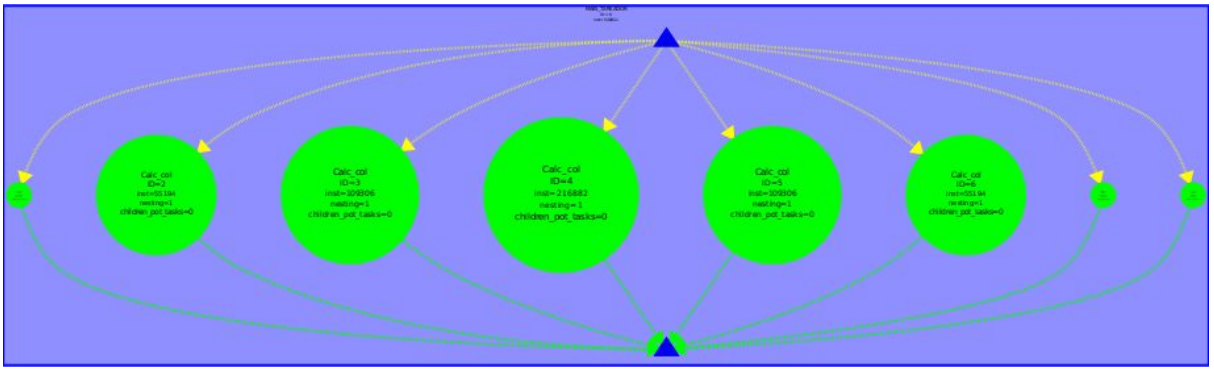
Práctica 3

Noel Carmona García

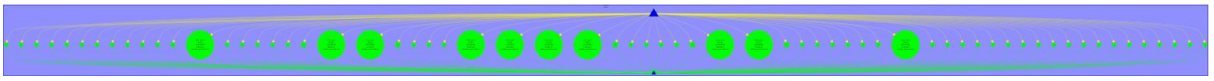
Granularity task analysis

1. Which are the two most important common characteristics of the task graphs generated for the two task granularities (Row and Point) for the non-graphical version of `mandel-tareador`? Obtain the task graphs that are generated in both cases for `-w 8`.

Row:



Point:



Características:

1. El número de instrucciones de cada tarea está bastante descompensado, especialmente en la versión row.

2. No existe dependencia de datos, ambas versiones son muy paralelizables, ya que no existen dependencias.

2. Which section of the code is causing the serialization of all tasks in mandeld-tareador? How do you plan to protect this section of code in the parallel OpenMP code?

La parte del código que causa la serialización de las tareas en mandeld-tareador corresponde a las líneas de código que generan la imagen. En OpenMP podemos solucionar este problema creando una zona de exclusión mutua mediante *#pragma omp critical*.

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

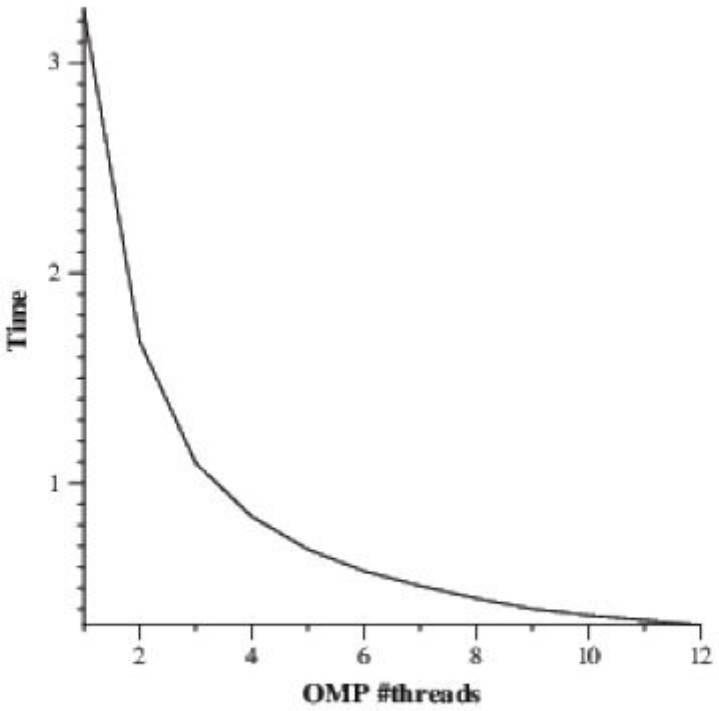
OpenMP task-based parallelization

1. For the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000). Reason about the causes of good or bad performance in each case.

