PAR Laboratory Assignment
Lab 3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

Natalia Dai
Daniel Ruiz Jiménez

Username: par2306
par2317
Date: 20/4/2022
Spring 2021-22

ÍNDICE

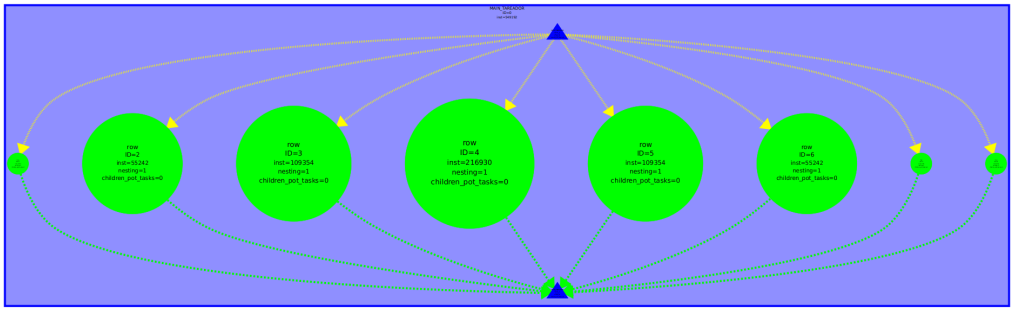
INTRODUCCIÓN

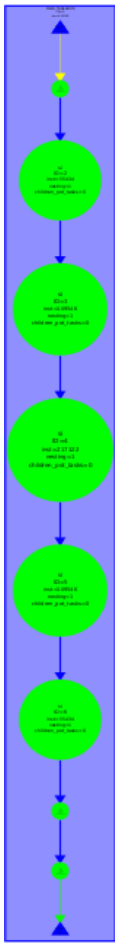
En este laboratorio (dividido en dos sesiones) emplearemos la librería OpenMP para realizar varias versiones de un mismo código: el set de Mandelbrot. Analizaremos el código y aplicaremos varias estrategias de paralelismo para intentar mejorar su rendimiento, usando el conocimiento adquirido de las sesiones anteriores, como el Tareador para descomponer código en grafos de dependencias y Paraver para interpretar gráficos.

1. Task decomposition analysis for the Mandelbrot set computation

Row Strategy

Para esta estrategia, ejecutaremos varias veces el archivo mandel-tar.c. Hemos escrito dos líneas adicionales para empezar (tareador_start_task()) y finalizar (tareador_end_task()) tareas, para poderlas ver con el Tareador. La primera ha sido entre los dos fors, y la segunda antes del penúltimo corchete (dentro del segundo for).
Podemos observar que en el grafo de dependencias, todas las tareas se ejecutan en paralelo. No obstante, las tareas tienen diferentes tamaños, por lo tanto, algunas terminarán antes que otras.

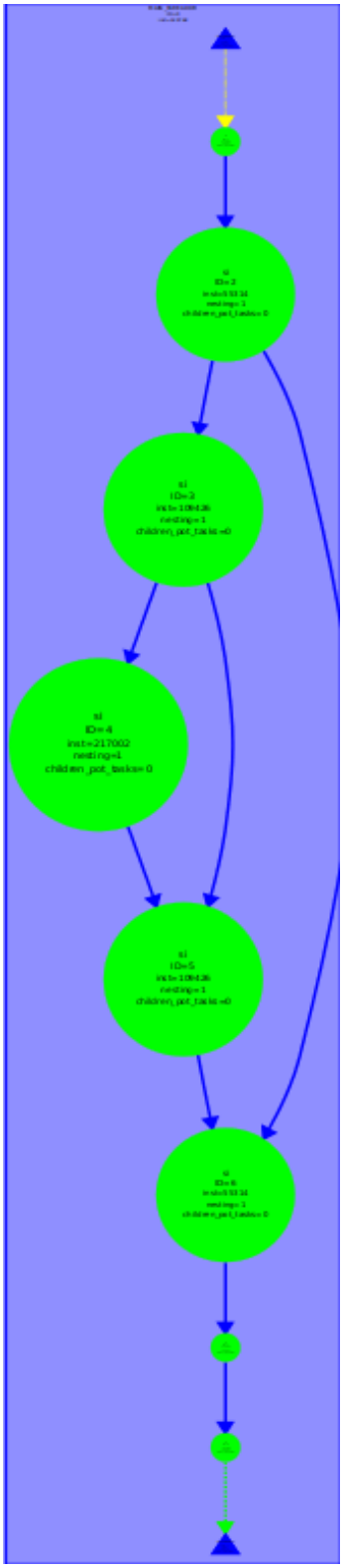




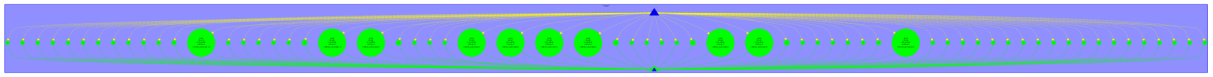
Con la opción **-d**, el código se serializa debido a una dependencia, así que no tenemos paralelismo. Esto se debe a que, cuando ejecutamos el programa con esta opción, se ejecuta un if del código que crea la dependencia (en concreto, la variable X11_COLOR_fake). Tenemos una data race ya que todos los threads quieren modificarla. Para solucionar este asunto, simplemente tenemos que usar una sentencia de sincronización de threads, en concreto "#pragma omp critical", aunque no sea la opción más eficiente, ya que conlleva un gran overhead.

