3rd Deliverable

par1103
Iqbal, Omair
Victor Gallego
2015-2016

Introduction

In this document we will explore different ways of rendering a mandelbrot set. More exactly we will use two kind of optimizations for that: Row parallelization strategy and point parallelization strategy. And then we will use openMP two basic directives, omp task and omp for. All in all, at the end of this document we will know the differences within these methods and we will be able to pick up the optimal one for a concrete situation.

Parallelization strategies

2 diferent kind of strategies are being used to improve the performance of the code:

- **Row parallelization strategy:** a task corresponds with the computation of a whole row of the Mandelbrot set.

- **Point parallelization strategy:** a task corresponds with the computation of a single point (row,col) of the Mandelbrot set.

Task granularity 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? Include the task graphs that are generated in both cases for -w 8.

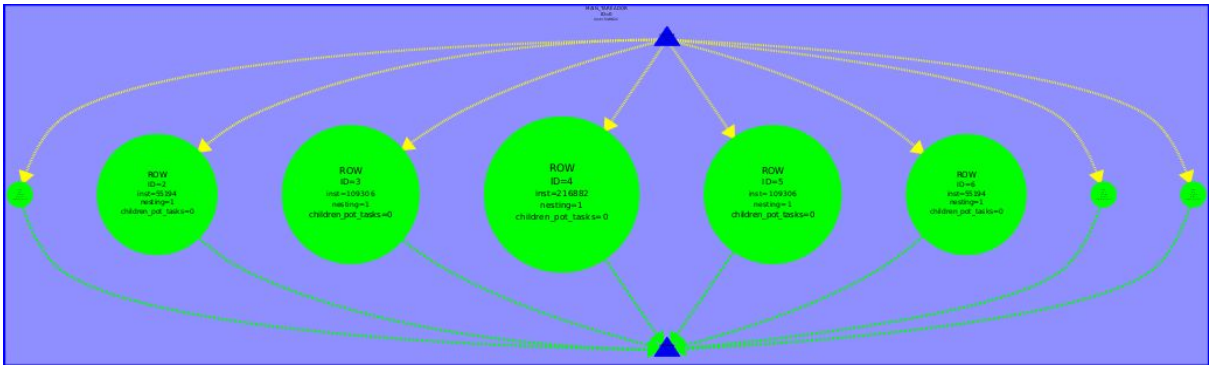


Figure 1: *Mandel-tareador.c with row strategy*

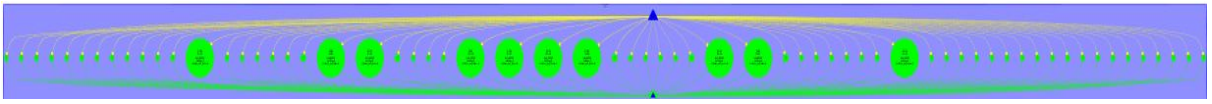


Figure 2*: Mandel-tareador with point strategy*

According to the previous figures we can see that main common characteristic is the lack of dependencies between the task.

The other shared characteristic between the task dependency graphs is that both of them have a unbalanced workload for each task, therefore none of them will scale up correctly.

2. Which section of the code is causing the serialization of all tasks in mandeld-tareador? How have you protected this section of code in the parallel OpenMP code?

Using Tareador we found that the serialization is caused by the variable X11_COLOR_fake which is used in the following piece of code:

