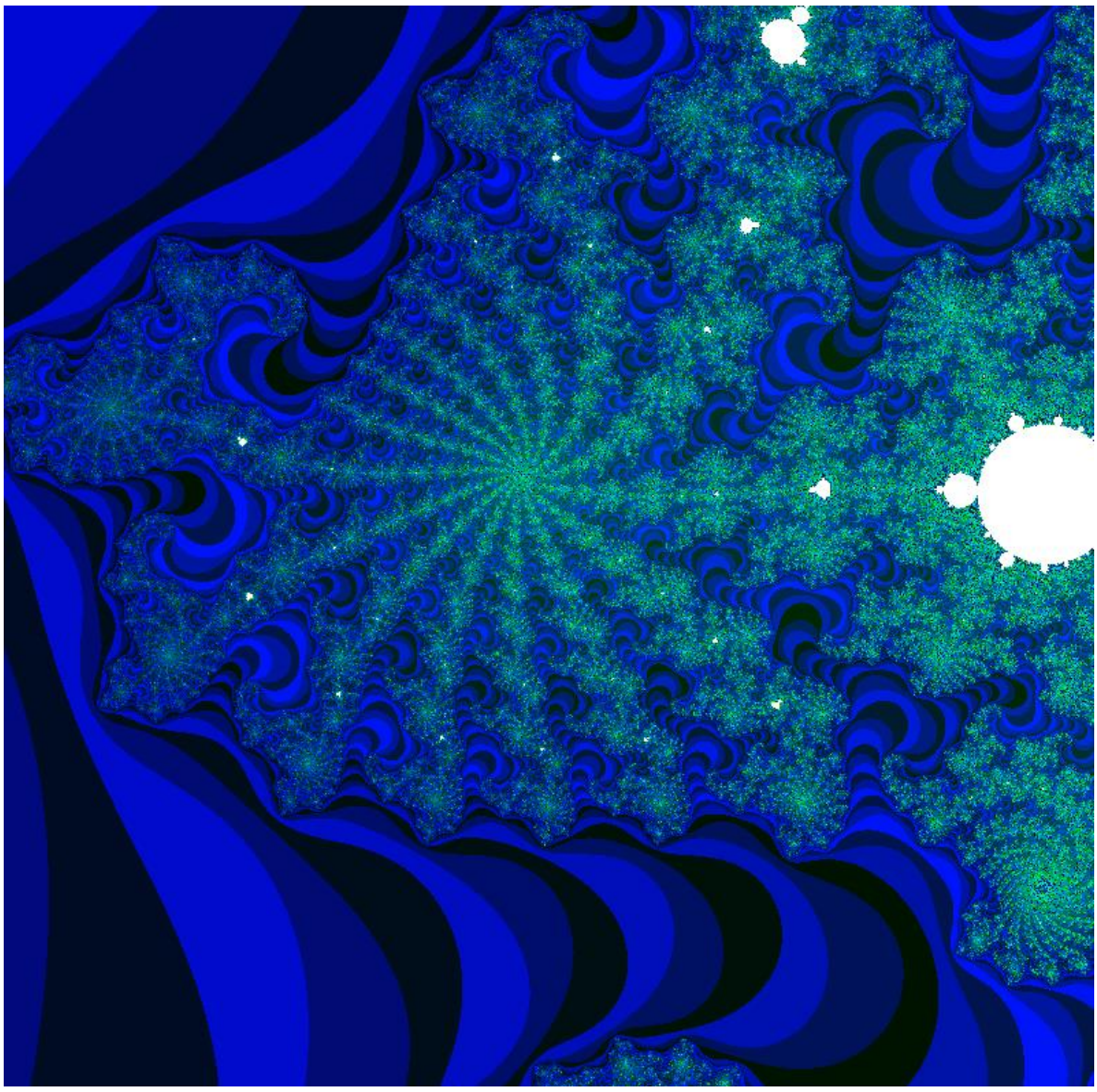
ENTREGABLE 3,

# The Mandelbrot set



Jordi Foix Esteve

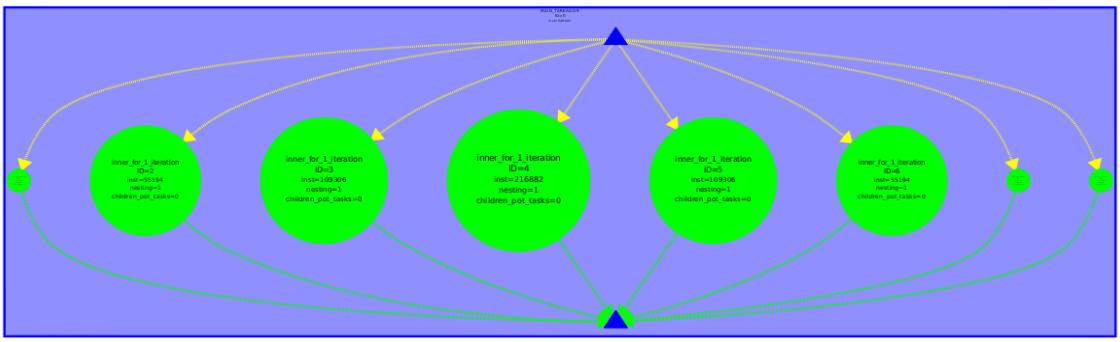
Albert Figuera Pérez

Par1305

## 6.1 Task granualarity 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.

En el graf de depèndencies del row strategy podem observar que gairebé tot el cost del codi es pot fer en paral·lel. La primera característica comú amb el point strategy es que no hi ha apenes depèndencies. Una altre característica comuna es que la càrrega no està balancejada.




*F*igura 1: Graf de dependències, R*ow strategy*.

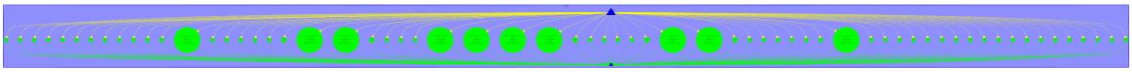


Figura 2: Graf de dependències, *Point strategy*

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?

En la versió gràfica podem observar que s'executa seqüncialment degut a una part del codi que té una directiva *#pragma omp critical.* Això ho fa per limitar a un unic thread aquesta regió.

```
#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

            output[row][col]=k;


#endif
```

Figura 4. Fragment de codi que produeix les dependències.

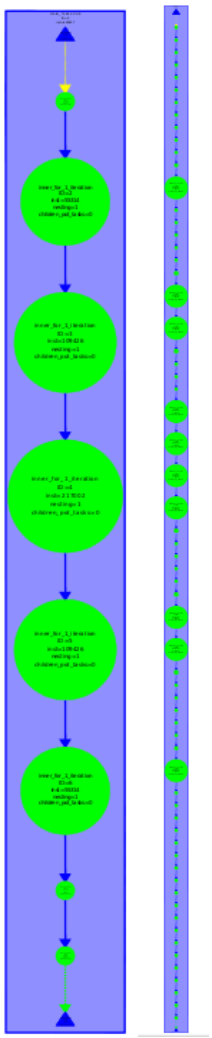


*Figura 3: Graf de depèndencies (Row i Point)*

# Parallelization strategies

## 6.2 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.

La Row strategy te un bon rendiment en general. La granualitat no és massa petita i els costos d'overhead no són gaire elevats com podem observar als següents gràfics. Podem observar que el plot del speed-up (Figura 7) es gairebé lineal. Veiem que el temps disminueix de forma significant.

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
    #pragma omp task firstprivate(row) private(col)

    for (col = 0; col < width; ++col) {
        complex z, c;
```

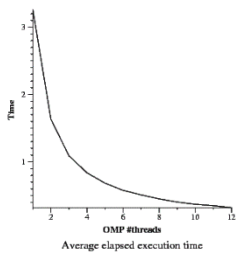
Figura 5. Row strategy code.

La point strategy té un rendiment bastant menor degut a que les tasques són molt més petites i n'hi ha més. Això crea un overhead major que no es compensat per la complexitat de les tasques. Podem observar aquest decrement de rendiment a les gràfiques.

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        #pragma omp task firstprivate(col,row)
        {
        complex z, c;
```

Figura 6. *Point strategy code.*

Per tant la millor estrategia, en aquest cas, és Row strategy degut a que la complexitat dels càlculs es petita. Si els càlculs fossin molt més costos el point strategy seria millor ja que tenim més procesadors.



