PAR Laboratory Assignment

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

Semester 2 - 2018

Group par2203

01/05/2018

Bernat Gené Škrabec

Ferran Ramírez Navajón

Index

Introduction: The Mandelbrot set.

In this lab session we are going to analyze different parallelization strategies and tools. To do so we will gather data regarding the behaviour of a piece of code which computes an image of the Mandelbrot set. We will change certain parts of the code to implement parallelization elements and observe what strategies and tools work best and try to reason why. Any version of the code that we use in this session can be find in the source folder delivered together with this document.

The Mandelbrot set is a particular set of points, in the complex domain, whose boundary generates a distinctive and easily recognizable two-dimensional fractal shape (see Fig0).

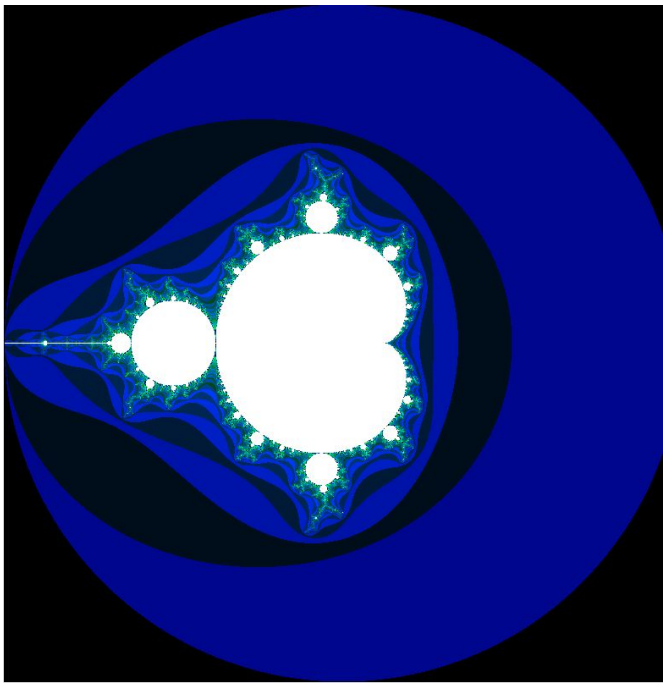

Fig0. Fractal shape. Image computed using our code.

For each point c in a delimited two-dimensional space, the complex quadratic polynomial recurrence zn+1 = zn^2 + c is iteratively applied in order to determine if it belongs or not to the Mandelbrot set. The point is part of the Mandelbrot set if, when starting with z0 = 0 and applying the iteration repeatedly, the absolute value of zn never exceeds a certain number however large n gets. A plot of the Mandelbrot set is created by coloring each point c in the complex plane with the number of steps max for which $|z_{max}| \geq 2$ (or simply $|z_{max}|\ 2 \geq 2 * 2$ to avoid the computation of the square root in the modulus of a complex number). In order to make the problem doable, the maximum number of steps is also limited: if that number of steps is reached, then the point c is said to belong to the Mandelbrot set.

Task granularity analysis

We will analyze, using Tareador, the potential parallelism for two possible task granularities that can be exploited in this program: a) Row: a task corresponds with the computation of a whole row of the Mandelbrot set; and b) Point: a task corresponds with the computation of a single point (row,col) of the Mandelbrot set.

As we can see in the following task graphs, in the first one, where we defined a task for each row, there are only 8 tasks. However in the second graph, which the tasks are defined at point level, there are many more tasks. This is not surprising, the second version works on a much finer granularity level and thus defines a far greater number of tasks.

