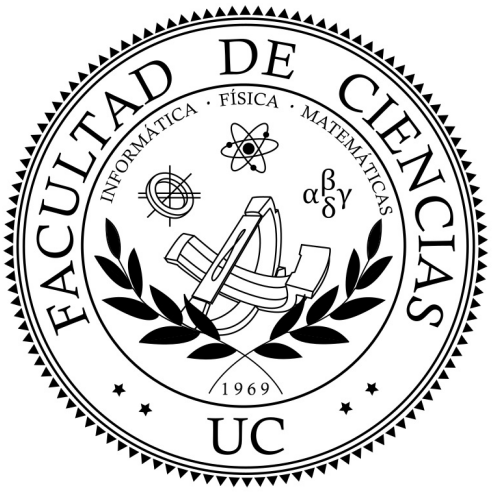

Universidad de Cantabria

Facultad de Ciencias

PROGRAMACIÓN PARALELA

MEMORIA DE LAS PRÁCTICAS

G653 - Programación Paralela, Concurrente y de Tiempo Real
Curso 2022-2023

PABLO GOITIA GONZÁLEZ

Índice

Información relevante sobre el equipo de pruebas

Lenovo IdeaPad L340 (17.ªMD)

PROCESADOR:
AMD Ryzen™ 5 3500U, de la familia AMD Ryzen™ 5 Mobile Processors
- N.° de núcleos de CPU: 4
- N.° de subprocesos: 8
- Reloj base: 2.1GHz
- Reloj de aumento máx: Hasta 3.7GHz
- Cache 384KB (L1); 2MB (L2); 4MB (L3)

SISTEMA OPERATIVO:
Ubuntu 20.04.5 LTS 64 bits

Notas acerca de los Scripts

Con el objetivo de automatizar y simplificar el procesamiento de los tiempos de ejecución de algunos de los programas de estas prácticas, se han desarrollado scripts en Python. Es importante tener en cuenta que no son flexibles en cuanto al formato en el que reciben los datos, por lo que en cualquier caso debe respetarse el formato solicitado en el apartado "Notes" de la cabecera de cada script. De lo contrario, el script finalizará con un error.

Práctica 1: Paralelización de algoritmos sencillos en C++

Ejercicio 1

Dibuja la gráfica del tiempo de ejecución usando entre 1 y 16 threads. Recuerda utilizar un tamaño de vector suficientemente grande como para justificar la ejecución paralela del algoritmo.

El algoritmo desarrollado, por simplicidad, sumará siempre un conjunto de N elementos con valor 1. Esto nos permite obtener tiempos algo más precisos y verificar que los resultados son correctos sin añadir complejidad al programa. El valor de N será de 100.000.000 enteros, que en una arquitectura de 64 bits equivale a 800MB de información, un tamaño que sin duda justifica el uso de mecanismos de paralelización.

Para obtener la gráfica, se ha desarrollado un algoritmo "graphics.py" en Python que toma los datos de la salida estándar del programa descartando los valores anómalos, los representa en una gráfica y los exporta los datos a un fichero de texto.

El resultado es el siguiente:

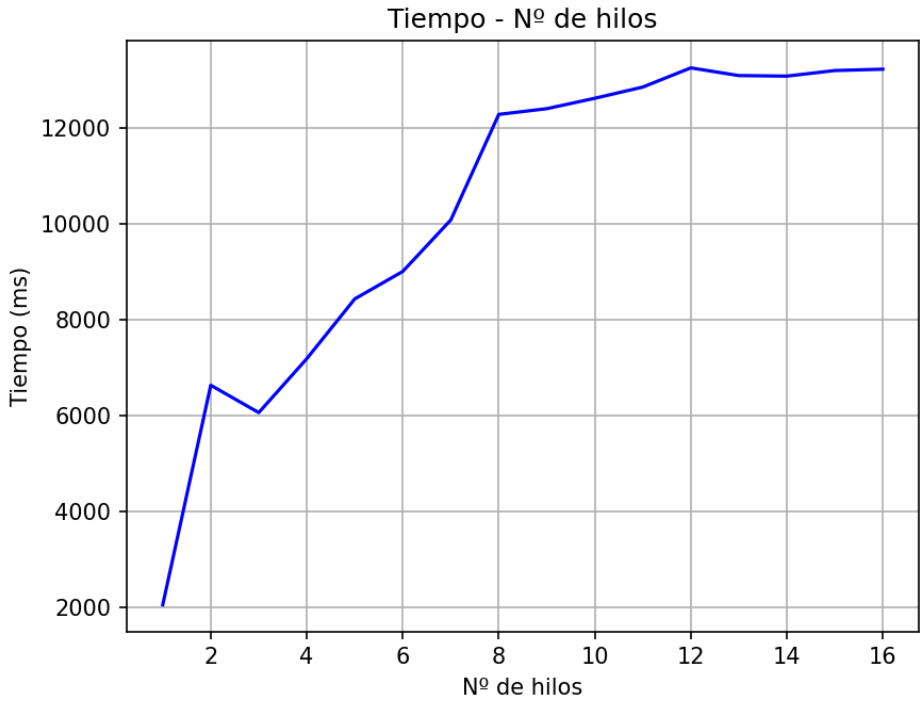

Figura 1: Representación de los tiempos de ejecución en la primera alternativa del programa "reduccion.c".

Los tiempos (ms) representados en la gráfica son:

T(1) = 2052.17	T(5) = 8431.25	T(9) = 12398.49	T(13) = 13088.98
T(2) = 6631.02	T(6) = 9004.25	T(10) = 12615.48	T(14) = 13077.36
T(3) = 6060.23	T(7) = 10076.48	T(11) = 12849.34	T(15) = 13193.8
T(4) = 7183.89	T(8) = 12279.3	T(12) = 13249.32	T(16) = 13222.22

Dibuja la gráfica del tiempo de ejecución usando entre 1 y 16 threads. Utiliza el mismo tamaño de vector que en el apartado anterior. ¿A qué se debe la diferencia de rendimiento entre ambas versiones? ¿Cuántos cores tiene el procesador en el que has ejecutado las pruebas? ¿Qué ocurre cuando utilizas un número de threads mayor que los cores disponibles?

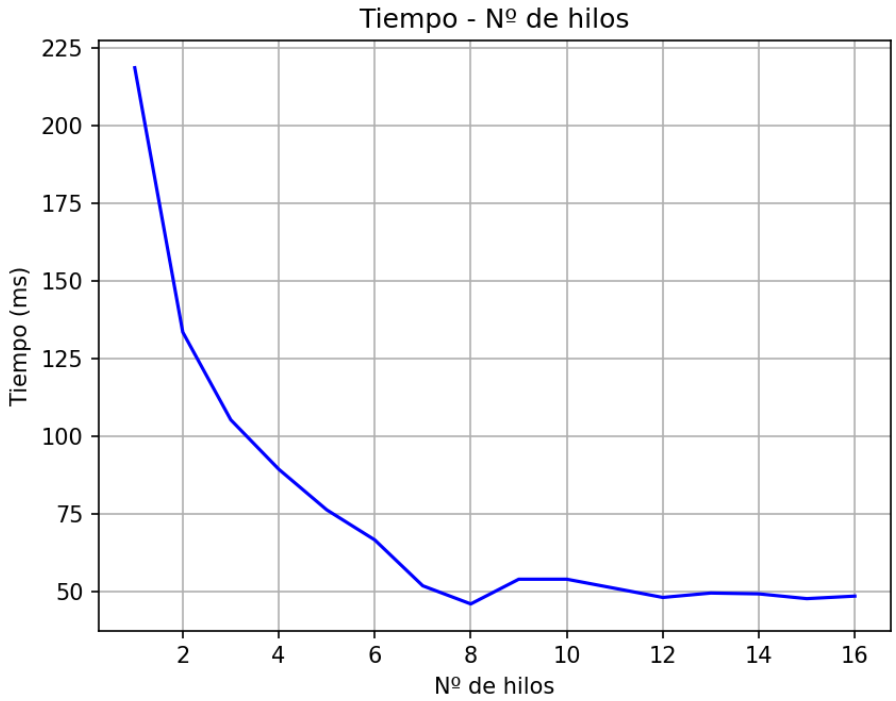

Figura 2: Representación de los tiempos de ejecución en la segunda alternativa del programa "reduccion.c".

