MEMORIA PRÁCTICA 2 CPA

ÍNDICE:

EJERCICIO 1 --- Página 2

EJERCICIO 2 --- Página 3

EJERCICIO 3 --- Página 10

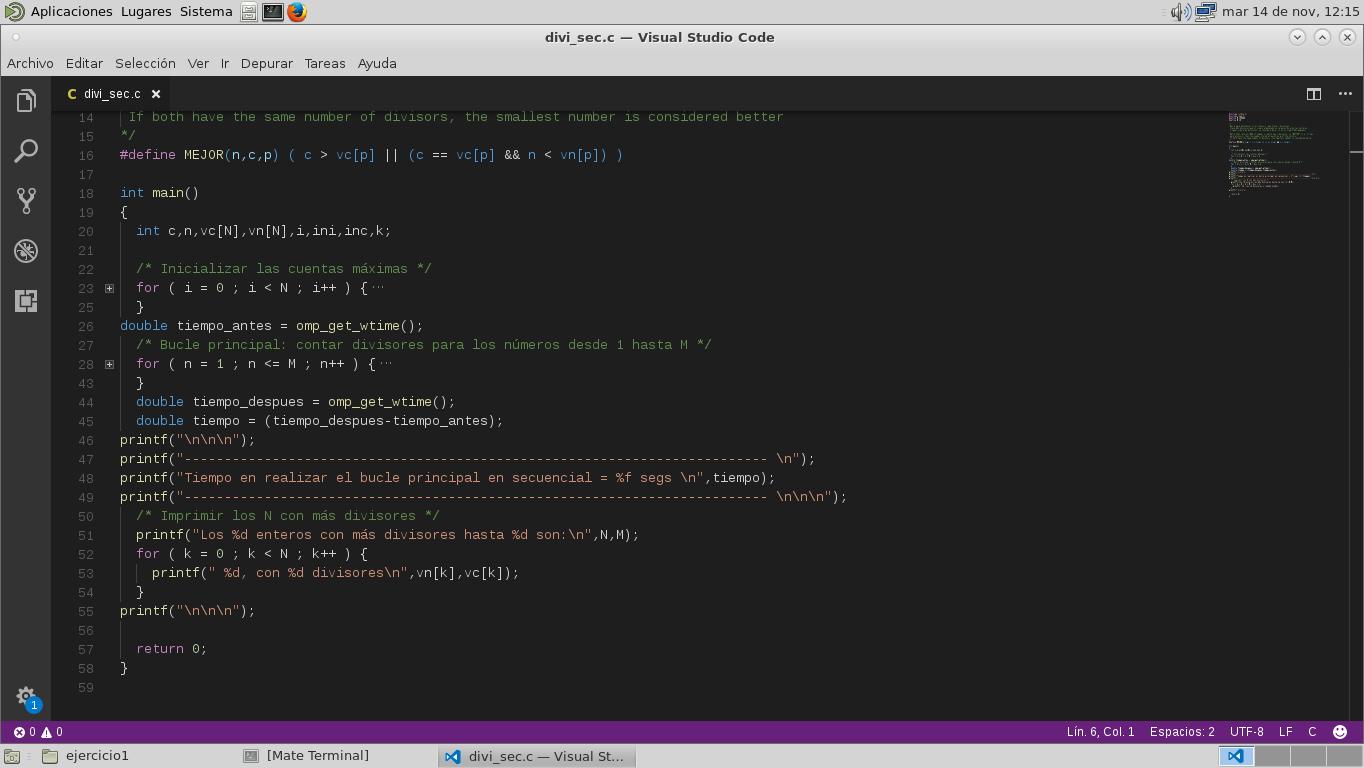
EJERCICIO 4 --- Página 11

ANEXOS --- Página 13

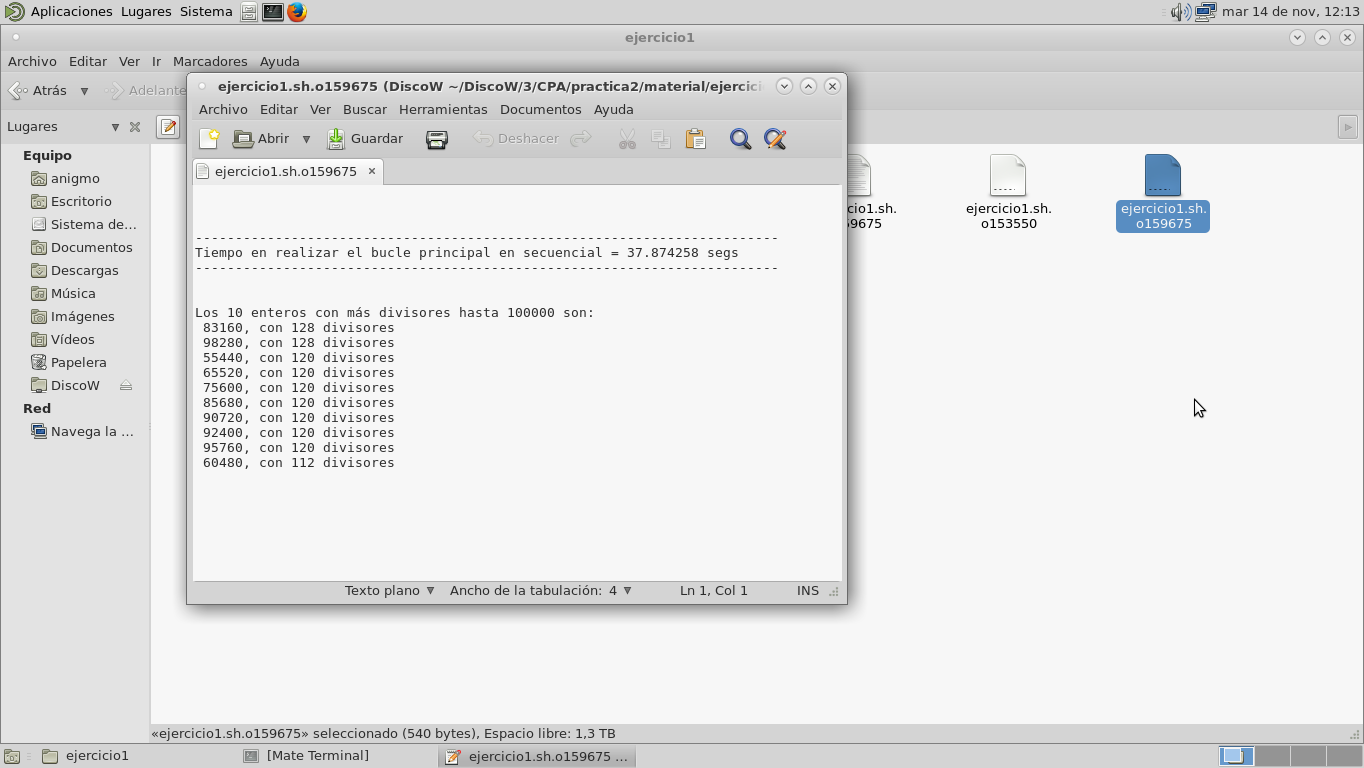
EJERCICIO 1

La función a utilizar para calcular el tiempo será omp\_get\_wtime(), tomaremos la medida del tiempo antes e inmediatamente después del bucle principal, asignando cada valor a una variable, para finalmente obtener los segundos que han transcurrido restando el tiempo final por el tiempo inicial.

Tras esto realizaremos un printf para mostrar el resultado (de tipo float)



En este caso obtenemos como resultado :



