en ella antes, leerá el valor “original” con independencia de que el resto de hilos la hubieran modificado. Si se usara para escribir (aunque fuera en otra variable) sí que se leería el valor modificado

Al finalizar la región paralela, la variable no retoma el valor que tenía antes, sino que continúa con el valor actualizado por los hilos. De hecho, podemos observar que la dirección de la variable pública es siempre la misma a lo largo del programa, tanto en el hilo maestro como en los hilos en paralelo.
Para evitar problemas de condiciones de carrera o de sobreescritura, OpenMP se encarga internamente de proteger las variables compartidas. Esta coordinación entre hilos puede ralentizar levemente la ejecución de los mismos, de manera que conviene no tener variables compartidas innecesarias. Por otra parte, al no tener que inicializar una variable extra para cada hilo al comenzar la región paralela, la inicialización en si debería ser levemente más rápida.

Ejercicio 2:

1 ¿En qué caso es correcto el resultado?

El resultado es correcto en la versión 2

2 ¿A qué se debe esta diferencia?

La versión 1 da un resultado distinto (y erróneo) al que da la versión 2 debido a que falta hacer "reduction(+:sum)". Al no tener esta cláusula, cada hilo calcula un valor para sum independiente (el correspondiente a su parte del bucle) pero no se calcula el valor real, es decir, cada hilo realiza el producto escalar de alguna parte de los vectores, pero no se calcula el resultado del producto escalar, (que sería la suma de esos resultados parciales)

si v y u son vectores de tamaño n, v = (V1 V2 … Vn), u = (U1 U2 … Un), el producto escalar de u y v es:

<v, u> = V1*U1 + V2*U2 + … + Vn*Un

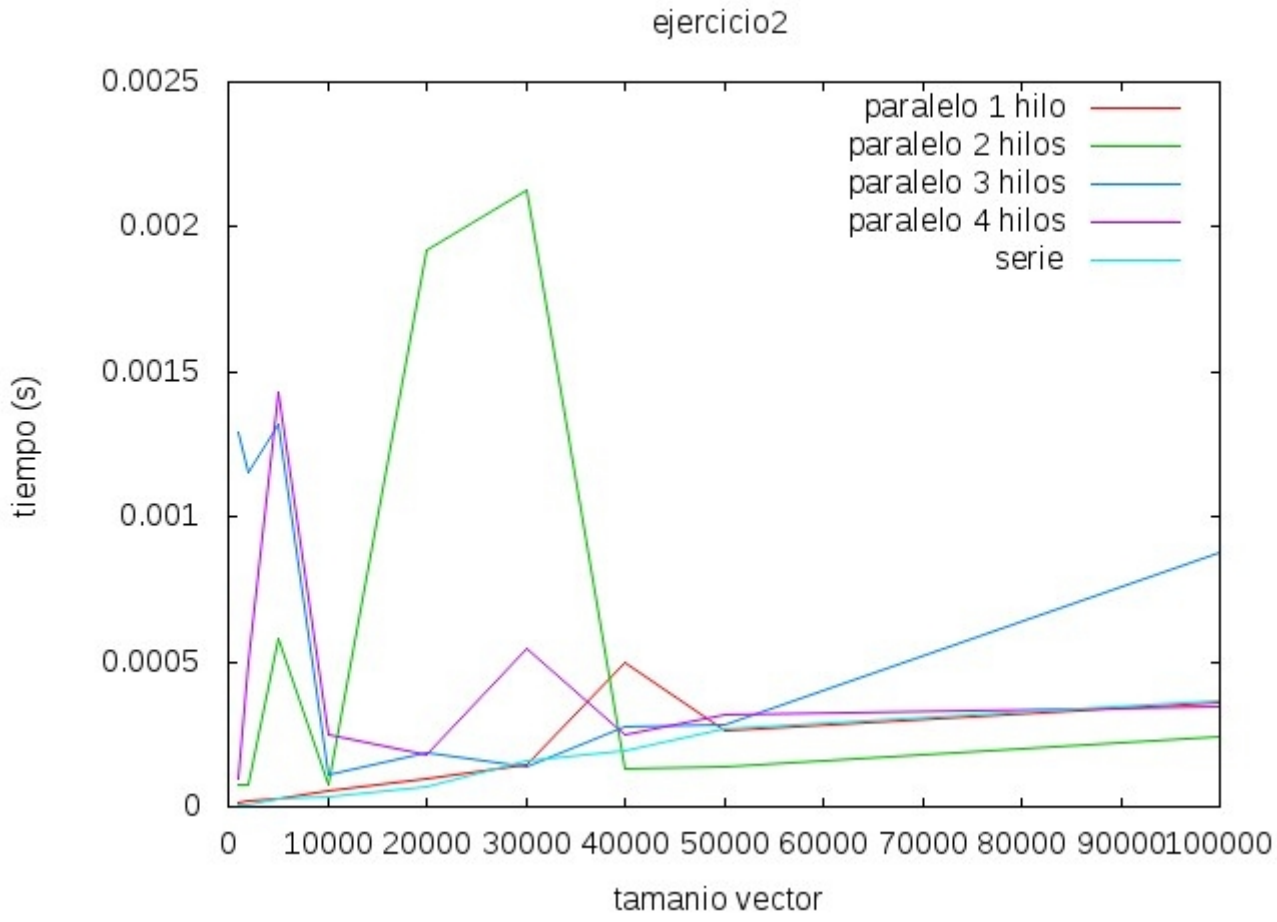
si los hilos se dividen el trabajo de manera que uno de ellos hace el producto escalar desde 1 hasta i (siendo i < n) se realiza lo siguiente:

