UNIVERSITAT POLITÈCNICA DE CATALUNYA

Paralelismo

Laboratorio 3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

21/04/2022

Haopeng Lin Ye

Joan Sales de Marcos

ÍNDICE

1 - Task decomposition analysis for the Mandelbrot set computation

1.1 - The Mandelbrot set (introduction)

En los dos primeros deliverables hemos visto formas de paralelizar programas a través del hardware y el código. En este caso usaremos esos conocimientos para descubrir la mayor eficiencia posible de un programa y cómo se comporta con las diferentes maneras de enfocar la paralelización.

Concretamente estudiaremos un código que calcula mediante iteraciones en una recursión si las coordenadas de un píxel están incluidas en el set o conjunto de Mandelbrot.

Para más información de este conjunto ver:
https://es.wikipedia.org/wiki/Conjunto_de_Mandelbrot

Al ejecutar el programa base y secuencial de este set (`./mandel-seq -d`) se obtiene una imagen que muestra con colores diferentes cuáles píxeles conforman el mismo, se pueden apreciar las iteraciones del código en los cambios de color de la figura 1.1.

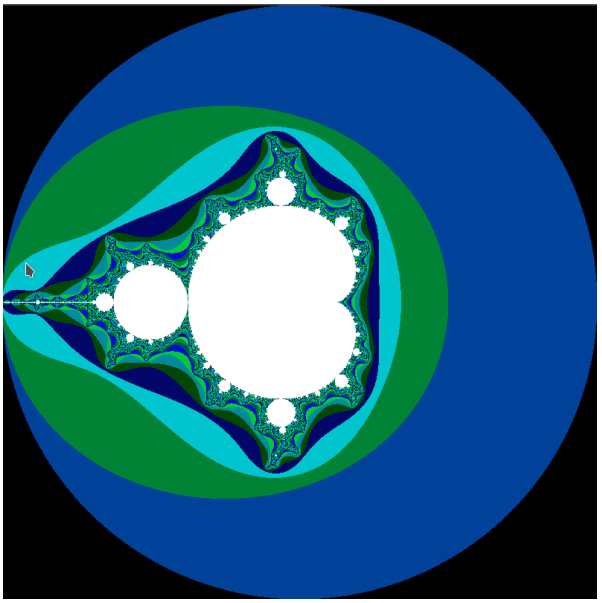


Figura 1.1: Mandelbrot Set “primigenio” (`./mandel-seq -d`)

En el comando de entrada de la terminal se pueden incluir también diversos parámetros para computar diferentes partes del set. Por ejemplo, usando `./mandel-seq -d -c -0.737 0.207 -s 0.01 -i 100000` como entrada obtendremos el subconjunto del set de Mandelbrot que muestra la figura 1.2:

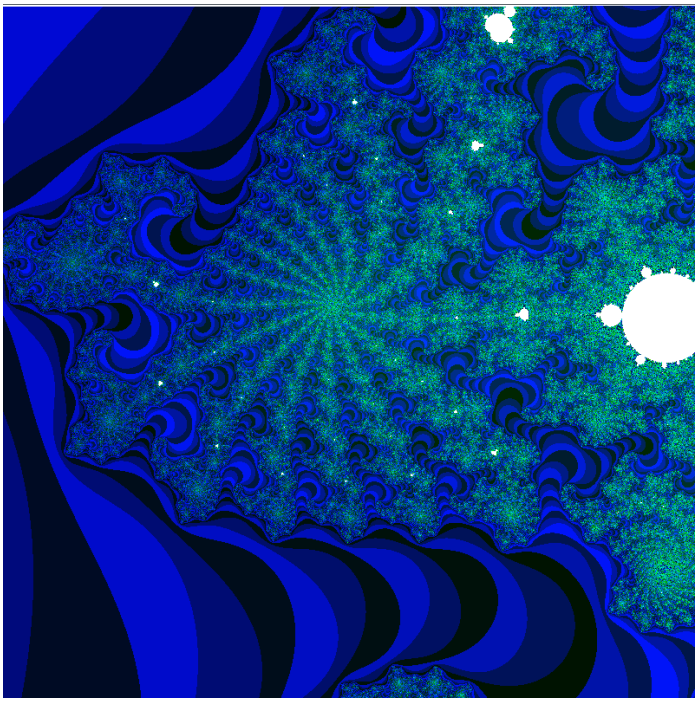


Figura 1.2: Imagen mostrada al ejecutar `./mandel-seq -d -c -0.737 0.207 -s 0.01 -i 100000`

Disponemos de diversas versiones de este programa para computar el set así que lo primero es analizar con Tareador lo que está pasando exactamente en cada una para pensar la mejor estrategia y así lograr la mayor eficiencia para la paralelización.

1.2 - Task decomposition analysis with Tareador

Como ya se ha dicho, para saber qué es lo que está ocurriendo en el código es muy útil usar Tareador. Para ello ejecutaremos el programa mandel-tar con diferentes opciones (y después con pequeñas modificaciones).

1.2.1 - Row Strategy

Primero veremos los TDG del código sin modificar, es decir, viendo las “Row”s (Filas) como tareas.
El grafo de dependencias generado al simplemente ejecutar `./mandel-tar` es el que se muestra en la figura 1.3:

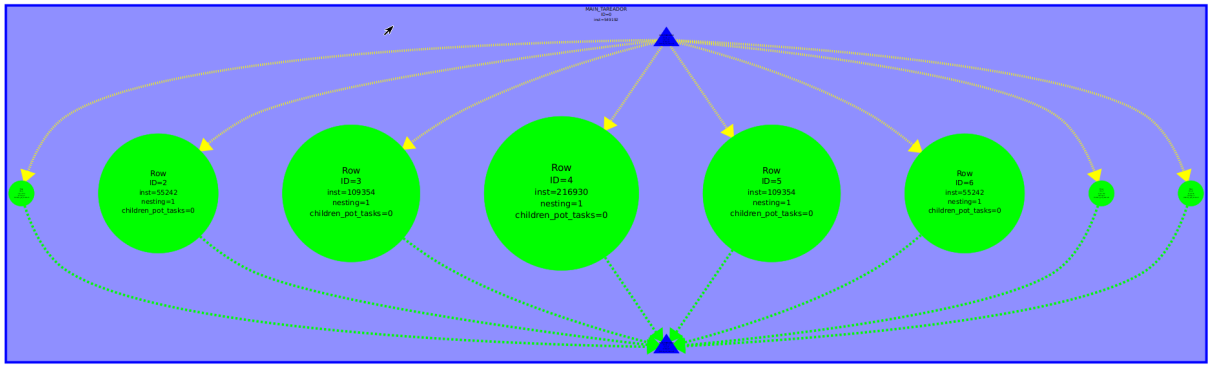


Figura 1.3: TDG generado por `./mandel-tar` Row strategy

Se puede apreciar que las 8 tareas son independientes y perfectamente paralelizables, esto se debe a que no hay dependencias entre las filas (Rows) que conforman la imagen para mostrar el set (por defecto son 8, ya que es el tamaño de la imagen definido en el propio código).

Otra característica destacable es que el tiempo empleado o el peso de las tareas intermedias es mucho más grande que el de las iniciales y las finales. Esto se puede deber a que las primeras y últimas rows para mostrar el set tienen más píxeles que quedan descartados inicialmente por el código, es decir, que en las primeras iteraciones el código identifica que esas coordenadas no forman parte del conjunto de Mandelbrot y no se han de procesar en futuras iteraciones (por la forma del set vista en la figura 1.1 se puede intuir).