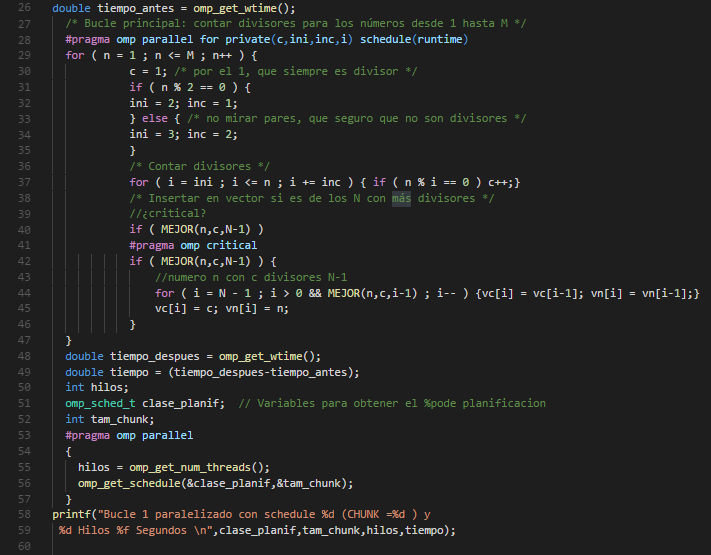
EJERCICIO 2

Para comenzar debemos señalar que es posible paralelizar los dos primeros bucles,

pero no el último, porqué como podemos ver, su condición depende de una condición booleana ( for ( i = N - 1 ; i > 0 && MEJOR(n,c,i-1) ; i-- )) que no puede ser conocida a priori.

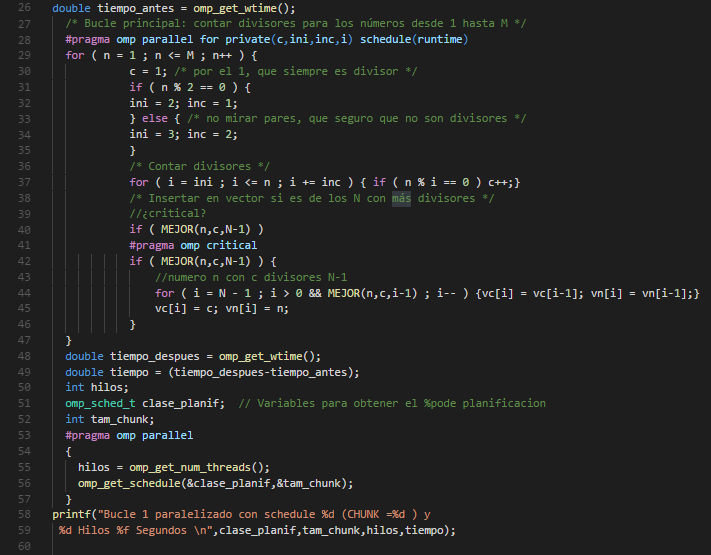
Paralelización Bucle 1

Comenzando por la paralelización del bucle más externo, observamos que cada hilo tendrá su propia variable c,ini e inc y que cada hilo hará completos los bucles i de su iteración n, por lo que estas 4 variables (c,ini,inc,i) deberán ser privadas.