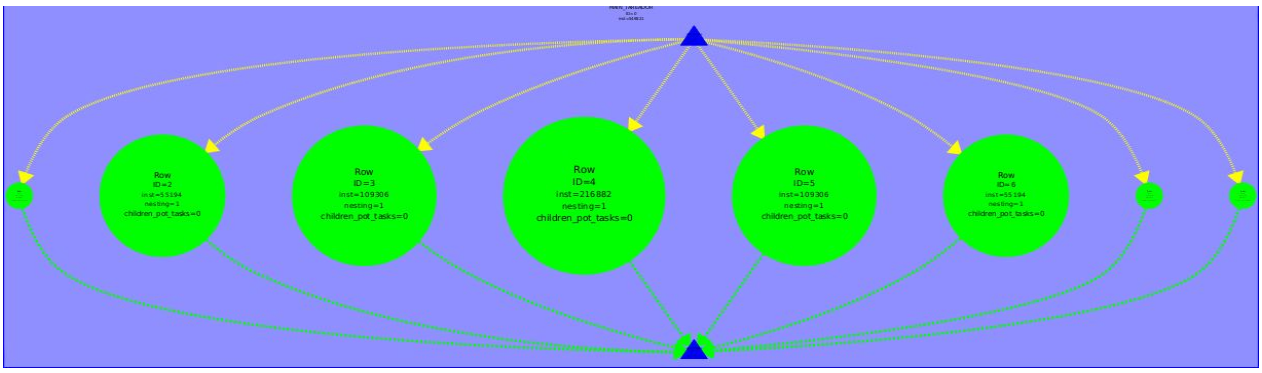

Fig1. Tareador graph of mandel-tareador in a row granularity version.

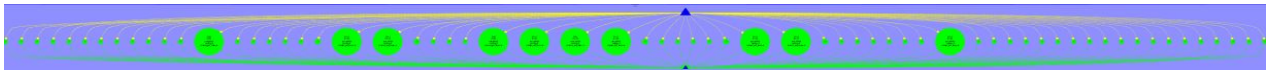

Fig2. Tareador graph of mandel-tareador in a point granularity version.

The two most important common characteristics we observe in the graphical versions of the task graphs are the following. First, the fact that the different tasks do not have any dependencies between them and second, that the size of the tasks is not the same, so the workload is not equally shared among all the parallel tasks.

For the graphical version of the code we obtain the following graphs:

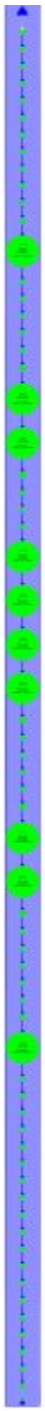

Fig3. Tareador graph of mandeld-tareador in a point granularity version.

Fig4. Tareador graph of mandeld-tareador in a row granularity version.

Looking at Fig3 and Fig4 we realize that, unlike the non-graphical version of the code, there is no potential parallelism with this one, on the grounds that every task is dependent from the one before it.

This makes little sense, so we will try to find what section of the code is causing this serialization and think about how it can be fixed.

The section of the code that is causing the serialization is the following:

```
/* 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);
}
```

Fig4: Region of the code causing serialization.

This is caused by a global variable from the xlib library used in one of the two functions called in this section of the code. Using the tools provided by Tareador we discover the name of that variable. Here is the information we get when investigating the dependencies.

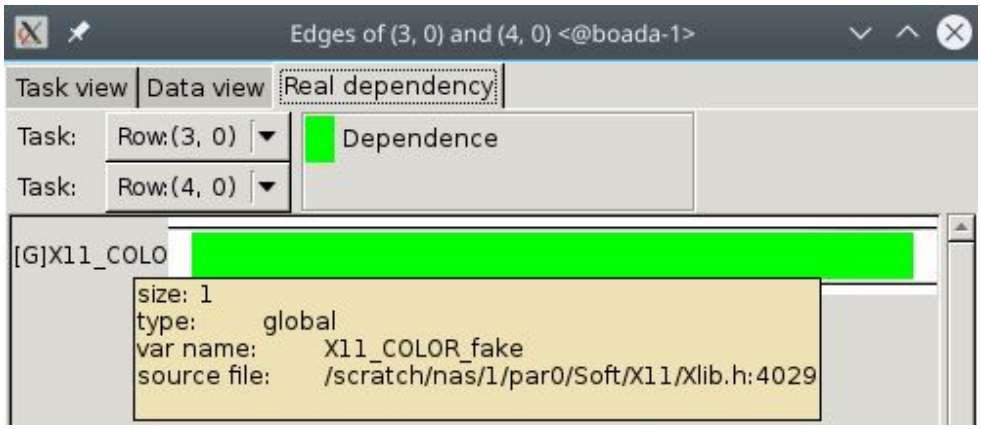

Fig6. Tareador information of the edges that are causing dependency.