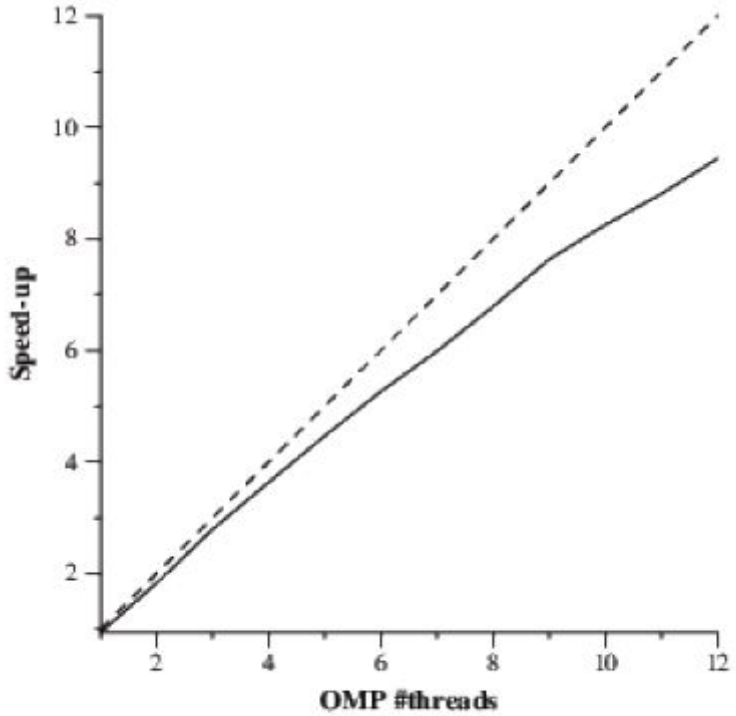
Row:

```
#threads        Elapsed average
1    3.25000000000000000000
2    1.63000000000000000000
3    1.09666666666666666666
4    .83666666666666666666
5    .68333333333333333333
6    .57666666666666666666
7    .50000000000000000000
8    .46333333333333333333
9    .42000000000000000000
10    .39666666666666666666
11    .35666666666666666666
12    .31000000000000000000

#threads        Speedup
1    1.00102564102564102564
2    1.99591002044989775050
3    2.96656534954407294834
4    3.88844621513944223110
5    4.76097560975609756099
6    5.64161849710982658965
7    6.50666666666666666666
8    7.02158273381294964033
9    7.74603174603174603173
10    8.20168067226890756315
11    9.12149532710280373847
12    10.49462365591397849461
```



Average elapsed execution time



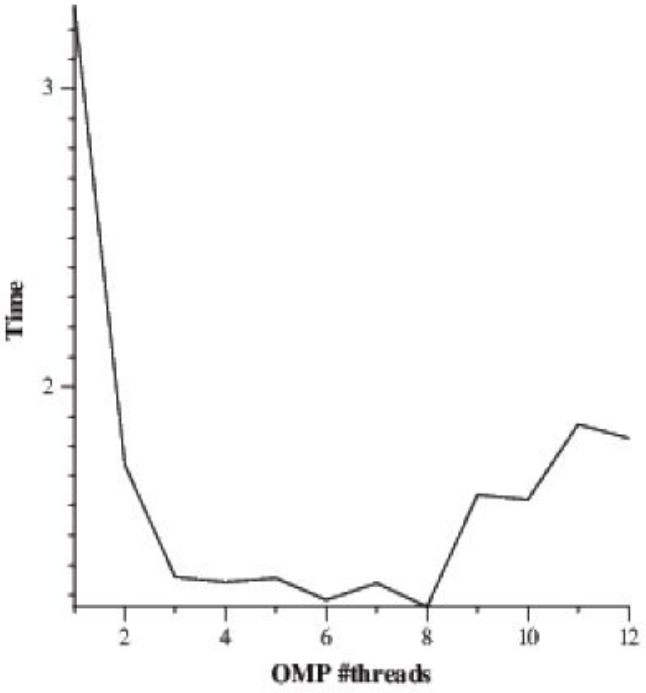
Speed-up wrt sequential time

Son buenos resultados y es lo que se esperaba, no hay dependencia de datos, cada tarea hace su fila y a medida que vamos aumentando el número de threads el speedup va en aumento.

