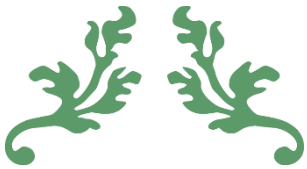


ARQUITECTURA DE COMPUTADORES

Técnicas de paralelización en arquitecturas de memoria compartida. OpenMP.

Álvaro Gómez Ramos 100307009 Grupo 81

Carlos Contreras Sanz 100303562 Grupo 81

Contenido

En este documento se describe el código de un programa secuencial que se nos ha pedido paralelizar utilizando OpenMP. Tras eso exponemos las diferentes soluciones que nos hemos planteado a lo largo del proceso de resolución de la práctica. Por último se selecciona una de ellas, y se responden todas las preguntas planteadas por los profesores sobre esta solución, cuyo código se entrega adjunto a este documento.

Descripción del código

Antes de plantearnos el paralelizar nada, es necesario analizar el código que se nos proporciona y ver que partes es posible paralelizar, que partes puede no ser rentable paralelizar, y cuales no es posible paralelizar.

Para ello empezamos estudiando el método main. En él, lo primero que se hace es declarar variables. Después se comprueba si es posible ejecutar (si hay archivo de entrada y de salida) y se comprueba si se ha activado la visualización o no. A continuación nos encontramos con un bucle, que se ejecuta tantas veces como numero de pasos nos diga el archivo de entrada. Tras el bucle solo queda lo referente a la visualización y a imprimir por pantalla el tiempo.

Nos damos cuenta de que la carga de trabajo está en el bucle for. Si nos paramos a analizar dicho bucle, vemos que prácticamente todo es por la llamada al método “calc_nbodies ()”. (Nota: toda la práctica se ha realizado sin tener en cuenta visualización, o lo que es lo mismo, teniéndola desactivada)

La llama al método “calc_nbodies” calcula, para cada paso, la posición final de todos los cuerpos. Para ello declara variables, reserva memoria y hace uso de tres bucles for. En el primero de ellos resetea las componentes de la fuerza que sufre cada cuerpo, en el segundo actualiza dichas componentes de fuerzas y en el tercero calcula las velocidades de cada uno en función de dichas fuerzas y actualiza la posición de los cuerpos en consecuencia.

Observando el código deducimos que el primer y el tercer bucle apenas tienen carga de trabajo comparados con el segundo bucle (el que se encarga de las fuerzas, que en realidad son dos bucles anidados). Si nos centramos en dicho bucle, vemos que además de algunas operaciones matemáticas, que podríamos considerar como básicas (y rápidas de ejecutar), se hace una llamada al método “mycos” y otra al método “mysin”, que junto con la función “atan” son las tres únicas cosas que no calificaríamos en primera instancia como operaciones matemáticas simples.

Llegados a este punto estamos casi seguros de que la mayor carga de trabajo viene por alguna de estas llamadas. Si las analizamos en profundidad, nos damos cuenta de que la función “atan” realiza el cálculo del arcotangente, pero esto es en realidad una simple división. Es por esto que nos quedamos con dos llamadas a métodos que serán las que en nuestra opinión le costarán más trabajo al programa.

Para comprobar todo lo dicha hasta ahora, hemos introducido cálculos de tiempos de ejecución (reutilizando el método de cálculo ya implementado en el archivo original). Los resultados de dichas medidas de tiempos son:

```
nbodies: 200    g var: 1000.00  interval: 0.10sec       nsteps: 10      min_dist
: 3.00p width 300       height: 300
nbody takes: 62.96sec    mean: 6.30sec    accuracy: 100
tiempo senos: 31.20243sec
tiempo cosenos: 31.37395sec
tiempo for i: 62.96292sec
tiempo for j: 62.96112sec
tiempo for actualiza: 0.00015sec
tiempo main: 62.97089sec
```

(Son consistentes para todos los archivos de entrada, pero solo se muestra para input1)

En esta captura podemos ver reflejado lo dicho hasta ahora:

- La duración del "calc_nbodies" (62.96 s) es la que se muestra como "nbody takes".
- El método main (62.97089 s) dura básicamente lo mismo que la ejecución del cálculo de posiciones (62.96 s). La diferencia entre la duración del main y la de "nbody takes" es lo que dura todo lo que no es bucle en el main.
- La duración del segundo bucle del "calc_nbodies" (el de los dos anidados, cálculos de fuerzas) del "calc_nbodies", sería la diferencia entre la duración de dicho método y el tiempo que se muestra como "tiempo for i" (62.96292 s).
- La duración del bucle de actualizar la posición (tercero del "calc_nbodies") es extremadamente pequeña (0.00015 s).
- La duración de los dos bucles anidados del "calc_nbodies" son 62.96292 s para el primero (exterior) y 69.96112 s para el interior.
- La duración de la llamada a "mysin" es de 31.20243 s,
- La duración de la llamada a "mycos" es de 31.37395 s.

Con esto, comprobamos que prácticamente todo el tiempo de ejecución del programa corresponde a las llamadas a "mysin" y mycos".

Si siguiésemos ahondando en el código veríamos que estos métodos hacen uso de método "mypow" y del método "factorial", siendo estas llamadas dentro de un bucle for. Dicho método factorial, por ejemplo, es recursivo (se llama a sí mismo), por lo que salvo que se implementase de otra forma (programación dinámica, por ejemplo) no se mejoraría el tiempo (dependencia de datos), y aun así no creemos que fuese demasiado beneficioso (sin demasiada mejoría).

Antes de meternos con eso, evaluamos la situación. Tenemos varios sitios en los que intentar paralelizar:

- En las llamadas a las funciones "mysin" y "mycos".
- En el / los bucles que contienen a dichas llamadas.
- En los métodos "mysin" y "mycos".