sum_1 = V1*U1 + … + Vi*Ui
sum_2 = V(i+1)*U(i+1) + … Vn*Un

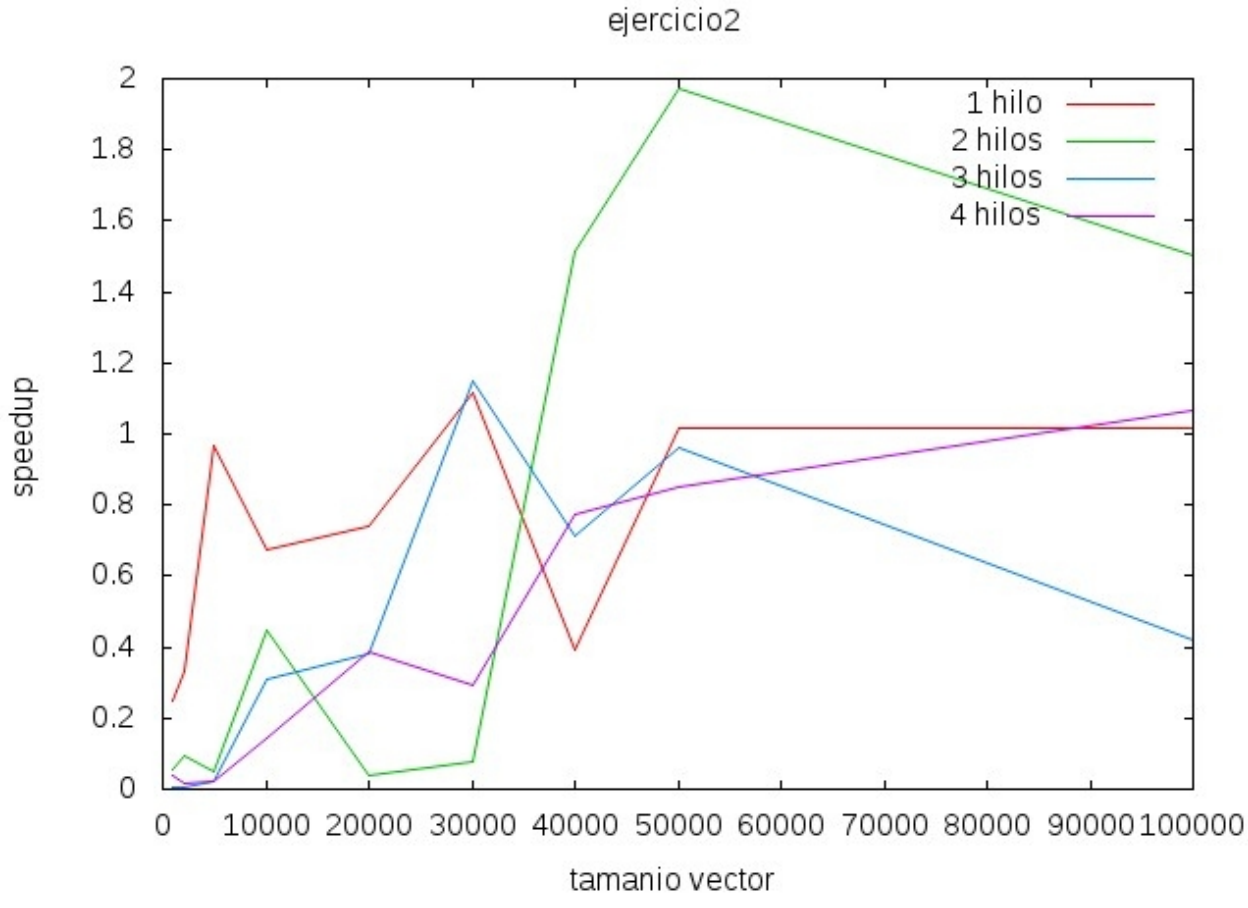
Y por tanto: <v, u> = sum_1 + sum_2
Al no incluir la cláusula reduction, no se realiza esta última suma, quedando un resultado incorrecto