Al ejecutar este mismo código con la opción “-d” el grafo de dependencias cambia radicalmente y se convierte en el de la figura 1.4:

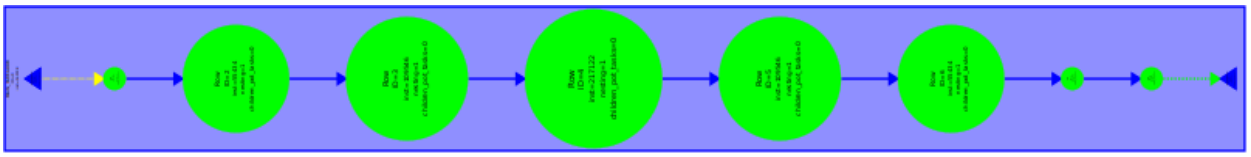


Figura 1.4: TDG generado por `./mandel-tar -d` rotado 90º (va de izquierda a derecha) Row strategy

A simple vista el grafo pasa a ser completamente secuencial sin haber cambiado nada interior del código. Esto seguramente sea provocado porque la opción del display -d hace llamadas a una librería de X11 que sí provoca dependencias.

El peso de las diferentes tareas no ha cambiado respecto a la ejecución básica. Las primeras y últimas tienen menos tiempo de ejecución que las intermedias.

Por último ejecutaremos el mismo código con la opción -h. Esto provoca un grafo de dependencias un poco más peculiar, visto en la figura 1.5:

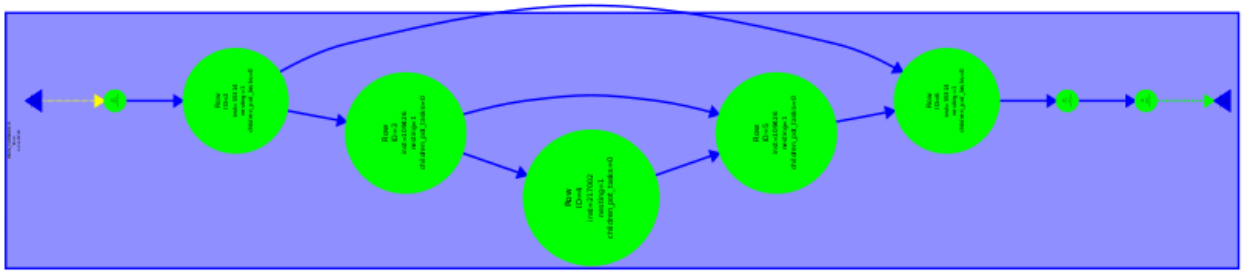


Figura 1.5: TDG generado por `./mandel-tar -h` rotado 90º (va de izquierda a derecha) Row strategy

En este caso el Task Dependency Graph muestra que solamente las tareas intermedias tienen dependencias entre ellas. Este efecto seguramente sea provocado porque con la opción -h la máquina ha de generar un histograma con los valores del Set de Mandelbrot y este histograma debe mantenerse coherente entre las diferentes iteraciones. En las primeras y últimas tareas (el peso de las cuales es idéntico a ejecuciones anteriores) no hay dependencias ya que inicialmente los píxeles son descartados y no se procesan.

1.2.2 - Point Strategy

Al hacer lo mismo que en el apartado anterior pero usando “Point” como tarea se han obtenido los siguientes grafos (son muy largos así que no se ven del todo bien para que quepan):

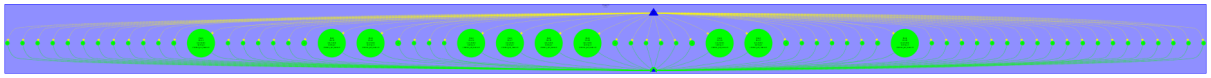


Figura 1.6: TDG generado por `./mandel-tar` Point Strategy

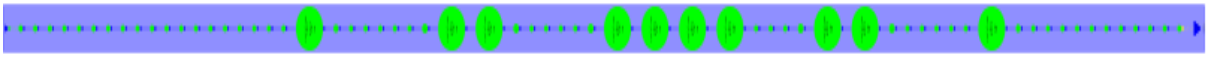


Figura 1.7: TDG generado por `./mandel-tar -d` rotado 90º (va de izquierda a derecha) Point Strategy

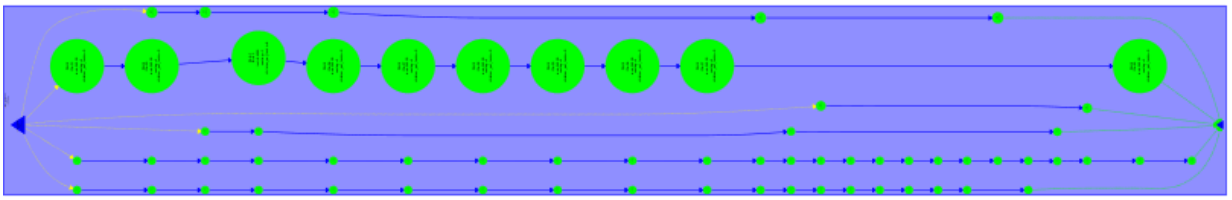


Figura 1.8: TDG generado por `./mandel-tar -h` rotado 90º (va de izquierda a derecha) Point Strategy

En la figura 1.6 se puede ver un grafo más o menos equivalente en cuanto a características con el de la estrategia de tareas por filas. Es un TDG perfectamente paralelo donde hay algunas tareas intermedias que tienen un coste mayor que las de los extremos, suponemos que por la misma razón: los píxeles son inicialmente descartados.

De la misma manera el grafo de la figura 1.7 posee características similares al de la figura 1.4: es un grafo completamente secuencial. La razón es equivalente a la del otro caso.

Sin embargo, el TDG mostrado por la figura 1.8 tiene algunas diferencias con su antecesor (el de la figura 1.5) principalmente porque las tareas de menos peso tienen dependencias entre ellas. Suponemos que este cambio es debido a que el histograma debe mantener coherencia también sobre los píxeles descartados y al haber muchas más tareas esta coherencia se puede perder. Otra diferencia es que las tareas de más peso forman una rama secuencial dentro del grafo.

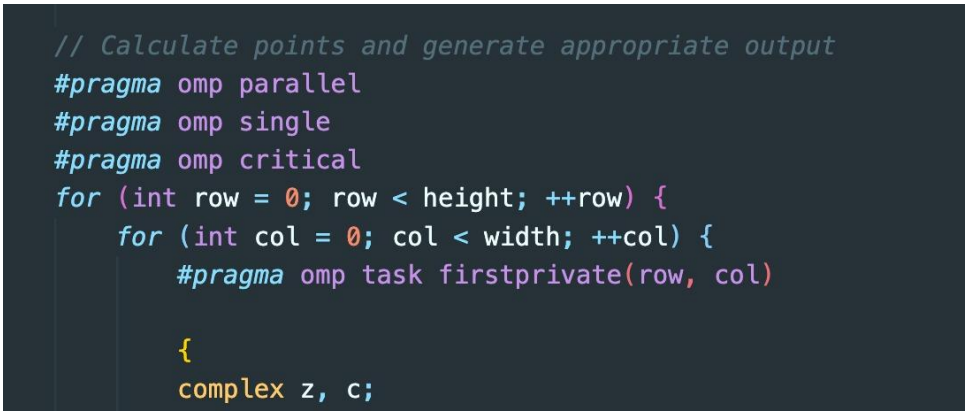
En general, al haber usado como tareas las iteraciones de un “for” interior al de row, hay muchísimos más nodos en el grafo. Esto provoca un tiempo de sincronización mucho más alto que al usar la estrategia de filas.