Esto es así porque entre el bucle interno y externo (del que contienen las llamadas a mysin y mycos) no hay dependencia de datos entre distintas iteraciones (cada una se refiere a un cuerpo, o bien del que se actualizan fuerzas o bien con el que se compara). Lo mismo sucede con distintas iteraciones de los bucles de "mysin" y de "mycos", y con los resultados de dichas funciones. Como ya hemos mencionado, llegar hasta el factorial no aportaría grandes beneficios (el cálculo

es el que es, y no se puede mejorar de forma significativa) y “mypow” no reduciría el tiempo de forma significativa.

Con estas posibles paralelizaciones sabemos, por la ley de Amdahl, que la máxima mejora que conseguiremos será reducir a la mitad el tiempo de ejecución de la parte paralelizable (toda la práctica, salvo el primer apartado opcional está realizada en las aulas indicadas por los profesores, cuyos ordenadores tienen dos cores).

En nuestro caso la parte paralelizable (viendo código y tiempos) es prácticamente toda, por lo que podemos bajar prácticamente a la mitad el tiempo (teóricamente) respecto a la versión sin paralelizar.

Pasamos por tanto a implementar estas soluciones para evaluar cuál sería la más conveniente (la evaluación inicial de todas ellas se ha realizado con 2 hilos).

(Nota: siempre que se haga referencia a bucle interno o externo en este documento, se refiere al bucle externo for[i] que contiene al bucle interno for [j] que contiene las llamadas de “mysin” y “mycos”, en el método “calc_nbodies” y en los que se realizan las actualizaciones de fuerzas)

Primera posible solución

Intentamos primero paralelizar el cálculo dentro de los métodos “mysin” y “mycos”. Para ello usamos un parallel for reduction, con acumulación de la variable result, en el bucle for de ambos métodos. Con esto estarían creándose tantos hilos como tuviésemos configurados, repartiendo las iteraciones de dichos bucles, para cada uno de ellos. (Esta creación de hilos y reparto sucede gran número de veces, quizá no es lo más adecuado. Sin contar que no todas las iteraciones de los bucles duran lo mismo, habría que pensar en usar planificadores)

Con esto obtenemos unos resultados de 47.31 segundos para el input1 sin errores.

Como sabemos que esto es mejorable, seguimos buscando alguna solución mejor.

Segunda posible solución.

La segunda solución que buscamos fue la que paralelizaba las llamadas a las funciones. Esto lo hicimos mediante el uso de las secciones de OpenMP. Para ello creamos justo antes de la llamada a cada función una sección que la contuviese. Dichas secciones contenían las llamadas a las funciones y la actualización del valor de la componente de fuerza correspondiente (fx para mycos y fy para mysin).

Esta solución, a pesar de ser muy buena (prácticamente la mejor) para dos núcleos (los laboratorios en los que se nos pedía realizar la práctica) tanto en tiempo (input1: 33s; input2: 25.95s input3: 41.04s) como en resultados (no variaban), no escalaba. Es decir no iba a mejorar más que eso, sin importar cuantos cores pudiésemos dedicar (asignamos “a dedo” el cálculo de una de las dos funciones a los cores).

Tercera posible solución

Es por eso que surge esta otra solución, cuyo objetivo es hacerlo más escalable. Consiste en combinar las secciones con la paralelización del bucle que contiene sus llamadas. Para ello se usan las secciones antes mencionadas y además se añade paralelización en el bucle externo (parallel for) que contiene sus llamadas. La paralelización del bucle requiere que se especifiquen las variables i, j,f, aux, fax, fay, alpha, dist como privadas, para que cada hilo no modifique estas variables afectando a todos los demás y a sus resultados.

Esto nos daba un tiempo bajo, con errores en la mitad de los decimales, de alguna de las columnas del 40% de las filas del archivo de resultados.

Pero como no se trataba de la solución que buscábamos (las secciones no eran la mejor implementación), seguimos intentando mejorar.

Cuarta posible solución

En este punto decidimos intentar paralelizar el bucle externo solamente, de la misma forma que se paralelizó en la solución anterior (con parallel for y variables privadas para protegerlas). Con esto estaríamos separando los cálculos para cada cuerpo: por ejemplo el hilo 1 calcula del cuerpo 0 al 99 y el hilo 2 del cuerpo 100 al 199. Esto puede presentar problemas a la hora del reparto de trabajo entre los hilos (en función del planificador), ya que las primeras iteraciones, los primeros cuerpos, tienen más carga de trabajo que los últimos (se calculan las fuerzas para los siguientes cuerpos en la lista, nunca con los anteriores, luego cuanto más cerca este del final de la lista, menos cálculos).

Los resultados con esta forma de paralelización fueron de 46.89s y fallos en el último decimal de alguna de las columnas de la mitad de filas (se incluye sección critica para la actualización del vector de fuerza, para evitar condiciones de carrera, esto no empeora el rendimiento una exageración, y es necesario).

Para intentar solventar esta descompensación de carga de trabajo entre hilos en lugar de hacer uso del planificador en este punto, se decidió paralelizar el bucle interno.

Solución definitiva

Para paralelizar el bucle interno se ha hecho uso de la misma instrucción de OMP usada en soluciones anteriores. Es básicamente lo mismo que en la cuarta solución posible, pero aplicado al bucle interno: uso de parallel for con variables privadas (j,f, aux, fax, fay, alpha, dist) y sección critica para las actualizaciones de fuerza.

De esta forma conseguimos equilibrar la carga de trabajo de cada hilo, ya que lo que se va a conseguir es que para cada cuerpo que fije el bucle externo, el número de cálculos que haya que hacer en el interno (sea el que sea, de hecho varía) se divida entre los hilos.

Con esto no notaremos demasiado el efecto de los planificadores (schedulers), quizá incluso empeore con ellos respecto a la solución anterior (por el número de veces que se crean hilos y se dividen tareas –con cada nueva iteración del bucle externo-, que también afecta al rendimiento).

Los resultados de esta solución son (para dos hilos) de 31.29s para el input1, 25.09s para el input2 y 39.26s para el input3.

