Informe práctica 3

Zdravko Dimitrov Arnaudov
04/01/2021

Prefacio

La multiplicación de matrices a gran escala es un trabajo extenso y complicado de computar. En ocasiones como esta y cuando el tiempo de ejecución es crucial, conviene aprovechar todo el potencial que dispone nuestro computador para operar más deprisa. La práctica en cuestión nos hace comprender cómo paralelizar problemas del estilo en un multiprocesador de memoria compartida usando OpenMP y apreciar el impacto que tiene el desbalanceo de carga de trabajo en el rendimiento.
Sin duda, al haber terminado la actividad contamos con un montón de cosas nuevas aprendidas y con más conocimiento sobre el poder de cómputo que tiene nuestro propio ordenador, lo cual personalmente me llama bastante la atención.
Desde mi punto de vista, me he notado con más soltura a la hora de analizar el reparto de trabajo entre los threads y cómo influyen los distintos algoritmos de equilibrio de carga para conseguir resultados mejores. La parte más complicada en mi lugar ha sido la lectura y comprensión del código, pues pienso que no estamos del todo acostumbrados a hacerlo con agilidad. Por último, no creo que hubiese añadido o eliminado algún contenido con respecto a la práctica. Es bastante completa y contempla todo lo necesario para satisfacer su propósito.

Índice

1. Sistema

Con intención de obtener una descripción del sistema lo más detallada posible, buscaremos una serie de comandos equivalentes a '*cpuinfo*' o '*lscpu*', puesto que disponemos de Mac OS y dichas instrucciones no son reconocidas.

Uno de ellos se corresponde con '*system_profiler SPHardwareDataType'*, el cual nos muestra los aspectos hardware más generales y relevantes a tener en cuenta:

```
Hardware Overview:

      Model Name: MacBook Pro
      Model Identifier: MacBookPro17,1
      Processor Speed: 2,4 GHz
      Number of Processors: 1
      Total Number of Cores: 8
      L2 Cache (per Core): 4 MB
      Memory: 8 GB
```

Otro de los comandos encontrados es '*sysctl -a | grep machdep.cpu* ' y este en particular nos aporta información adicional sobre el número de cores físicos y lógicos del procesador:

```
machdep.cpu.cores_per_package: 8
machdep.cpu.core_count: 8
machdep.cpu.logical_per_package: 8
machdep.cpu.thread_count: 8
machdep.cpu.brand_string: Apple M1
machdep.cpu.features: FPU VME DE PSE TSC MSR PAE MCE CX8 APIC SEP MTRR
PGE MCA CMOV PAT PSE36 CLFSH DS ACPI MMX FXSR SSE SSE2 SS HTT TM PBE
SSE3 PCLMULQDQ DTSE64 MON DSCPL VMX EST TM2 SSSE3 CX16 TPR PDCM SSE4.1
SSE4.2 AES SEGLIM64
machdep.cpu.feature_bits: 151121000215084031
machdep.cpu.family: 6
```

Por último, emplearemos un tercer comando '*sysctl hw*' para observar más en detalle el reparto de cores y también información adicional asociada a la memoria cache:

```
hw.ncpu: 8
hw.byteorder: 1234
hw.memsize: 8589934592
hw.activecpu: 8
hw.physicalcpu: 8
hw.physicalcpu_max: 8
hw.logicalcpu: 8
hw.logicalcpu_max: 8
hw.cputype: 7
hw.cpusubtype: 4
hw.cpu64bit_capable: 1
hw.cpufamily: 1463508716
hw.cpusubfamily: 0
hw.cacheconfig: 8 1 1 0 0 0 0 0 0 0
hw.cachesize: 3612786688 65536 4194304 0 0 0 0 0 0 0
hw.pagesize: 4096
hw.pagesize32: 16384
hw.busfrequency: 100000000
hw.busfrequency_min: 100000000
hw.busfrequency_max: 100000000
hw.cpufrequency: 2400000000
hw.cpufrequency_min: 2400000000
hw.cpufrequency_max: 2400000000
hw.cachelinesize: 64
hw.l1icachesize: 131072
hw.l1dcachesize: 65536
hw.l2cachesize: 4194304
hw.tbfrequency: 24000000
```