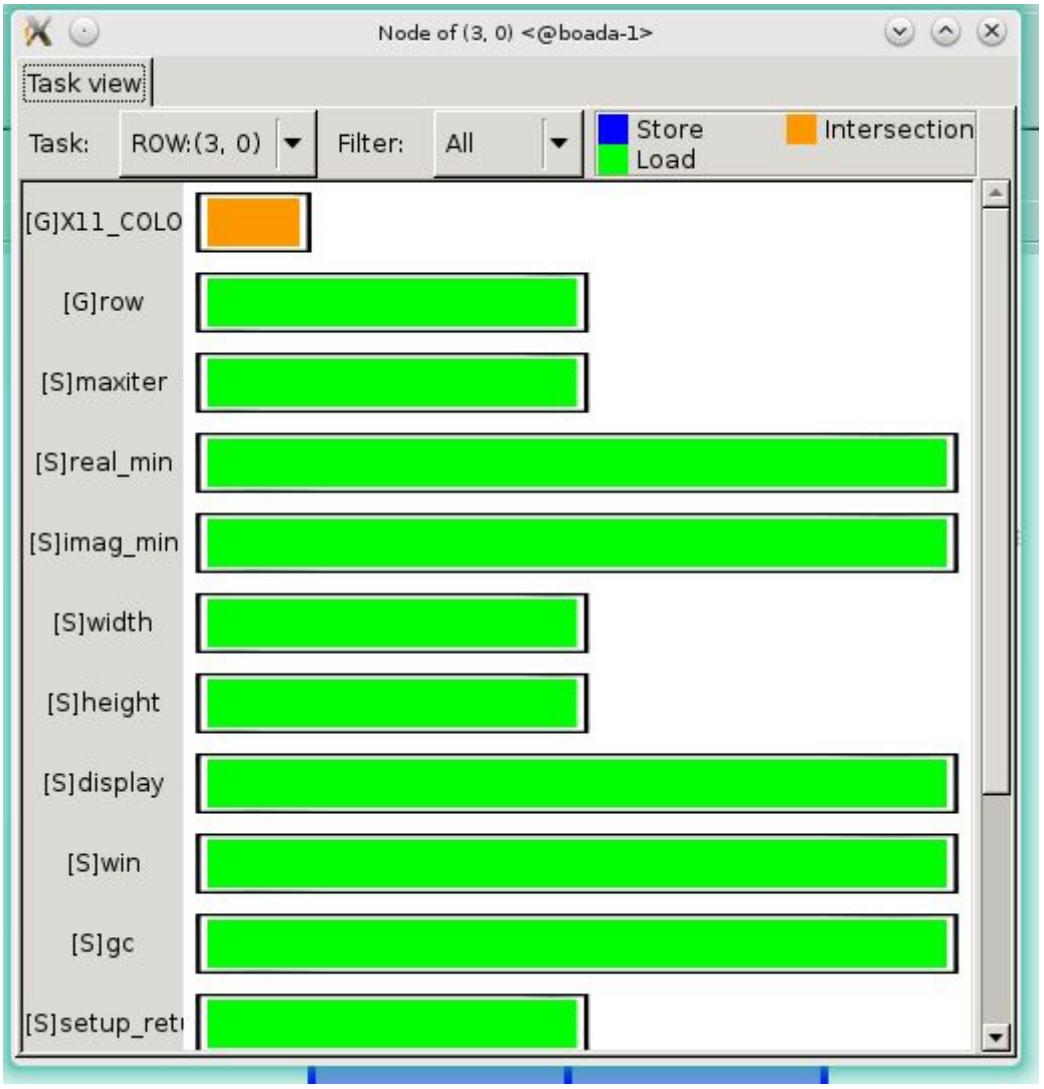


Figure 3: *Proof that the variable is causing serialization*

```
#if _DISPLAY_
            /* 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);
            }
```

Figure 4: *Part of the code where the variable is used.*

To protect this part of the code we can add a #pragma omp critical directive to avoid the data race.

```
#pragma omp critical
if (setup_return == EXIT_SUCCESS) {


    XSetForeground (display, gc, color);
    XDrawPoint (display, win, gc, col, row);
}
```

Figure 5: *Part of the code with data race solved.*

But we still have no parallelism as all tasks are running sequentially with dependencies. So we decided to disable the 11_COLOR_fake variable and were able to parallelize all the tasks. This is the result we obtained:

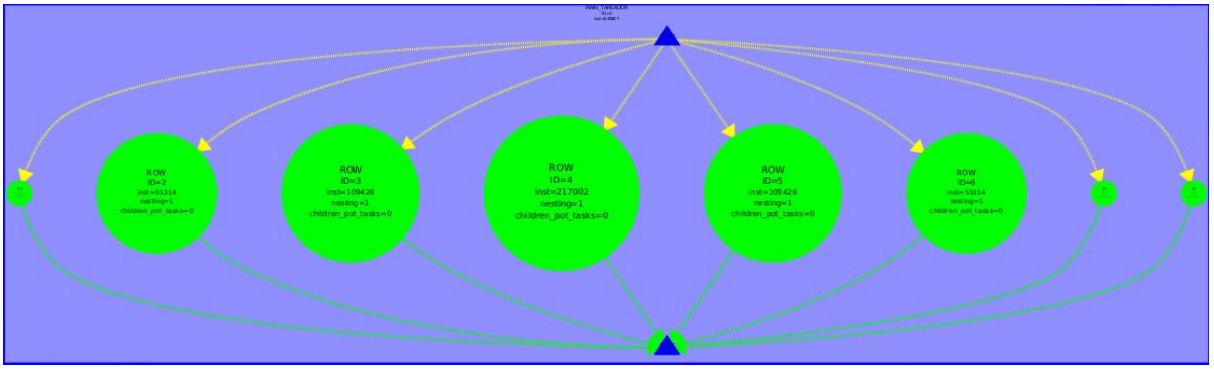


Figure 6: *mandeld-tareador.c with row strategy disabling the 11_COLOR_fake variable.*

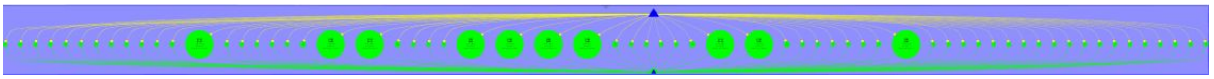


Figure 7: *mandeld-tareador.c with point strategy disabling the 11_COLOR_fake variable.*

OpenMP task–based parallelization

1. Include the relevant portion of the codes that implement the task-based parallelization for the Row and Point decompositions (for the non–graphical and graphical options), commenting whatever necessary.

```
    #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) {
                {
```

Figure 8: Mandel-omp.c non-graphic Task row parallelization strategy

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

                {
```

Figure 9: Mandel-omp.c non-graphic Task point parallelization strategy

```
    #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) {
                {
```

Figure 10: Mandel-omp.c graphic Task row parallelization strategy

```
#if _DISPLAY_
            #pragma omp critical
                {
            /* 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);
            }
}
```

Figure 11: Mandel-omp.c graphic Task row parallelization strategy

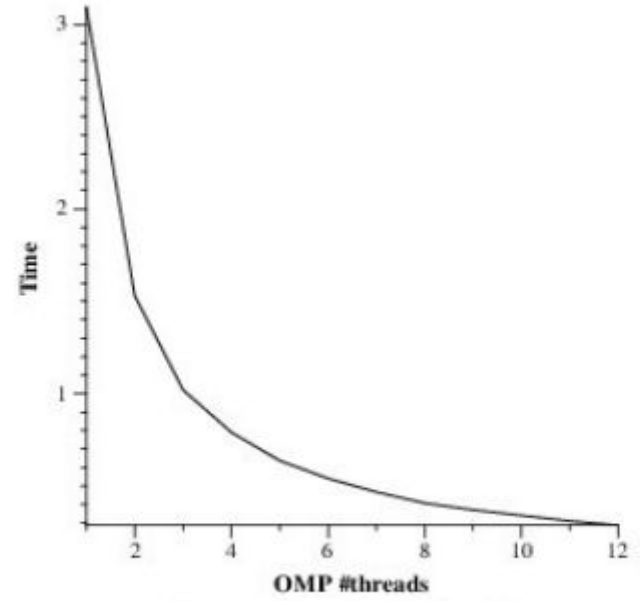
```
    #pragma omp parallel
    #pragma omp single
    for (row = 0; row < height; ++row) {
        for (col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)