Si ponemos atención en la instrucción if tras el segundo bucle (el primer bucle i), observamos que en caso de que nuestro número tenga más divisores que el ultimo de los mejores por el momento ( o tenga igual numero de divisores siendo menor), realizaremos el tercer y último bucle for para encontrar su posición adecuada, modificando el array de números y el de divisores.Estos arrays son globales y podría darse el caso de que varios hilos los actualizarán a la vez obteniendo un resultado erróneo, por esto, establecemos una sección crítica(con #pragma omp critical ), y para mejorar las prestaciones,

Duplicamos el if, colocando el critical tras el primer if, de forma que sólo entrarán a la sección crítica aquellos que hayan cumplido la condición y los demás continuarán sin detenerse.



Teniendo cubierto el aspecto de la paralelización, cabe señalar, que para ejecutar esta versión cambiando el tipo de planificación (Schedule) en tiempo de ejecución, colocaremos en #pragma omp parallel for lo siguiente schedule(runtime).Sin esto, al ejecutar el programa, aún ejecutando el cambio de planificación, este no surtiría efecto.

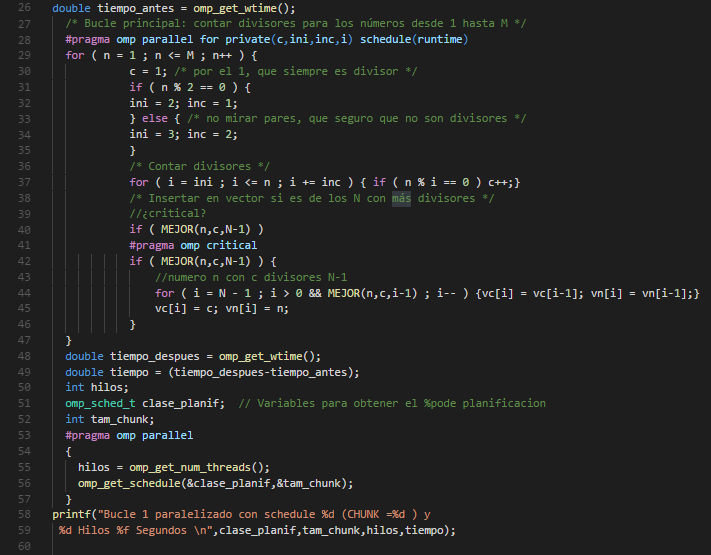
Otra opción sería fijar la planificación en el #pragma omp parallel for , por ejemplo #pragma omp for schedule(dynamic).

Para mostrar los resultados ( tiempo y número de hilos), abriremos otra sección paralela(# pragma omp parallel , para obtener el tiempo seguiremos el mismo procedimiento que en el ejercicio anterior y para obtener el número de hilos, usaremos omp\_get\_num\_threads() ( siempre en la región paralela).Cerraremos la región paralela y realizaremos el printf fuera de esta para que solo lo ejecute un hilo.

Como añadido, hemos añadido dos variables, una de tipo omp\_sched\_t y otra de tipo int, su función, es servir de contenedor del método que es llamado en la región paralela para obtener el tipo de planificación y chunk.

En la región paralela usamos el método omp\_get\_schedule(&clase\_planif,&tam\_chunk) para almacenar el tipo de Schedule en nuestra variable clase\_planif( de tipo omp\_sched\_t) y el chunk en nuestra variable tam\_chunk (de tipo int).

En nuestro caso también imprimimos el tipo de planificación y chunk en los resultados.

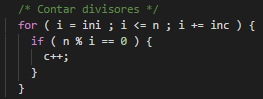


Cabe destacar que si ejecutamos un ejecutable x indicando el número de hilos y la planificación y en el código no aparece Schedule(runtime), el método omp\_get\_schedule nos devolverá la planificación y el chunk indicados pero NO serán aplicados, se aplicará la planificación por defecto de la máquina en la que trabajemos (cada máquina tiene la suya , en Kahan es dynamic) .

\*\*\*\*\*El código completo está en el anexo 1\*\*\*\*\*

Paralelización Bucle 2

El bucle 2 tiene este aspecto:



Podemos ver que cada hilo hará una iteración, cada hilo tendrá su propia i (la i(variable del bucle) es privada por defecto) y que todos (cuando se cumpla la condición del if) aumentarán en 1 el valor de c. Nosotros hemos optado por una reducción a la suma (reduction (+:c)) en este caso, pero también se podría usar la clausula atomic.



De nuevo, añadimos Schedule(runtime) porque nuestro propósito es cambiar la planificación en tiempo de ejecución.