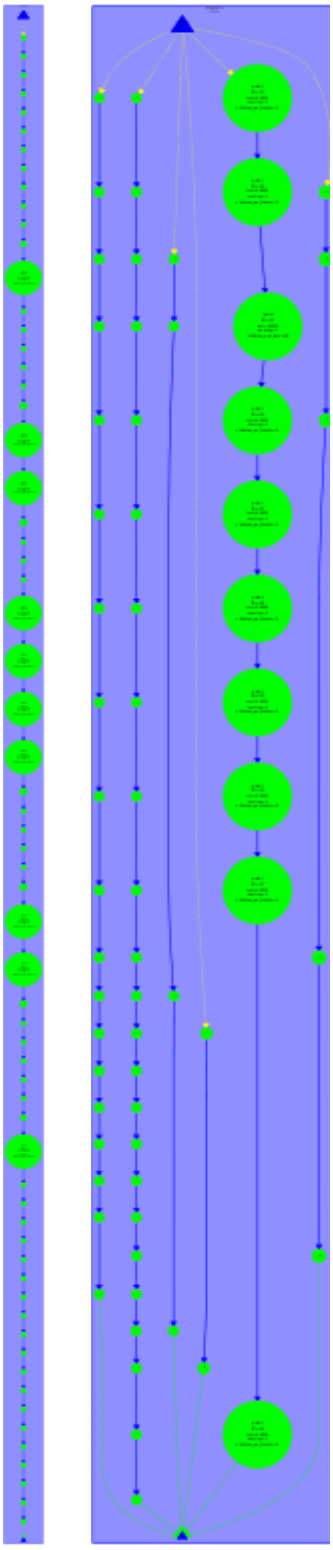
```
if (output2display) {
    /* Scale color and display point  */
    long color = (long) ((k-1) * scale_color) + min_color;
    if (setup_return == EXIT_SUCCESS) {
        XSetForeground (display, gc, color);
        XDrawPoint (display, win, gc, col, row);
    }
}
```

Para la última versión de esta estrategia, la **-h** usa una variable global llamada histogram a la que acceden todos los threads, y se crea una datarace “limpia”. Usando el mismo código de sincronización que con la estrategia anterior obtenemos el grafo de dependencias siguiente:



Point strategy





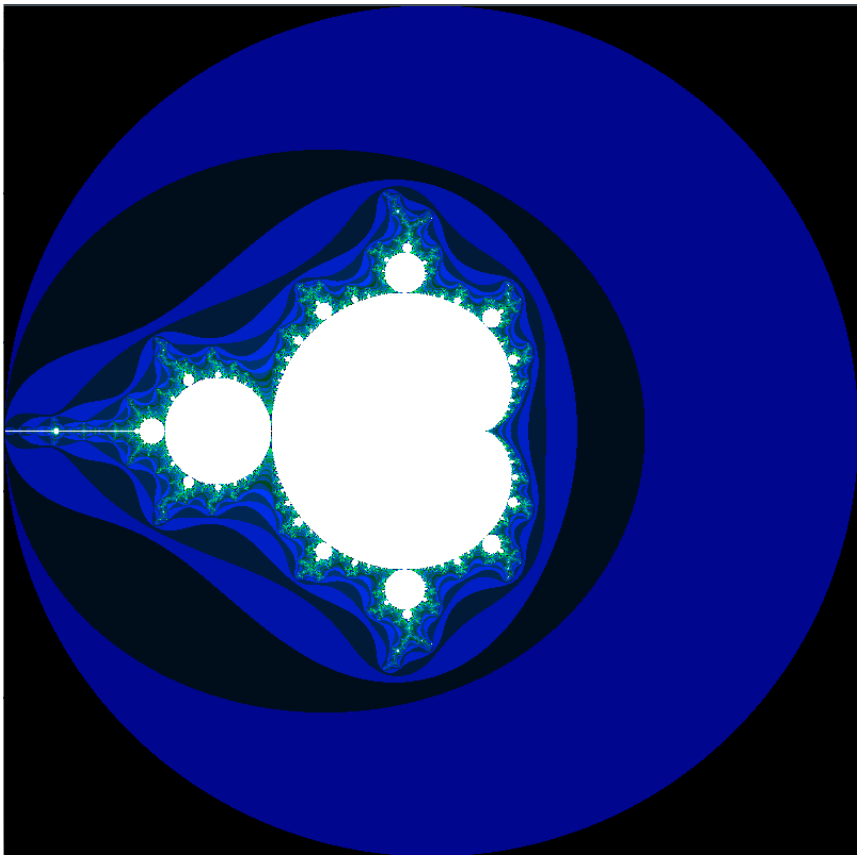
Para esta estrategia, cambiaremos el lugar de las llamadas a las funciones de empezar y terminar tareas para que cada tarea se corresponda a un punto del set de Mandelbrot. Las vamos a mover a lo más adentro del bucle. La ejecución sin opciones nos dibuja un grafo similar al de la estrategia Row pero en esta ocasión tenemos 8*8 = 64 tareas diferentes, todas en paralelo.

Aún así, sigue estando desbalanceado, ya que unas tareas son mayores que otras.

La opción **-d** (izquierda) nos muestra que el comportamiento anterior se repite, se crean dependencias y se pierde el paralelismo, y tenemos una cola muy larga de tareas.

La solución para la datarace es la misma, y lo mostramos en la opción **-h** (derecha). Hay un número mayor de tareas debido a que la granularidad es más alta.