Una vez hecha la modificación, nos queda la siguiente gráfica:
Los tiempos (ms) representados en la gráfica son:

T(1) = 218.74	T(5) = 76.22	T(9) = 53.86	T(13) = 49.41
T(2) = 133.51	T(6) = 66.53	T(10) = 53.86	T(14) = 49.15
T(3) = 105.26	T(7) = 51.74	T(11) = 50.93	T(15) = 47.60
T(4) = 89.32	T(8) = 45.90	T(12) = 47.99	T(16) = 48.42

El procesador del equipo de pruebas tiene 4 cores físicos y 8 hilos.

Los resultados obtenidos son los esperados para ambos casos. En el primer caso, cada vez que un hilo suma un elemento necesita pedir el mutex para garantizar la consistencia de la variable global que actualiza, dando lugar a largos tiempos de bloqueo mientras el mutex está tomado por otro hilo. En la gráfica (Figura 1) se observa una tendencia a la alza motivada por los tiempos de bloqueo crecientes que supone tener más hilos compitiendo por tomar el mutex.

Sin embargo, con la optimización realizada en la segunda alternativa, todo ese tiempo de espera desaparece, dado que todos los valores se suman en una variable local. La mayor parte del tiempo se invierte directamente en las operaciones, no como en el caso anterior. Cada hilo solo tendrá que pedir el mutex una vez para actualizar el resultado final. Ahora sí, obtenemos unos tiempos de ejecución con una tenedencia que corresponde más con la de un programa concurrente eficiente. Cuantos más hilos haya, mejor distribución del tiempo de cómputo se hace.

Cuando el número de threads supera al número de cores virtuales, en ambos casos, los valores de los tiempos se estabilizan debido a que ya hemos hecho el mejor reparto de la carga posible. Utilizando más threads de los que el procesador permite desestabiliza en pequeña medida ese reparto y añade cierto overhead debido a que el sistema operativo debe de intervenir para hacer los cambios de contexto correspondientes según la planificación.

Ejercicio 2

Dibuja la gráfica del tiempo de ejecución usando entre 1 y 16 threads. Recuerda utilizar un tamaño de vector suficientemente grande como para justificar la ejecución

paralela del algoritmo.

De igual manera que en el programa anterior anterior, todos los elementos adquirirán valor 1. En total, trabajaremos con un vector V de tamaño 100.000.000 elementos que ocupan 800MB, valor que se duplicaría al generar el vector Resultado.
Para este ejercicio también se ha desarrollado un script en Python que recoge los resultados de "rap.c" y los procesa para representarlos en una gráfica y guardarlos en un fichero de texto. El resultado es el siguiente:

Figura 3: Representación de los tiempos de ejecución del programa "rap.c".

Los tiempos (ms) representados en la gráfica son:

T(1) = 399.47	T(5) = 435.63	T(9) = 645.94	T(13) = 865.29
T(2) = 333.90	T(6) = 496.83	T(10) = 700.68	T(14) = 934.41
T(3) = 384.50	T(7) = 539.46	T(11) = 762.64	T(15) = 996.18
T(4) = 409.24	T(8) = 590.19	T(12) = 808.10	T(16) = 1050.44

¿Te llama la atención algo en los tiempos de ejecución? ¿A qué se debe? ¿Cómo podríamos resolverlo?

El comportamiento que puede observarse en la gráfica nos indica que, tal y como está implementado el programa, usar muchos hilos no es la opción más eficiente. Un único hilo resuelve el problema en unos 399.47ms. Al distribuir la carga entre 2 hilos, el tiempo baja hasta los 333.90ms. Esto quiere decir que ejecutar el programa con 2 hilos parece lo más apropiado para obtener los resultados más rápido. El problema es la tendencia a la alza a partir de los 2 threads. Para este caso particular, vemos que aproximadamente a partir de los 4 hilos (entre 3 y 4 para ser más precisos) ya sería más eficiente ejecutar el código de forma secuencial. Los tiempos de ejecución se disparan a partir de ese valor.

Se trata de un comportamiento que al fin y al cabo esperábamos, debido a que por cada uno de los hilos recorremos todo el vector sumando sus valores hasta la posición inicial del bloque que se le ha asignado. Llega un momento en el que se están realizando operaciones de más y se empiezan a sufrir las consecuencias.

Considero que la única solución para hacer más eficiente este algoritmo es reformularlo. Está mal planteado de base, y de por sí ya es muy ineficiente. Lo más recomendable sería buscar otra

técnica algorítmica que nos ofrezca un rendimiento más uniforme.

Ejercicio 3

Describe el trabajo que OpenMP habría hecho por tí de haberlo utilizado, en lugar de haber trabajado con una implementación manual utilizando threads. ¿Cuáles son las ventajas de OpenMP?

El trabajo de paralelización en estos programas ha consistido en distribuir la carga de trabajo de forma uniforme entre todos los threads que hemos creado, y en proteger con semáforos todas las variables compartidas si las hubiera. Este trabajo que hemos realizado manualmente lo habría hecho OpenMP por nosotros utilizando las directivas apropiadas. Distribuir la carga de trabajo no es una tarea trivial, y cómo hacerlo depende de algunos factores como la uniformidad de los tiempos de cómputo por cada iteración. Dependiendo de si esta carga es más o menos constante entre las iteraciones, tendremos que distribuir los bloques de iteraciones de una manera u otra. Un equivalente a lo que hemos hecho con las posibilidades que nos ofrece OpenMP sería la directiva *for* con el algoritmo de distribución de carga "static". Por otro lado, OpenMP nos ofrece un mecanismo mucho más intuitivo para establecer secciones críticas. En resumen, OpenMP nos habría ahorrado tiempo gracias a su relativa facilidad de uso (comparándolo con realizar todas las operaciones manualmente).

Realiza una descripción de alto nivel de cómo implementarías una aproximación a las tasks de OpenMP utilizando threads de C++

Se puede crear un mecanismo similar a las *tasks* de OpenMP simplemente creando un Thread Pool de hilos asíncronos (detach), a los que el hilo master iría llamando para ejecutar secciones de código según conveniencia. Para ello, podríamos crear una clase ThreadPool, que tendría algunos elementos indispensables como un mutex para para gestionar la cola de tareas, otro para el vector de resultados, un vector con los threads existentes y un atributo para saber cuántos threads estamos usando.

Práctica 2: Paralelización de un Programa de Multiplicación de Matrices Triangulares

Ejercicio 1

Explica brevemente la función que desarrolla cada una de las directivas y funciones de OpenMP que has utilizado.

Directivas:

- Directiva ***parallel***: define una región paralela sobre su ámbito. El bloque de código se ejecutará en paralelo por varios threads que se crean cuando un thread "padre" alcanza esta región. Al final de esta región existe una barrera implícita, en la que se espera a que todos los threads finalicen su trabajo antes de que el thread maestro continúe la ejecución del programa.

  Junto a esta directiva hemos añadido algunas cláusulas: ***private***, que establece que las variables indicadas sean privadas a los threads, manteniendo una copia local cada uno de ellos, y ***shared***, que establece que las variables especificadas sean compartidas, existiendo una única copia en memoria que puede ser leída o escrita por los threads, aunque no garantiza la exclusión mutua.

- Directiva ***for***: es una directiva de paralelismo de datos. Se ejecuta dentro de una región paralela y aprovecha sus threads para ejecutar las iteraciones del bucle for asociado. Al igual que la directiva parallel, también tiene una barrera implícita.

  A esta directiva se le ha asignado la cláusula ***schedule*** según lo indicado en cada actividad con diferentes parámetros. Establece el algoritmo de equilibrio de carga (por defecto static) y el tamaño del bloque de iteraciones que corresponde a cada thread.