                {
```

Figure 12: Mandel-omp.c graphic Task point parallelization strategy

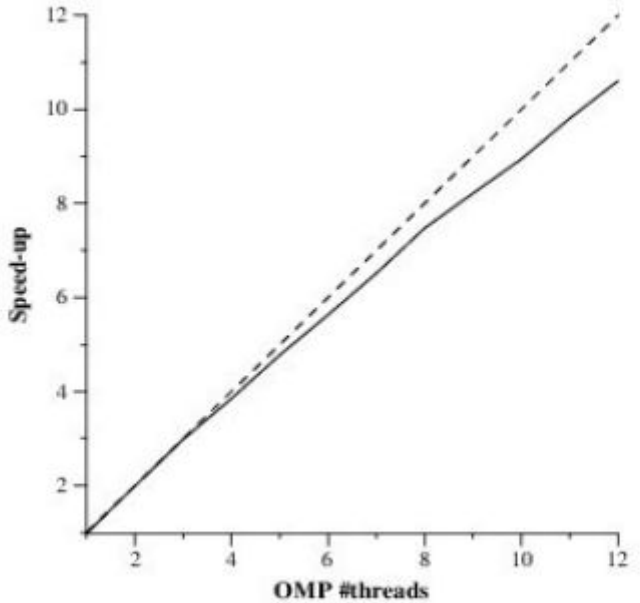
```
#if _DISPLAY_
            #pragma omp critical
                 {
            /* Scale color and display point  */
            long color = (long) ((k-1) * scale_color) + min_color;
            if (setup_return == EXIT_SUCCESS) {
                XSetForeground (display, gc, color);
             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

Figure 13: Mandel-omp.c graphic Task point parallelization strategy

2. For the 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.

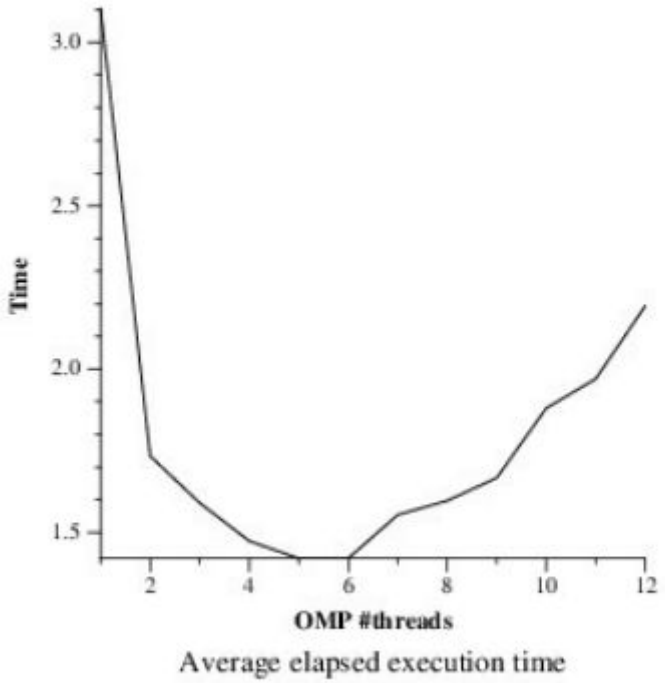


Average elapsed execution time



Speed-up wrt sequential time

Figure 14: Execution time and speedup of mandel-omp.c using row parallelization strategy with 1, 2, 4, 6, 8, 10 and 12 threads.