2 - Implementing task decompositions in OpenMP

En esta sección exploraremos las diferentes maneras de usar las opciones que nos ofrece OpenMP y analizaremos las distintas formas de ejecutar el código y su escalabilidad. Usaremos la versión del código “mandel-omp” para estos experimentos.

2.1 - Point strategy implementation using task

Para que el código funcionase correctamente hemos añadido el pragma `#critical` ya que es el que mantiene más restrictivo el paralelismo haciendo así que las variables estén seguras de no ser machacadas. La parte modificada se puede observar en la figura 2.1:



```
// Calculate points and generate appropriate output
#pragma omp parallel
#pragma omp single
#pragma omp critical
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)

        {
        complex z, c;
```

Figura 2.1: modificación del código mandel-omp

Después de probar que funcionaba bien con 1 y 2 threads hemos hecho pruebas de tiempo con uno y ocho threads (todas con 1000 iteraciones). Hemos optado por hacer dos experimentos: uno con el pragma critical fuera de los “for”s y otro dentro. Los resultados se pueden ver en las siguientes figuras:

pragma critical en el interior

```
par2311@boada-1:~/lab3-original$ OMP_NUM_THREADS=1 ./mandel-omp -o

Computation of the Mandelbrot set with:
    center = (0, 0)
    size = 2
    maximum iterations = 1000

Total execution time (in seconds): 0.426461

Mandelbrot set: Computed
Histogram for Mandelbrot set: Not computed
Writing output file to disk: output_omp_1.out

par2311@boada-1:~/lab3-original$ OMP_NUM_THREADS=8 ./mandel-omp -o

Computation of the Mandelbrot set with:
    center = (0, 0)
    size = 2
    maximum iterations = 1000

Total execution time (in seconds): 2.239170

Mandelbrot set: Computed
Histogram for Mandelbrot set: Not computed
Writing output file to disk: output_omp_8.out
```

Figura 2.2: Salida en la terminal de mandel-omp con critical en el interior

pragma critical en el exterior:

```
par2311@boada-1:~/lab3-original$ OMP_NUM_THREADS=1 ./mandel-omp -o

Computation of the Mandelbrot set with:
    center = (0, 0)
    size = 2
    maximum iterations = 1000

Total execution time (in seconds): 0.399666

Mandelbrot set: Computed
Histogram for Mandelbrot set: Not computed
Writing output file to disk: output_omp_1.out

par2311@boada-1:~/lab3-original$ OMP_NUM_THREADS=8 ./mandel-omp -o

Computation of the Mandelbrot set with:
    center = (0, 0)
    size = 2
    maximum iterations = 1000

Total execution time (in seconds): 1.137811

Mandelbrot set: Computed
Histogram for Mandelbrot set: Not computed
Writing output file to disk: output_omp_8.out
```

Figura 2.3: Salida en la terminal de mandel-omp con critical en el exterior

Se aprecian tiempos de ejecución diferentes en las dos versiones, resumidos en la siguiente tabla:

Texe (segundos)	**1 thread**	**8 threads**
Critical interior	0.426461	2.239170
Critical exterior	0.399666	1.137811

Se puede ver que a medida que se aumentan los threads la diferencia entre colocar el critical dentro o fuera es mayor. Esto es debido a que como se ha demostrado en apartados anteriores, el número de tareas aumenta si las tareas se conforman dentro o fuera de los “for”s provocando un aumento del tiempo de sincronización. Al usar critical el tiempo de overhead se ve muy afectado a más tareas haya.

En la siguiente figura se muestra el tiempo de ejecución y el speedup a medida que se aumentan los threads (esta vez con 10000 iteraciones):

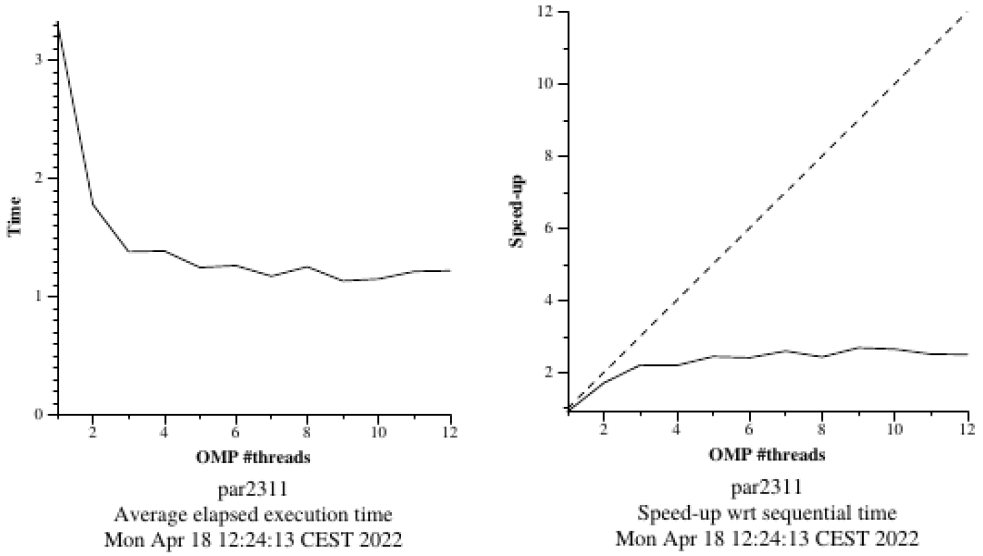


Figura 2.4: Gráficas que muestran la evolución del tiempo de ejecución y el speedup a medida que aumentan los threads.

Se puede ver que a partir de los 4~6 threads tanto el tiempo de ejecución como el speedup no varía mucho. Esto nos dice que el número óptimo de threads con los que ejecutar el código está entre estos valores, ya que al usar más no estamos consiguiendo ninguna mejora contundente, es más, el utilizar más threads quizá sea peor por los recursos usados de la máquina.

Para conocer más exactamente qué está ocurriendo en la ejecución hemos usado Paraver para ver la carga de trabajo de cada thread. Se puede ver en la figura 2.5 que el thread número 4 ha sido el responsable en la creación de las tareas y los demás las han ejecutado:

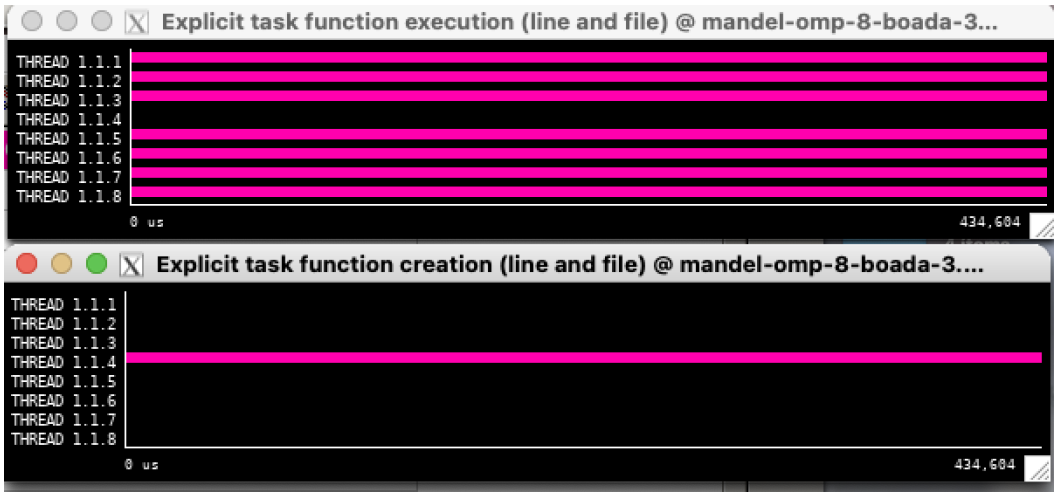


Figura 2.5: Gráficas del Paraver que muestran el trabajo de los threads.