Teniendo en cuenta las especificaciones de nuestra máquina y que la arquitectura es ARM, para poder hacer uso de las herramientas que ofrece OpenMP es necesario usar una compilación distinta para nuestros programas. Inicialmente, ha sido imprescindible emplear una consola (Terminal, en Mac) corriendo un software específico que traduce instrucciones de x86 a ARM (Rosetta). Después, para compilar nuestro programa deseado en C y emplear tanto las directivas como funciones de OpenMP, el comando de compilación será: '*gcc -Xpreprocessor -fopenmp file.c -o file -lomp*'

2. Resolución de las actividades propuestas

En este apartado, expondremos las contestaciones a las preguntas planteadas en cada una de las actividades.

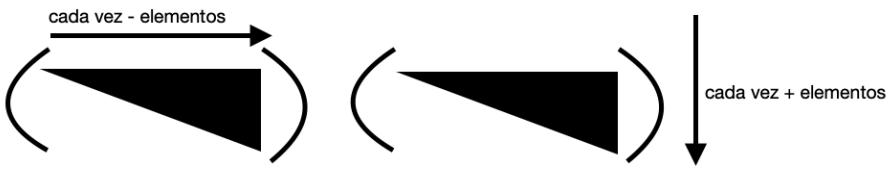
2.1. Ejercicio 1

El objetivo de este ejercicio será paralelizar la función matmul del fichero matmul.c usando directivas y funciones de OpenMP. Además, nuestro programa deberá ser capaz de calcular en tiempo de ejecución el número máximo de threads en la región paralela y posteriormente crearlos. Como requisito final, se deberá medir el tiempo de ejecución de la función con las funciones correspondientes de OpenMP e imprimirlo por pantalla.

A continuación, definiremos brevemente la funcionalidad de cada una de las directivas y funciones de OpenMP empleadas. La primera de ellas se corresponde con la directiva '*#pragma omp parallel'* para delimitar una sección paralela en la que computarán varios threads. Seguidamente después, nos encontramos con otra directiva para repartir las iteraciones de un bucle for por varios threads, que se denomina '*#pragma omp for'***.**
En cuanto a las funciones de OpenMP utilizadas, podemos destacar dos: '*omp_get_wtime()'* para calcular el tiempo de ejecución mediante la diferencia de un tiempo final con el inicial y '*omp_get_max_threads()'* para obtener el número máximo de threads a computar en la región paralela.

Una vez detalladas las directivas y funciones abordaremos las etiquetas OpenMP asociadas a las variables usadas en el código. En la directiva *'#pragma omp parallel'* hemos añadido una serie de etiquetas para delimitar variables compartidas, privadas y con otra función. Las variables que se han considerado necesarias declarar como privadas son aquellas que iteran los bucles para recorrer las matrices: i, j y k.
En ese caso, usamos '*private(i,j,k)*'. Las variables que deben ser necesariamente compartidas son las matrices en cuestión y su dimensión correspondiente. Entonces, emplearemos '*shared (A,B,C, dim)'*. Por último, como el enunciado nos pide calcular el número máximo de threads en tiempo de ejecución y usarlos en la región paralela, debemos señalarlo también en la directiva con '*num_threads (max_threads)'*.

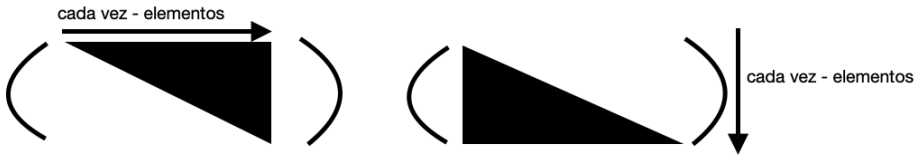
En cuanto al reparto de trabajo de los threads, sabemos que la carga de trabajo no será equilibrada. Para empezar, la función matmul opera de forma normal la multiplicación de matrices, filas por columnas. Como estamos empleando matrices triangulares superiores únicamente, se tendrán en cuenta aquellas operaciones en las que el resultado va a ser cero. Además, unas multiplicaciones costarán más que otras. Partiendo de esta premisa, podemos deducir que la carga de cómputo no será la misma para todos los threads aunque trabajen sobre el mismo número posible de elementos distintos de cero a medida que se recorren las filas y columnas. Por lo tanto, se da lugar a desbalanceo de carga empobreciendo los tiempos de ejecución.