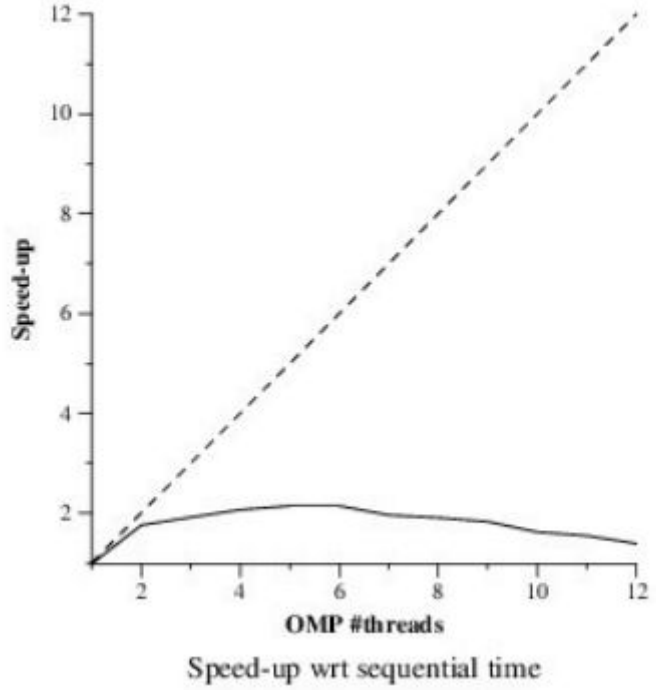


Figure 15: Execution time and speedup of mandel-omp.c using point parallelization strategy with 1, 2, 4, 6, 8, 10 and 12 threads.

The row parallelization strategy haves a better scalability due to a lower overhead in creating the task, while in the point parallelization strategy we have a high overhead.

The cost of managing all the created tasks is expensive, which hardly prevent a better performance.

OpenMP for–based parallelization

1. Include the relevant portion of the codes that implement the for-based parallelization for the Row and Point decompositions (for the non–graphical and graphical options), commenting whatever necessary.

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

Figure 16: Mandel-omp.c non-graphic for row parallelization strategy

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

Figure 17: Mandel-omp.c non-graphic for point parallelization strategy

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

Figure 18: Mandel-omp.c graphic for row parallelization strategy

```
#if _DISPLAY_
            #pragma omp critical
                {
            /* 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);
            }
}
```

Figure 19: Mandel-omp.c graphic for row parallelization strategy

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

Figure 20: Mandel-omp.c graphic for point parallelization strategy

```
#if _DISPLAY_
            #pragma omp critical
                {
            /* 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);
            }
}
```

Figure 21: Mandel-omp.c graphic for point parallelization strategy

2. 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.

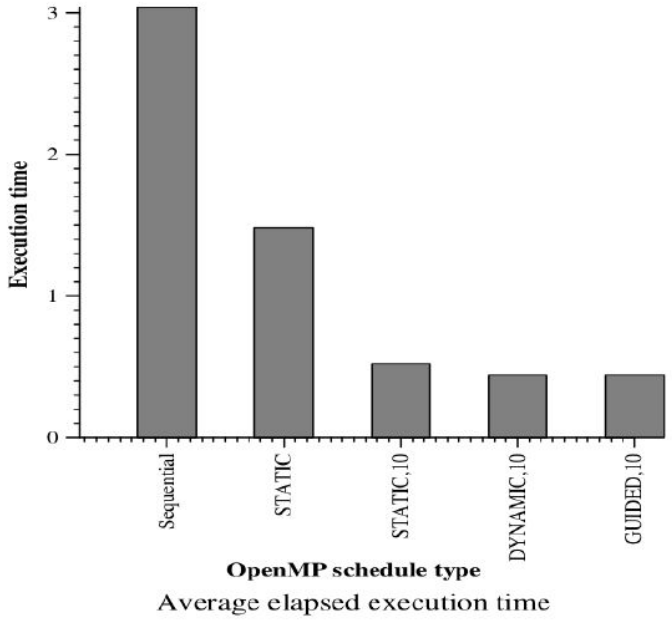


Figure 20: *Execution time obtained for 4 different loop schedules when using 8 threads and 10.000 iterations for the Point decomposition of non graphical version.*