Tras hacer otra prueba de ejecución ha pasado exactamente lo mismo pero esta vez otro thread ha sido el que ha creado las tareas, concretamente el cinco. En la tabla de la figura 2.6 se muestran las tareas realizadas de cada thread en el trabajo especificado:

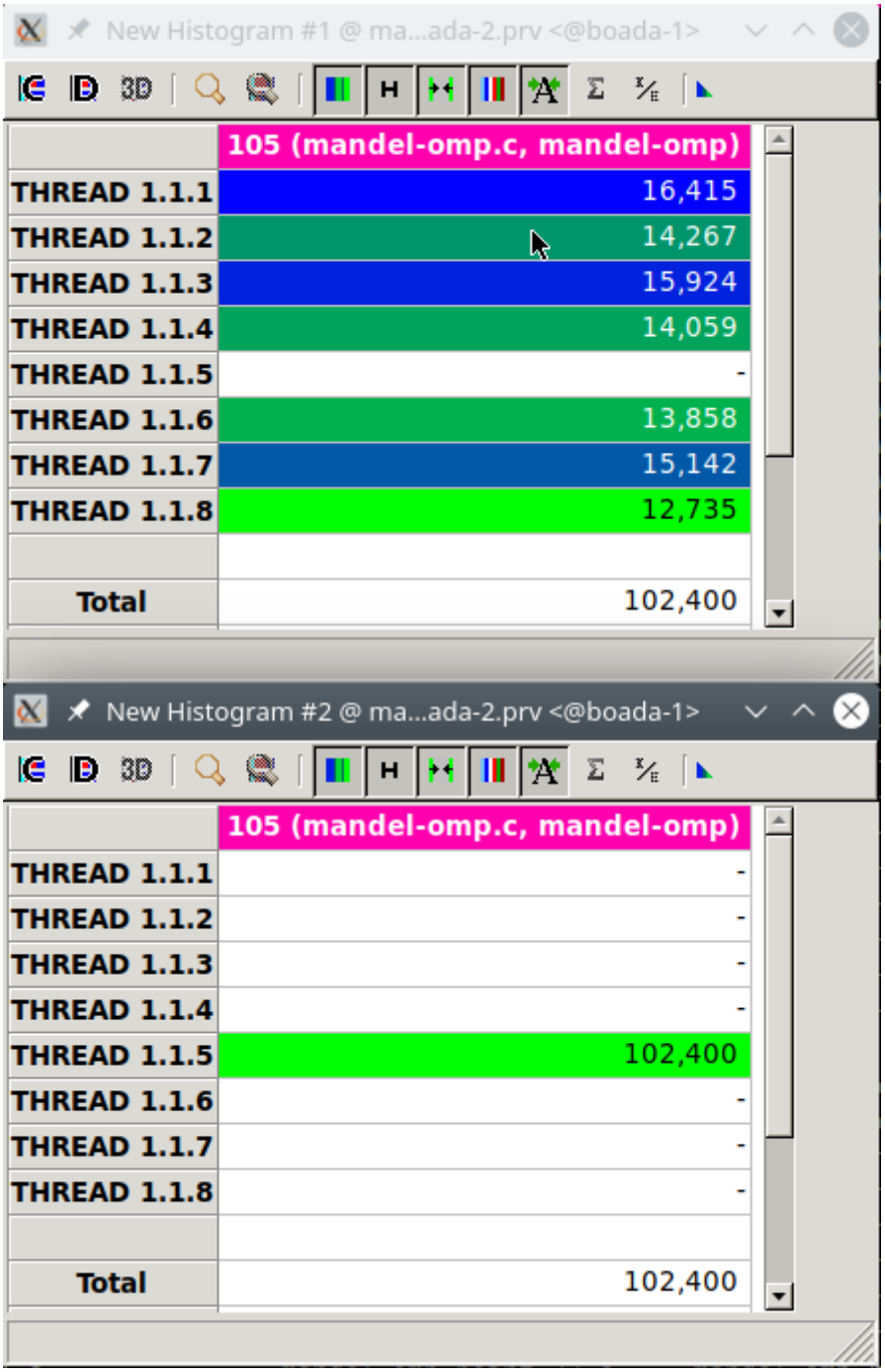


Figura 2.6: Tablas con los números de tasks realizados por cada thread.

Como se puede ver, el trabajo realizado por el thread que crea las tareas es mucho mayor al resto, esto no es nada balanceado.

Se puede ver más claro en el siguiente histograma (obtenido de la primera ejecución):

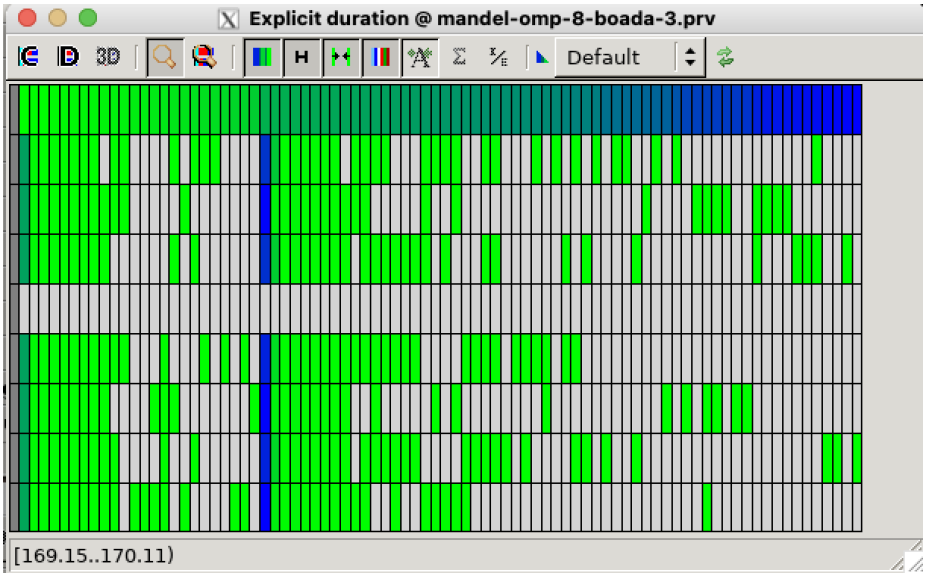
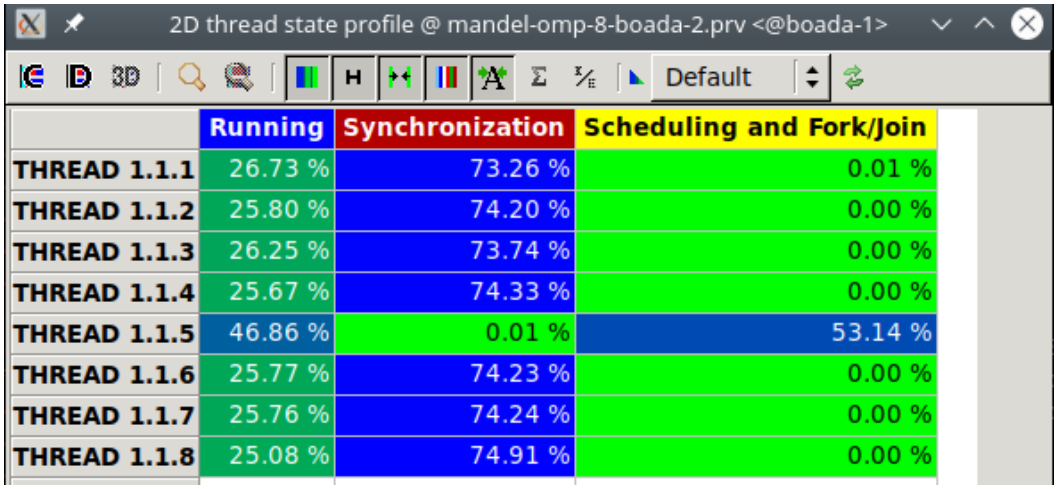


Figura 2.7: Histograma con el número de tareas ejecutadas por cada thread.