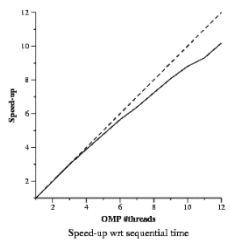
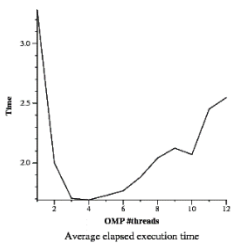




Figure 7. Average execution time and speed-up plots of Row version(task).



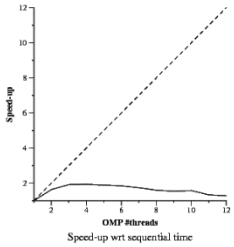




Figure 8. Average execution time and speed-up plots of Point version(task).

## 6.3 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.

La *Row strategy* te un bon rendimient i el speed up es gairebé lineal.

```
#pragma omp parallel
#pragma omp single
#pragma omp taskloop private(col,row) grainsize(8)
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {


        complex z, c;
```

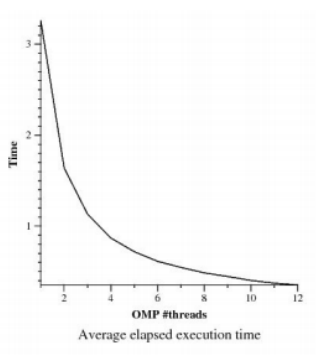
Figura 9. *Row strategy code 2*

La *Point strategy* com en el cas anterior també és pitjor però no tant. Torna a pasar que el overhead produeix un nombre elevat de tasques.

```
#pragma omp parallel
#pragma omp single
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        #pragma omp task firstprivate(col,row)
        {
        complex z, c;
```

Figura 10. *Point strategy code 2.*

Un altre cop la millor estrategia es la Row strategy degut a que té un overhead menor a la Point strategy.



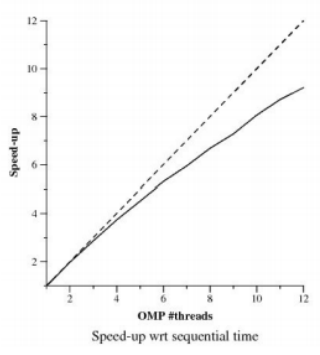
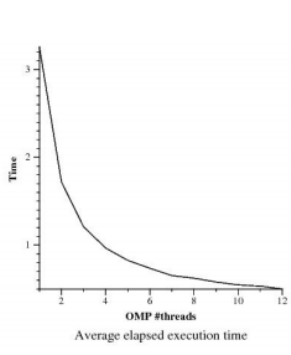




Figura 11 Plots of Row version (task-loop).



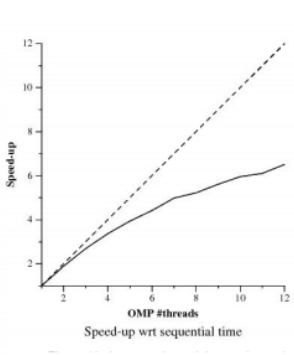




Figura 12 Plots of Row version (task-loop).