Funciones:

- Función ***omp_get_wtime()***: Devuelve un double igual al número de segundos transcurridos desde algún punto en el pasado, manteniendo una coherencia. La utilizamos para obtener el tiempo de ejecución de un fragmento de código concreto.

Etiqueta todas las variables y explica por qué le has asignado esa etiqueta.

Privadas:

- Variables ***i, j, k***: son índices para identificar las iteraciones de los bucles, y como cada hilo recorrerá los vectores simultáneamente, es necesario darles una copia privada para que no se pisen entre sí.

Compartidas:

- Variables ***dim, A, B***: dim es un entero, y A y B son vectores de enteros. Aunque su naturaleza es distinta, estas tres variables tienen en común que poseen valores constantes que ninguno de los hilos modifica. Es razonable ahorrar memoria y permitir que todos los hilos lean la misma copia.

- Variable ***C***: es un vector de resultados. Esto implica que va a ser escrito por varios hilos a la vez. Sin embargo, cada uno de ellos va a realizar modificaciones sobre una región concreta a la que sólo el hilo en cuestión tiene acceso, luego podemos mantener una única copia compartida de este vector y, además, no sería necesario protegerlo con otros mecanismos.

Explica cómo se reparte el trabajo entre los threads del equipo en el código paralelo.

Por defecto, las iteraciones de un for paralelo se distribuyen por todos los hilos en bloques de un tamaño fijo (static), a no ser que se indique un tamaño de bloque específico con la cláusula *"Schedule (static, size)"*.

Calcula la ganancia (speedup) obtenido con la paralelización.

Para hacer el análisis de resultados se ha utilizado un tamaño de la matriz de 1224 elementos, de tal manera que el tiempo de ejecución secuencial sea de unos 10 segundos.
El tiempo de ejecución secuencial es 10.25 s, y en paralelo 2.28 s, luego el speedup será de:

$SpeedUp = \frac{T_S}{T_P}$, siendo T_S el tiempo de la ejecución secuencial y T_P el de la paralela.

$SpeedUp = \frac{10,25}{2,28} = 4,4956$

Ejercicio 2

Determina el tiempo de ejecución, tanto secuencial como paralelo, así como la ganancia obtenida. Añade la clausula schedule para modificar el reparto de trabajo entre los threads. Mide el tiempo de respuesta con los tres algoritmos disponibles, static, dynamic, guided, sin poner ningún tamaño de bloque.

Para automatizar el proceso de toma de medidas y mejorar la calidad del análisis, se ha desarrollado un algoritmo "exec_times.py" en Python que ejecuta el programa MatMul 10 veces y recopila los datos de la salida estándar para calcular una media y eliminar las desviaciones.

Sin schedule:

- ***Tiempo de ejecución secuencial***: 4.54 segundos
- ***Tiempo de ejecución paralelo***: 1.50 segundos

La ganancia obtenida sería: $SpeedUp = \frac{4,54}{1,50} = 3,03$

Con schedule (Static):

- ***Tiempo de ejecución secuencial***: 4.57 segundos
- ***Tiempo de ejecución paralelo***: 1.52 segundos

Con schedule (Dynamic):

- ***Tiempo de ejecución secuencial***: 4.51 segundos
- ***Tiempo de ejecución paralelo***: 1.03 segundos

Con schedule (Guided):

- ***Tiempo de ejecución secuencial***: 4.52 segundos
- ***Tiempo de ejecución paralelo***: 1.50 segundos

¿Qué diferencias observas con el ejercicio anterior en cuanto a la ganancia obtenida sin la cláusula schedule? Explica con detalle a qué se debe esta diferencia.

La ganancia se ve perjudicada debido a la considerable reducción del tiempo secuencial y a la poca mejora con respecto a la ejecución paralela del ejercicio anterior. Esto se debe a que al no utilizar la cláusula *schedule*, el algoritmo de equilibrio de carga que se utiliza por defecto es "static", que reparte las iteraciones en "bloques" que se asignan a los threads de manera equitativa. Esta solución, como veremos más adelante, es óptima si la carga de trabajo de las diferentes iteraciones es más o menos constante, pero al evolucionar al algoritmo *"Multiplicar_Matrices_Sup"* lo que hemos conseguido es omitir muchas operaciones en las que intervienen los ceros de la mitad superior de la matriz, por lo que ahora el tiempo requerido por cada bloque de iteraciones sería irregular, y al mantener el mismo algoritmo de equilibrio de carga, que no es adecuado para el nuevo escenario, estaríamos distribuyendo mal la carga de trabajo entre los hilos.

¿Cuál de los tres algoritmos de equilibrio de carga obtiene mejor resultado? Explica porqué ocurre esto, en función de cómo se reparte la carga de trabajo y las operaciones que tiene que realizar cada thread.

Podremos analizar y comparar los resultados más fácilmente si calculamos la ganancia de cada una de las implementaciones con respecto a su versión secuencial:

Static:	Dynamic:	Guided:
$S_S = \frac{4{,}57}{1{,}52} = 3,01$	$S_D = \frac{4{,}51}{1{,}03} = 4{,}37$	$S_G = \frac{4{,}52}{1{,}50} = 3{,}01$

De los tres algoritmos de equilibrio de carga el dynamic es el que nos ofrece los resultados más rápido. La razón por la que dynamic es la mejor opción puede explicarse mediante un contraejemplo. El scheduling static reparte el número de iteraciones del bucle equitativamente entre todos los threads, y su uso es muy recomendable cuando encontramos un bucle cuya carga de trabajo no varía entre iteraciones. El problema es que, dada la naturaleza del algoritmo que estamos paralelizando, esta carga varía mucho. Lo que sucede al utilizar static es que los threads que ejecutan un bloque de iteraciones que requiere un menor tiempo de cómputo finalizarán antes y quedarán a la espera de que los demás hilos finalicen también. Para evitar este problema se puede establecer una política dynamic, que asigna la carga a los hilos de manera dinámica conforme van finalizando el bloque de iteraciones (de tamaño fijo) que tenían asignado previamente. Con esto se consigue que ese hilo que permanecía a la espera de sus "hermanos" siga computando y liberando algo de carga a los demás.

La ganancia de guided dista mucho también de la que consigue dynamic, pese a que ambos algoritmos funcionan de forma parecida, con la diferencia de que la asignación dinámica de la carga tiende a la baja conforme van quedando menos iteraciones.

Para el algoritmo static piensa cuál será el tamaño del bloque óptimo, en función de cómo se reparte la carga de trabajo.

Teniendo en cuenta que el algoritmo static asigna bloques de igual cantidad de iteraciones a todos los hilos de la región paralela, y que el equipo de pruebas tiene 8 cores lógicos, el tamaño del bloque óptimo sería $1224/8 = 153$. Si ajustamos los parámetros en el programa, obtendremos un tiempo de ejecución secuencial de 4.56s y 1.50 para la ejecución paralela, luego el SpeedUp será:

$$S_{S_{153}} = \frac{4{,}56}{1{,}50} = 3,0604$$

Se observa cómo los valores de $S_S = 3,0066$ y $S_{S_{153}} = 3,0604$ se aproximan según lo previsto, luego el tamaño del bloque razonado es el correcto.

Para el algoritmo dynamic determina experimentalmente el tamaño del bloque óptimo. Para ello mide el tiempo de respuesta y speedup obtenidos para al menos 10 tamaños de bloque diferentes. Presenta una tabla resumen de los resultados y explicar detalladamente estos resultados.

Para aprovechar todo el potencial de cómputo de nuestro procesador empleando el algoritmo dynamic, deberíamos considerar tamaños de bloque inferiores a $1224/8 = 153$, pero no exactamente 153 porque obtendríamos un rendimiento similar al static.