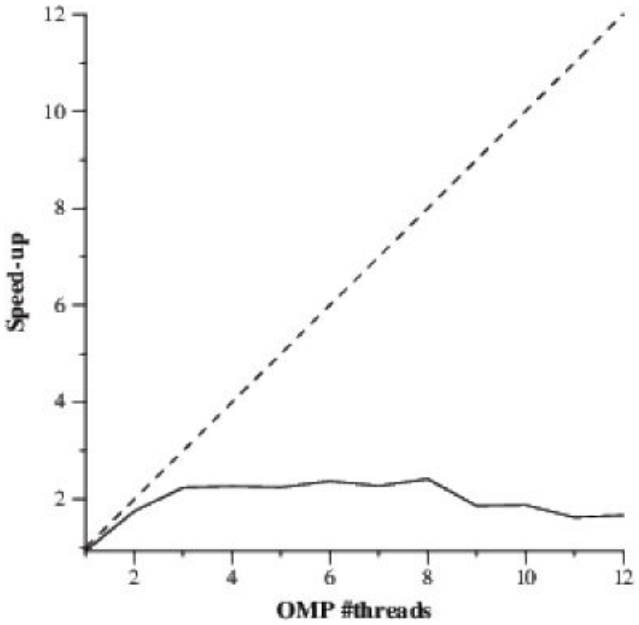
Point:

```
#threads        Elapsed average
1    3.27000000000000000000
2    1.76333333333333333333
3    1.41333333333333333333
4    1.35000000000000000000
5    1.31666666666666666666
6    1.42666666666666666666
7    1.63333333333333333333
8    1.61000000000000000000
9    1.79333333333333333333
10    1.84333333333333333333
11    1.99666666666666666666
12    2.17666666666666666666

#threads        Speedup
1    .99490316004077471967
2    1.84499054820415879017
3    2.30188679245283018868
4    2.40987654320987654320
5    2.47088607594936708861
6    2.28037383177570093458
7    1.99183673469387755102
8    2.02070393374741200827
9    1.81412639405204460966
10    1.76491862567811934900
11    1.62938230383973288815
12    1.49464012251148545176
```



Average elapsed execution time



Speed-up wrt sequential time

Los resultados no son muy buenos, esto es debido por dos motivos:

1. Estamos creando una tarea para cada punto de la imagen, por lo que estamos creando muchas tareas las cuales realizan muy poco trabajo.

2. Diferentes threads se encargan de tratar la misma línea, por lo cual tenemos un problema de false sharing.

Tal y como podemos ver en la imagen el utilizar más threads puede generar una bajada de rendimiento, en cuanto al false sharing lo podemos solucionar haciendo padding.

OpenMP taskloop-based parallelization

1.For the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots obtained in the strong scalability analysis (with -i 10000). Reason about the causes of good or bad performance in each case.

OpenMP for-based parallelization

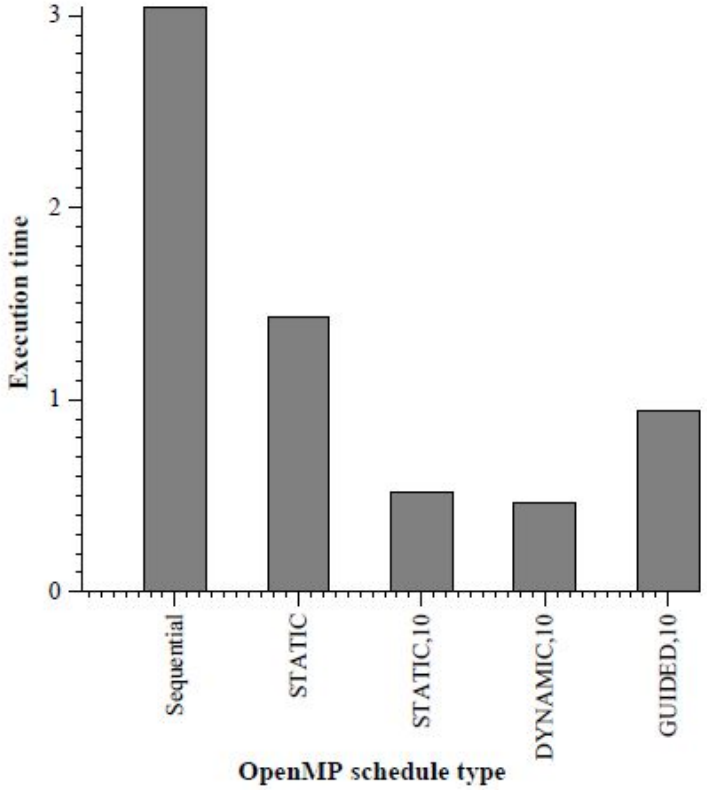
1. For the the Row and Point decompositions of the non-graphical version, include the execution time and speed–up plots that have been obtained for the 4 different loop schedules when using 8 threads (with `-i 10000`). Reason about the performance that is observed.