Hay una zona en medio (la línea vertical azul que indica muchas tareas) donde aparentemente hay una especie de cuello de botella que podría ser perfectamente evitado si se aprovechara más el tiempo que emplea el thread número 4.

Finalmente, queremos ver si realmente es despreciable o no el tiempo de sincronización así que usaremos Paraver para descubrirlo. En la figura 2.8 podemos observar una tabla con estos mismos datos:



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	26.73 %	73.26 %	0.01 %
THREAD 1.1.2	25.80 %	74.20 %	0.00 %
THREAD 1.1.3	26.25 %	73.74 %	0.00 %
THREAD 1.1.4	25.67 %	74.33 %	0.00 %
THREAD 1.1.5	46.86 %	0.01 %	53.14 %
THREAD 1.1.6	25.77 %	74.23 %	0.00 %
THREAD 1.1.7	25.76 %	74.24 %	0.00 %
THREAD 1.1.8	25.08 %	74.91 %	0.00 %

Figura 2.8: Tabla con los porcentajes del tiempo de cada thread.

Aparentemente el programa está de media un 74% del tiempo sincronizándose. Esto es una vergüenza para el tiempo de ejecución final, claramente el código no es del todo eficiente con demasiados threads en esta versión.

2.2 - Point strategy with granularity control using taskloop

En esta parte usaremos el taskloop para controlar mejor la granularidad y creación de tareas de los “for”s.

```
// Calculate points and generate appropriate output
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
        #pragma omp taskloop
    for (int col = 0; col < width; ++col){
        #pragma omp task firstprivate(col)
        {
        complex z, c;

        z.real = z.imag = 0;
```

Figura 2.9: Modificación básica de mandel-omp para incluir taskloop en “Point” (row + col)

Haremos los mismos experimentos que usando critical. Estos son los resultados:

Salida en la Terminal:

```
par2311@boada-1:~/lab3-original$ OMP_NUM_THREADS=1 ./mandel-omp -o

Computation of the Mandelbrot set with:
    center = (0, 0)
    size = 2
    maximum iterations = 1000

Total execution time (in seconds): 0.401138

Mandelbrot set: Computed
Histogram for Mandelbrot set: Not computed
Writing output file to disk: output_omp_1.out
```

```
par2311@boada-1:~/lab3-original$ OMP_NUM_THREADS=8 ./mandel-omp -o

Computation of the Mandelbrot set with:
    center = (0, 0)
    size = 2
    maximum iterations = 1000

Total execution time (in seconds): 0.960643

Mandelbrot set: Computed
Histogram for Mandelbrot set: Not computed
Writing output file to disk: output_omp_8.out
```

Figura 2.10: Salida de la terminal del nuevo código con 1 y 8 threads.

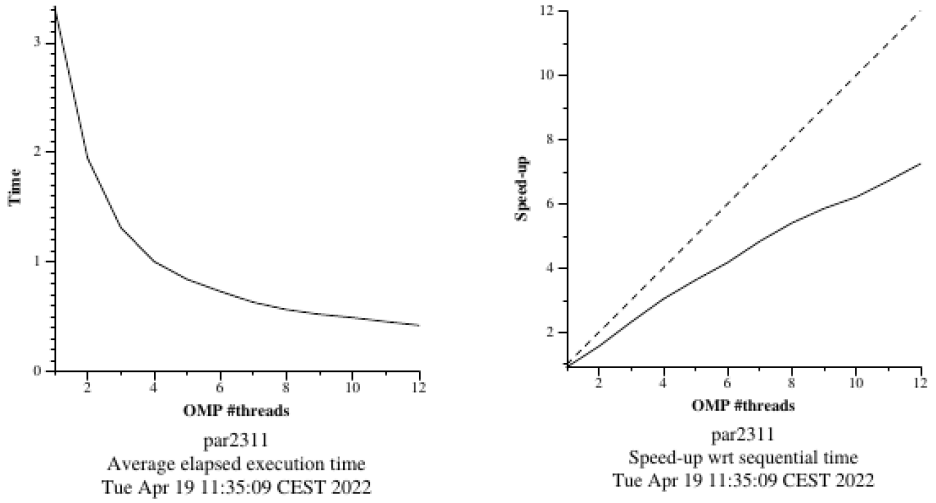


Figura 2.11: Gráficas que muestran el tiempo de ejecución y el speedup con diferentes threads.

Se puede apreciar una clara diferencia con los resultados obtenidos al usar critical. En este caso el tiempo de ejecución sigue bajando (aunque cada vez menos, esta vez no se queda linealmente “encallado”) y el speedup sigue aumentando. Ninguna de las dos gráficas se queda en un estado fijo. Esto es una buena señal de que este método es mejor escalable que el anterior.

Al igual que en el caso de critical, observaremos los histogramas de Paraver para así visualizar mejor qué está pasando:

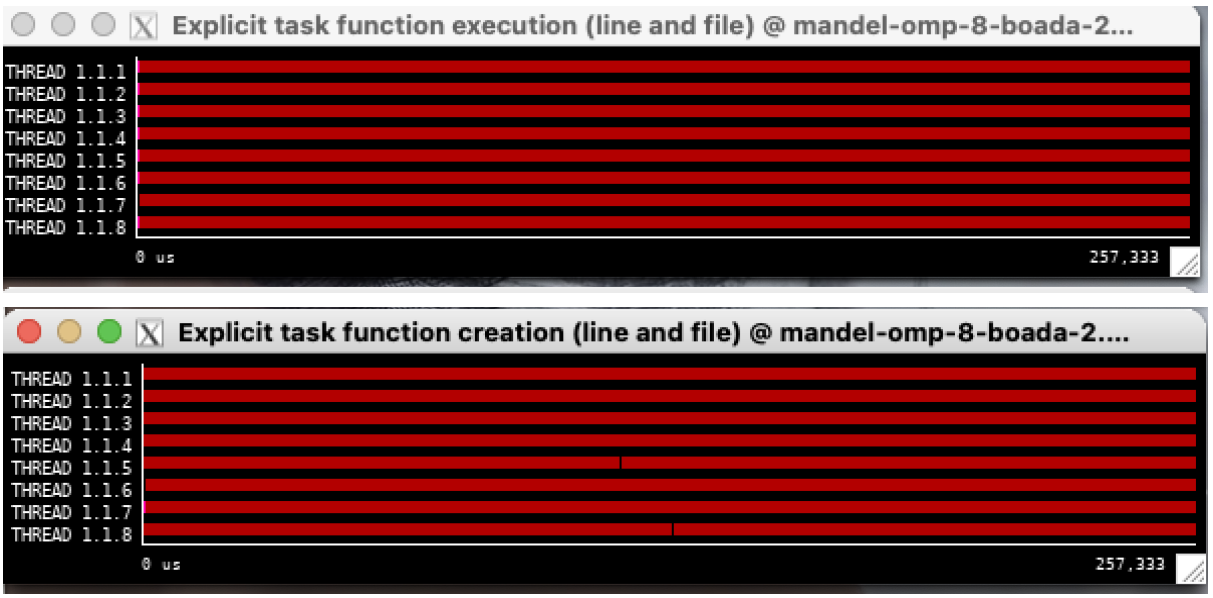


Figura 2.12: Histogramas de creación y ejecución de tareas de los threads.