Tras realizar la simulación para los distintos tamaños tal y como se especifica en el enunciado, hemos obtenido las siguientes gráficas que representan el rendimiento de las distintas ejecuciones variando el tamaño y el número de hilos:

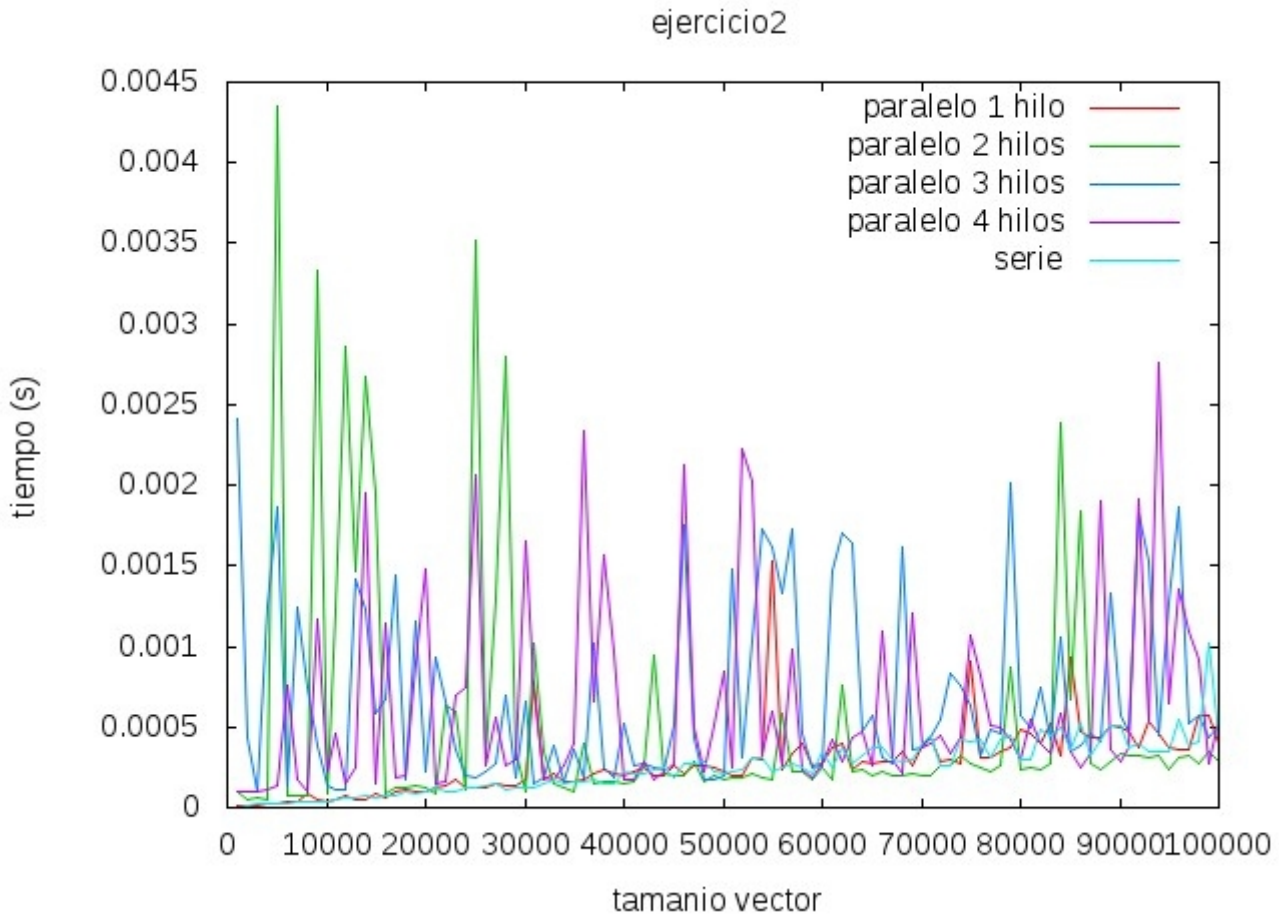
Tiempo de Ejecución



Speedup



Dado que los resultados parecían un poco extraños en principio, decidimos tomar un mayor tamaño muestral con el siguiente resultado:



Como podemos observar, las gráficas fluctúan bastante a excepción de las que utilizan un solo hilo en paralelo y la versión en serie. Creemos que esto es debido a que la máquina, al tener 2 procesadores con 2 cores cada uno (y no permitir hyperthreading), tiene que "expulsar" periódicamente a algún hilo del procesador para poder ejecutar, por ejemplo, el sistema operativo (nótese que todas estas mediciones se hicieron sin procesos extra ejecutándose en segundo plano). Y dado que en la región paralela se espera a que terminen ambos hilos para continuar con la ejecución, si uno de ellos se bloquea, se ha de esperar a que pueda retomar y acabar la ejecución. Por tanto, a mayor número de hilos, más rápido se debería ejecutar el código pero a la vez es más probable que alguno de ellos quede bloqueado, ralentizando así la ejecución.

1 En términos del tamaño de los vectores, ¿compensa siempre lanzar hilos para realizar el trabajo en paralelo, o hay casos en los que no?

No es seguro decir que siempre compensa lanzar hilos para realizar el trabajo, pues para valores pequeños del tamaño de los vectores, observamos que todas las versiones en paralelo son más lentas que la versión en serie. Esto es debido a que, aunque entre varios hilos tarden menos en realizar la ejecución, la fase de inicialización de los hilos consume suficiente tiempo como para que en general no compense.
Para tamaños mayores del vector, podemos observar en las gráficas que lanzar un solo hilo en

paralelo tiene mejores resultados que la versión en serie y que, salvo que alguno de los hilos sea bloqueado por algún otro proceso, cuanto más ocupen el procesador los hilos, más rápido se realiza la ejecución.
Con esto nos referimos a que, en nuestro caso la situación ideal sería la de tener dos hilos ejecutándose en cada uno de los dos procesadores (un total de 4 hilos, que es el máximo número de hilos que la máquina soporta). Lanzar más hilos es una pérdida de tiempo, pues siempre habría uno bloqueado, esperando a poder utilizar la cpu para ejecutarse y lanzar menos hilos implicaría que parte de la cpu no estuviera siendo utilizada (por tanto no es del todo eficiente)
Sin embargo, como ya hemos puntualizado antes, a mayor número de hilos, más probable es que alguno sea bloqueado para que algún proceso del sistema pueda utilizar el procesador. Por tanto tampoco es seguro decir que lanzar 4 hilos (en el caso de este ordenador) siempre dará mejores resultados.
Este último problema podría solucionarse de varias maneras:

-añadiendo más procesadores, o más núcleos o permitiendo hyperthreading
-implementando un sistema operativo de política apropiativa (no expulsará a los hilos hasta que terminen o se bloqueen esperando una señal de entrada/salida)
-diseñando el hardware para este problema en específico, de manera que no se requirieran procesos de un sistema operativo

A pesar de todo lo anteriormente dicho, sí que podemos decir que para tamaños mayores de los vectores suele compensar lanzar más hilos.

2 Si compensara siempre ¿En qué casos no compensa y por qué?

No compensa sobre todo para casos en los que los hilos no ejecuten demasiadas instrucciones durante la región paralela (por ejemplo, que el bucle tuviera pocas iteraciones), pues la fase de inicialización sería más costosa en tiempo que la ejecución en sí del código.