Normalmente los tiempos de ejecución del programa secuencial varían mucho de acuerdo a la carga del procesador en el momento de la prueba. Por este motivo, se ha automatizado el proceso de toma de medidas de tal forma que se tomen 10 por cada tamaño de bloque, descartando las desviaciones, para obtener valores más precisos. El script desarrollado es "exec_times_block.py". Los resultados obtenidos para los distintos tamaños de bloque son los siguientes:

Bloque	Secuencial	Paralelo	SpeedUp
1 (default)	4.78	1.05	4.54
10	4.85	1.06	4.57
20	4.62	1.05	4.39
30	4.72	1.06	4.43
40	4.64	1.12	4.15
50	4.63	1.35	3.42
60	4.64	1.48	3.14
70	4.62	1.46	3.15
80	4.63	1.47	3.15
90	4.58	1.55	2.97
100	4.55	1.57	2.9
110	4.84	1.58	3.07
120	4.53	1.62	2.79
130	4.54	1.63	2.79

Según los resultados obtenidos, cuanto menor es el tamaño del bloque mejores resultados conseguimos. Las alternativas más eficientes corresponden a los tamaños de bloque desde 1 hasta 30. No elegiríamos nunca un valor mayor debido a la gran disminución del SpeedUp que se da con los valores posteriores.

Es difícil determinar el tamaño de bloque óptimo, dado que en los valores del 1 al 30 tenemos ganancias que se ven afectadas por los tiempos de ejecución secuencial, que en ocasiones varían demasiado pese a que sea una media de varias ejecuciones y sin picos. Por ello, aunque el mejor SpeedUp nos lo ofrece el bloque de 10 iteraciones, cualquiera de estos valores (1, 10, 20, 30) nos devolvería el resultado en un tiempo mínimo.

Explica porqué el algoritmo guided obtiene los peores resultados de tiempo, en función del reparto de carga de trabajo que realiza.

Aunque el algoritmo guided puede parecer la opción más eficiente dada su alta capacidad de adaptación en tiempo de ejecución, lo cierto es que su uso es más recomendable para los casos en los que tengamos un conjunto de hilos que realizan sus operaciones en una sección sin barrera y seguidamente encuentren un for "guided" que empezarán a ejecutar en distintos momentos, en el que además la carga computacional de las iteraciones es más o menos constante. Según los hilos van llegando al bucle, se les asigna un tamaño de bloque en función de las iteraciones restantes. De esta forma, los últimos en llegar obtendrían una carga más equilibrada con respecto a los threads que ya estaban en ese instante. De hecho, esta es la motivación de el algoritmo: ir balanceando el trabajo que realiza el conjunto de hilos para intentar que finalicen en tiempos similares.

Nuestro algoritmo no se adapta al caso expuesto: todos los threads empezarán a ejecutar el bucle en el mismo instante, y la carga de las distintas iteraciones no está muy bien equilibrada, por lo que el reparto no va a ser tan ajustado y acabaremos sufriendo retrasos.

Ejercicio 3

Realiza la misma paralelización que en el ejercicio anterior. Determina el tiempo de ejecución, tanto secuencial como paralelo, así como la ganancia obtenida en este caso. Añade la clausula schedule para modificar el reparto de trabajo entre los threads. Mide el tiempo de respuesta con los tres algoritmos disponibles, static, dynamic, guided, sin poner ningún tamaño de bloque.

Sin schedule:

- ***Tiempo de ejecución secuencial***: 3.89 segundos
- ***Tiempo de ejecución paralelo***: 1.29 segundos

La ganancia obtenida sería: $SpeedUp = \frac{3,89}{1,29} = 3,01$

Con schedule (Static):

- ***Tiempo de ejecución secuencial***: 3.85 segundos

- ***Tiempo de ejecución paralelo***: 1.29 segundos

Con schedule (Dynamic):

- ***Tiempo de ejecución secuencial***: 3.87 segundos
- ***Tiempo de ejecución paralelo***: 1.03 segundos

Con schedule (Guided):

- ***Tiempo de ejecución secuencial***: 3.90 segundos
- ***Tiempo de ejecución paralelo***: 1.04 segundos

¿Cuál es el mejor algoritmo de equilibrio en este caso? Explica las diferencias que aprecias con el ejercicio anterior.

Calculamos la ganancia de cada uno de los algoritmos para poder analizar y comparar más fácilmente los resultados:

Static:	Dynamic:	Guided:
$S_S = \frac{3,85}{1,29} = 2,97$	$S_D = \frac{3,87}{1,03} = 3,79$	$S_G = \frac{3,90}{1,04} = 3,69$

Considerando los resultados, el mejor algoritmo para este caso sigue siendo el dynamic pese a una bajada considerable del SpeedUp con respecto a la del ejercicio anterior. El guided ha mejorado poniéndose a la par del dynamic, por lo que también podríamos considerarlo como la primera opción. El static se mantiene constante con una leve caída.

¿Cuál es en este caso el peor algoritmo? Explica porqué obtiene unos resultados peores que en el caso anterior. ¿Puedes mejorar de alguna manera esos resultados?

El peor algoritmo es el static dado que tiene la ganancia más baja de las tres alternativas. Sin embargo, como mencionamos antes el SpeedUp ha disminuido algo con respecto al del caso anterior, pero puede deberse al hecho de haberse ejecutado en momentos distintos en los que el sistema podía estar más o menos saturado. De hecho, la evolución de los tiempos de ejecución ha sido favorable, siendo menores en este caso manteniendo una ganancia similar, por lo que no podríamos afirmar que obtenga unos resultados peores.

Práctica 3: Filtrado de un Vídeo mediante Tareas OpenMP

Ejercicio 1

Explica brevemente la función que desarrolla cada una de las directivas y funciones de OpenMP que has utilizado salvo las que ya hayas explicado en la práctica anterior (por ejemplo parallel).
Directivas:

- Directiva ***master***: Se asegura de que la región de código de su ámbito sea ejecutada únicamente por el thread master. Cualquiera de los demás hilos ignoran esta sección y continúan su ejecución. No posee barreras implícitas ni al principio ni al final.
- Directiva ***task***: Genera una tarea que entra en el threadpool de la región paralela, donde cada thread posee una cola de trabajos pendientes. Dicha tarea entraría en una de esas colas a la espera de ser ejecutada, por lo que su ejecución es asíncrona. El uso de task's no es la alternativa de paralelización más eficiente, pero facilita mucho el trabajo cuando tenemos una sección de código difícil de paralelizar, como el bucle "do while" de nuestro programa.

  A esta directiva la hemos añadido una nueva cláusula: ***firstprivate***, que permite asignar a cada uno de los hilos una copia de las variables indicadas, inicializada con el valor que tiene el hilo master en ese momento.
- Directiva ***taskwait***: Establece una barrera a modo de punto de sincronización entre el hilo master y sus hijos.

No se han utilizado nuevas funciones.

Etiqueta todas las variables y explica por qué le has asignado esa etiqueta.

Privadas (ámbito del parallel):

- La región paralela no establece variables privadas.

Compartidas (ámbito del parallel):

- Variables ***pixels, filtered***: pixels es el array que representa "seq" imágenes del vídeo, y filtered representa esas "seq" imágenes con el filtro ya aplicado. Los threads deben de poder acceder a estas variables porque serán los encargados de aplicar el filtro gausiano y, además, como cada hilo trabaja sobre una región concreta, no hay que establecer otros mecanismos para la exclusión mutua.
- Variable ***width, height***: indican el tamaño de cada imagen. Los hilos necesitan acceder a estas variables para poder aplicar el filtro.

  Existen otras variables que se utilizan dentro de la región paralela, como ***size, in, out o seq*** que son usadas solo por el thread master y no las etiquetamos como shared porque no es necesario darlas visibilidad frente al resto de los hilos.

Privadas (ámbito de la task):

- Variable ***i***: es el índice que indica sobre qué imagen del vector pixels se está trabajando. El hilo master es el encargado de actualizarlo, y los hilos deben de tener una copia local para garantizar que al aplicar el filtro sólo lo hagan sobre la imagen que se les ha ordenado.