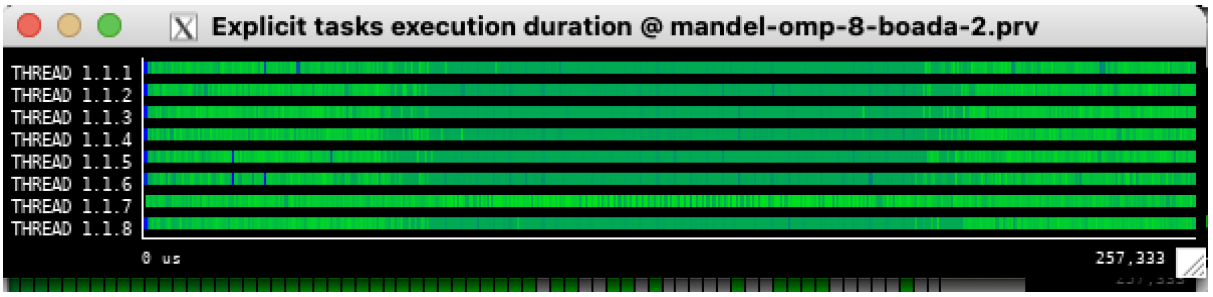


Figura 2.13: Histograma con la duración de las tareas de cada thread.

Por lo que se ve en las figuras 2.11, 2.12 y 2.13 ahora se está aprovechando todo el tiempo de los threads y no hay "tiempo muerto" perdido como ocurría usando critical. De momento esta versión del código tiene mejor rendimiento que la primera.

Para comprobar si es necesario proteger las variables de los diferentes grupos de tareas hemos ejecutado el código con y sin la cláusula `nogroup`. los resultados están en las figuras 2.15 y 2.16:

```
// Calculate points and generate appropriate output
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
        #pragma omp taskloop nogroup firstprivate(row)
    for (int col = 0; col < width; ++col){
        // #pragma omp task firstprivate(col)
        {
        complex z, c;

        z.real = z.imag = 0;
```

Figura 2.14: Modificación del código para incluir `nogroup`.

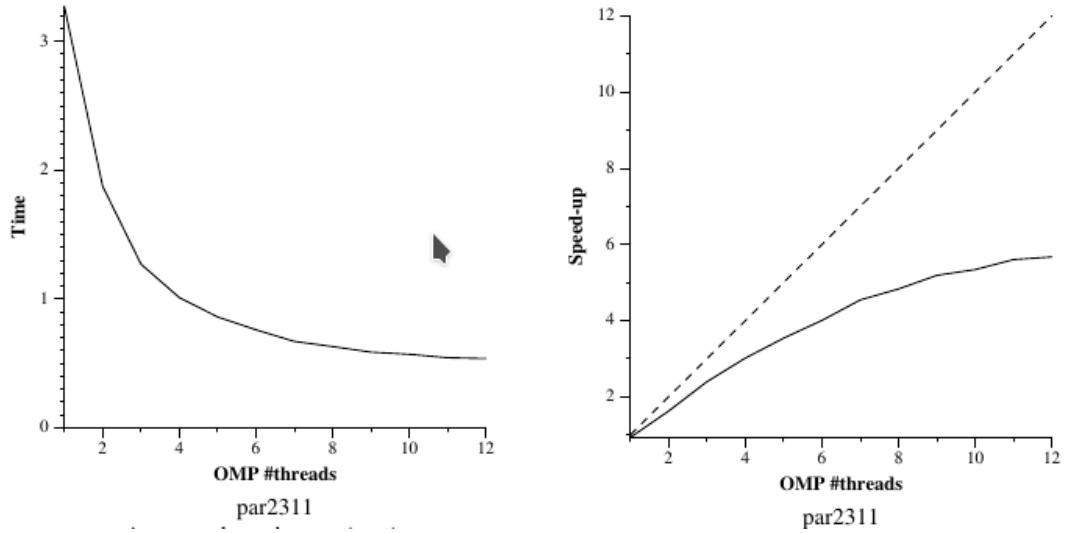


Figura 2.15: Gráficas del tiempo de ejecución y Speedup según el número de threads (sin nogroup)

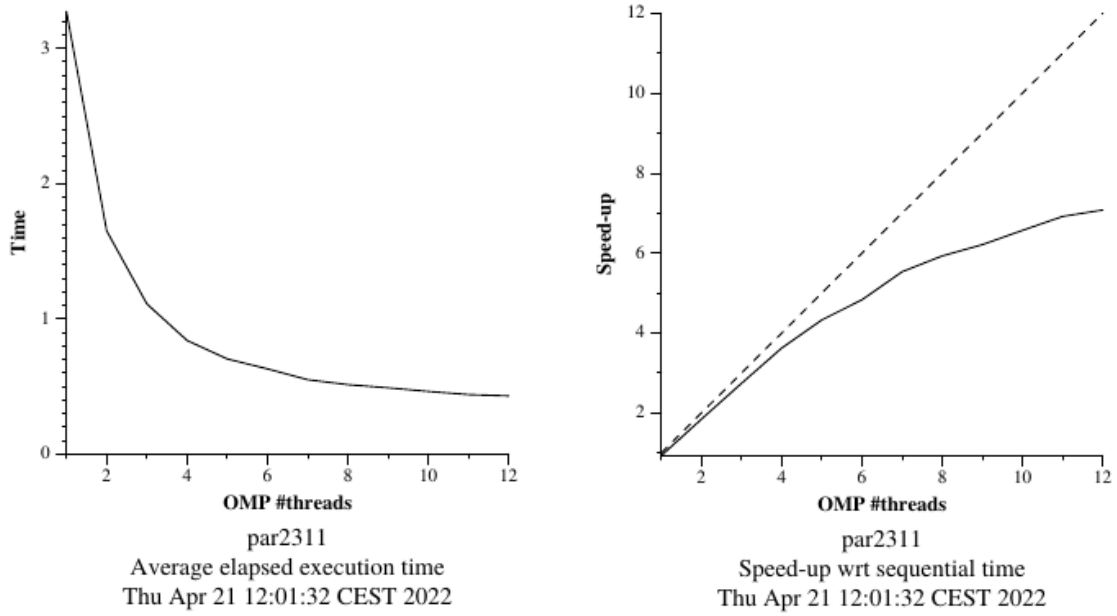


Figura 2.16: Gráficas del tiempo de ejecución y Speedup según el número de threads (con nogroup)

Hay una pequeña diferencia entre las dos figuras, el resultado es que al usar `nogroup` el tiempo de ejecución es un pelín menor. Esto es debido a que al estar las tareas agrupadas hay menos tiempo de sincronización en general. De este pequeño cambio podemos concluir que en este caso en concreto es útil usar `nogroup`.