La impresión de resultados y el cálculo de tiempo es igual que en el apartado anterior

\*\*\*\*\*El código completo está en el anexo 2\*\*\*\*\*

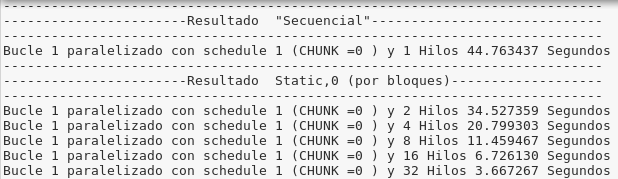
Resultados Paralelización

A continuación, se expondrán los resultados obtenidos:

En la exposiciones, vemos que aparecen tiempos “secuenciales” que se toman del programa paralelo ejecutado con un hilo, que como vemos distan mucho del tiempo secuencial real calculado en el ejercicio 1

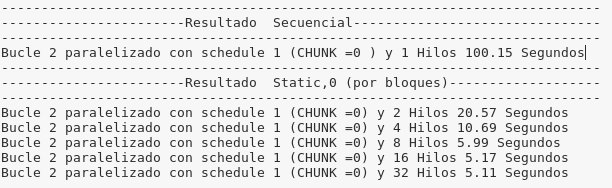
Comenzaremos con los resultados de las paralelizaciones de ambos bucles con la planificación static,0 (estática por bloques) en esta planificación los bloques se reparten a priori en misma cantidad (aproximadamente) para cada hilo, y es posible que a un hilo le toque mucho más trabajo que a los otros, retrasando el final.

Resultados del bucle 1 (estática por bloques):



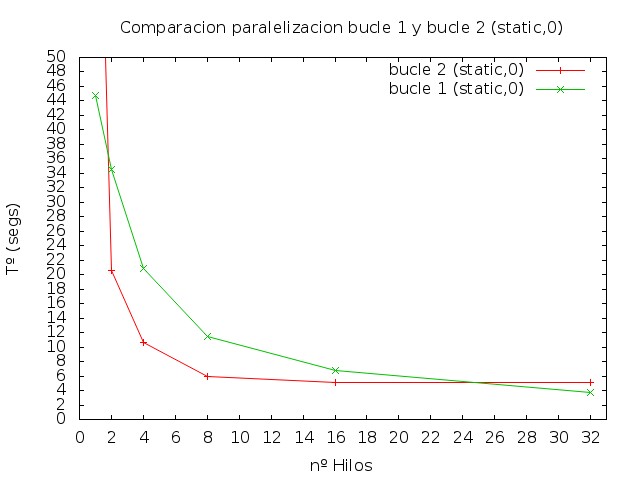
Se puede ver que la mejora no llega a ser la ideal, pues en ningún caso doblando el número de hilos tardamos la mitad de tiempo, esto puede deberse a que algún hilo obtiene más trabajo en esta planificación y mientras los demás están ociosos

Resultados del bucle 2 (estática por bloques):



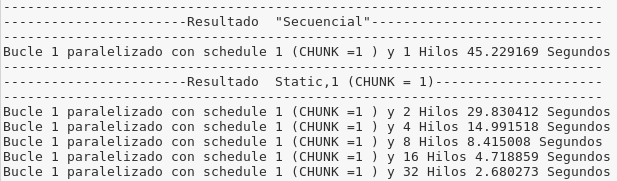
Como vemos, la paralelización del bucle 2 estática por bloques es contra todo pronóstico mejor con 2,4,6, e incluso 16 hilos.

Lo tenemos representado en esta gráfica:



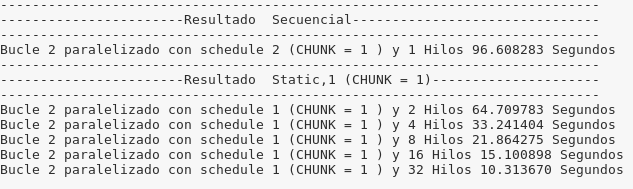
Continuaremos con los resultados de las paralelizaciones de ambos bucles con la planificación static,1 ( estática con chunk=1) en esta planificación los bloques se reparten a priori de uno en uno en misma cantidad (aproximadamente) para cada hilo, y es posible que a uno o varios hilos les toques más trabajo que a los otros, pero como mucho será una iteración más, lo que disminuye la penalización.

Resultados del bucle 1 (estática con chunk=1):



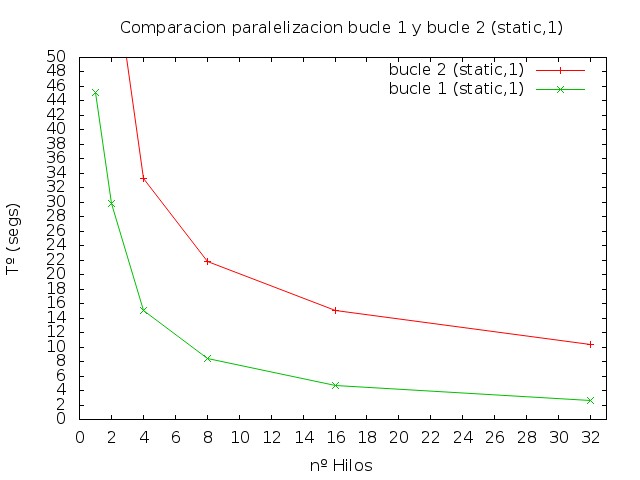
Podemos observar que todos los tiempos son mejores que con static,0 por lo mencionado.