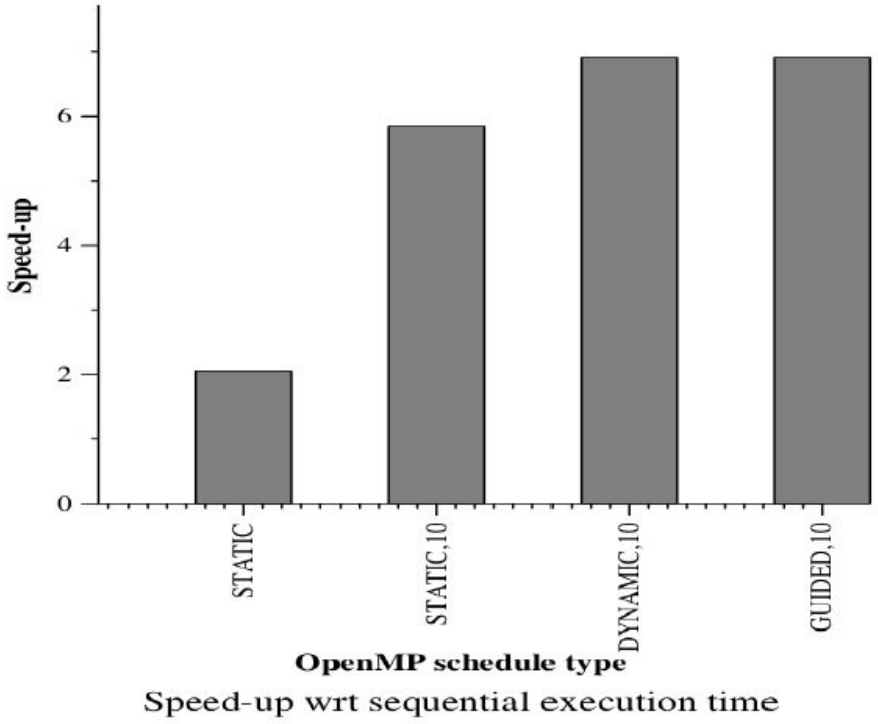


Figure 21: *Speedup obtained with respect to sequential time for 4 different loop schedules when using 8 threads and 10.000 iterations for the Point decomposition of non graphical version.*