3 ¿Se mejora siempre el rendimiento al aumentar el número de hilos a trabajar?

Como hemos visto, no siempre se mejora el rendimiento al aumentar el número de hilos por causa de limitaciones hardware (tener más hilos que cores no es eficiente) y del sistema operativo, el cuál requiere los procesadores para ejecutarse periódicamente.
De hecho, en la gráfica de tiempos, observamos que la ejecución con 2 hilos es más rápida que la de 3 y 4 hilos.

4 Si no fuera así, ¿a qué se debe este efecto?

Esto es debido a las características del hardware, pues si hubiera más cores, los resultados serían de otra forma (probablemente las ejecuciones con 3 o 4 hilos acabaran siendo más eficientes)
También hay que tener en cuenta que al aumentar el número de hilos, se disminuyen el número de iteraciones del bucle que hace cada hilo y esto tiene impacto en el rendimiento.
Por ejemplo, en un caso extremo en el que hubiera tantos hilos como iteraciones del bucle se fueran a hacer, se consumiría mucho más tiempo inicializando los hilos que el tiempo que se debería ahorrar al paralelizar el bucle.

Ejercicio 3:

ejecutando el código de multiplicación de matrices en sus distintas versiones (paralelizando los distintos bucles for y haciéndolo en serie) obtenemos las siguientes tablas de datos:

Tablas para matrices de tamaños 1000x1000

Tabla con tiempos de ejecución (en segundos)

Version/hilos	1	2	3	4
MatrixComputeSerie	6.344164	6.382536	6.401066	6.336224
MatrixComputePar1	7.920110	5.424810	17.713832	16.549775
MatrixComputePar2	7.355925	3.857233	8.163340	5.753792
MatrixComputePar3	7.202077	3.667365	3.629160	3.341416

Tabla con aceleraciones

Version/hilos	1	2	3	4
MatrixComputeSerie	1	1	1	1
MatrixComputePar1	0.8010	1.1765	0.3613	0.3828
MatrixComputePar2	0.8624	1.6546	0.7841	1.1012
MatrixComputePar3	0.8808	1.7403	1.7637	1.8962

1 ¿Cuál de las tres versiones obtiene peor rendimiento? ¿A qué se debe?