Resultados del bucle 2 (estática con chunk=1):



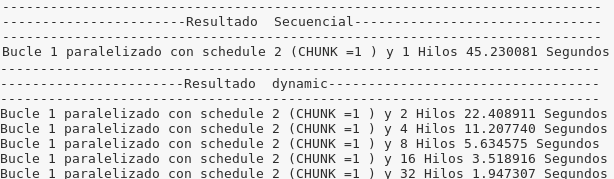
Como vemos, las planificaciones con pequeños chunk penalizan mucho más a la paralelización del bucle interno, consiguiendo un tiempo peor paralelizando este con 32 hilos que si paralelizaramos el primer bucle y usasemos 8 hilos

Lo tenemos representado en esta gráfica:



Por último analizaremos la paralelización de ambos bucles con la planificación dynamic. Esta planificación reparte las iteraciones cuando los hilos acaban su trabajo actual, asique los hilos más rápidos tendrán más trabajo y se repartirá mejor la carga entre todos, así la penalización será mínima.

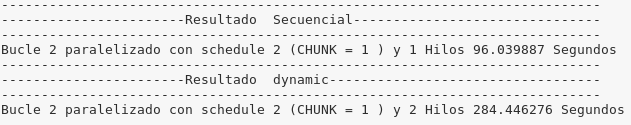
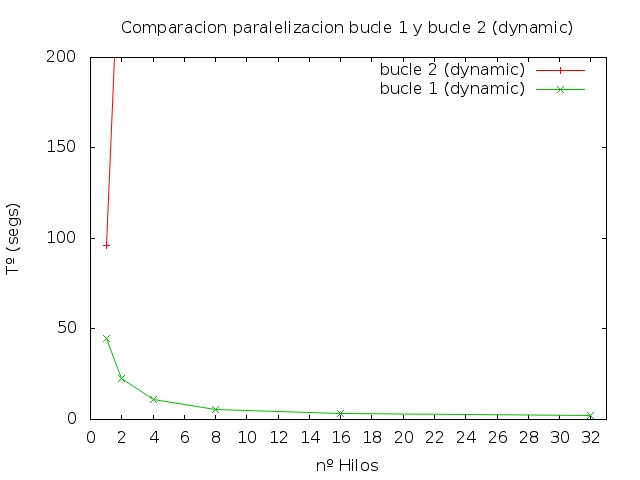
Resultados del bucle 1 (dynamic):



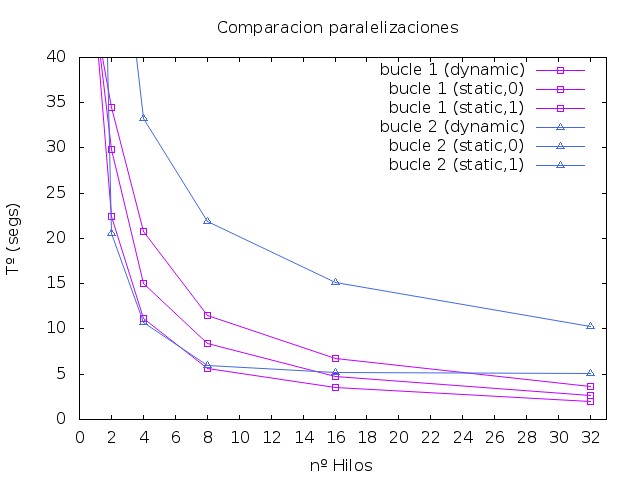
Como observamos, la penalización es mínima y obtenemos los mejores tiempos.

Hemos pasado de 37 segundos en secuencial “real” a 45,2 en “secuencial” con un hilo y 1,94 segundos en paralelo con 32 hilos

Resultados del bucle 2 (dynamic):

La sobre carga es tan grande en la paralelización del bucle 2 que el proceso sufre un kill a partir de usar 2 hilos.

Como conclusión tenemos que la mejor opción es paralelizar el bucle 1, que la planificación que da mejores prestaciones es la dynamic y que no es necesario cambiar la planificación por defecto pues es Dynamic. Cabe destacar que en el único caso en el que una paralelización del bucle 2 sería mejor que la del bucle 1 sería en static 0, y usando entre 2 y 16 hilos(en este caso, la Dynamic del bucle 1 sería muy similar o mejor (con 16 hilos).