Finalmente, calcularemos la ganancia obtenida fruto de la paralelización. Antes de calcular los tiempos, debemos saber qué tamaño de matriz usaremos como referencia. Probando varios, para un tiempo de cómputo secuencial de 10s necesitamos un tamaño de matriz cuadrada de 2140. Una vez establecida dicha dimensión, procedemos con el speedup.
El tiempo de ejecución que obtenemos de media en modo secuencial es 10,0017s. Aplicando paralelización, el nuevo tiempo que obtenemos es de 2,36805s. La ganancia resultante será 4,2236.
En base a los resultados obtenidos, podemos apreciar que el cálculo es superior a 4 veces el tiempo de cómputo en modo secuencial. Además, podemos extraer una serie de conclusiones y es que el speedup teórico no se corresponde con el obtenido, ronda aproximadamente la mitad de lo que podría haber sido. Estudiando más en detalle la carga de cómputo que se impone sobre un thread, es evidente que no es excesiva, al final el código se reduce a sumas y multiplicaciones medianamente complejas.
Nuestra hipótesis es que este aspecto puede ser un factor decisivo a la hora de obtener la ganancia. Para comprobar si es válida, calcularemos nuevamente el speedup aumentando drásticamente las dimensiones de las matrices, con tal de incrementar la carga de trabajo en modo secuencial y paralelo.
Con unas dimensiones de matriz de 5000, el tiempo en modo secuencial se eleva a los 290s y en paralelo se estanca en los 55s. La ganancia en esta segunda muestra es superior a 5, pudiendo concluir que a mayor intensidad de cómputo mayor ganancia obtendremos paralelizando.

2.2. Ejercicio 2

En esta segunda actividad, nos centraremos en la manipulación y ajuste del reparto de trabajo en una función de multiplicación de matrices más optimizada, mediante los distintos algoritmos de equilibrio de carga que dispone OpenMP.
La nueva función sobre la que trabajaremos es matmul_sup y está optimizada para multiplicar matrices triangulares superiores con inferiores. Antes de tocar el código, enfocamos nuestra atención en el algoritmo para comprender en qué se basa la optimización y qué aspecto tiene la carga de cómputo a medida que se progresa con la multiplicación.
Leyendo atentamente el código, nos percatamos de todas las modificaciones que se han implantado sobre el algoritmo original para optimizarlo. En particular, solo se computan los elementos por encima o por debajo de la diagonal de la matriz, sea triangular superior o inferior. Entonces aquellas operaciones que vayan a dar cero se descartan.
El primer cambio aparente es que el contador i que recorre las filas de la matriz triangular superior va desde 0 a (dim -1) porque la última fila es nula al igual que la última columna de la matriz triangular inferior. El segundo cambio que podemos apreciar es la inicialización del contador k que empieza en (i+1) para saltarse los ceros de las matrices triangulares.
Una vez entendido como funciona el algoritmo, nos queda revisar la distribución de la carga de cómputo. Haciendo otro pequeño esquema, vemos que la intensidad de trabajo decrece progresivamente, lo cual puede provocar desbalanceo de carga si el reparto no es adecuado.



Después de los planteamientos realizados, comenzaremos a realizar ejecuciones y obteniendo resultados. Para empezar, calcularemos la ganancia obtenida de paralelizar la función. Los tiempos que obtenemos de media son 4,6568s en secuencial y 1,4955s en paralelo con un speedup de 3,1138.
Seguidamente, ejecutaremos probando los algoritmos de equilibrio de carga.
Con static tenemos de media 1,4806, con dynamic 1,0152s y guided 1,0787.

Podemos observar algunas diferencias respecto a la ganancia obtenida con el ejercicio anterior sin la cláusula schedule. En este caso, es apreciable que la ganancia es significativamente menor porque paralelizando un algoritmo optimizado, se obtiene menos speedup que paralelizando un algoritmo sin optimizar. Esto quiere decir que si a un thread le lleva mucho menos esfuerzo cumplir su trabajo, no se obtiene tanto beneficio paralelizando el cómputo.