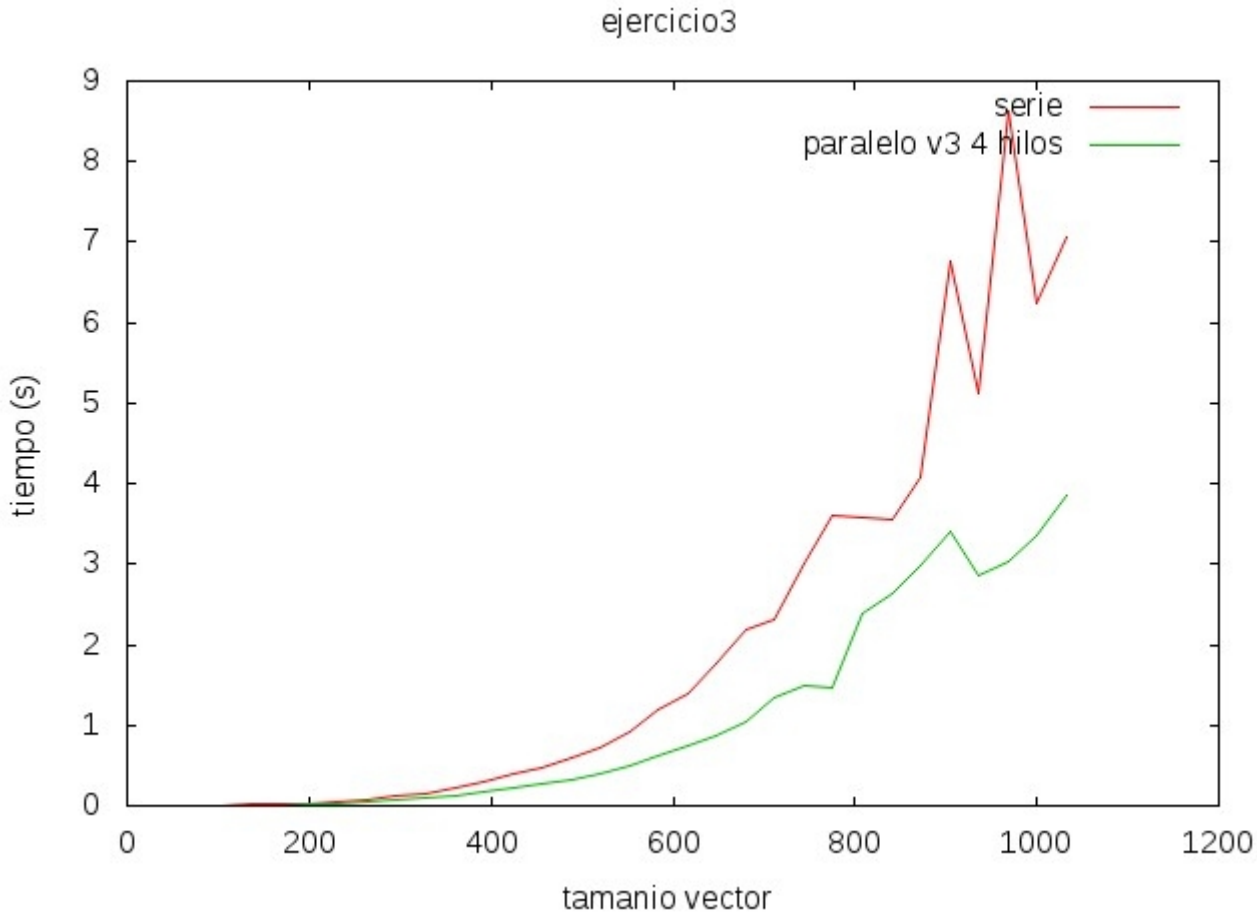
La versión que obtiene peor rendimiento es la versión 1, que se corresponde con la versión que tiene la paralelización en el bucle más interno.
Esto se debe a que el programa entra en el bucle paralelizado un total de 1.000.000 veces y por tanto lanza $1*10^6$, $2*10^6$, $3*10^6$ y $4*10^6$ hilos respectivamente en las 4 simulaciones hechas. Por tanto, cualquier mejora de rendimiento que pudiera suceder en el bucle, pierde impacto debido