2. Implementing task decompositions in OpenMP

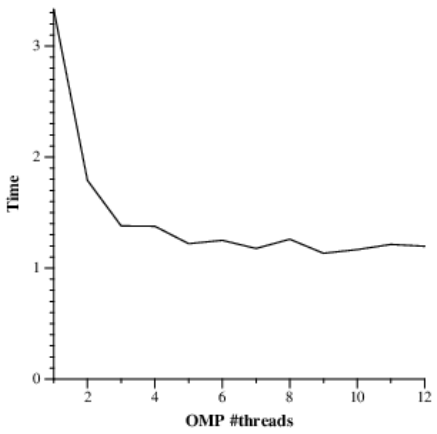


La imagen generada es la correcta pero al ser secuencial, la versión con 1 thread tarda más en generar la imagen que con 2 y con 8. Para la versión con 2 threads, al principio el programa se ejecutaba pero no mostraba el dibujo en pantalla, pero pusimos la cláusula critical y solucionamos el problema.

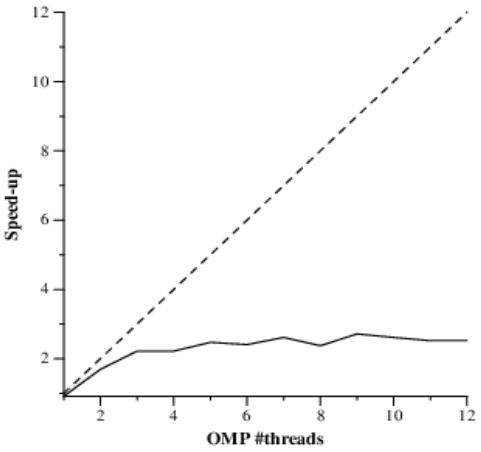
```
if (output2display) {
    /* Scale color and display point  */
    long color = (long) ((k-1) * scale_color) + min_color;
    #pragma omp critical
    if (setup_return == EXIT_SUCCESS) {
        XSetForeground (display, gc, color);
        XDrawPoint (display, win, gc, col, row);
    }
}
```

POINT TASK

Para la primera estrategia no cambiamos nada del fichero original, y los tiempos de ejecución son significativamente diferentes de acuerdo al diferente número de procesadores:
con 1 procesador 3.315496 segundos y con 8, 1.221644 segundos.

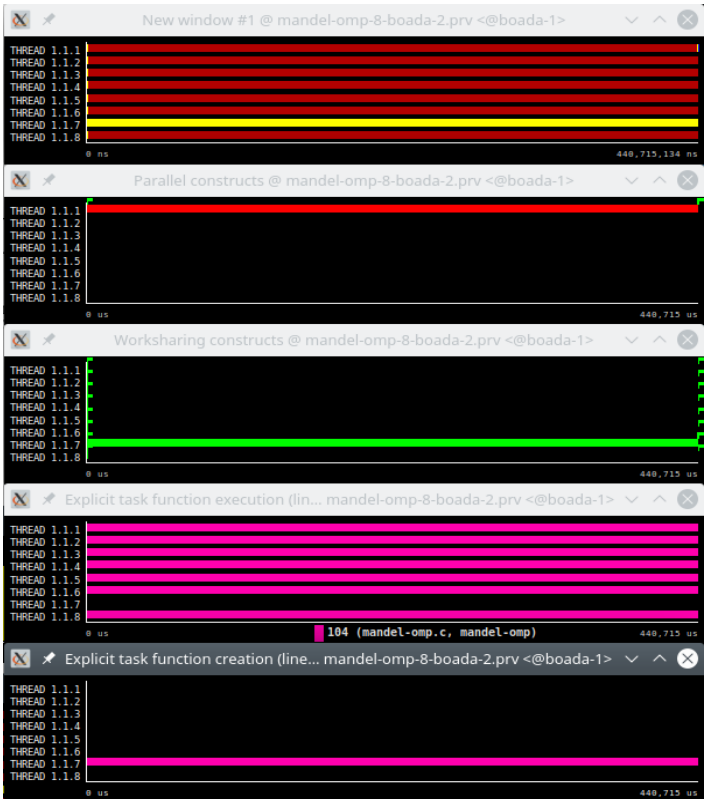


par2317
Average elapsed execution time
Fri Apr 8 12:41:11 CEST 2022



par2317
Speed-up wrt sequential time
Mon Apr 11 10:45:10 CEST 2022

En cuanto a la escalabilidad, podemos observar que no es muy buena. Cuando llegamos a los 5 procesadores, el tiempo de ejecución se vuelve más o menos constante, y el speedup es deficiente, ya que se desvía del patrón ideal lineal muy rápidamente.



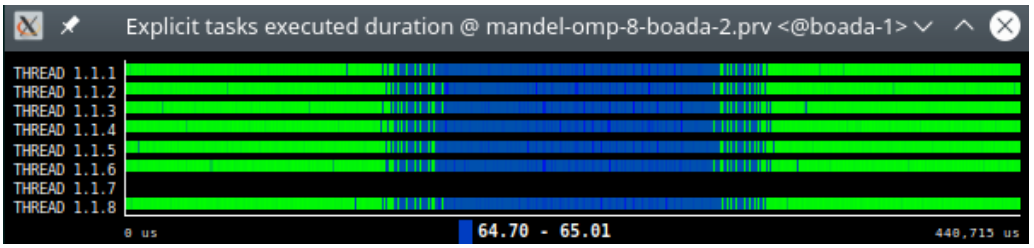
New Window, Parallel constructs, Worksharing constructs, Explicit task function execution & creation