Esto es bastante cercano a la mejora máxima posible en este ejercicio con dos núcleos (la mitad prácticamente). Por esto y por el relativamente bajo número de errores (ultimo decimal de las cifras) del input1, se decide coger esta solución. (Nota: tal y como se nos ha indicado nos hemos centrado en los errores el input1 a la hora de paralelizar, siendo los del input2 exageradamente grandes y los del input3 siendo fallos en las unidades).

Por tanto, lo que calculamos y dedujimos teóricamente sobre rendimiento y paralelización ha resultado corroborarse con los datos.

Evaluación de rendimiento

Ejercicios obligatorios

En este apartado se evalúa la solución seleccionada.

Para ellos nos valemos principalmente del uso de tres archivos de entrada diferentes:

- Input1: 200 cuerpos y 10 pasos de cálculo
- Input2: 50 cuerpos y 200 pasos de cálculo
- Input3: 100 cuerpos y 50 pasos de cálculo

Ejercicio 1

Ejecute el programa paralelo con 1, 2, 4, 8 y 16 threads para cada uno de los conjuntos de entrada. Represente los tiempos de forma gráfica y comente los resultados.

Resultados

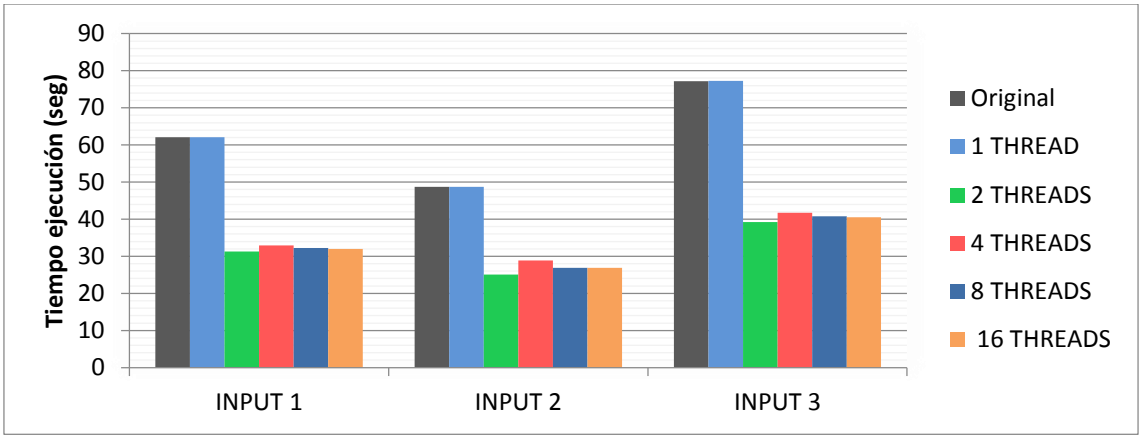
En este ejercicio se nos pide que tomemos tiempos con 1, 2, 4, 8 y 16 hilos, los representemos gráficamente y comentemos resultados.

Los distintos tiempos son:

	Original	1 THREAD	2 THREADS	4 THREADS	8 THREADS	16 THREADS
INPUT 1	62,07	62,09	31,29	32,93	32,21	31,95
INPUT 2	48,69	48,71	25,09	28,87	26,87	26,85
INPUT 3	77,21	77,25	39,26	41,72	40,74	40,51

(nota: se muestran tiempos en segundos)

Representados gráficamente:



¿Cuál es la evolución de rendimiento? ¿Por qué?

El tiempo original es desde el que partimos, que es la ejecución secuencial. El tiempo con un hilo debe ser prácticamente igual a la secuencial, ya que en el fondo se ejecutara usando solo un núcleo, sin paralelizar nada.

Con dos hilos sí que se intentará paralelizar, y dado que el ordenador sobre el que se ejecutan las pruebas tiene dos hilos, aquí sí que se ejecutan en paralelo realmente. Como la forma en que hemos paralelizado se reparte el trabajo de cada hilo bastante bien, y como la parte que es posible paralelizar conlleva la práctica totalidad de la carga de trabajo, este tiempo de ejecución del programa cae a casi la mitad.

Con más hilos lo que debería pasar es que subiese el tiempo de ejecución, ya que al crear, destruir y repartir hilos consume tiempo, y no es posible ejecutar realmente más de dos en paralelo.

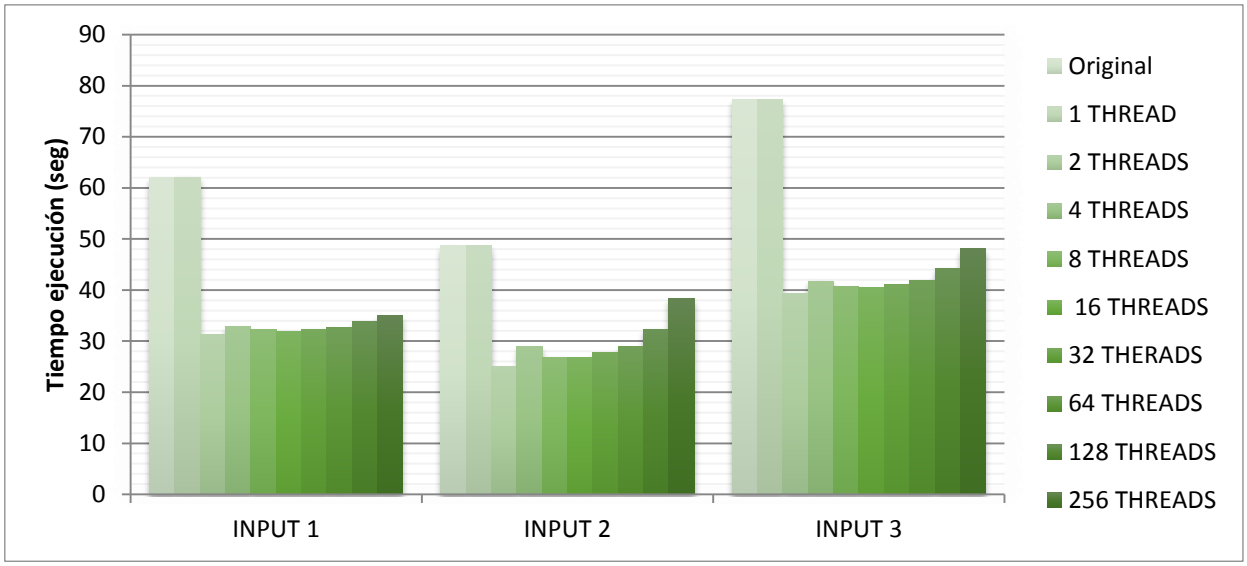
Sí que podría darse el caso de que con 4 u 8 hilos, por ejemplo (teniendo dos cores) mejorase el rendimiento. Esto sería porque podrían aprovecharse “huecos” en que no se está usando un core (por ejemplo al acceder un hilo a memoria) para ejecutar otro hilo.