a la cantidad de veces que se han de inicializar los hilos. Además de esto, en este bucle se ha de realizar una cláusula reduction y por tanto, la misma finalización de la región paralela debe ejecutar alguna instrucción más, las cuales acaban teniendo cierto impacto dado que dichas instrucciones se ejecutarían 10^6 veces

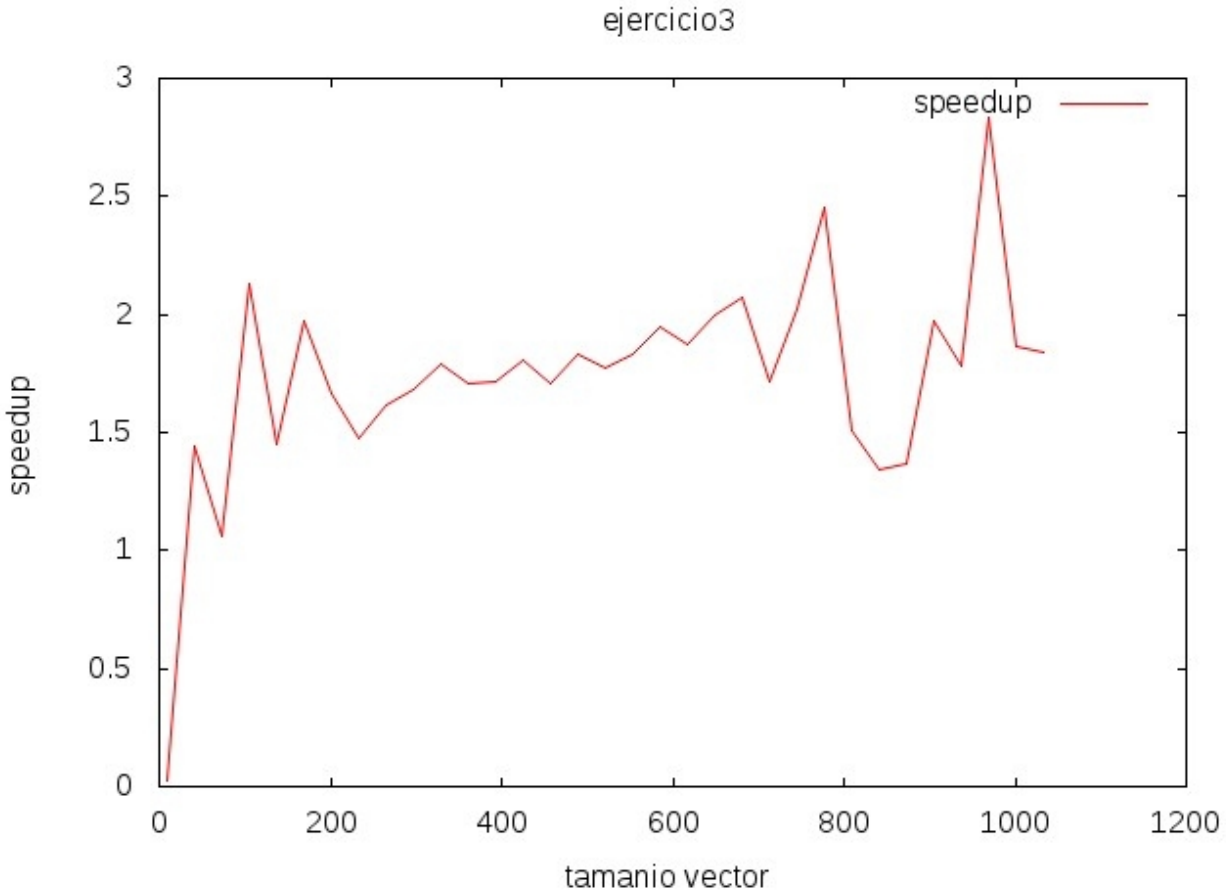
En definitiva, obtiene peor rendimiento por lanzar cantidades tan inmensas de hilos y tener que inicializarlos.