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	25.85 %	74.14 %	0.01 %
THREAD 1.1.2	25.26 %	74.74 %	0.00 %
THREAD 1.1.3	25.33 %	74.66 %	0.00 %
THREAD 1.1.4	25.00 %	75.00 %	0.00 %
THREAD 1.1.5	25.75 %	74.25 %	0.00 %
THREAD 1.1.6	24.85 %	75.14 %	0.00 %
THREAD 1.1.7	47.63 %	0.00 %	52.37 %
THREAD 1.1.8	24.38 %	75.62 %	0.00 %
Total	224.06 %	523.56 %	52.39 %
Average	28.01 %	65.44 %	6.55 %
Maximum	47.63 %	75.62 %	52.37 %
Minimum	24.38 %	0.00 %	0.00 %
StDev	7.43 %	24.74 %	17.32 %
Avg/Max	0.59	0.87	0.13

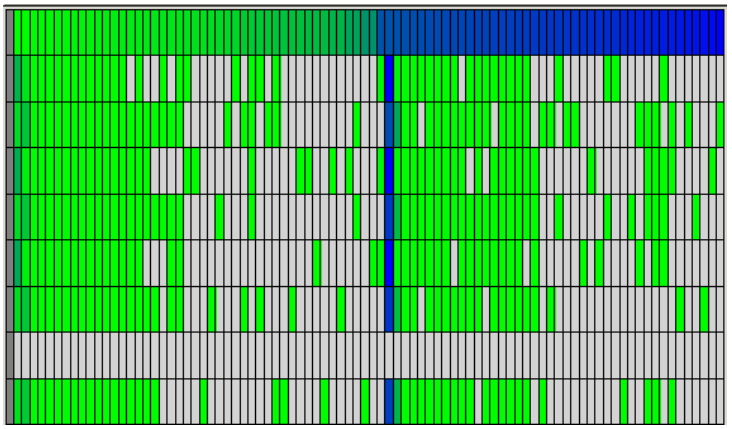
Histrograma de New Window

	104 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	15,905
THREAD 1.1.2	14,519
THREAD 1.1.3	15,464
THREAD 1.1.4	13,933
THREAD 1.1.5	16,295
THREAD 1.1.6	13,635
THREAD 1.1.7	-
THREAD 1.1.8	12,649
Total	102,400
Average	14,628.57
Maximum	16,295
Minimum	12,649
StDev	1,225.09
Avg/Max	0.90

Histrograma Explicit Tasks Function Execution de Point Task



Explicit tasks executed duration



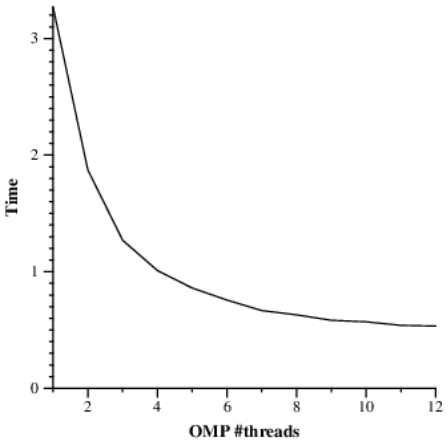
Histograma de la duración de tareas de Point Task

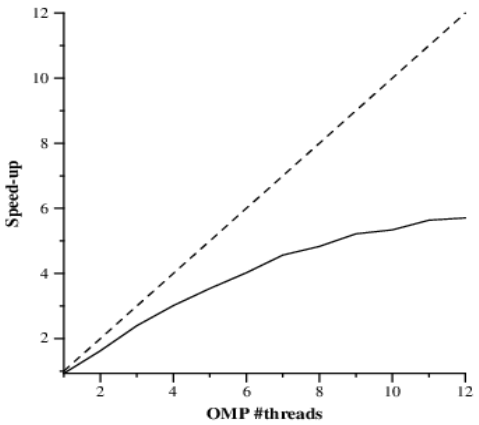
Viendo ahora los gráficos de paraver, podemos observar que hay un thread que se encarga exclusivamente de crear las tareas (en nuestro caso el thread 7) y el resto de threads las ejecutan, por lo que se pierde un poco de rendimiento, ya que no todos los threads se dedican a ejecutar las tareas. Esto se puede ver en el primer histograma. En el segundo, nos dice el número de tareas que ejecuta cada thread, unas 14628 tareas por thread de media. Podemos observar que el thread que crea las tareas no ejecuta ninguna, como es lógico.

Finalmente, tenemos el gráfico y el histograma de la duración de las tareas. Si nos fijamos en el gráfico, hay una zona azul en el medio de la ejecución y verde en los extremos. Esto significa que las tareas que se crean en el medio de la ejecución tardan más que las de los lados. Por este motivo, cuando ejecutamos el programa y se dibuja el Mandelbrot, las partes de arriba y abajo se generan rápido y las del centro tardan más.

POINT TASKLOOP:

Para la segunda estrategia cambiamos la sección paralela al bucle exterior y escribimos #pragma omp taskloop firstprivate(row). Los tiempos de ejecución denotan mejoría en tiempo: con un procecsador 3.260644 segundos y con 8, 0.624581 segundos.





Pasando a la escalabilidad, hay una mejoría pequeña pero notable. Esto confirma que estamos usando una estrategia mejor.