Explica cómo se consigue el paralelismo en este programa.

Analizando los tiempos de ejecución del programa secuencial y de la región del código que hemos propuesto como candidata para su paralelización con un vídeo de 80 imágenes de 1920x1440px, hemos obtenido que el programa completo tarda en ejecutarse 21.99 segundos, y sólo las llamadas a *fgauss* duran un total de 21.08 segundos. Esto supone un 95.86 % del tiempo de ejecución, por lo que nos vamos a centrar en paralelizar el filtrado de las imágenes. Lo que sucede, es que paralelizar el "do while" en el que se encuentra no es trivial, por lo que se ha optado por aplicar un mecanismo asíncrono basado en task's que nos facilita el proceso.

En el programa secuencial, lo que se hace es leer una imagen del fichero de vídeo de entrada, aplicarla el filtro gausiano y guardarla en el fichero de salida. Este proceso se repetiría para todas las imágenes. Esto supone un coste temporal demasiado grande.

Entonces, la idea para paralelizar este programa se basa en aplicar el filtro a varias imágenes a la vez. Esta cantidad de imágenes está definida en la variable *"seq"*, que por defecto tiene valor 8. Se ha decidido no cambiar este valor dado que es el número de cores lógicos disponibles en el equipo de pruebas. El proceso consiste en establecer una región paralela que es ejecutada por el thread principal (master). Empezaría cargando una de estas imágenes en una posición del vector *pixels*, e inmediatamente crearía una tarea (task) que aplicaría el filtro a la posición del vector sobre la que cargó la imagen leída. Esta tarea será asignada a uno de los hilos del threadpool, y quedaría a la espera en la cola del hilo correspondiente hasta poder ejecutarse. De este modo ganamos tiempo al dejar que los threads vayan ejecutando las tareas mientras el master sigue leyendo.

Cuando el master finaliza la lectura de las imágenes y ya ha ordenado su procesamiento, se queda esperando en una barrera definida por la directiva *taskwait* a que sus hijos finalicen todas las tareas antes de comenzar a escribir las "seq" imágenes del vector *filtered* que se han procesado sobre el fichero de salida. El proceso se repetiría hasta que se hayan leído todas las imágenes del vídeo.

Al terminar de leer todas las imágenes pueden darse 2 situaciones: que el número de imágenes sea múltiplo de "seq", caso en el que no haría falta hacer gestión alguna, o que no lo sea. En este último caso lo que podría ocurrir es que el hilo master siga generando tareas que aplican el filtro a imágenes del vector *pixels* que ya se habían procesado, y además las guarde en una posición que no las corresponde en el fichero de salida. Este fichero ocuparía espacio de más, claro está. La situación debe evitarse u obtendremos un vídeo defectuoso.

El mecanismo implementado para evitar este error es el mismo que se utiliza en el algoritmo secuencial, pero algo más evolucionado: cuando el valor devuelto por la función *fread* es 0, según el manual, se debe entre otros posibles errores a que se han leído menos elementos de los esperados o que nos encontremos al final del fichero. Aprovechamos esto para decidir cuándo aplicar el filtro y sobre qué rango de *pixels*. Si el valor devuelto no es 0, el programa se comportará según lo esperado: lee cada una de las imágenes, ordena su procesamiento una por una, y espera para poder escribirlas todas de golpe en el fichero de salida. Si nos encontrásemos en el caso contrario, significa que la cantidad de imágenes restantes es menor a "seq", y automáticamente cambiaríamos este valor al número de imágenes leídas correctamente en el ciclo correspondiente y forzaríamos la escritura de las imágenes restantes. No es necesario realizar ninguna gestión más, ya que a no ser que el fichero de entrada esté corrupto y el tamaño que ocupan las imágenes fuera irregular, en la siguiente iteración del "do while" la condición de permanencia no se cumpliría porque se ha llegado al 'eof'.

Calcula la ganancia (speedup) obtenido con la paralelización para todo el programa y explica los resultados obtenidos.

Tomaremos como referencia el tiempo que tarda en ejecutarse la parte del "do while" tanto en el código secuencial como el paralelo, ya que el resto del programa se mantiene igual en ambas versiones y no afecta al cálculo del SpeedUp. Como en prácticas anteriores, el proceso se ha automatizado con un script "exec_times.py" programado en Python que toma el tiempo de la salida estándar de los programas secuencial y paralelo, que son ejecutados varias veces para obtener un tiempo medio sin considerar los picos.

Los tiempos de ejecución resultantes, en segundos, son:

- ***Tiempo de ejecución secuencial***: 22.06 segundos
- ***Tiempo de ejecución paralelo***: 7.00 segundos

Luego el SpeedUp conseguido sería el siguiente: $S_S = \frac{22{,}06}{7{,}00} = 3{,}15$

La ganancia obtenida sorprende, debido a que en el apartado anterior habíamos estimado que la fracción paralelizable de este programa era un 95.86 %. Llegados a este punto nos interesa comparar el resultado teórico con el práctico, por lo que aplicaremos la Ley de Amdahl para obtener la máxima ganancia a la que podríamos aspirar con los 8 cores lógicos de nuestro equipo:

$$S_{max} = \frac{T_1}{T_P} = \frac{1}{s + \frac{p}{P}} = \frac{1}{0,0414 + \frac{0,9586}{8}} = 6,20$$

Pese a la gran diferencia entre el caso ideal y la realidad, hemos conseguido una ganancia considerable teniendo en cuenta la magnitud del problema que estamos enfrentando. La versión paralela de nuestro programa nos ofrece una salida 3 veces más rápido que el programa secuencial.

Práctica 4: Fractal de Mandelbrot

Ejercicio 1

Explica brevemente la función que desarrolla cada una de las directivas y funciones de OpenMP que has utilizado (sólo las que no hayas explicado en prácticas anteriores).

Directivas:

- Directiva ***single***: Es una directiva de reparto de trabajo que consiste en que solo un thread ejecuta secuencialmente el código de su ámbito. Este bloque de código sería ejecutado por el primer hilo que llegue y los demás se quedarían esperando en la barrera implícita que tiene al final.
- Directiva ***barrier***: Es un mecanismo para sincronizar los threads de la región paralela. Cuando un thread llega a una barrera, se queda esperando a que el resto de los threads finalicen sus operaciones y la alcancen también, antes de continuar la ejecución. Es necesario que todos los hilos alcancen la barrera en orden para que el conjunto pueda avanzar.
- Directiva ***critical***: Esta directiva establece un mecanismo de sección crítica que garantiza la exclusión mutua sobre un bloque de código. Solo un thread puede acceder a esta sección al mismo tiempo. Primero comprueba si está libre, y de ser así, bloquearía el acceso a los otros hilos y empezaría la ejecución. Si no estuviera libre, se queda esperando la liberación por parte del thread que la estuviera ejecutando. Emplea nombres que identifican las distintas secciones críticas y, las regiones de código que no tengan dicho identificador serán tratadas como si pertenecieran a la misma sección.

No se han utilizado nuevas funciones.

Etiqueta todas las variables y explica por qué le has asignado esa etiqueta.

Privadas:

- Variables ***i, j***: son los índices de los bucles. Cada thread debe de tener una copia privada para poder iterar sin pisar el trabajo de los demás hilos.
- Variables ***x, y***: se utilizan para computar los parámetros que recibe la función *explode* en cada iteración del bucle que determina los valores de "count". Son una especie de variable auxiliar y cada hilo requiere una copia.
- Variable ***c***: se trata también de una variable auxiliar para el cálculo del color que recibirá cada píxel de la imagen. Por su naturaleza, cada hilo tendrá una copia local.