Pero como hemos dicho, en general, a la larga (aumento de hilos) lo que sucede es que el tiempo de ejecución crece, ya que el tiempo “perdido” en crear, destruir, cambiar y repartir hilos no compensa al que se pudiese ganar con esos hilos creados. Además, llega un momento es que no es posible usar más tiempo los núcleos, porque ya están siendo usados casi el 100% del tiempo.

Recapitulando, en general, hasta llegar al número de cores baja el tiempo, al aumentar los hilos por encima del número de cores baja el rendimiento.

Para comprobar y demostrar esto hemos realizado una serie de medidas con más hilos, para así ver el comportamiento general, y cuál es la tendencia (muestra tiempos de ejecución en segundos):

	32 THERADS	64 THREADS	128 THREADS	256 THREADS
INPUT 1	32,36	32,58	33,76	34,96
INPUT 2	27,82	28,9	32,36	38,29
INPUT 3	40,98	41,92	44,31	48,23



Es por esto que nuestros datos nos resultan un poco extraños, concretamente la medida de 4 hilos. Es lógico hasta cierto punto que con 8 y 16 hilos el rendimiento se mantenga o incluso mejore algo por la paralelización de nuestro programa y lo ya explicado, pero ese pico de tiempo con 4 hilos no parece razonable (cuando los demás hilos mejoran esa medida, y el anterior mejora esa medida). Lo atribuimos por tanto a fallo en la medida de tiempos, ya que podría haber afectado algún factor externo o el estado de la máquina durante la realización de esas pruebas en concreto.

Como norma general diremos que debe descender el tiempo hasta un punto mínimo (generalmente cerca del número de cores del procesador, salvo que haya muchas detenciones y mucho tiempo la CPU parada), y a partir de ese momento subir, llegando incluso a empeorar el tiempo original. Esto sucede así en nuestra implementación, salvo por los valores atípicos con 4 núcleos ya mencionados.

Conjuntos de entrada

¿Qué diferencias hay entre los distintos conjuntos de entrada? ¿Cómo afectan estas diferencias al rendimiento?

Si nos fijamos en los tiempos vemos diferencias notables en los tiempos de ejecución del programa en función del archivo de entrada (en la magnitud, no así en el comportamiento).

Vamos a analizar por tanto a que se debe esto:

Los conjuntos de entrada tienen los siguientes datos:

- Número de cuerpos
- Gravedad
- Intervalos de tiempo entre pasos
- Numero de pasos
- Distancia mínima
- Ancho de región
- Alto de región

De estos parámetros, los que podrían afectar a la ejecución del programa son: número de cuerpos, gravedad y numero de pasos (además son los únicos que varían de un archivo a otro).

De modo que podemos aplicar el método científico para averiguar en qué medida depende el tiempo final de cada uno de esos parámetros.

Tomaremos medidas variando la gravedad, después variando el número de pasos, después el número de cuerpos y por ultimo comprobaremos lo deducido.

Se adjunta la tabla de medidas usadas para deducir como se ve afectado el rendimiento (para 2 hilos):

	Tiempo (s)	Steps	Gravedad	nª Bodys
Input 1	31,35	10	1000	200
Input 2	26,09	200	1000	40
Input 3	39,17	50	1200	100
Input 3	39,21	50	1000	100
Input 3	39,15	50	5000	100
Input 1	62,53	20	1000	200
Input 1	93,96	30	1000	200
Input 1	125	40	1000	200
Input 1	7,85	10	1000	100
Input 1	1,97	10	1000	50
Input 1	0,09	10	1000	10
Input 1	125,41	10	1000	400
Input 1	250,54	20	1000	400

En esta tabla se muestra:

- En las tres primeras filas, los parámetros originales para cada archivo de entrada y sus tiempos.
- En las dos siguientes filas se modifica solamente la gravedad, y se observa cómo cambia el tiempo de ejecución.
- En las tres siguientes se cambia el número de pasos, y se observa cómo cambia el tiempo de ejecución.
- En las cuatro siguientes filas se cambia el número de cuerpos, y se observa cómo cambia el tiempo de ejecución.
- En la última fila se combina la variación de pasos y cuerpos y se observa y comprueba cómo varia el tiempo de ejecución.

Nos damos cuenta de que:

- La gravedad no afecta al rendimiento.
- El tiempo crece de forma lineal con el número de pasos (a doble número de pasos doble tiempo).
- El número de cuerpos afecta de manera exponencial al tiempo (doble de cuerpos es cuádruple de tiempo, mitad de cuerpos es un cuarto de tiempo).

Se concluye por tanto que es la combinación de número de pasos (afecta lineal) y número de cuerpos (afecta exponencial) lo que hace más o menos rápido el cálculo. Basta con multiplicar la variación que tendrían cada uno de ellos por separado.

Esto se comprueba en la última fila de la tabla. Se combina el aumentar al doble el número de pasos (doble tiempo) y el aumentar al doble el número de cuerpos (cuádruple tiempo). Eso nos da que el tiempo debería ser (4*2) ocho veces superior, lo cual es correcto (250.54/31.35 = 7.991).