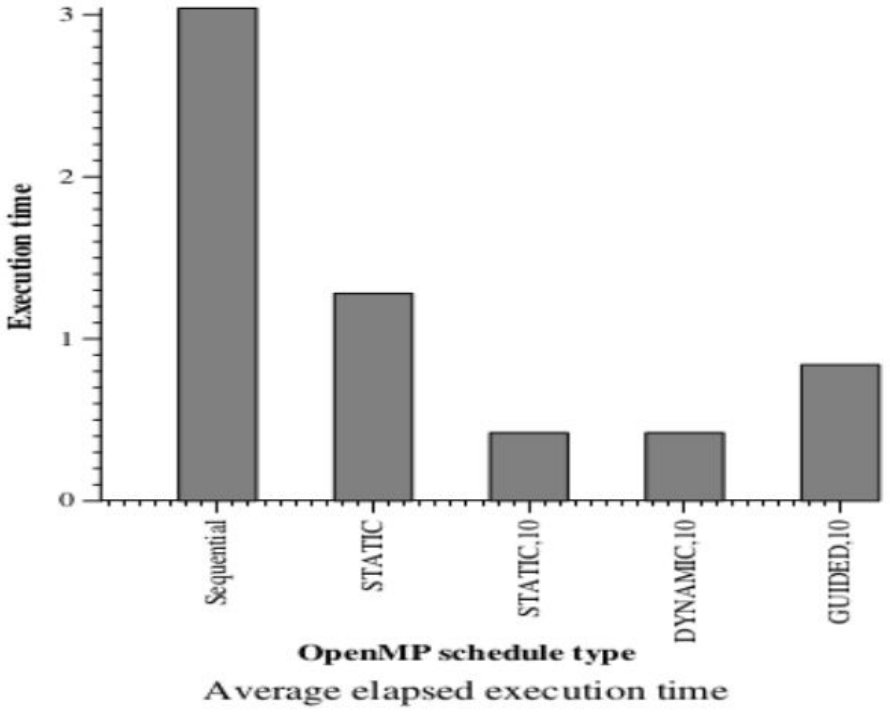


Figure 22: *Execution time obtained for 4 different loop schedules when using 8 threads and 10.000 iterations for the Row decomposition of non graphical version.*

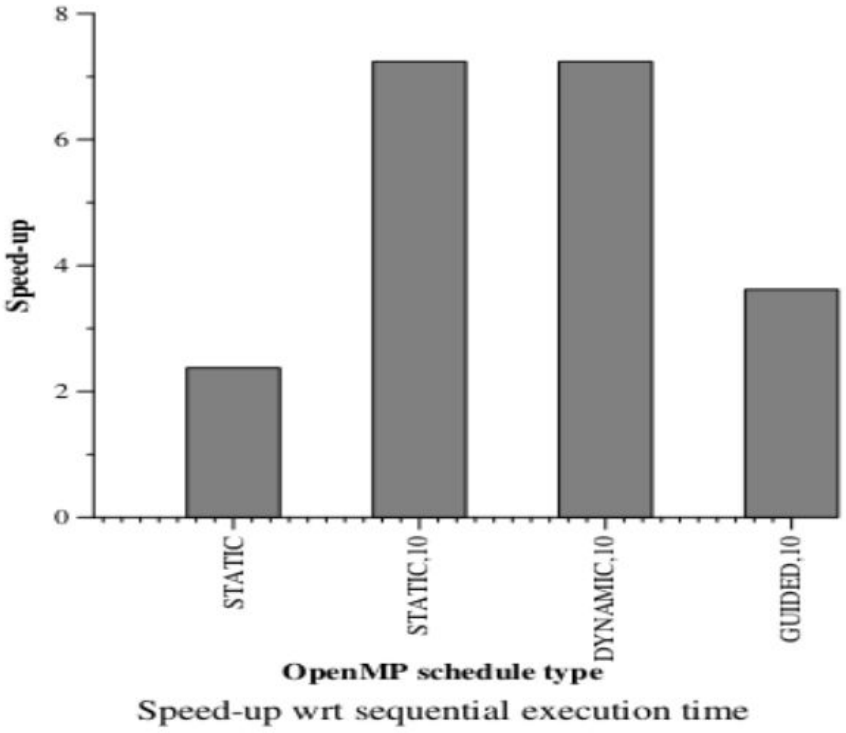


Figure 23: *Speedup obtained with respect to sequential time for 4 different loop schedules when using 8 threads and 10.000 iterations for the Row decomposition of non graphical version.*

3. 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
Running average time per thread(ns)	387,554,525.80	416,074,516.50	411,412,232.12	412,489,579.75
Execution unbalance (average time divided by maximum time)	0.32	0.87	0.90	0.39
SchedForkJoin(average time	127076.65	115672.87	225586.23	128465.13

per thread or time if only one does) (ns)				

Optional

The mathematics behind the mandelbrot formulas were difficult to understand but still it was a interesting way of visualizing data. We've appreciated the power of parallelism in order to compute things. And last but not least we've seen how the overhead can dramatically afect the execution time.

Performance evaluation

Using tasks we don't get a good scalability in point parallelization strategy but we have a good one in the row one.
On the other hand using for directives we a good improvement of the performance, the results can be seen in the tables.

Conclusion

Depending in which case is better to use one or the other one because the overhead must be taken into consideration but we can implement more strategies to parallelization.