El algoritmo con mejor resultado es sin duda el dinámico. El estático es el más ineficiente porque en una función cuya tendencia de intensidad de trabajo es desigual, reparte de forma equivalente las iteraciones de los bucles induciendo al desbalanceo de carga. Esto quiere decir que unos threads terminarán de computar antes que otros y deberán esperar al resto para seguir trabajando o finalizar. El algoritmo dinámico solventa este problema, repartiendo de forma dinámica bloques de trabajo a medida que los hilos van acabando de computar, alcanzando un mejor reparto de trabajo. El guiado comparte la misma funcionalidad que el dinámico, salvo el tamaño de los bloques de trabajo que reparte. Inicialmente, distribuye bloques más grandes y los va reduciendo progresivamente ajustando el desbalanceo de carga al final. Como ya sabemos, matmul_sup concentra la mayor parte de los cálculos al principio, por este motivo, no llegaremos a aprovechar del todo el potencial de este algoritmo. Si la cantidad de trabajo fuese mucho más elevada, es posible que el dinámico llegase a sufrir por el overhead que supone repartir dinámicamente el trabajo en tiempo de ejecución, en ese caso el guiado podría llegar a destacar.

Cuando empleamos la cláusula '*schedule(static)'*, se reparten entre los threads las iteraciones del bucle en bloques cuyo tamaño es aproximadamente size / num_threads. Es decir, si nuestro tamaño de matriz fuese 64 y tuviésemos 8 threads, a cada thread le correspondería un bloque de 8 iteraciones. De esta forma, si el algoritmo se comportase como matmul_sup, los threads que se ocupasen de las últimas iteraciones podrían llegar a terminar antes y esperarían a la finalización de los primeros.
Nuestro interés es reducir al máximo el desbalanceo de carga, minimizando las esperas para poder seguir computando o terminar. Por lo tanto, el tamaño de bloque óptimo en static podría ser 1 y cuanto mayor es este tamaño, menos rendimiento.
Para comprobar si nuestros planteamientos son correctos, usaremos el algoritmo estático con tamaño de bloque 1 y compararemos los resultados con la versión anterior. Después de realizar varias ejecuciones, obtenemos un tiempo de media de 1,0629s, aún mejor que el guiado. Podemos concluir que en nuestro caso, es el tamaño más rápido.

Para el algoritmo dinámico, debemos determinar experimentalmente cuál es el tamaño del bloque óptimo. Probaremos con 10 tamaños distintos obteniendo su tiempo de respuesta y speedup correspondiente.

Con respecto a los tamaños de los bloques, utilizaremos 1, 2, 4, 6, 8, 16, 32 ,64, 128 , 256. En cada caso, realizaremos de nuevo diez ejecuciones obteniendo la media y la ganancia resultante.
A la vista de los resultados generados, podemos observar una región de estabilidad que se delimita entre el tamaño de bloque 1 y 8. A partir de 16, los tiempos crecen y la ganancia disminuye progresivamente hasta alcanzar la del algoritmo estático. La conclusión que podemos extraer es que aumentar el tamaño de bloque para el algoritmo dinámico, lo que hace es empeorar la ganancia y generar desbalanceo de carga.
Sobre el rango de valores en los que hemos identificado semejanza en cuanto a resultados, realizaremos una segunda tanda de ejecuciones para analizar más en detalle los tiempos. Una vez terminada la segunda prueba, apreciamos la misma tendencia y es que ningún tamaño destaca de los demás con diferencia.
Finalmente, para la magnitud y teniendo en cuenta las circunstancias de nuestro problema, lo ideal es seleccionar el tamaño de bloque que viene por defecto al emplear la cláusula '*schedule(dynamic*)', es decir, tamaño de bloque 1. En nuestro caso, el overhead que supone el reparto dinámico de trabajo en tiempo de ejecución, no implica una penalización apreciable. Si escalásemos más el problema, quizás si se comenzaría a apreciar y lo deseable sería buscar un tamaño de bloque que reduzca la cantidad de overhead pero tampoco genere demasiado desbalanceo de carga.