Ejercicio 2

Calcule el speedup obtenido en el apartado anterior para cada uno de los casos y represéntelo de forma gráfica. Comente los resultados

En este ejercicio se nos pide calcular el speedup para cada una de las medidas del ejercicio anterior.

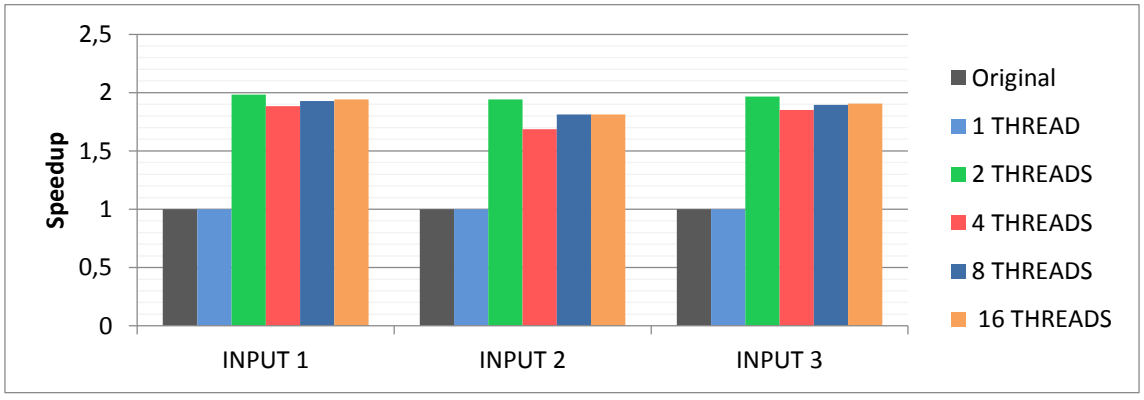
El speedup es básicamente la aceleración que se consigue sobre el código original. Se calcula dividiendo el tiempo original entre aquel obtenido que lo mejora. El resultado es adimensional y representa el factor de mejora que se ha conseguido. A mayor speedup, mayor mejora.

Volviendo a mencionar lo mismo que en el primer apartado de esta memoria, y según la ley de Amdahl, en este caso el speedup no será nunca mayor de dos.

La tabla con las medidas de speedup calculadas según la formula mencionada es:

	Original	1 THREAD	2 THREADS	4 THREADS	8 THREADS	16 THREADS
INPUT 1	1	0,999677887	1,983700863	1,884907379	1,927041292	1,942723005
INPUT 2	1	0,999589407	1,94061379	1,686525805	1,812058057	1,813407821
INPUT 3	1	0,999482201	1,966632705	1,850671141	1,895189003	1,905949148

El grafico con estas medias es:



Lo primero en que nos fijamos es que efectivamente no hay ningún speedup mayor (o igual) a dos. Vemos en estas medidas reflejado lo mismo que veíamos en los gráficos anteriores.

Vemos además que efectivamente con dos hilos se produce la mayor mejora, llegando a prácticamente el doble de velocidad al ejecutar. Vemos además los valores atípicos ya

mencionados del hilo 4. Si incluyésemos las medidas con más hilos, veríamos como cae el speedup (de la misma forma que subía el tiempo).

Esta medida de speedup es casi como ver las medidas de tiempo, podríamos decir que expresan lo mismo sobre cada caso pero de diferente forma.

Vemos además que cuando aumenta el número de cuerpos / pasos se pierde speedup.

Ejercicio 3

Tome tiempos con 4 threads variando el tipo de scheduling del bucle que más afecte al rendimiento del programa. Ejecute las pruebas con static, dynamic y guided para cada uno de los conjuntos de entrada. Represente los tiempos obtenidos de forma gráfica y comente los resultados.

En este caso se fija a 4 el número de hilos y se nos pide que variemos la planificación (scheduling), el planificador, del bucle que más afecte.

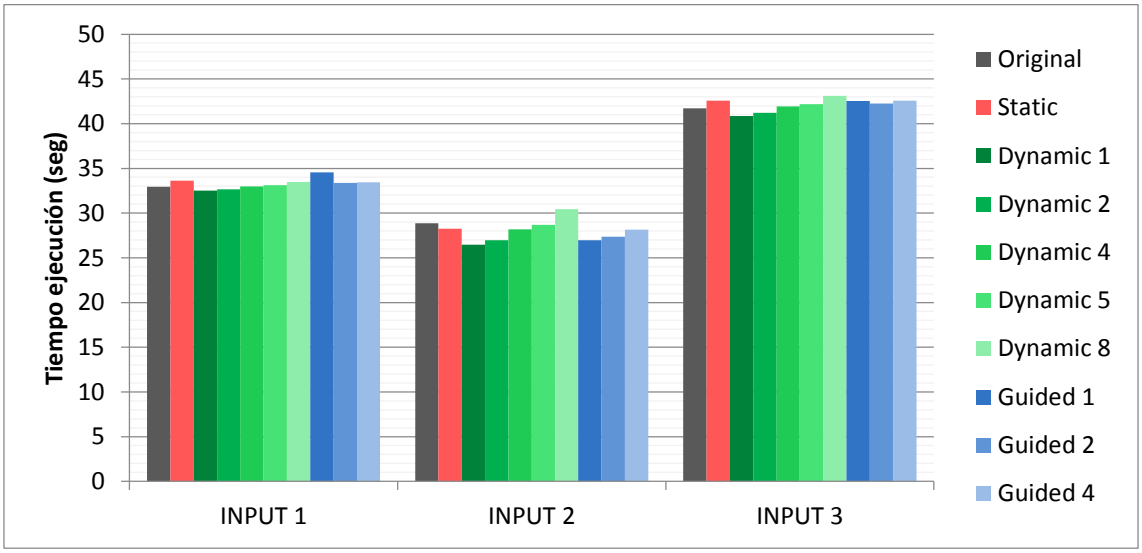
En nuestro caso se ha añadido al bucle que paralelizamos (el interior del método calc_nbodies).