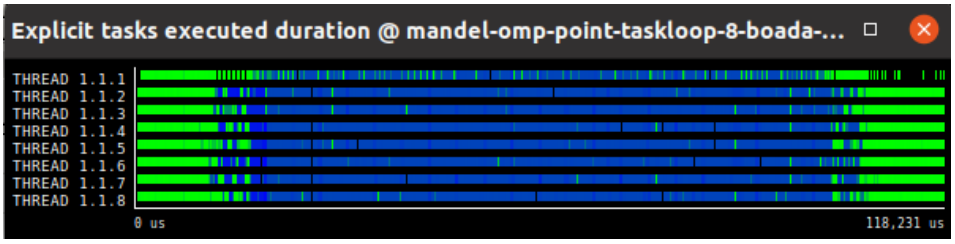
New Window, Parallel constructs, Worksharing constructs, Explicit task function execution & creation

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	53.06 %	25.78 %	21.16 %
THREAD 1.1.2	68.17 %	31.82 %	0.01 %
THREAD 1.1.3	67.14 %	32.85 %	0.01 %
THREAD 1.1.4	66.91 %	33.09 %	0.01 %
THREAD 1.1.5	66.91 %	33.08 %	0.01 %
THREAD 1.1.6	67.31 %	32.69 %	0.01 %
THREAD 1.1.7	69.01 %	30.98 %	0.01 %
THREAD 1.1.8	67.15 %	32.84 %	0.01 %
Total	525.65 %	253.14 %	21.21 %
Average	65.71 %	31.64 %	2.65 %
Maximum	69.01 %	33.09 %	21.16 %
Minimum	53.06 %	25.78 %	0.01 %
StDev	4.83 %	2.32 %	7.00 %
Avg/Max	0.95	0.96	0.13

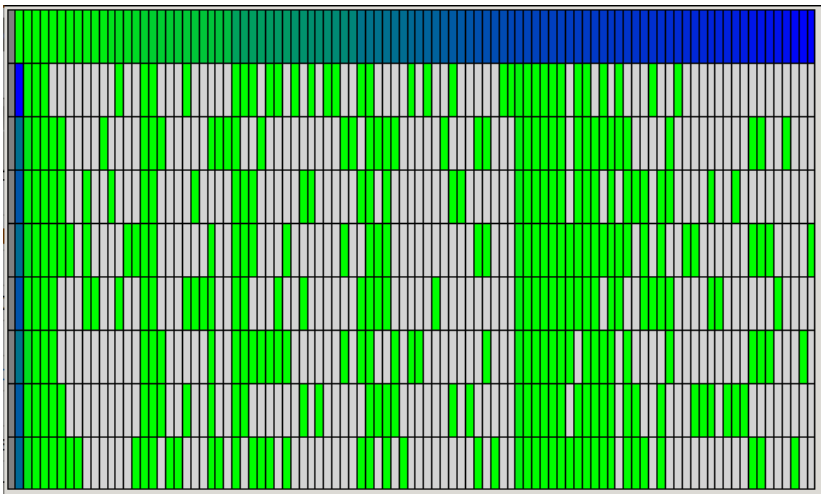
Histograma de New Window

	103 (mandel-o..skloop.c)
THREAD 1.1.1	4,282
THREAD 1.1.2	2,945
THREAD 1.1.3	3,123
THREAD 1.1.4	3,016
THREAD 1.1.5	3,157
THREAD 1.1.6	2,980
THREAD 1.1.7	3,066
THREAD 1.1.8	3,031
Total	25,600
Average	3,200
Maximum	4,282
Minimum	2,945
StDev	414.17
Avg/Max	0.75

Histograma de Explicit Tasks Function Execution de Point Taskloop



Explicit tasks executed duration



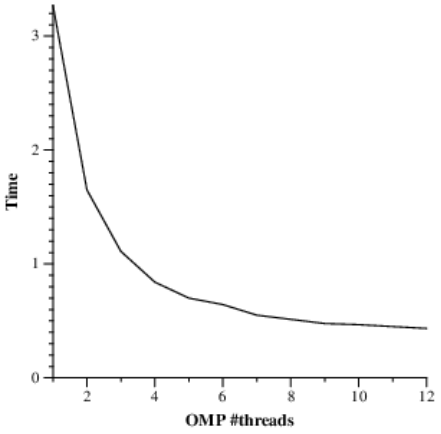
Histograma de la duración de tareas de Point Taskloop

En los gráficos de Paraver, podemos observar que el thread que antes se encargaba únicamente de crear tareas ahora también participa en la ejecución de las mismas de forma intercalada. Esta diferencia puede haber afectado al rendimiento. El primer histograma lo confirma, ahora el thread 1 tiene un % de región roja (sincronización) diferente de 0%.

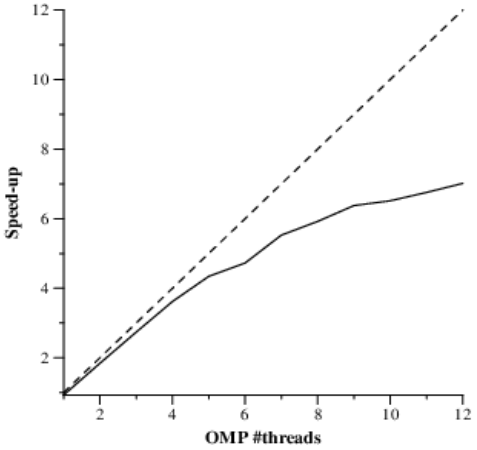
En el segundo histograma vemos que ahora la región azul está más extendida en el gráfico pero ahora el número de ráfagas por thread es mucho menor, ha bajado de una media de 14000 a una media de 3200.

POINT TASKLOOP NOGROUP:

En la siguiente revisión del código hemos introducido la cláusula nogroup. Esto permite al código quitar la agrupación de tareas implícita del código, y hará que al paralelizarlo gane un poco de eficiencia, como se puede ver en el tiempo de ejecución. Los tiempos en cuestión son los siguientes: 3.261381 s y 0.504881 para las versiones con 1 y 8 threads respectivamente.



par2317
Average elapsed execution time
Tue Apr 19 20:29:20 CEST 2022



par2317
Speed-up wrt sequential time
Tue Apr 19 20:29:20 CEST 2022

En cuanto a la escalabilidad, vemos que sigue mejorando, el tiempo de ejecución va bajando constantemente y a los 12 procesadores empieza a estabilizarse, mientras que el speedup se va juntando cada vez más a la línea imaginaria ideal.