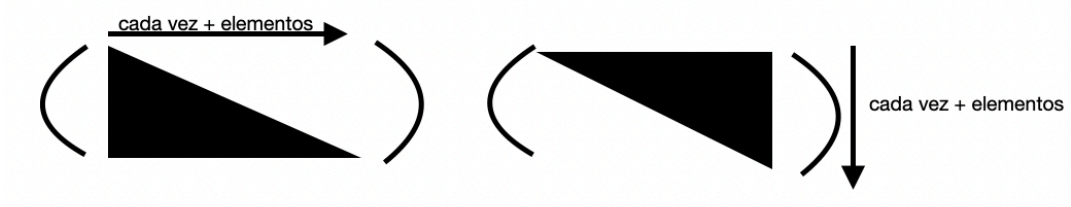
En relación al algoritmo guiado, como ya hemos explicado antes, obtiene un tiempo mejor que el estático porque reduce el desbalanceo de carga pero no tanto como el dinámico. Además, no puede apreciarse su potencial porque donde más intenso es el cómputo es al principio y este algoritmo reduce el desequilibrio al final.

2.3. Ejercicio 3

En este último ejercicio, estudiaremos el comportamiento y funcionalidad del algoritmo matmul_inf. Esta función optimiza la multiplicación de matrices triangulares inferiores con superiores. Para entender cómo funciona, leeremos en detalle su implementación y observaremos cómo se diferencia de los algoritmos anteriormente abordados.

Para empezar, el iterador i que recorre las filas comienza en 1 porque la primera matriz es triangular inferior y su primera fila es nula, al igual que la primera columna de la matriz triangular superior. La segunda distinción que podemos reconocer es el límite del iterador k, el cual avanza hasta k < i, para no calcular los ceros de las matrices triangulares.

Al haber descrito el funcionamiento del algoritmo, analizaremos la distribución de la carga de trabajo a medida que se recorren las matrices. Si nos volvemos a hacer un esquema, contemplamos que el cómputo más intenso reside al final de la multiplicación, a diferencia de matmul_sup que lo situaba al principio.



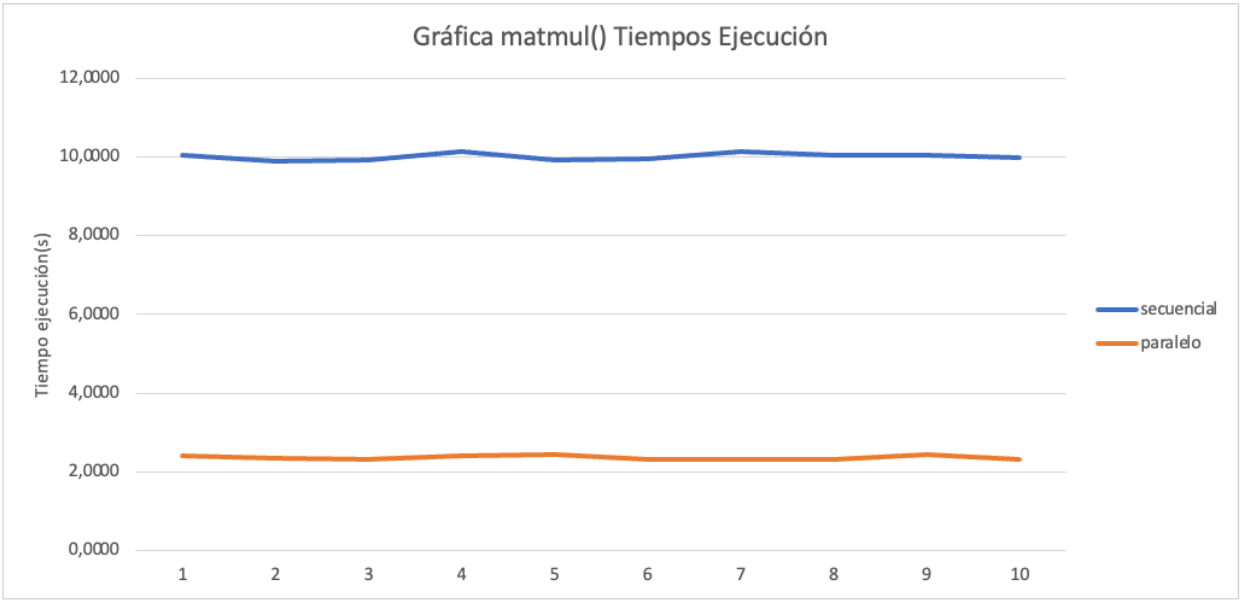
El algoritmo de equilibrio que mejor se adaptaría dadas las propiedades de esta función, sería el guiado porque ajusta el desbalanceo al final del cómputo, siendo esta la región donde más carga se genera. El algoritmo dinámico seguramente obtendría un tiempo muy bueno como en el anterior ejercicio, pero si aumentásemos el tamaño de las matrices, el guiado lo superaría con diferencia por reducir el overhead que implica repartir dinámicamente bloques de trabajo. También porque puede aprovechar su potencial como consecuencia del comportamiento de la función matmul_inf.
El peor algoritmo para este caso, sería de nuevo el estático porque el reparto de carga seguiría siendo desigual y se produciría desbalanceo de carga.