Los datos obtenidos variando el planificador han sido los siguientes:

	Original	Static	Dynamic 1	Dynamic 2	Dynamic 4	Dynamic 5	Dynamic 8	Guided 1	Guided 2	Guided 4
INPUT 1	32,93	33,63	32,52	32,67	32,98	33,11	33,49	34,57	33,36	33,46
INPUT 2	28,87	28,27	26,46	26,96	28,19	28,67	30,43	26,98	27,34	28,13
INPUT 3	41,72	42,57	40,87	41,23	41,92	42,18	43,1	42,53	42,24	42,56

(nota: se muestran tiempos en segundos || los números para dynamic y guided representan el chunk)

Esto en forma de gráfica es:



¿Cuál es la evolución de rendimiento? ¿Por qué?

Antes de entrar a analizar los datos, diremos que con nuestra solución elegida el efecto positivo que pudiesen tener los planificadores y los distintos chunks pasa prácticamente desapercibido (se nota muy levemente), mientras que los efectos negativos o la dinámica general sí que se puede apreciar algo mejor (pero, una vez más, no de forma tan acusada como con otras posibles soluciones). Esto ya ha sido explicado en la parte correspondiente a esta solución en el principio del documento.

Esto es así porque los planificadores se usan para repartir el trabajo (iteraciones del bucle) de forma más o menos equitativa a los distintos hilos.

El original y el estático son similares (estático puede variar chunk, pero no lo hemos considerado relevante, ya que lo asigna de forma estática). Básicamente se coge el total de iteraciones y se divide entre los hilos, sin tener en cuenta que alguno reciba más trabajo que otro (por ejemplo con la cuarta solución, las primeras iteraciones tienen más carga de trabajo que las ultimas, por lo que si las primeras 50 por ejemplo van al hilo 1, tardarán más que las que se le asignen al hilo dos y muchísimo más que las que se asignen al hilo 3, por lo que habrá que quedarse esperando a que el primer hilo termine).

El dinámico divide el número total de iteraciones en grupos de tamaño igual al chunk. Después reparte los primeros 4 grupos a los 4 hilos. Al primero que termine le da el quinto, al siguiente le dará el sexto… etc. Sin importar cuál de ellos sea el que se lo pida. Si por ejemplo alguna iteración es exageradamente lenta, no se bloqueara todo el programa, si no que ese hilo se encargará de esa tarea hasta que termine, y los demás mientras pueden seguir recibiendo trabajo y realizándolo.

Cabe destacar sobre el chunk que si fuese tan grande como el número de iteraciones del bucle sería equivalente a un secuencial, si fuese la mitad sería equivalente a uno con dos hilos, y si fuese un cuarto es el estático para 4 hilos.

El guiado es como el dinámico salvo por que el tamaño de bloque que se asigna decrece con cada nueva asignación (mínimo el del chunk). Se divide número de iteraciones restantes entre número de hilos, y se asigna un bloque de esas iteraciones al que lo solicite.

Es decir, en función de la implementación de la solución el tiempo podría primero decrecer al aumentar el chunk, pero en todo caso va a tender a crecer a medida que crece el chunk, bien porque no se aprovechan todos los hilos a 100%, bien porque se hace un reparto que no es equitativo y se da más trabajo a unos que a otros.

Por el contrario si es demasiado bajo, podría darse el caso de que tantas divisiones y repartos mermasen el rendimiento.

Si analizamos los resultados, vemos que el planificador estático es bastante similar al tiempo original con 4 hilos. En ocasiones un poco mejor, y en ocasiones un poco peor. Es el por defecto, luego puede deberse a variaciones en las medidas o estado del computador en el que se realizaron.

En cuanto al dinámico y al guiado, hay que tener en cuenta que, como ya hemos mencionado (al tener nuestra solución repartido el trabajo de forma equitativa), no vamos a encontrar grandes diferencias.

Pero sí que se ve la suficiente variación para poder hacer una valoración. En general el guiado es peor que el dinámico porque divide primero un bloque muy grande, y lo va reduciendo paulatinamente hasta llegar al tamaño del chunk. Esto hace que el trabajo no se reparta de igual forma que con el dinámico (puede ser que el primer grupo, que es muy grande, sea además el que más tarde, por lo que no solo no gano, si no que pierdo).

En cualquier caso guiado y dinámico mejoran al estático con chunks bajos. Esto es porque se asigna de forma dinámica a los hilos según vayan terminando la ejecución de los bloques previamente asignados a ellos.

El motivo de que con chunks bajos sea más rápido (para nuestro ejercicio), ya se ha explicado, se reparte mejor el trabajo (y se menciona más en el siguiente ejercicio).

El motivo por el que con chunk de 1 es en ocasiones peor que el de dos también se ha dicho (la carga de trabajo de crear bloques y repartirlos hace contraproducente crear tantos bloques. Cada vez que termina una iteración un hilo, tengo que volver a repartir. Eso no siempre compensa).

¿Qué diferencias hay entre los distintos conjuntos de entrada? ¿Cómo afectan estas diferencias al rendimiento? ¿Qué planificador se comporta mejor en cada uno de los casos?

Las diferencias entre los distintos conjuntos de entrada ya han sido expuestas en el primer ejercicio obligatorio.

En cuanto a si esto afecta al rendimiento, analizando los datos arriba expuestos, no parece que sea significativo el cambio al variar los parámetros del archivo de entrada, y parece que en general se mantiene la misma tendencia, ya comentada, para todos los conjuntos.

Ejercicio 4

Averigüe y justifique razonadamente cuál es el tamaño de chunk óptimo para cada uno de los tres casos propuestos.