New Window,

Parallel constructs,

Worksharing constructs,

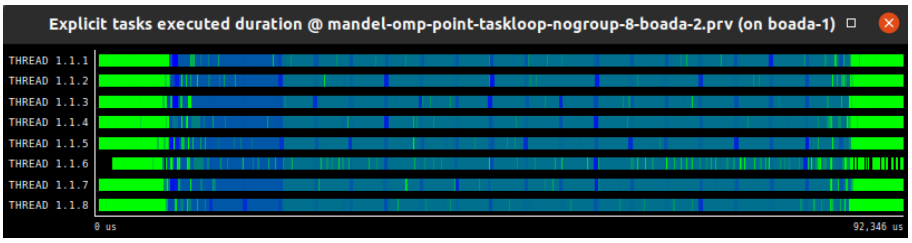
Explicit task function execution & creation

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	87.57 %	12.40 %	0.03 %
THREAD 1.1.2	87.68 %	12.31 %	0.01 %
THREAD 1.1.3	87.02 %	12.97 %	0.01 %
THREAD 1.1.4	87.16 %	12.83 %	0.01 %
THREAD 1.1.5	87.36 %	12.63 %	0.01 %
THREAD 1.1.6	78.21 %	0.12 %	21.67 %
THREAD 1.1.7	86.88 %	13.12 %	0.01 %
THREAD 1.1.8	87.53 %	12.46 %	0.01 %
Total	689.41 %	88.84 %	21.75 %
Average	86.18 %	11.10 %	2.72 %
Maximum	87.68 %	13.12 %	21.67 %
Minimum	78.21 %	0.12 %	0.01 %
StDev	3.02 %	4.16 %	7.16 %
Avg/Max	0.98	0.85	0.13

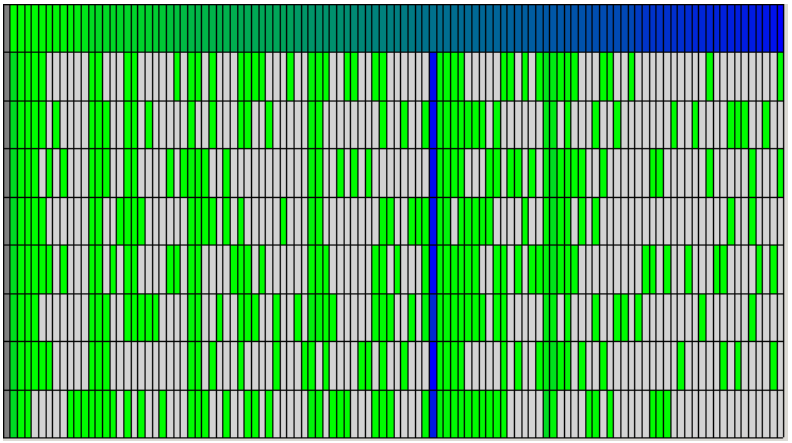
Histograma de New Window

	103 (mandel-o..ogroup.c)
THREAD 1.1.1	2,984
THREAD 1.1.2	3,363
THREAD 1.1.3	3,135
THREAD 1.1.4	3,514
THREAD 1.1.5	3,052
THREAD 1.1.6	2,924
THREAD 1.1.7	3,191
THREAD 1.1.8	3,437
Total	25,600
Average	3,200
Maximum	3,514
Minimum	2,924
StDev	203.22
Avg/Max	0.91

Histograma de Explicit Tasks Function Execution de Point Taskloop Nogroup



Explicit tasks executed duration

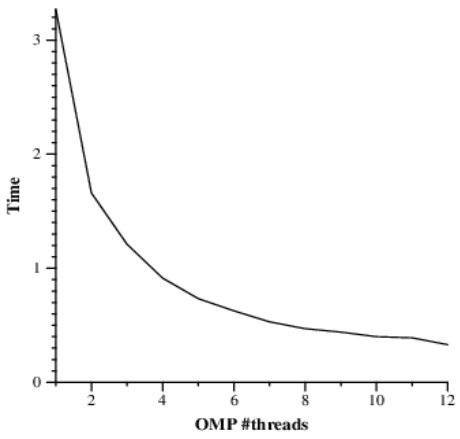


Histogramas de la duración de tareas de Point Taskloop Nogroup

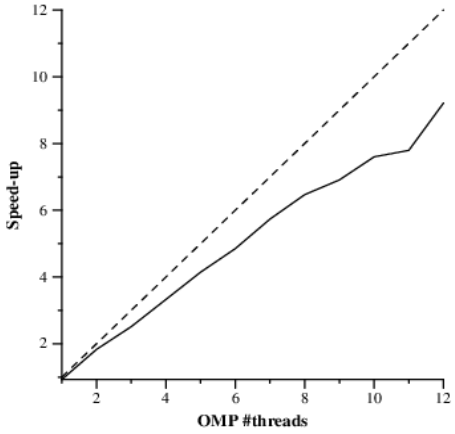
En los gráficos de Paraver vemos ahora que hay motas azules a lo largo de la New Window, apreciables sin tener que hacer mucho zoom. Esto señala que ahora se van ejecutando más tareas, cosa que se refleja en el histograma, donde los porcentajes de tiempo de la sección azul (Running), han subido considerablemente. Los overheads de sincronización se han reducido entre un 15 y un 20%, y la creación de tareas sigue igual. El histograma de la creación de tareas muestra que la media de tareas por thread se mantiene, pero están más distribuidas. Vemos que hay threads de color verde oscuro y azul, es decir, que realizan más tareas. Finalmente, la duración de las tareas podemos ver que se ha reducido un poco y se ha extendido más por el programa. Las tareas duran menos en general pero durante la ejecución del programa se ejecutan más tareas que duran más tiempo.