3. Metodología y desarrollo de las pruebas realizadas

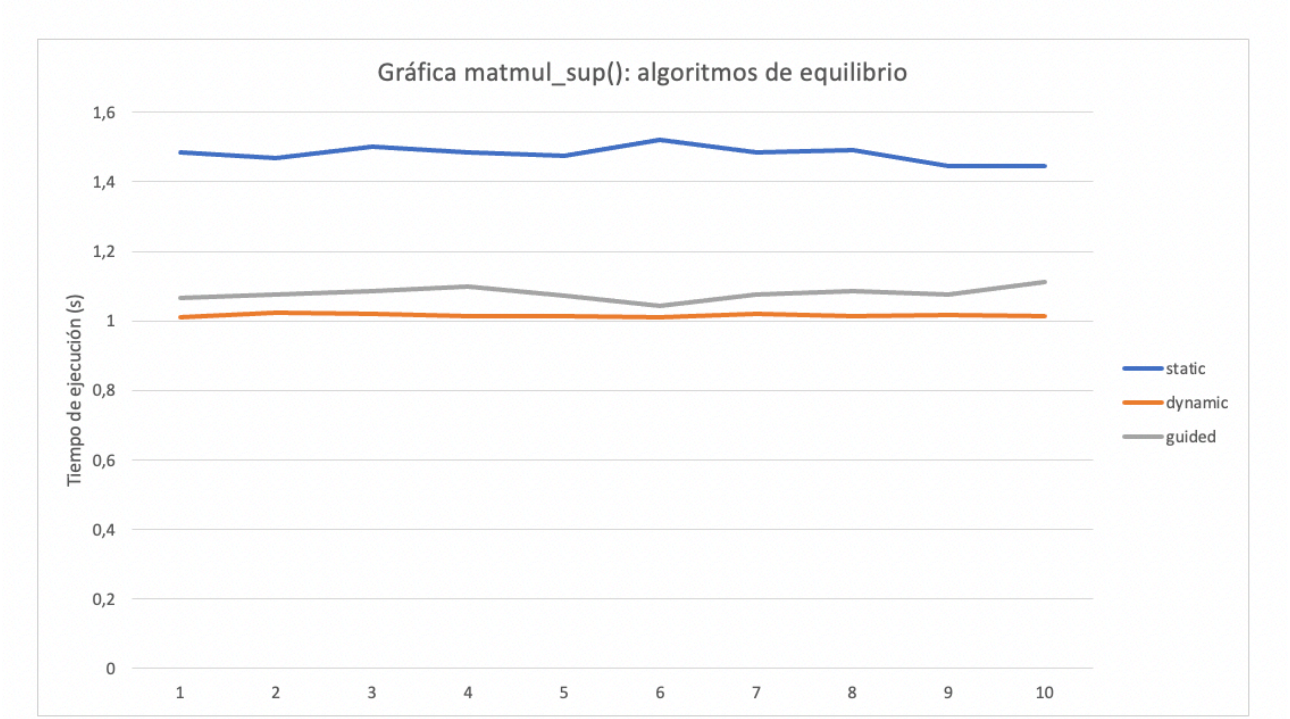
En este apartado de la memoria, contemplaremos la metodología y explicación de las pruebas realizadas.
En cada una de las actividades, nuestro objetivo ha sido anotar y comparar tiempos. Para cada caso, hemos realizado 10 ejecuciones seguidas, con tiempo de reposo entre activaciones, mediante un script sencillo que hemos llamado *matmulScript.sh*. Cabe destacar que el tamaño seleccionado para las matrices es 2140 porque en la primera actividad en modo secuencial, supone un tiempo de cálculo muy aproximado a los 10s. Todos los resultados anotados se encuentran en un fichero excel que se adjuntará con el programa.

```
#! /bin/bash
for i in {0..9}
do
        sleep .05
        ./matmul 2140
done
```