\*\*\*\*\*Gráficas b1 y b2 en el anexo 5 y 6\*\*\*\*\*

Calculo de prestaciones:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PRESTACIONES STATIC 0 ( ESTÁTICA POR BLOQUES) | | | | |
| Nº hilos | PARALELIZACIÓN BUCLE 1 | | PARALELIZACIÓN BUCLE 2 | |
|  | Speed-up | Eficiencia | Speed-up | Eficiencia |
| 2 |  |  |  |  |
| 4 |  |  |  |  |
| 8 |  |  |  |  |
| 16 |  |  |  |  |
| 32 |  |  |  |  |

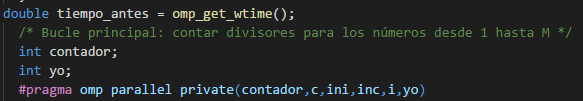
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PRESTACIONES STATIC,1 ( ESTÁTICA CHUNK=1) | | | | |
| Nº hilos | PARALELIZACIÓN BUCLE 1 | | PARALELIZACIÓN BUCLE 2 | |
|  | Speed-up | Eficiencia | Speed-up | Eficiencia |
| 2 |  |  |  |  |
| 4 |  |  |  |  |
| 8 |  |  |  |  |
| 16 |  |  |  |  |
| 32 |  |  |  |  |

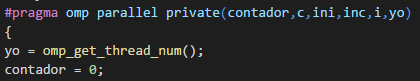
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PRESTACIONES DYNAMIC ( DINÁMICA) | | | | |
| Nº hilos | PARALELIZACIÓN BUCLE 1 | | PARALELIZACIÓN BUCLE 2 | |
|  | Speed-up | Eficiencia | Speed-up | Eficiencia |
| 2 |  |  |  |  |
| 4 |  |  |  |  |
| 8 |  |  |  |  |
| 16 |  |  |  |  |
| 32 |  |  |  |  |

EJERCICIO 3

Debemos paralelizar en este caso el bucle 1, puesto que en él cada hilo tendrá un número y encontrará su número de divisores (almacenándolos en la variable c),nosotros sumaremos los divisores que ha encontrado de cada número en un contador y lo devolveremos.

Además de las 4 variables privadas que había, añadiremos dos, contador (int) y yo(int), crearemos las variables sin inicializarlas, crearemos en este caso una región paralela que envuelve más instrucciones que el for con #pragma parallel private(contador,c,ini,inc,i,yo) .

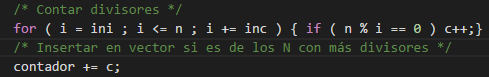
 Dentro de la región inicializamos yo al número de hilo actual (cada hilo tiene el suyo) con omp\_get\_thread\_num() y contador a 0.



Lo hacemos así para que cada hilo tenga su valor, dado que si los inicializamos fuera el hilo 0 tendria valor y los demás no.

Tras esto repetimos la paralelización del bucle externo con la salvedad de que será #pragma omp for , ya que ya está en una región paralela.

Dentro del bucle sólo hay un cambio y es añadir una instrucción tras el segundo for para cuando ya tengamos c, añadirlo a nuestro contador con contador+=c;



Tras el for colocaremos el printf con el número de hilos y contador.



\*\*\*\*\*El código completo está en el anexo 7\*\*\*\*\*

EJERCICIO 4

Dado que la mejor paralelización es la del bucle 1, vamos a paralelizar este de forma manual, la idea es repartir las iteraciones entre los hilos de forma eficiente, para esto, tendremos dos variables, nh y yo, nh será el número de hilos y yo será el número de hilo.Cada hilo comenzará haciendo la iteración de su número de hilo y incrementará esta en nh, es decir, n (variable del bucle) pasará de incrementarse en 1 a incrementarse en nh.

Por ejemplo; si tenemos 4 hilos, el hilo uno hará la iteración 1,5,9…

Para lograr esto, en primer lugar, creamos las dos variables a usar (ambas int)



Abrimos una región paralela en la que además de añadir las 4 variables privadas del bucle 1 añadiremos yo ( cada hilo tendra su identificador) pero no nh porque este debe ser global para que todos los hilos lo conozcan y puedan aumentar correctamente su iteración en el bucle.

Dentro de la región inicializamos nh con omp\_get\_num\_threads(); y yo con omp\_get\_thread\_num();

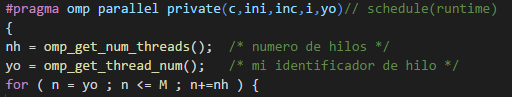
En esta ocasión, no utilizamos #pragma omp for porque lo hacemos manualmente.

En el for debemos hacer las siguientes modificaciones:

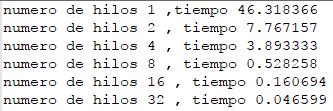
La variable del bucle se inicializa a n (cada hilo la pone a su número de hilo).

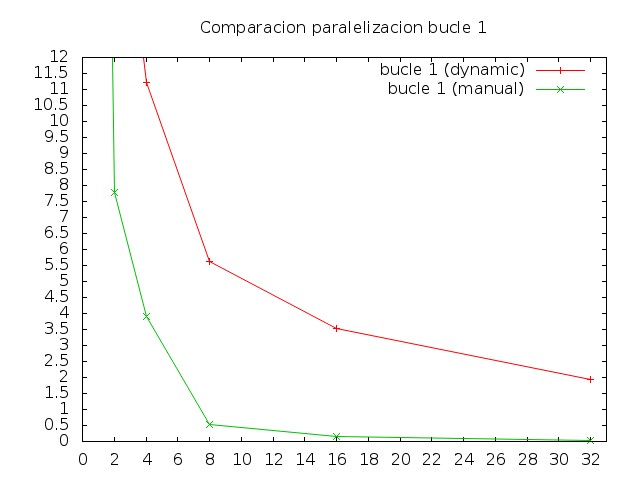
La variable n ya no se aumenta en 1 sino en nh (n+=nh).