ROW TASKLOOP :

Cambiamos totalmente la estrategia, y volvemos a la estrategia Row. Para este caso hemos escrito la cláusula #pragma omp taskloop antes del comienzo de los dos bucles del for, que nos da como tiempo de ejecución 3.259005 segundos con 1 procesador y 0.457698 segundos con 8, bastante parejo con la mejor variante de la estrategia anterior (point taskloop nogroup).

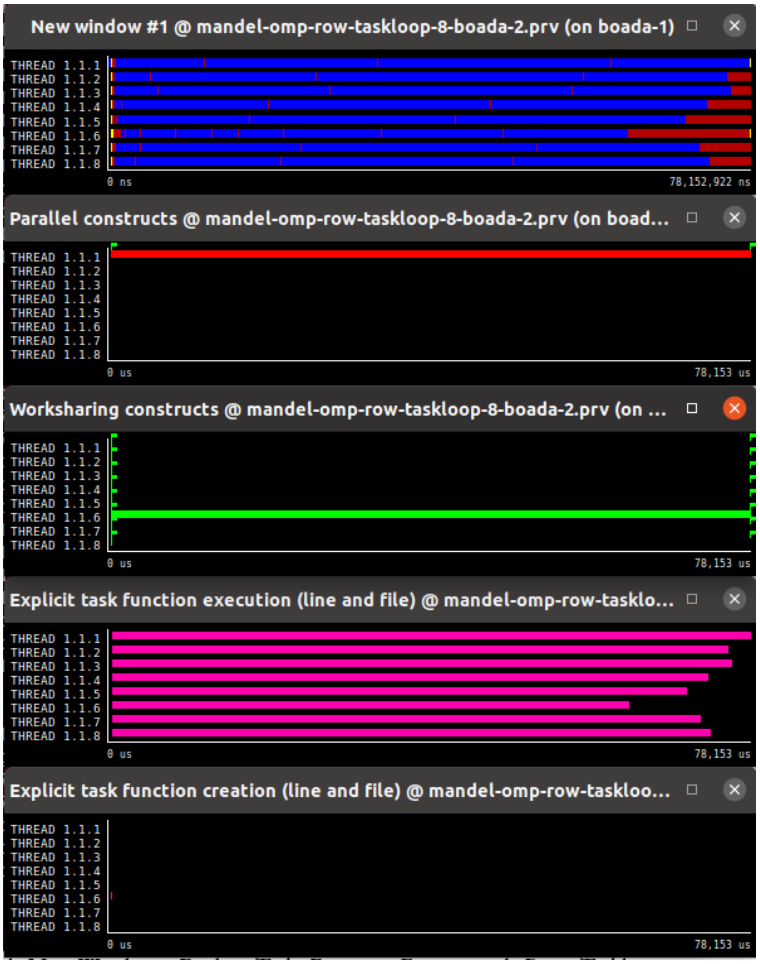


par2306
Average elapsed execution time
Tue Apr 19 20:38:33 CEST 2022



par2306
Speed-up wrt sequential time
Tue Apr 19 20:38:33 CEST 2022

La escalabilidad es muy buena, la mejor de todas, con un tiempo de ejecución menguante en la totalidad del gráfico y el speedup es muy, muy bueno, se acerca mucho a la línea ideal.



New Window,

Parallel constructs,

Worksharing constructs,

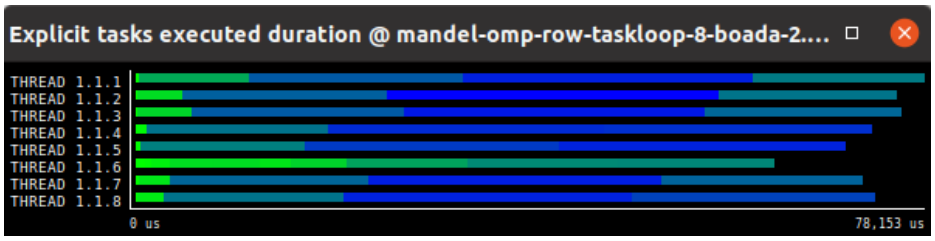
Explicit task function execution & creation

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	99.76 %	0.20 %	0.04 %
THREAD 1.1.2	96.32 %	3.66 %	0.01 %
THREAD 1.1.3	96.81 %	3.18 %	0.01 %
THREAD 1.1.4	93.06 %	6.94 %	0.01 %
THREAD 1.1.5	89.66 %	10.33 %	0.01 %
THREAD 1.1.6	80.48 %	19.27 %	0.25 %
THREAD 1.1.7	91.85 %	8.14 %	0.01 %
THREAD 1.1.8	93.43 %	6.56 %	0.01 %
Total	741.36 %	58.29 %	0.34 %
Average	92.67 %	7.29 %	0.04 %
Maximum	99.76 %	19.27 %	0.25 %
Minimum	80.48 %	0.20 %	0.01 %
StDev	5.47 %	5.41 %	0.08 %
Avg/Max	0.93	0.38	0.17