2 ¿Cuál de las tres versiones obtiene mejor rendimiento? ¿A qué se debe?

La versión que obtiene mejor rendimiento es la 3, que se corresponde con la versión que paraleliza en el bucle más externo.
Esto se debe a que el programa solo lanza 1, 2, 3 y 4 hilos respectivamente, que paralelizan una gran cantidad de instrucciones, por tanto, ni los hilos se entorpecen los unos a los otros, ni se inicializan cantidades absurdas de hilos (lo cual ralentiza al programa), esto implica que la mejora obtenida por la paralelización es lo más eficiente posible.
Además de esto, dado que un solo hilo tarda tanto en terminar su propia ejecución, si el sistema operativo lo bloquea, el número de “instrucciones perdidas” por este motivo son despreciables en comparación con el total de instrucciones ejecutadas.

Tomando la mejor versión y simulando tal y como se especifica en el enunciado, hemos obtenido las siguientes gráficas:





Como podemos observar, para tamaños pequeños de las matrices, ambas versiones se asemejan bastante, pero se van distanciando, resultando en mucho menos tiempo las ejecuciones de la versión en paralelo.
Con estas gráficas, podemos ver que la versión en paralelo, aunque no llega a ser 4 veces más rápida que la versión en serie, sí que se acerca mucho a ser 3 veces más rápida.
Una aceleración de 4 sería la aceleración teórica que se podría alcanzar, para un resultado práctico, es una cota superior. No es posible alcanzar esta cota debido a las ejecuciones del sistema operativo, las inicializaciones de variables, la sincronización entre hilos, etc.

En general, podemos observar que para tamaños grandes de las matrices, la aceleración tiende a ser 2. Esto es debido a que el sistema operativo acaba concediendo 2 cores de media durante la ejecución del programa, de manera que durante la mayor parte de la ejecución solo se ejecutan dos hilos a la vez. Sin embargo, al ser un total de 4 hilos, es posible que el sistema conceda más (o menos) cores de media para que se vayan ejecutando los otros hilos (por este motivo, la ejecución con 4 hilos acaba siendo más rápida que la de 2 hilos). Además, esto explica la existencia de picos en las gráficas.

Con respecto a los tamaños, es fácil observar en la gráfica de aceleración, que la versión en paralelo comienza a ser más rápida que la versión en serie en torno a las matrices de tamaño 75x75, manteniéndose con un rendimiento superior para matrices mayores.
El crecimiento de la aceleración se debe a lo anteriormente comentado de que la región paralela se aprovecha mucho más