Row:

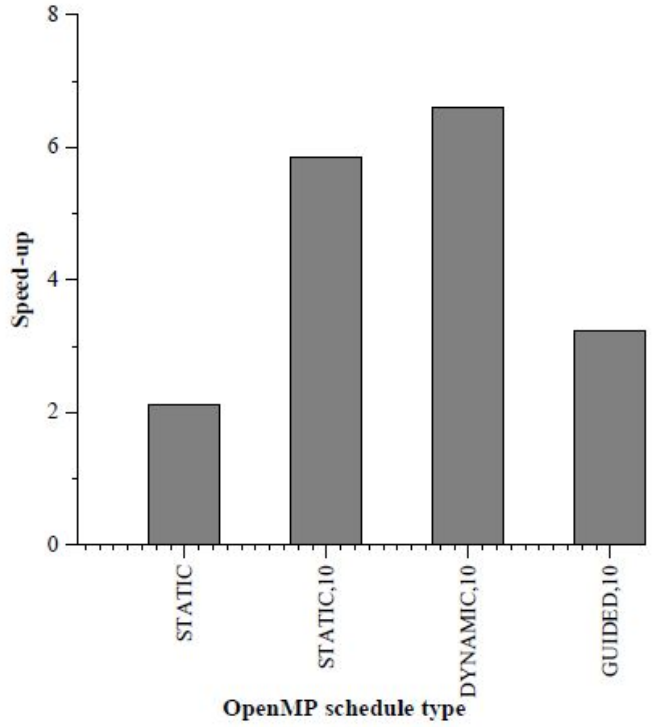
- El speed-up es bajo ya que divide el trabajo en bloques y el primer thread es el encargado de la repartición de trabajo, lo cual hace que la repartición de trabajo no sea la mejor.
- La repartición de trabajo en static,10 es mucho mejor que la de static.

- En dynamic,10 el overhead generado durante la repartición de trabajo entre threads se compensa con un mejor speed-up.

- El speed.up de guided,10 es más bajo de lo esperado.



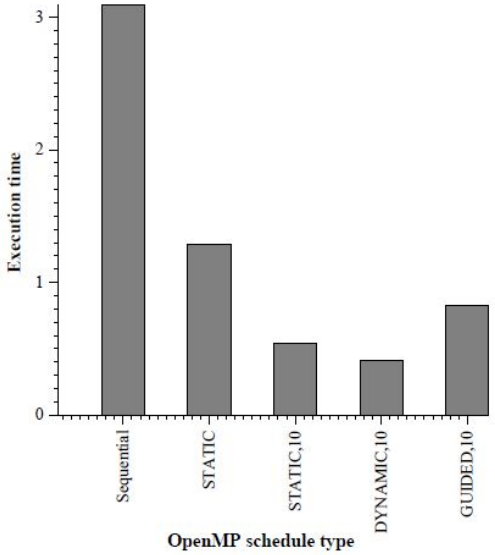
Average elapsed execution time



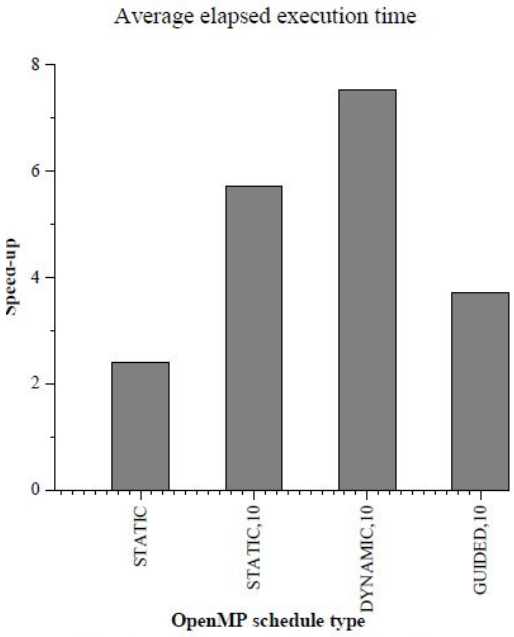
Speed-up wrt sequential execution time

Point:

Son unos resultados similares a la versión row, lo más destacable es el incremento de speed-up en dynamic,10. Al repartidos los puntos en bloques de 10 parece que el falsesharing no le afecta tanto.



Average elapsed execution time



Speed-up wrt sequential execution time

2.-For the Row parallelization strategy, complete the following table with the information extracted from the Extrae instrumented executions (with 8 threads and `-i 10000`) and analysis with Paraver, reasoning about the results that are obtained

	static	static,10	dynamic,10	guided,10
Runing average time per thread	423.241.173ns	473.418.734 ns	455.264.282 ns	445.215.457 ns
Execution unbalance (average time divided by maxium time)	0,3054398 ns	0,88369830 ns	0,952106508 ns	0,482500977ns
SchedForkJoin (average time per thread of time if only one does)	173.355.481 ns	750.665 ns	5.126.016 ns	106.088.837 ns

- En el tiempo medio por cada thread.

- La repartición de trabajo en static es la peor. *Dynamic,10* tiene mejor reparto de trabajo.

- En la creación de threads podemos observar como dynamic tarda bastante en comparación con *static,10.* Al estar static,10 peor repartido el trabajo tiene que esperar más para hacer join.