Para concluir este experimento hemos analizado con Paraver los histogramas de creación y ejecución de tareas y se puede observar lo siguiente:

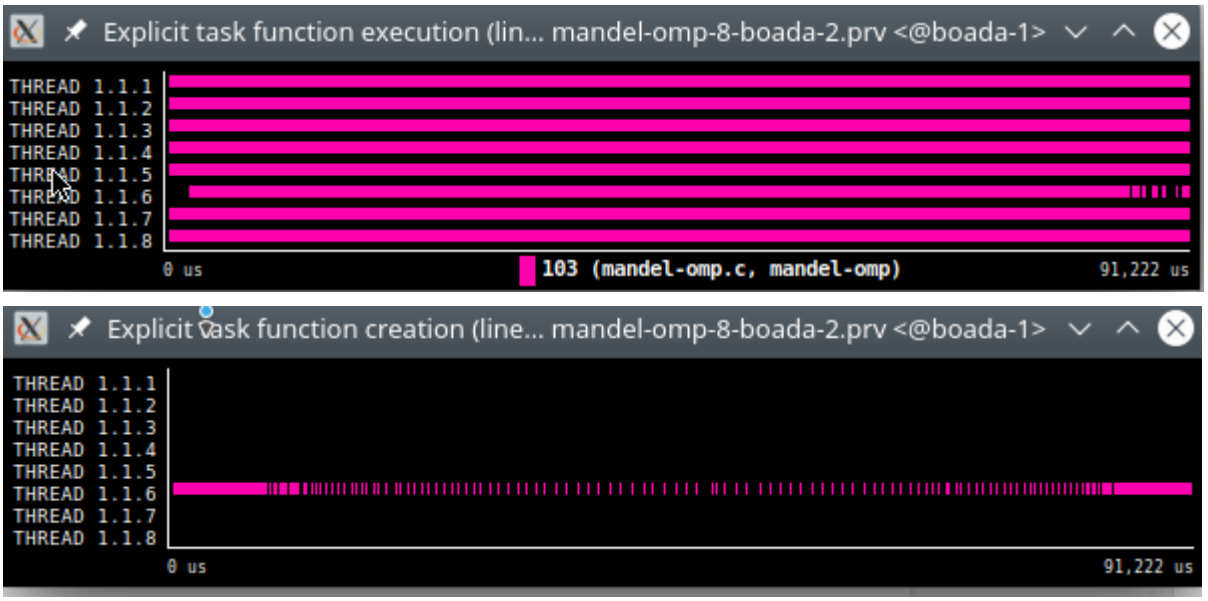


Figura 2.17: Histogramas de la creación y ejecución de tareas de Paraver usando nogroup.

En este caso se aprecia que el thread encargado de crear las tasks (el número 6) también está trabajando en la ejecución intermitentemente. Con esta pequeña modificación estamos aprovechando mucho mejor el tiempo que en la primera versión del código.

2.3 - Row strategy implementation

Tras analizar *Point decomposition* para el programa de computación del conjunto de Mandelbrot en la sección previa, ahora haremos lo mismo pero con *Row decomposition*, así descubriremos si esta es otra alternativa válida.

```
// Calculate points and generate appropriate output
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
        int col = 0;
    #pragma omp task firstprivate(row) private(col)
    for (int col = 0; col < width; ++col){
        // #pragma omp task firstprivate(col)
        {
```

Figura 2.18: Modificación del código mandel-omp para hacer la descomposición por filas.

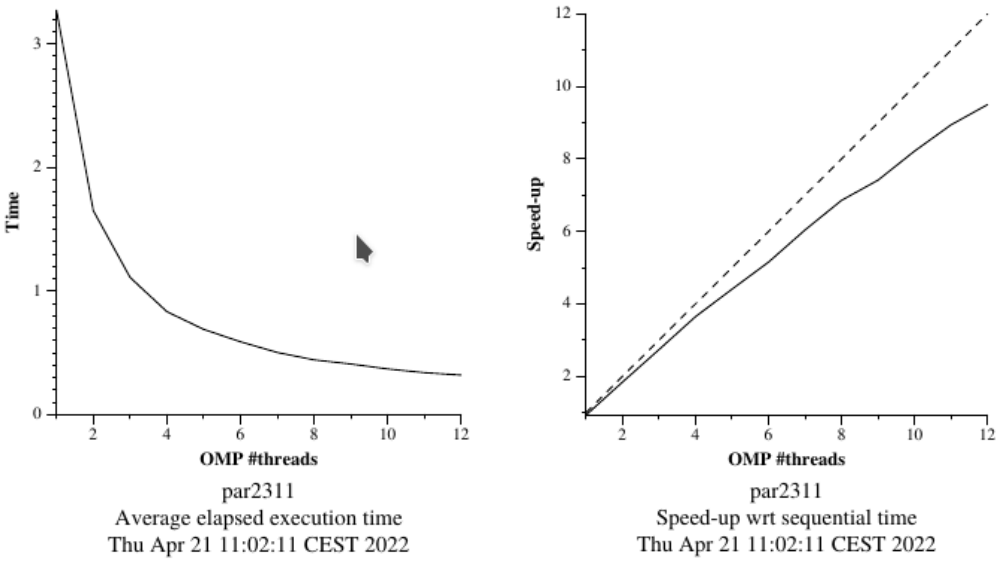


Figura 1.19: Gráficas que muestran el tiempo de ejecución y el Speedup de esta versión.

Como ya es costumbre, primero y más importante y principal de todo analizaremos el tiempo de ejecución y la escalabilidad. Se puede observar que las gráficas son menos planas que las anteriores, es decir, tanto el tiempo de ejecución como el speedup van mejorando mucho a medida que aumentan los threads. Esto es muy buena señal de que esta versión del código es bien escalable.

A continuación en la figura 2.20 se puede ver que el thread de la creación de tareas está haciendo eso mismo al principio del tiempo de ejecución total. Esto permite que el resto del tiempo sea exclusivamente utilizado para la ejecución como se puede observar en el histograma de arriba.