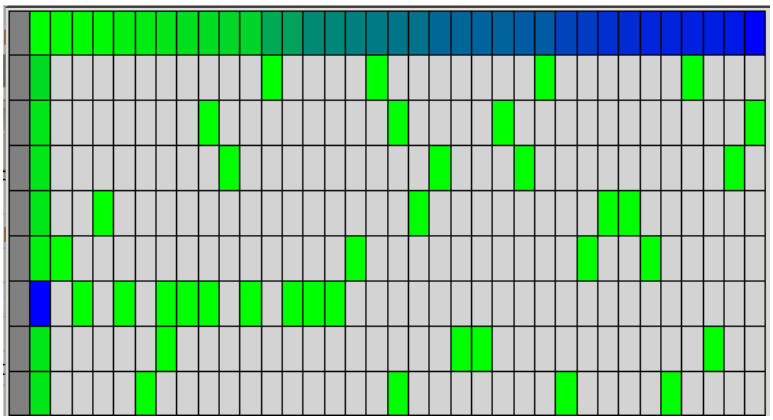
Histograma de New Window

	102 (mandel-o..skloop.c)
THREAD 1.1.1	8
THREAD 1.1.2	7
THREAD 1.1.3	7
THREAD 1.1.4	7
THREAD 1.1.5	6
THREAD 1.1.6	31
THREAD 1.1.7	7
THREAD 1.1.8	7
Total	80
Average	10
Maximum	31
Minimum	6
StDev	7.95
Avg/Max	0.32

Histograma de Explicit Tasks Function Execution de Row Taskloop



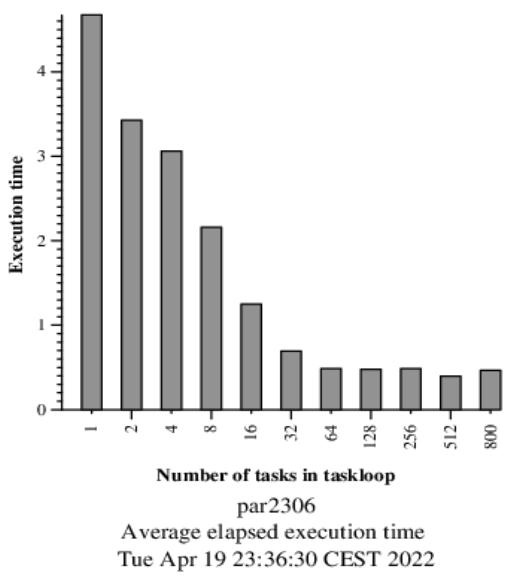
Explicit tasks executed duration

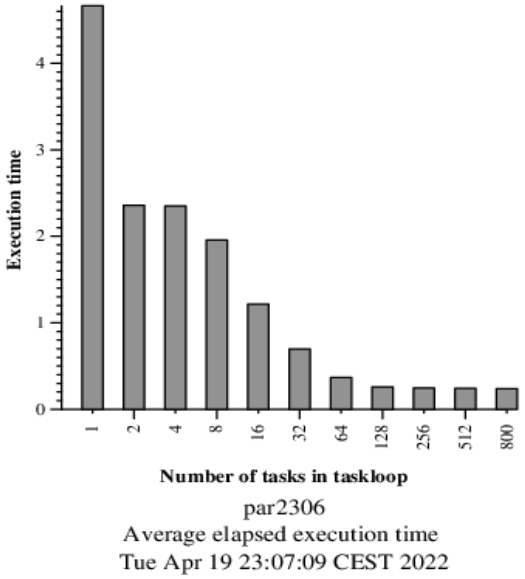


Histograma de la duración de tareas de Row Taskloop

En los gráficos de Paraver, vemos que son bastante diferentes a los de la estrategia Point. El gráfico de la New Window está casi todo azul. Esto quiere decir que las tareas están casi todo el programa funcionando, cosa que vemos en el histograma (el thread que menos tareas ejecuta las ejecuta un 89% del tiempo). Los overheads son inexistentes ya que se crean muy pocas tareas al poner la cláusula paralela fuera de los dos bucles. Algo a destacar es la cantidad de tareas que se crean, una media de 10, comparado con la versión Point, que tenía 3200 de media. En el gráfico e histograma de la duración de tareas, vemos que cada procesador está a cargo de muy pocas tareas, se pueden distinguir a simple vista cada una de ellas. En el histograma final vemos que hay una única tarea al principio de la ejecución que se encarga de la porción más pesada del programa al inicio, y el resto se van ejecutando en la segunda mitad.

3. Opcional





POINT TASKLOOP NOGROUP NUMTASKS **ROW TASKLOOP NUMTASKS**

En el ejercicio opcional se pide comentar el rendimiento de dos versiones del programa según un cierto número de tareas. Al ejecutarlas y generar los gráficos, podemos observar que son bastante parecidos, pero la versión Row Taskloop Numtasks es la que mejor funciona, ya que al pasar de 1 a 2 threads y a 4 threads, su tiempo de ejecución se reduce a la mitad, y en el resto de variaciones de tareas, es mejor en tiempo que la otra versión. Esto se debe a que la estrategia Point crea muchas tareas (row), y esto añade overheads de creación y sincronización al tiempo de ejecución del programa. No obstante, la carga de trabajo de las tareas es inferior. Sin embargo, la estrategia Row crea menos tareas que ejecutan los dos bucles cada una.

Conclusiones

Tras hacer esta sesión de laboratorio, podemos concluir que utilizar una estrategia de paralelismo u otra puede afectar en gran medida al rendimiento de un programa. El primer día vimos gracias a los grafos de Tareador las dependencias que se pueden crear al utilizar regiones paralelas en una parte específica del código. El segundo sacamos resultados de la ejecución de varias estrategias: tiempos de ejecución, gráficos de escalabilidad y gráficos de Paraver. Cada estrategia ha ido obteniendo mejores resultados que la anterior, y esto se ha reflejado en los gráficos que hemos ido interpretando a lo largo de la práctica.