Compartidas:

- Variable ***n***: es la cantidad de píxeles, tanto en el eje X como en el eje Y, ya que la imagen es cuadrada. Se utiliza para delimitar los bucles de todas las secciones paralelizadas del código, por lo que permitiremos que todos los hilos accedan a la misma copia. No es necesario proteger el acceso, ya que sobre ella sólo se realizan lecturas y su valor permanece invariable.
- Variables ***x_max, x_min, y_max, y_min***: también son variables con valor constante, a las que se accede cuando se va a computar el valor de "x" e "y" que se pasará como parámetro a la función *explode*. Por esa razón, se permite que los hilos accedan a la misma copia.
- Variable ***count, count_max, c_max***: "count" es un vector que almacena un valor de "count" para cada píxel, y "count_max" y "c_max" son variables utilizadas para el cómputo del color que les corresponde más adelante. Deben ser visibles para todos los hilos. En el caso del vector, no es necesario garantizar la exclusión mutua de sus accesos porque cada hilo trabaja sobre un rango concreto. "count_max" funciona a modo de valor constante y "c_max" es una variable que puede ser actualizada a la vez por distintos hilos y que se protege mediante mecanismos de exclusión.

Explica brevemente cómo se realiza el paralelismo en este programa, indicando las partes que son secuenciales y las paralelas.

En primer lugar, se ha hecho un profiling para saber en qué regiones del código debemos concentrar todos los esfuerzos si fuera posible. Hemos empleado la herramienta del sistema "*gprof*" [1].

El resultado es el siguiente:

analysis.txt

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 93.80     70.08    70.08 64016001     0.00     0.00  explode
  5.35     74.08     3.99                             main
  0.94     74.78     0.70        1   702.55   702.55  ppma_write_data
  0.28     74.99     0.21        1   210.76   913.31  ppma_write
  0.00     74.99     0.00        2     0.00     0.00  timestamp
  0.00     74.99     0.00        1     0.00     0.00  ppma_write_header

Copyright C 2012-2020 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
```

Observamos que el mayor cuello de botella de nuestro programa son las llamadas a la función ***explode***, a la que se realizan un total de 64.016.001 de llamadas, consumiendo el 93.80 % del tiempo de ejecución del conjunto del programa. Aunque el enunciado pide crear una región paralela que cubra todo el cuerpo del programa, daremos prioridad a la paralelización de la sección en la que se realizan estas llamadas a explode y, posteriormente, paralelizaremos todo lo demás para conseguir el mayor SpeedUp posible.

Realizaremos el procedimiento en orden, ya que vamos a paralelizar todo el cuerpo del programa de todos modos. En primer lugar, se busca un espacio en la memoria para almacenar los valores del vector *count*. De esta parte se encarga únicamente el thread máster. Para evitar que los hilos hijo ignoren esta sección sin más y empiecen a trabajar sobre un espacio de memoria aún indeterminado, se coloca una barrera con la directiva *barrier*. De este modo, todos los hilos esperarán a que se haya asignado el espacio de memoria sobre el que van a operar.

La siguiente sección corresponde a la que contiene las llamadas a *explode()*, y las mejoras que podemos aplicar se limitan a asignar la directiva *for* al bucle principal. En un principio decidí no incluir la cláusula schedule, para que la distribución de la carga por defecto la realizase el algoritmo static. Más adelante, en el ejercicio 3, me di cuenta de que los números impares de threads ofrecían un SpeedUp mucho menor que los pares, y caí en la cuenta de que el reparto de la carga se estaba realizando mal: algunos hilos trabajaban mientras los demás esperaban. Después de hacer unas pruebas, he optado por incluir la cláusula para utilizar el algoritmo *dynamic*, que ofreció un SpeedUp muchísimo superior para cualquier número de threads. Se han ido etiquetando todas las variables que en algún momento dado van a utilizar los threads que ejecuten esta sección del código, tal y como se expuso en el apartado anterior.

La siguiente sección, "Determine the coloring of each pixel", contiene otro un bucle al que asignaremos la directiva *for*, que va acompañada en este caso de la cláusula *nowait* para eliminar la barrera implícita que hay al final de la sección y permitir que los hilos que hayan finalizado sus tareas avancen hacia la siguiente para ganar tiempo. Nuestra intención es que el primer hilo que finalice sus operaciones pase a la sección "Set the image data" y vaya reservando la memoria mientras los demás terminan.

Esa reserva de memoria está asociada a la directiva "single" con el objetivo de que la ejecute el primer hilo en llegar. De nuevo, al estar reservando la memoria con la que se va a trabajar seguidamente, hay que establecer una barrera para evitar que los hilos empiecen a acceder a posiciones indeterminadas. Por suerte, la directiva *single* posee una implícita al final. El for siguiente se paralelizaría incorporándolo al ámbito de una nueva directiva *for*.

[1] Documentación sobre *gprof* utilizada: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/text/gprof.txt

Ejercicio 2

El segundo bloque de código etiquetado con el comentario Determine the coloring of each pixel, obtiene el valor de la variable c_max, que debe ser compartida. Esto genera la necesidad de proteger el acceso a esta variable mediante una sección critica. Realiza cuatro implementaciones distintas de la sección crítica y comprueba el rendimiento de cada una de ellas, calculando el speedup obtenido sólo por ese bucle. Explica los resultados obtenidos.

De aquí en adelante, para poder arrojar más datos sobre cómo se está comportando nuestro programa y sacar mejores conclusiones, tomaremos las medidas de tiempos de ejecución tanto para la parte paralelizable del programa como para la sección correspondiente al bucle que actualiza la variable compartida.

El tiempo de la parte paralelizable del programa secuencial es de 73.001875 segundos, y la parte del bucle tarda en ejecutarse solamente 0.143191 segundos incluidos en el tiempo anterior. Utilizaremos estos datos más adelante para calcular las ganancias.

Existe una constante definida en la cabecera que toma un valor del enumerado "*protection_mode*" para indicar cuál de las cuatro implementaciones vamos a utilizar.

```
#define PROT_MODE OMP
enum protection_mode{OMP, RUN, SEC, PRIV};
```

Cada una de estas 4 implementaciones se ha considerado en la sección a proteger. Se ejecutarán unas acciones u otras en función de la constante ***PROT_MODE*** gracias a un switch. Para cada una de las implementaciones, a continuación, además de los tiempos de ejecución y la ganancia obtenida se incluye una versión simplificada de esee código tan extenso, considerando solo el caso correspondiente para facilitar la comprensión:

- **Implementación mediante directivas OpenMP.**

  El código que se ejecutaría en este primer caso es el siguiente:

  Caso OpenMP (OMP)

```
#pragma omp for nowait
for ( j = 0; j < n; j++ )
{
    for ( i = 0; i < n; i++ )
    {
        #pragma omp critical
        {
            if (c_max < count[i+j*n] )
            {
              c_max = count[i+j*n];
            }
        }
    }
}
```

  Sección paralela al completo: Tarda en ejecutarse 25.365280 segundos. La ganancia obtenida con respecto del programa secuencial será:

$$S_{OMP_{completo}} = \frac{73{,}00}{25{,}37} = 2,88$$

  La ganancia es algo más baja de lo que esperábamos teniendo en cuenta que estamos usando todos los recursos del equipo de pruebas. Como veremos ahora, el culpable de esta poca ganancia es el bucle.

  Solo el bucle: Del tiempo obtenido anteriormente, 11.618857 segundos corresponden a la ejecución del bucle. El SpeedUp con respecto al tiempo de ejecución del bucle del programa secuencial es:

$$S_{OMP_{bucle}} = \frac{0{,}14}{11{,}62} = 0,012$$

El resultado no es bueno, aunque era de esperar. Todo ese tiempo de más se debe a que tal y como está planteado el problema, los hilos en cada iteración necesitarán acceder a la sección crítica. Estamos introduciendo un overhead considerable, y el resultado de esto es que hemos aumentado el tiempo de ejecución de esta parte en casi 100 veces, por lo que por el momento, nos inclinaríamos más por la opción de mantener la ejecución secuencial de esta sección.

- **Implementación mediante funciones de runtime.**

  El código en este caso es:

Caso Runtime (RUN)

```
omp_lock_t mutex;
omp_init_lock(&mutex);