El resto sigue igual que en el ejercicio2.



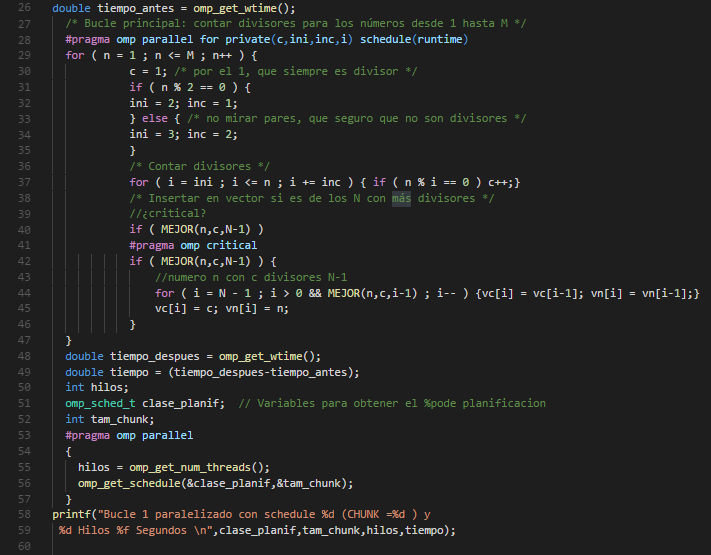
\*\*\*\*\*El código completo está en el anexo 8\*\*\*\*\*



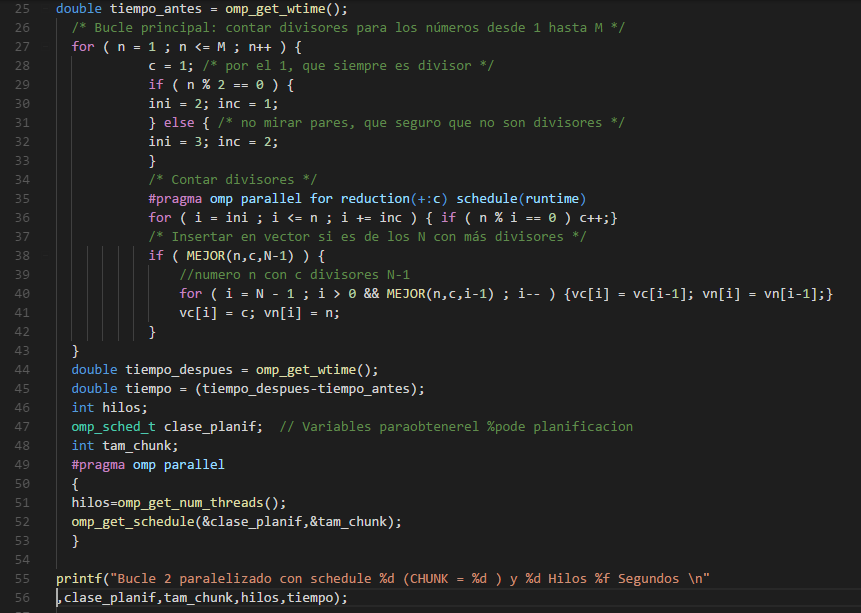


Como observamos la paralelización manual es mejor en todos los casos

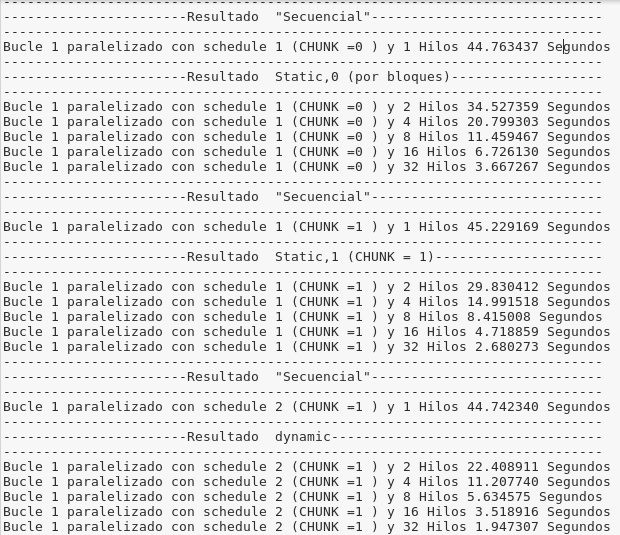
ANEXO 1 (CÓDIGO B1 EJERCICIO 2)