## 6.4 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 Decomposition

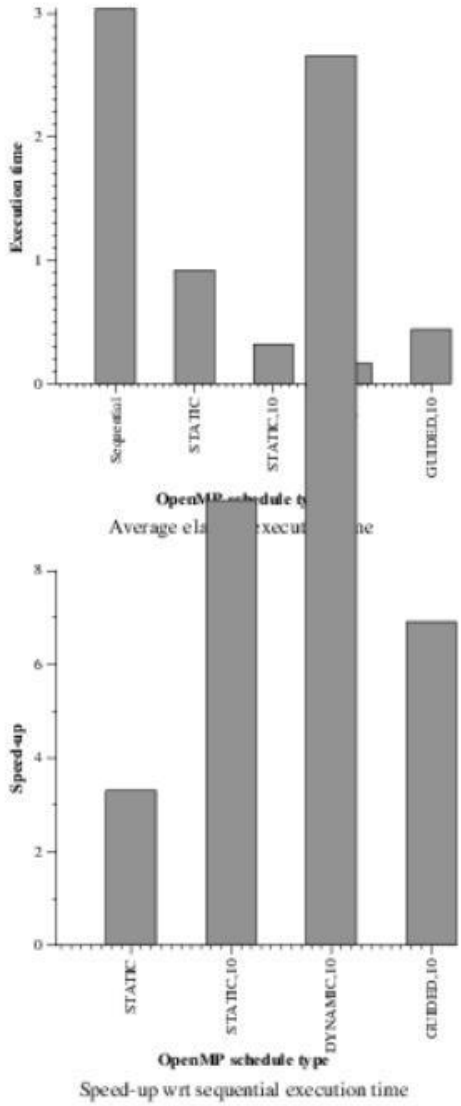



Figura 11. Row decomposition graphs. Time is in seconds.

The static version, without the 10 iterations per thread specified, divides the for loop in the number of threads, in this case 8. That means each thread executes 800/8=100 iterations of the loop. With the 10 specified each thread executes 800/10=80 iterations. The dynamic scheduler decides at runtime the size of the chunk. The guided divides the loop into chunks of more than 10 iterations (optional parameter) and resizes them on runtime to balance all the threads. Typically starts with big chunks as then will decrease the size of them. On the graphic we can see a very big improvement between sequential and the other schedulers. Specially, the Dynamic, 10 have the best execution time with an outstanding speed-up.

Point Decomposition

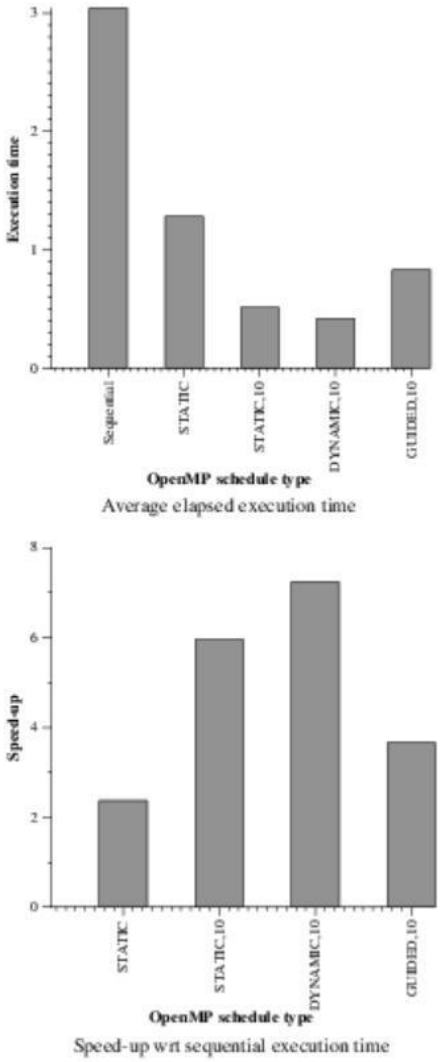



Figura 12. Dot decomposition graphs. Time is in seconds.

With the point decomposition we have not so extreme results. Dynamic, 10 still wins in execution time, even having more overhead than static. Guided is a little bit slower than dynamic, maybe because resizing the chunks in real time adds overhead.

> 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.

The most fastest schedule is guided but not with a very big difference. The average time per thread is better with the dynamic execution and the static having a very bad result. About the load unbalance (the bigger the better), dynamic has the best one with almost 1 (1 means perfect balance).

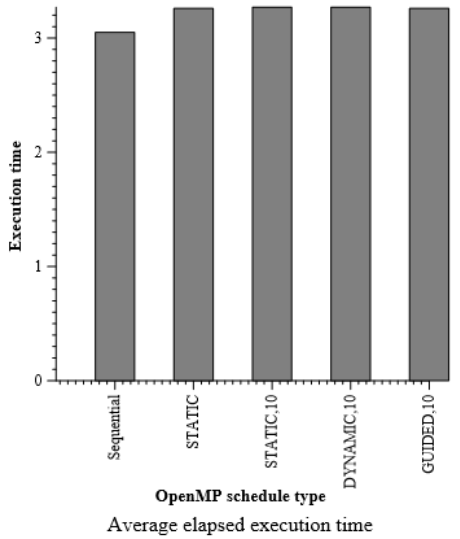
| In ns | static | static,10 | dynamic,10 | guided,10 |
|---|---|---|---|---|
| **Running average time per thread** | 415,785 | 449,761 | 440,152 | 421,654 |
| **Execution unbalance (average time divided by maximum time)** | 0.29 | 0.87 | 0.89 | 0.45 |
| **SchedForkJoin (average time per thread or time if only one does)** | 1,401,340 | 458,1 | 382,22 | 821,665 |