#pragma omp for nowait
for ( j = 0; j < n; j++ )
{
    for ( i = 0; i < n; i++ )
    {
        omp_set_lock(&mutex);
        if (c_max < count[i+j*n])
        {
            c_max = count[i+j*n];
        }
        omp_unset_lock(&mutex);
    }
}
```

  Sección paralela al completo: Tarda en ejecutarse 26.532635 segundos. La ganancia obtenida con respecto del programa secuencial es:

$$S_{RUN_{completo}} = \frac{73{,}00}{26{,}53} = 2,75$$

  La ganancia ha disminuido, lo que nos hace sospechar que el tiempo de ejecución del bucle ha aumentado. Lo vemos:

  Solo el bucle: Tarda 12.838814 segundos en ejecutarse, y como decíamos antes, es más tiempo que en el caso anterior, luego el SpeedUp disminuirá también:

$$S_{RUN_{bucle}} = \frac{0{,}14}{12{,}84} = 0,0109$$

  Por la misma razón que antes este resultado era el esperado. Estamos haciendo lo mismo, aunque con un mecanismo que nos da más control sobre el mutex que ahora hemos declarado de forma explícita.

- **Implementación secuencial de esa parte del código.**

  Además del código interno del bucle, en esta implementación también cambia la directiva asociada al bucle principal. Dejaríamos de ejecutarlo en paralelo con la directiva *for* para que lo ejecute únicamente el primer hilo en llegar con la directiva *single*.

  El código de esta implementación sería el siguiente:

Caso Secuencial (SEC)

```
#pragma omp single
for ( j = 0; j < n; j++ )
{
    for ( i = 0; i < n; i++ )
    {
        if (c_max < count[i+j*n])
        {
            c_max = count[i+j*n];
        }
    }
}
```

Sección paralela al completo: La sección completa tarda en ejecutarse 13.7796 segundos. La ganancia obtenida con respecto del programa secuencial es:

$$S_{SEC_{completo}} = \frac{73{,}00}{13{,}78} = 5,298$$

La ganancia ha aumentado considerablemente. Parece que lo que comentamos antes sobre dejar que la parte del bucle se ejecute de forma secuencial paralelizando otras partes del código ha dejado resultados prometedores. Por el momento nos inclinaremos hacia esta opción como la más beneficiosa.

Solo el bucle: A la vista de los resultados anteriores, el tiempo esta vez será menor. El bucle de la variable compartida nos da un tiempo de ejecución 0.146613 segundos. La ganancia en este caso sería aproximadamente 1, porque el modo de ejecución del bucle es igual al del programa secuencial. Utilizaremos todos los decimales en la operación para ser más precisos:

$$S_{SEC_{bucle}} = \frac{0{,}143784}{0{,}146613} = 0,9807$$

- **Implementación con variables privadas a cada thread y selección del máximo de todas ellas.**

La implementación esta vez es:

Caso Variables Privadas (PRIV)