Dado el razonamiento del ejercicio anterior, y visto nuestro código, podemos concluir que: como con nuestra solución el número de iteraciones que hay que repartir cambia con cada iteración del bucle de fuera, definir un chunk alto no sería de ayuda, sería más bien contraproducente (si por ejemplo seleccionamos chunk 8, cada vez que se vaya a ejecutar el bucle de dentro con menos de 8 iteraciones, se hace de forma secuencial). Por ello sabemos que necesitamos un chunk bajo, pero ¿Cómo de bajo?

Pues la respuesta no creemos que sea trivial. Sabíamos que tenía que ser cercano a uno, lo más bajo posible, pero en cuanto al valor óptimo no hemos sabido asegurar antes de realizar las pruebas si tenía que ser de 1, de 2 o incluso de 4 (la considerábamos bastante improbable).

Digamos por tanto que esta entre uno y dos el chunk óptimo por los motivos ya expuestos (para dinámico y guiado, para el estático lo mejor es por defecto, esto es equitativamente entre los hilos).

Ejercicios opcionales

Ejercicio 1

Realice de nuevos los puntos 1 y 2 del anterior apartado en una máquina externa a las aulas de los laboratorios de informática. Se debe especificar el tipo de procesador utilizado y justificar las diferencias de rendimiento con respecto al original en caso de que las hubiera.

¿Cuál es la evolución de rendimiento? ¿Por qué? ¿Qué diferencias hay entre los distintos conjuntos de entrada? ¿Cómo afectan estas diferencias al rendimiento?

Para este ejercicio hemos repetido los ejercicios 1 y 2 obligatorios en una maquina con el siguiente procesador:

"AMD FX Series FX-8350 4.0Ghz 8X Black Edition"

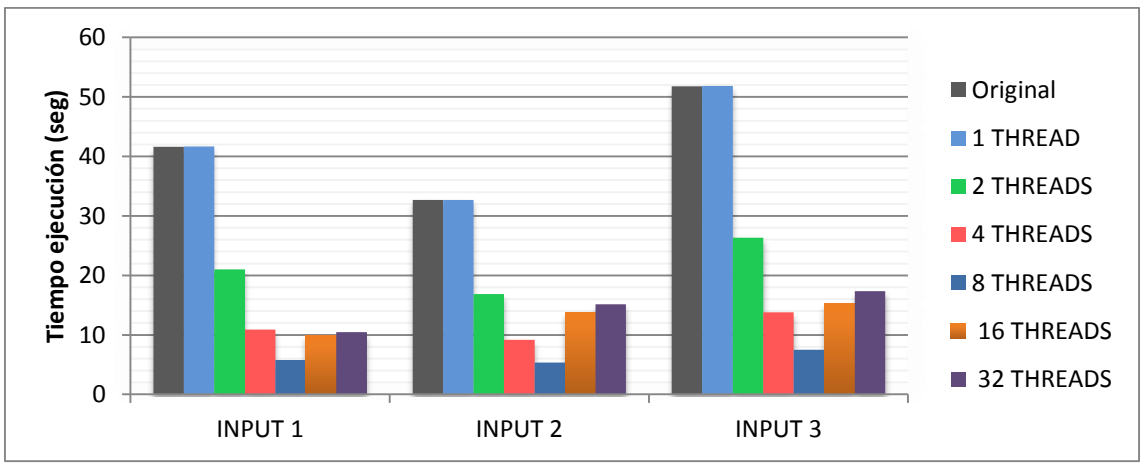
- Model number FX-8350
- Frequency 4000 MHz
- Turbo frequency 4200 MHz
- Package 938-pin micro-PGA package
- Socket Socket AM3+
- Architecture / Microarchitecture
- Platform Volan
- Processor core? Vishera
- Manufacturing process 0.032 micron
- Data width 64 bit
- The number of cores 8
- The number of threads 8
- Floating Point Unit Integrated
- Level 1 cache size
  - 4 x 64 KB shared instruction caches
  - 8 x 16 KB data caches
- Level 2 cache size 4 x 2 MB shared exclusive caches
- Level 3 cache size 8 MB shared cache
- Multiprocessing Uniprocessor
  - Features MMX
  - SSE
  - SSE2
  - SSE3
  - SSE4
  - SSE4a
  - AES instructions
  - Advanced Bit Manipulation
  - Advanced Vector Extensions
  - AMD64 technology
  - Virtualization technology
  - Enhanced Virus Protection
  - Turbo Core 2.0 technology
- Low power features PowerNow!
  - On-chip peripherals Dual-channel DDR3 memory controller
  - HyperTransport technology
- Electrical/Thermal parameters
  - Thermal Design Power 125 Watt

Tiempos

Las medidas de tiempo (en segundos) han sido las siguientes:

	Original	1 THREAD	2 THREADS	4 THREADS	8 THREADS	16 THREADS	32 THREADS
INPUT 1	41,64	41,65	21,02	10,91	5,77	9,8	10,48
INPUT 2	32,67	32,69	16,87	9,17	5,33	13,81	15,16
INPUT 3	51,81	51,82	26,32	13,82	7,5	15,2	17,35

Esto es forma gráfica es:



En este caso tenemos un procesador de 8 núcleos / 8 hilos. Por tanto el mejor rendimiento se alcanzará con 8 hilos teóricamente. Si atendemos a los datos vemos que esto se refleja fielmente. Además se aprecia con claridad lo comentado en el ejercicio uno de la parte obligatoria: se alcanza el mejor rendimiento con el número de hilos similar al número de cores, y a medida que nos alejamos de dicho número el rendimiento empeora.