## 6.5 Optional

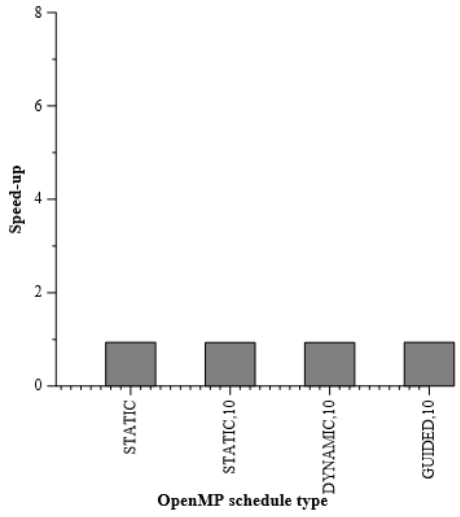
1. If you have done any of the optional parts in this laboratory assignment, please include your experience, additional information collected or the relevant portion of the code and performance plots obtained in your report.

**Optional 1:** How is the Mandelbrot space computed and what is the performance for the different schedules when using the collapse clause? Look at the following incomplete code:

```
#pragma omp for collapse(2) schedule(runtime)
for (row = 0; row < height; ++row) {
  for (col = 0; col < width; ++col) {
```



Average elapsed execution time



Speed-up wrt sequential execution time

Figura 13. Speedup and execution times of collapse. Time is in seconds.

As we can see the parallelization did not perform well. We should obtain speedups above 1 because the collapse clause distributes better the workload among the threads.

**Optional 2:** How is the Mandelbrot space computed and what is the performance for the different schedules when combining both for and task? Look at the following incomplete code:

```
#pragma omp for schedule(runtime)
for (row = 0; row < height; ++row) {
    #pragma omp task
    for (col = 0; col < width; ++col) {
```

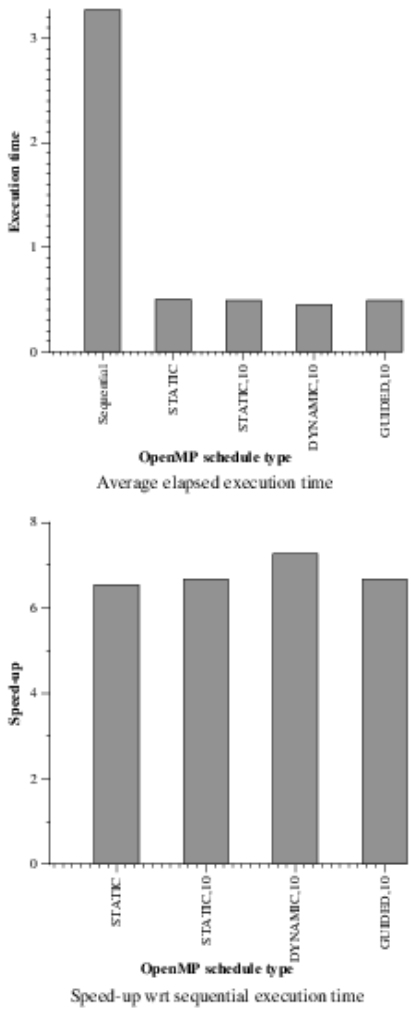



Figura 14. Speedup and execution times of for&task. Time is in seconds.

The speedup of all the schedules is very similar, being the dynamic,10 the best one by a tiny bit.

If we call mandeld-omp with an argument of -i 10000 iterations, we can see that the image is generated by rows, being each row a task created by the outermost loop. The iterations of that loop are divided by the omp for schedule (dynamic). The image generated is fine, no distortion or bad pixels.