```
#pragma omp for nowait
for ( j = 0; j < n; j++ )
{
    for ( i = 0; i < n; i++ )
    {
        if (c_privada < count[i+j*n])
        {
            c_privada = count[i+j*n];
            #pragma omp critical
            {
                if (c_max < c_privada)
                {
                    c_max = c_privada;
                }
            }
        }
    }
}
```

Sección paralela al completo: La ejecución dura 13.682419 segundos. La ganancia obtenida con respecto del programa secuencial es:

$$S_{PRIV_{completo}} = \frac{73{,}00}{13{,}68} = 5,336$$

Esta ganancia es la mejor que hemos conseguido hasta ahora, luego podemos concluir que la mejor de las 4 alternativas es la de las variables privadas, por lo que la mantendremos en el programa final.

Solo el bucle: Hemos obtenido el mejor tiempo de ejecución para esta sección hasta ahora, tardando solo 0.031203 segundos. La ganancia sin lugar a dudas será la mejor de todas:

$$S_{PRIV_{bucle}} = \frac{0{,}143784}{0{,}031203} = 4,608$$

Estos buenos resultados se deben a que en realidad la mayoría de las operaciones que los hilos realizan se hacen a nivel local. Se ha descubierto que el gran lastre en cuanto a tiempo de ejecución es tener que esperar para leer la variable compartida, cuando en realidad lo más probable es que no la vayas a modificar en esa iteración. Ahora en vez de comprobar la variable compartida "c_max" en cada iteración, los hilos tienen una variable propia que pueden consultar, y sólo pedirán el mutex (implícito en la directiva *critical*) cuando hayan encontrado un valor mayor y quieran actualizarlo a nivel global.

Ejercicio 3: Escalado fuerte.

Aplicando la Ley de Amdahl estimar el speedup teórico para un número de threads entre 1 y 8, sin modificar el tamaño del problema. Para ello debéis calcular la fracción serie sobre el programa secuencial, midiendo el tiempo de ejecución de las partes no paralelizables con función omp_get_wtime().

Se ha obtenido que el tiempo de ejecución de todo el programa es de 84.345105 segundos, y el de la parte paralelizable sería de 73.022412 segundos. Son unos resultados que van en la línea de lo que nos indicaba anteriormente el profiling realizado, en el que vimos que las 64.016.001 llamadas a ***explode*** consumían el 93.80 % del tiempo de ejecución del conjunto del programa.

Sabemos, entonces, que la **fracción paralela** del programa es $\frac{73{,}022412}{84{,}345105} = \mathbf{0{,}8658}$ y, por lo tanto, la **fracción serie** será $1 - 0,8658 = \mathbf{0{,}1342}$.

El SpeedUp teórico viene dado por la Ley de Amdahl. Vamos a adaptarla a nuestro programa y a dejarla en función del número de procesadores para realizar los cálculos más adelante:

$$S_{max} = \frac{T_1}{T_P} = \frac{1}{s + \frac{p}{P}} = \frac{1}{0,1342 + \frac{0{,}8658}{P}}$$

Entonces, a partir de la fórmula obtenemos los siguientes SpeedUp's teóricos:

Nº de threads	SpeedUp Teórico
1	1
2	1,76
3	2,37
4	2,85
5	3,25
6	3,59
7	3,88
8	4,12

Comprobar experimentalmente estos valores, hasta utilizar una cantidad de threads igual al número de cores del computador. Dibujar una gráfica con los resultados teóricos y experimentales obtenidos.

Para tomar las medidas hemos utilizado la versión paralela de las variables privadas, que ofrecía el mejor rendimiento. Con un script desarrollado en Python "graphics.py" hemos representado los SpeedUp's de las nuevas medidas con respecto del tiempo secuencial que hemos obtenido antes en función del número de threads a utilizar, ahora pasado como parámetro al programa *mandelbrot*. Debido al tiempo que duran las pruebas, esta vez nos limitaremos a tomar una muestra por cada número de hilos. El script nos ha devuelto los siguientes resultados:

Nº de threads	Tiempo de ejecución	SpeedUp Experimental
1	84,73	0.99
2	43.89	1.92
3	31.79	2.65
4	26.58	3.17
5	23.87	3.53
6	21.17	3.98
7	19.81	4.25
8	19.99	4.21

Si representamos los SpeedUp's teóricos y experimentales en una gráfica obtenemos lo siguiente:

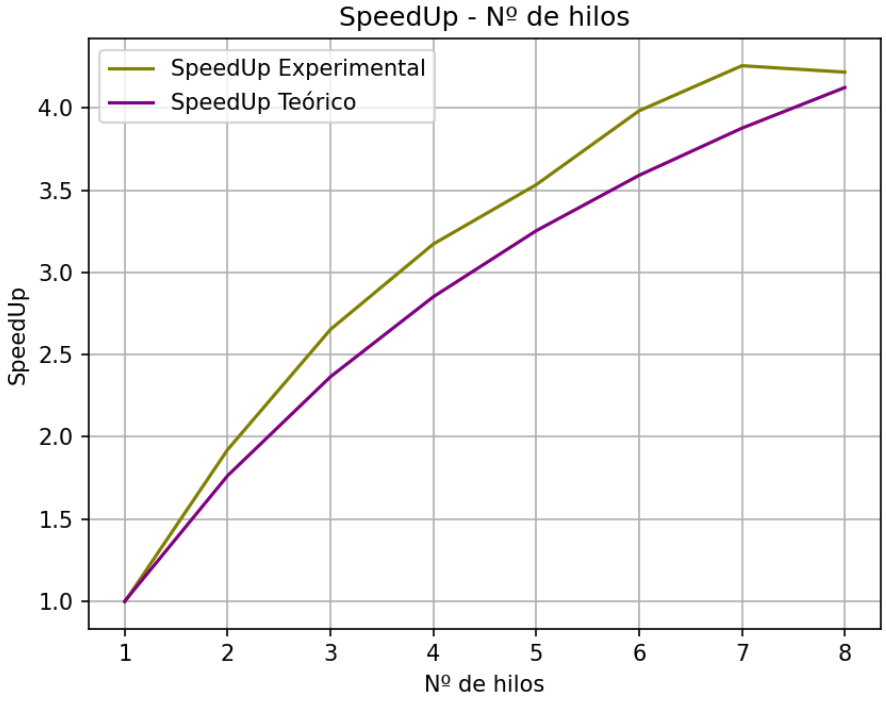

Figura 4: Representación de los SpeedUp's reales y experimentales de "mandelbrot.c".

Extrae todas las conclusiones que puedas, tanto sobre la escalabilidad fuerte de la aplicación, como sobre la fiabilidad de la Ley de Amdalh para predecir el escalado fuerte.

Nuestra aplicación escala hasta 7 threads porque como podemos observar en la gráfica, el SpeedUp crece proporcionalmente con respecto al número de hilos hasta llegar a dicho valor. Además, la ganancia de la ejecución con 8 hilos ha bajado con respecto a la anterior, por lo que a partir de 8 hilos el rendimiento será peor. La tendencia, además, no es lineal, por lo que no podemos afirmar que nuestro programa sea escalable en sentido fuerte.

La Ley de Amdahl es fiable para predecir el escalado fuerte. Si nos fijamos en los resultados, la tendencia que sigue nuestra función del SpeedUp Experimental sigue la tendencia de la función del SpeedUp Teórico.

Ejercicio 4: Escalado débil.

Aplicar la Ley de Gustafson para obtener el speedup escalado de este programa, para un número de threads entre 1 y 8, modificando adecuadamente el tamaño del problema. Para ello mide la fracción serie del programa secuencial para cada tamaño del problema, y calcula el speedup escalado correspondiente. Comienza con un tamaño de problema pequeño, en torno a 5 segundos. Ten en cuenta que la complejidad del algoritmo es cuadrática respecto a los datos de entrada, es decir $O(n^2)$.

Partimos de un tamaño de problema de 2001, que da más o menos el tiempo recomendado en el enunciado.

El SpeedUp Escalado viene dado por la Ley de Gustafson:

$$SpeedupEscalado = P + (1 - P) \times s$$

De nuevo, será un script (graphics_gustafson.py) el que realice todo el proceso por nosotros. Nos ofrece los tiempos de ejecución del programa secuencial completo y de su parte paralelizable, el tiempo de ejecución del programa paralelo para el mismo tamaño de problema que el secuencial, y calcula por nosotros la fracción serie y el SpeedUp Escalado. Los resultados se muestran en la siguiente tabla:

Nº Threads	Tamaño del problema	Fracción serie	SpeedUp Escalado
1	2001	1 - (4.41/5,14) = 0.14	1.00
2	$\sqrt{2 \times (2001)^2} = 2830$	1 - (8.97/10.29) = 0.12	1.87
3	$\sqrt{3 \times (2001)^2} = 3466$	1 - (13.40/15.62) = 0.14	2.72
4	$\sqrt{4 \times (2001)^2} = 4002$	1 - (17.80/20.37) = 0.12	3.62
5	$\sqrt{5 \times (2001)^2} = 4475$	1 - (22.42/26.03) = 0.13	4.45
6	$\sqrt{6 \times (2001)^2} = 4902$	1 - (27.18/31.12) = 0.12	5.37
7	$\sqrt{7 \times (2001)^2} = 5295$	1 - (31.74/36.58) = 0.13	6.21
8	$\sqrt{8 \times (2001)^2} = 5660$	1 - (36.34/41.66) = 0.12	7.11

Comprobar experimentalmente estos valores, hasta el número de threads que permita el computador. Dibujar una gráfica con los resultados teóricos y experimentales obtenidos.

#THREADS/SIZE	$T_{secuencial}$	$T_{paralelo}$	SpeedUp Experimental
1 (2001)	5.14	5.14	1.00
2 (2830)	10.29	6.29	1.63
3 (3466)	15.62	7.45	2.09
4 (4002)	20.37	8.98	2.26
5 (4475)	26.03	9.80	2.65
6 (4902)	31.12	10.71	2.90
7 (5295)	36.58	12.32	2.96
8 (5660)	41.66	14.34	2.90

Podemos comparar los SpeedUp's escalados y los experimentales en la siguiente gráfica:

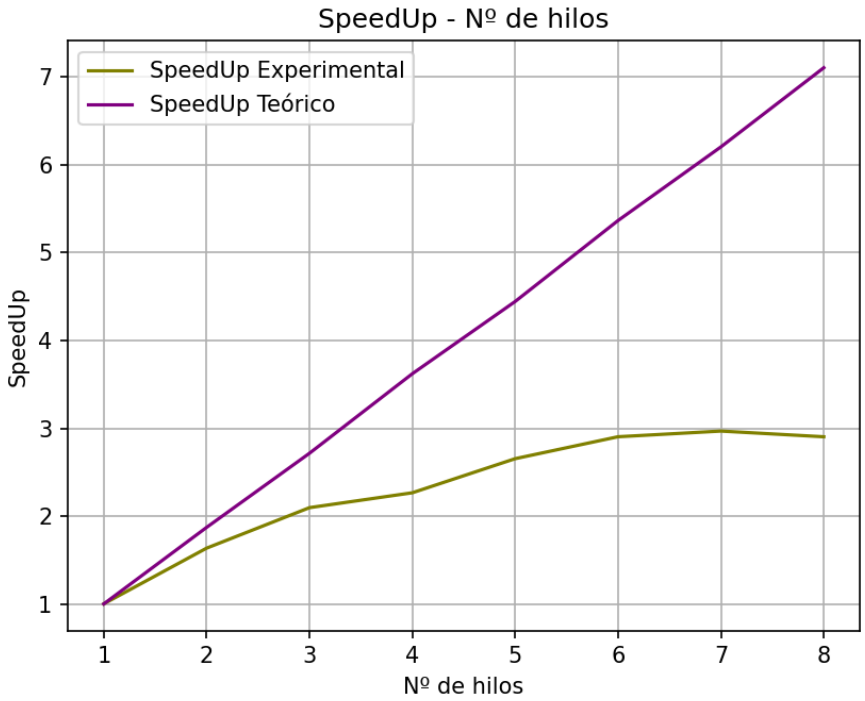

Figura 5: Representación de los SpeedUp's escalados y experimentales de "mandelbrot.c".

Extrae todas las conclusiones que puedas, tanto sobre la escalabilidad débil de la aplicación, como sobre la fiabilidad de la Ley de Gustafson para predecir el escalado débil.

Nuestro problema tampoco es escalable en el sentido débil. Lo ideal sería que la ganancia se mantenga constante según aumentan los recursos del sistema y el tamaño del problema, pero no lo hace. De hecho, dista mucho del caso ideal dado por la Ley de Gustafson al seguir una tendencia no lineal. Se puede observar que el problema es escalable para hasta 7 hilos. A partir de entonces, el rendimiento empeora con respecto a esa cantidad (véase el rendimiento para 8 hilos, menor al anterior).

A la vista de los resultados, podemos concluir que la Ley de Gustafson no solo es fiable, sino también realista, porque define perfectamente el mejor caso que podríamos alcanzar.