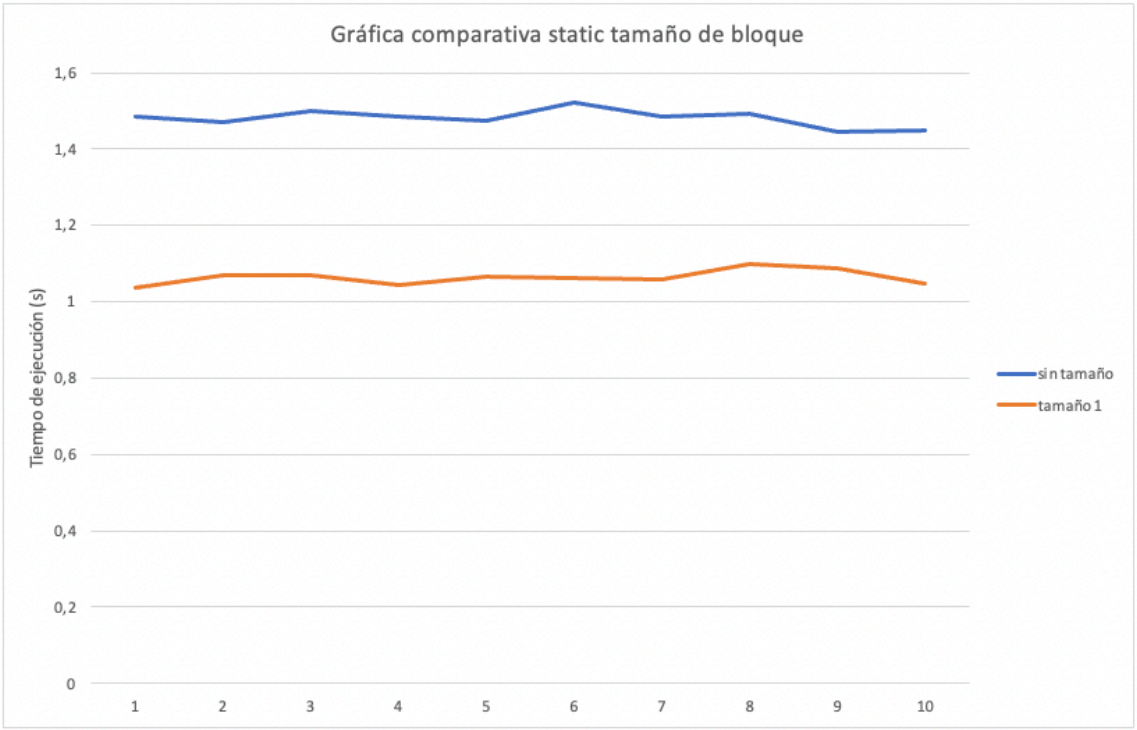
En el primer ejercicio, hemos tenido que determinar la ganancia que resulta de paralelizar la función matmul(). Se ha elaborado una tabla para obtener el tiempo medio de las ejecuciones en modo secuencial y paralelo, con el propósito de establecer un speedup más seguro y fiable. Antes de correr el programa, hemos tenido que compilar con:
‘*gcc -O3 -Xpreprocessor -fopenmp matmul.c -o matmul -lomp*