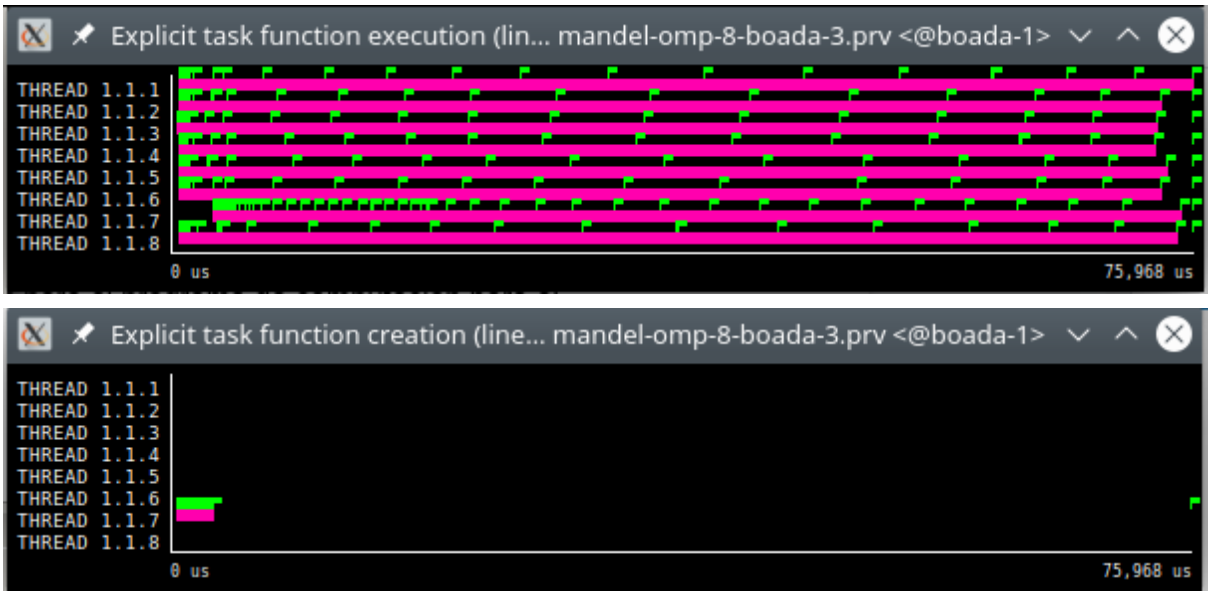
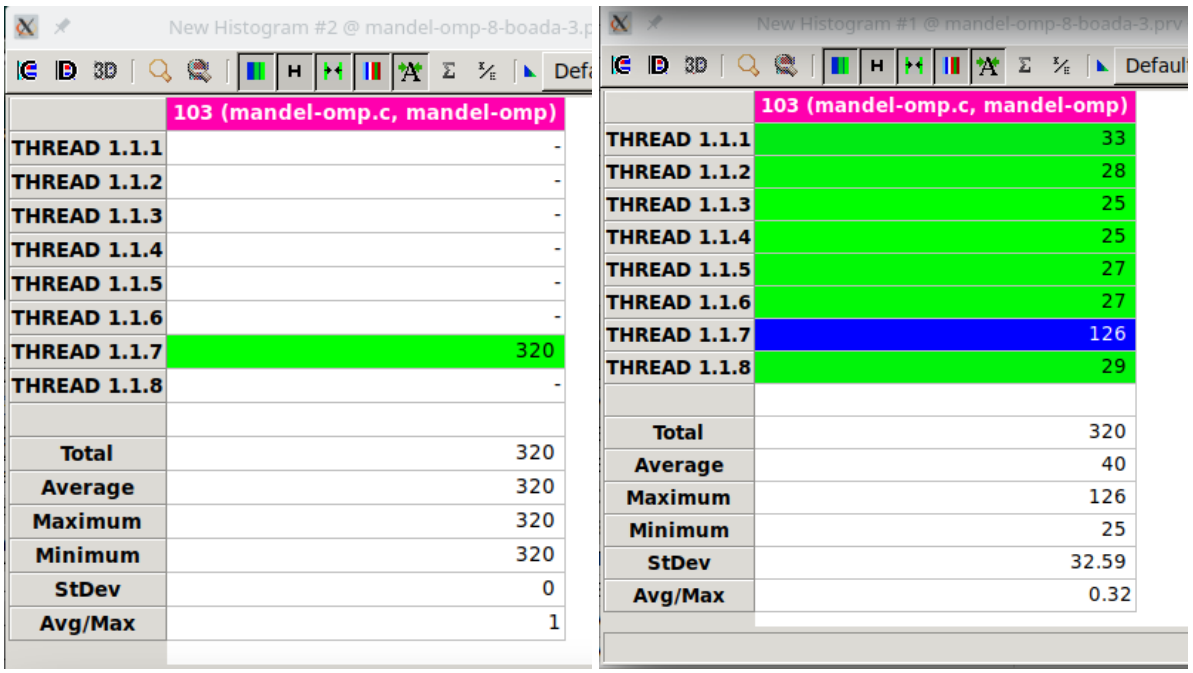


Figura 2.20: Histogramas con Paraver de la creación y ejecución de tareas.



	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	-
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	320
THREAD 1.1.8	-
Total	320
Average	320
Maximum	320
Minimum	320
StDev	0
Avg/Max	1

	103 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	33
THREAD 1.1.2	28
THREAD 1.1.3	25
THREAD 1.1.4	25
THREAD 1.1.5	27
THREAD 1.1.6	27
THREAD 1.1.7	126
THREAD 1.1.8	29
Total	320
Average	40
Maximum	126
Minimum	25
StDev	32.59
Avg/Max	0.32

Figura 2.21: Tablas con el número de tareas creadas y ejecutadas de cada thread.

En la figura 2.21 se aprecia que hay Load Unbalance, el thread número siete ejecuta 100 tareas más que los demás, esto es debido a que también es el encargado de la creación de tasks.

En esta modificación del código no se puede usar `grainsize` para mejorar la eficiencia ya que no hemos usado la cláusula `taskgroup`.

3 - Optional

Antes de terminar, queremos analizar cuál es el número de tasks óptimo para que el programa sea lo mejor posible.

Para ello usaremos un script que se nos ha otorgado (submit-numtasks-omp.sh) y veremos los resultados obtenidos con las dos versiones (Row strategy y Point strategy).

El script lo que hace es ejecutar el código con diferentes números de tasks creados, así obtendremos todos los resultados de golpe.

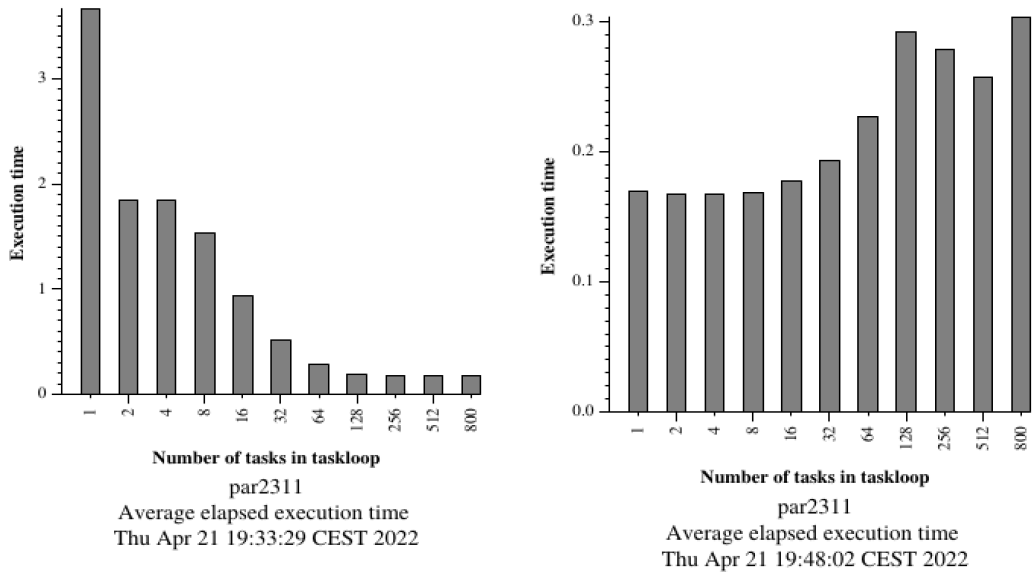


Figura 3.1: Gráficos que muestran el tiempo de ejecución según el número de tasks creados para ello (Izquierda Row Strategy, Perecha Point Strategy).

No hay duda al ver estos gráficos que para la estrategia de Filas el número de tasks óptimo es 128 ya que a partir de este número el tiempo de ejecución no disminuye lo suficiente como para ser realmente importante.

Sin embargo, para la estrategia enfocada a los Puntos el tiempo de ejecución aumenta desmesuradamente a más tasks hay, esto seguramente sea provocado por los tiempos de sincronización. No es escalable por el número de tareas.

La diferencia principal entre los dos gráficos es que uno disminuye mientras que el otro aumenta.

4 - Conclusions

Hemos visto que según lo que se considere una tarea el tiempo de ejecución de los programas y su granularidad puede variar muchísimo, por eso es importante antes de actuar pararse y analizar un poco el código

En el mundo real no es posible tener tanta variedad de opciones en tu máquina como la que nos ofrece boada pero aún así esto que hemos aprendido no solo sirve en programación e informática sinó también en muchas otras cosas que pueden ser similares como por ejemplo la organización de trabajos de una empresa o la cadena de creación y fabricación en una fábrica.

Es muy recomendable en la paralelización considerar todas las opciones y posibilidades de análisis y esto es muy útil en muchos ámbitos de trabajo.