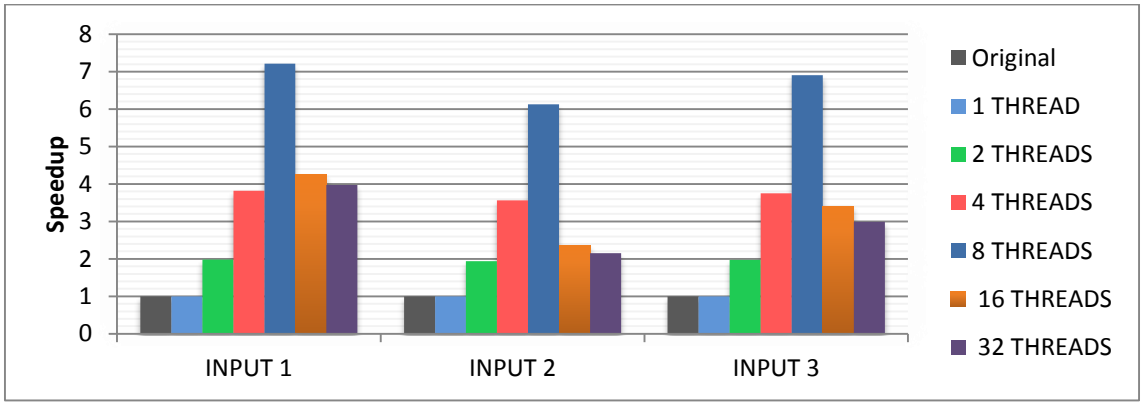
Esto es porque antes de llegar al número de cores, no se aprovecha todo el potencial de la máquina, y al pasarnos del número de cores es porque no es posible aprovechar la maquina más, y tanto crear hilos, destruirlos y cambiar entre hilos es contraproducente.

En cuanto a las diferencias en los conjuntos de entrada y a cómo afectan al rendimiento, se comporta todo como en el ejercicio 1 de los obligatorios.

Speedup

Si pasamos ahora a analizar el factor de mejora:

	Original	1 THREAD	2 THREADS	4 THREADS	8 THREADS	16 THREADS	32 THREADS
INPUT 1	1	0,999759904	1,980970504	3,816681943	7,216637782	4,248979592	3,973282443
INPUT 2	1	0,999388192	1,9365738	3,562704471	6,12945591	2,365677046	2,155013193
INPUT 3	1	0,999807024	1,968465046	3,748914616	6,908	3,408552632	2,986167147



Vemos ahora cuanto ha mejorado el rendimiento del programa en función del número de hilos. Una vez más es como mirar a la gráfica de tiempos. Donde allí baja el tiempo, aquí sube el speedup.

Vemos que al tener 8 cores, hasta que llegamos a esta cifra bajamos el tiempo, pero solo en el caso de 2 hilos es cuando tenemos un speedup prácticamente igual al límite teórico. Con 4 u 8 cores no se llega a al límite teórico. Esto puede deberse a la arquitectura o a la paralelización del programa.

Al igual que en los ordenadores de la universidad, con el conjunto de entrada 2 y 3 se pierde speedup (en general a mayor número de pasos y cuerpos).

Ejercicio 2

¿Por qué en algunas ocasiones los resultados de salida de la aplicación varían entre ejecuciones consecutivas con más de un hilo sin modificar los parámetros de ejecución?

Porque al estar trabajando con números en coma flotante tenemos un error asociado, y al realizar varios cálculos el error se propaga.

Este error del que hablamos se produce porque los números en coma flotante tienen un número de dígitos limitado para representar números reales infinitos, por lo que cuando un número no se puede representar con los dígitos disponibles se redondea al más cercano.

Esto ocurre en los siguientes casos:

- Denominadores grandes
- Dígitos periódicos
- Números no racionales
- Decimales
- Especialmente en números cercanos a 0

Estos "redondeos" de los que hablamos son los que provocan que los resultados obtenidos al paralelizar tengan errores, ya que por culpa de dichos redondeos, las operaciones con números en coma flotante no tienen las propiedades asociativas o distributivas.

Al paralelizar la ejecución en varios hilos, estamos haciendo que este orden de operaciones no sea el mismo (no se controla el orden en que se ejecuta). Al no tener el mismo orden en los cálculos, los resultados variarán.

Al paralelizar una sección del código con varios hilos estamos añadiendo errores que afectaran al resultado. El error no es siempre el mismo porque no siempre se ejecuta en el mismo orden, por lo que al ejecutar lo mismo varias veces seguidas obtenemos variaciones en los resultados.

¿Por qué esta variación tiene más impacto cuanto mayor es el número de iteraciones?

Porque con cada operación estamos arrastrando y extendiendo errores. Cuantas más iteraciones tengamos, el error de los resultados será mayor. Los errores se propagan y cuantas más iteraciones más notables se hacen, de mayor magnitud.

¿Por qué sólo ocurre con la versión paralela y no con la versión secuencial?

Porque la versión secuencial al ejecutarse seguida no añade el error que se comete al paralelizar una sección (es decir, si siempre lo hago en el mismo orden, a pesar de que pueda cometer errores de aproximación, siempre cometeré los mismos). La variación de los resultados se debe al error que añade cada hilo al paralelizarlo por el orden de las operaciones.

Conclusiones

Esta práctica en general ha resultado bastante frustrante de realizar, quizá porque al principio no estaban demasiado asimilados los conocimientos o quizá porque no teníamos muy claro que hacer, hasta donde llegar y que era lo permitido.

Es decir, hace un par de semanas llegamos a la solución de paralelizar el bucle exterior del calc_nbodies (después de estudiar el código,…). Pero tenía errores asumibles en el imput1, y auténticas burradas en el 2 y en el 3.

Esto supusimos que estaba mal.

Después conseguimos la solución de las secciones, pero resulto no ser válida por no poder escalar, así que volvimos a la solución anterior e intentar desarrollarla o hallar más soluciones (a todo esto, con esfuerzo y horas invertidas entre solución y solución). Y ahora con esta solución que hemos elegido, bastante parecida a la original pero dos semanas después, da la sensación de haber sido en parte una pérdida de tiempo y comedura de cabeza que quizá no era necesaria.

Como aspecto positivo decir que ha servido para asentar conocimientos de teoría, para dejar más clara la sincronización y todo lo relacionado con ella, incluidos los errores de aproximación.