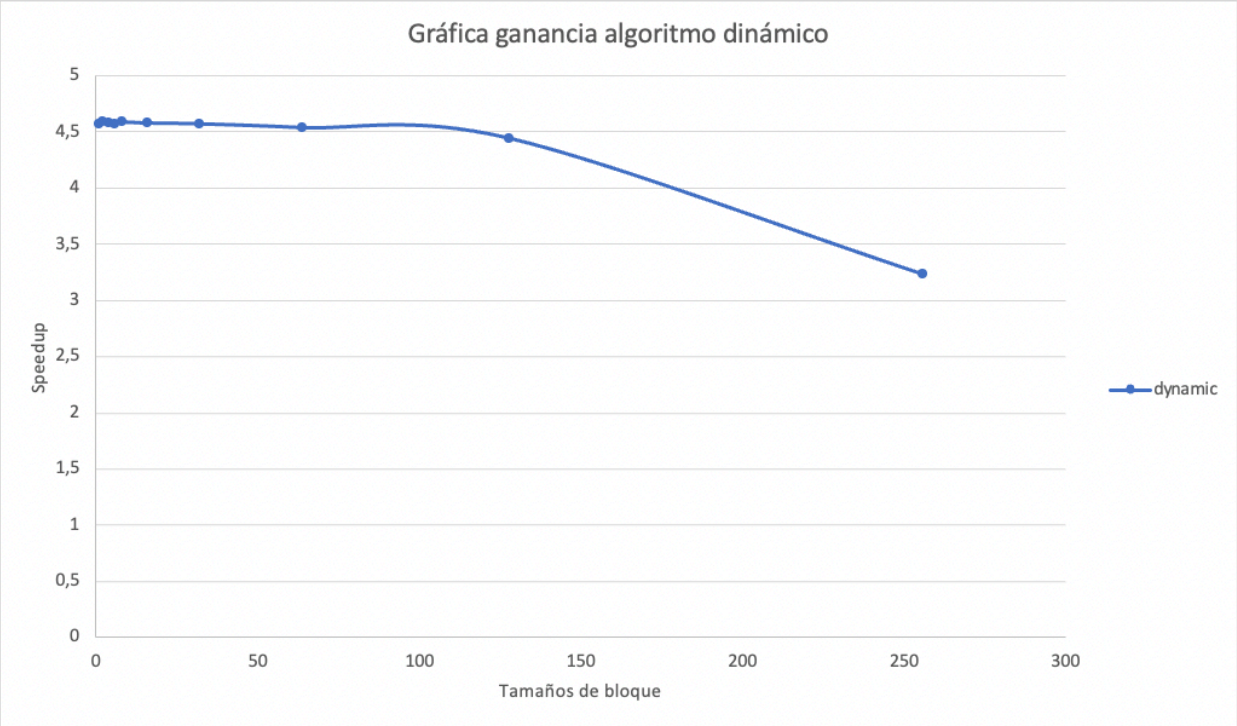
Es evidente que paralelizar este algoritmo representa una mejora significativa. Tal y como hemos comentado antes, si elevásemos la complejidad del cómputo, el speedup también incrementaría notablemente. La ganancia obtenida como ya se explicó antes fue de 4,22 aproximadamente.
En cuanto al segundo ejercicio para matmul_sup(), el procedimiento inicial ha sido el mismo. Hemos calculado la ganancia de paralelizar la función, de 3,11. Además, se han probado los distintos algoritmos de equilibrio sin especificar tamaño de bloque.



Estos resultados fueron explicados previamente por lo que seguiremos con las siguientes pruebas realizadas. Nuestro segundo paso fue determinar el tamaño de bloque óptimo para el algoritmo estático y dinámico. En el primer caso, confirmamos que el tamaño ideal era 1 para reducir al máximo el desbalanceo de carga.



Con el dinámico, tenía que verse reflejado experimentalmente con 10 tamaños distintos de bloque.
Finalmente, y después de dos tandas de ejecuciones, llegamos a la conclusión de que según las dimensiones de nuestro problema podríamos usar el tamaño por defecto con la cláusula *'schedule(dynamic)'*, es decir, 1. Para visualizar mejor la tendencia de los resultados con un simple vistazo, generamos una gráfica que muestra la ganancia obtenida según el tamaño de bloque. Sin duda alguna, a mayor tamaño menos rendimiento. A media que incrementamos las dimensiones del bloque, más nos acercamos al tiempo del algoritmo estático por el desbalanceo de carga.



4.Valoración

La práctica realizada me ha parecido super destacable por la facilidad que se han apreciado los repartos de trabajo según los algoritmos usados y cómo hemos aprovechado los conocimientos sobre los distintos algoritmos de equilibrio de carga para analizar los resultados que hemos obtenido en la práctica.
También me ha sorprendido lo mucho que importa un buen reparto de trabajo para minimizar los desbalanceos de carga cuando la distribución de trabajo del problema no es uniforme.
Con más tiempo, me habría gustado probar benchmarking con distintos números de threads en la región paralela para observar el comportamiento.