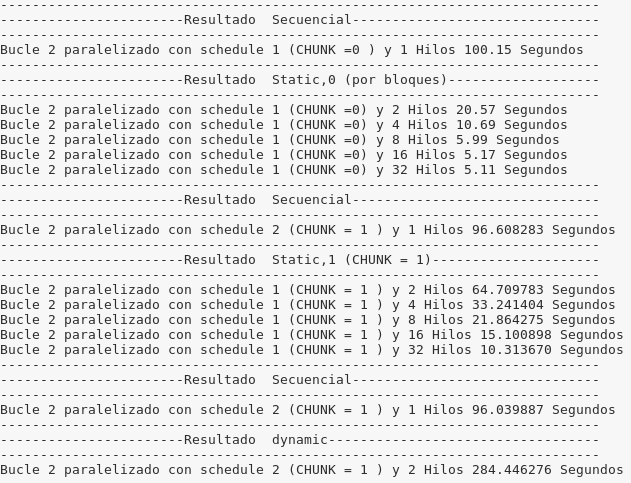
ANEXO 2 (CÓDIGO B2 EJERCICIO 2)



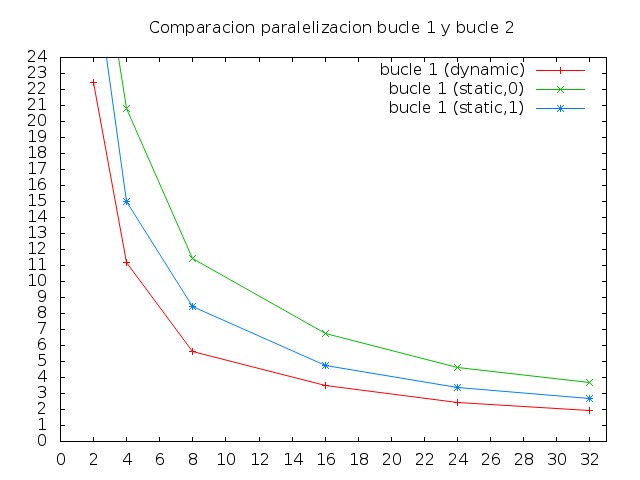
ANEXO 3 (RESULTADOS B1 EJERCICIO 2)



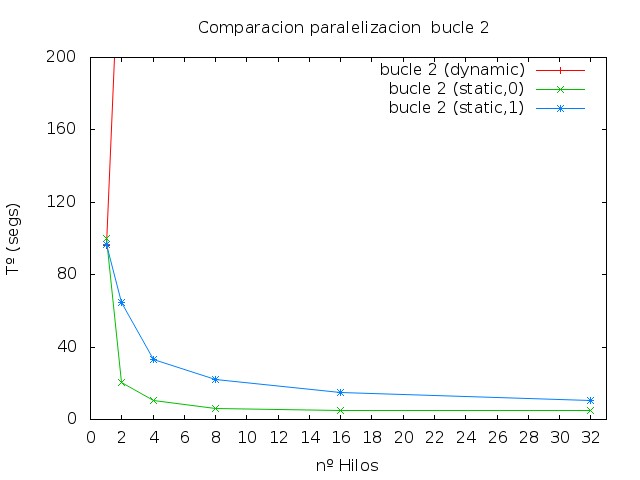
ANEXO 4 (RESULTADOS B2 EJERCICIO 2)

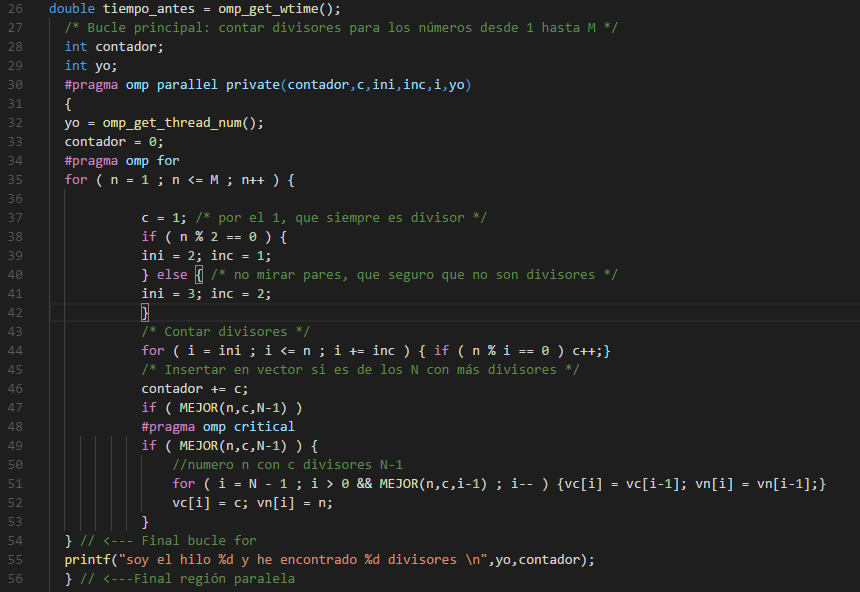


ANEXO 5 (GRÁFICA B1 EJERCICIO 2)



ANEXO 6 (GRÁFICA B2 EJERCICIO 2)

ANEXO 7 (CÓDIGO EJERCICIO 3)



ANEXO 8 (CÓDIGO EJERCICIO 4)