To avoid those dependencies with OpenMP code we can use the pragma “critical” right before the call to the function that is using this variable.

OpenMP task–based parallelization

Next we will study the different parallelization techniques provided by the OPenMP API. We will begin by trying to discover the best way to parallelize our code using the omp directive “task”.

First we must decide which strategies we are going to use to decompose our program in various tasks. In the previous section we analyzed two possibilities; row decomposition and point decomposition. We will start with the first one.

To implement a task decomposition based on the row strategy we must write a pragma which defines a task for the computation of each row of the mandelbrot set image. We must not forget to also add the *#pragma omp critical*, where the two functions from xlib library are called, which caused data dependencies between tasks as we saw in the previous section.

To check the changes made, we have added a file named “mandel-omp-row-task.c” which contains the code, in source folder.

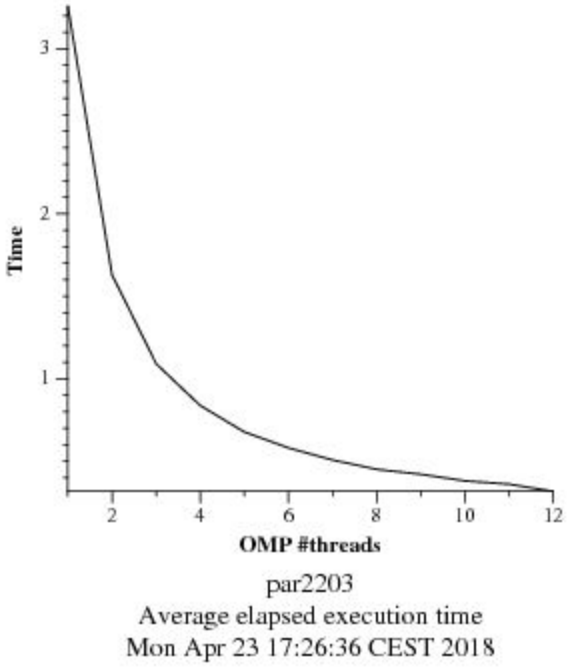

Fig7. Execution time plot using Row parallelization strategy

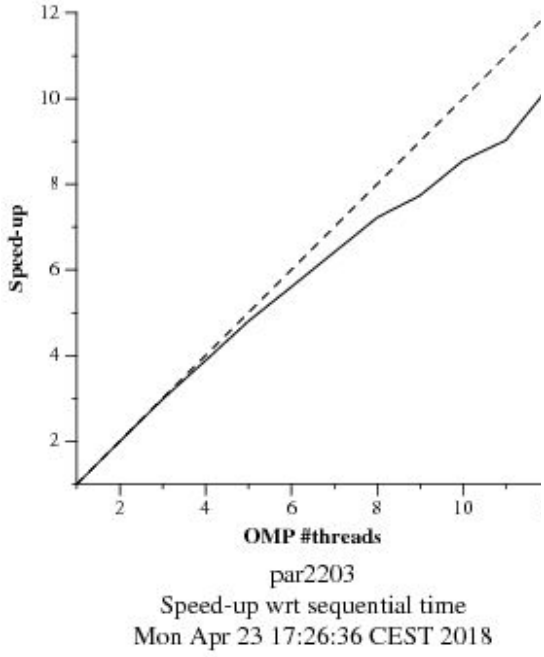

Fig8. Speed-up plot using Row parallelization strategy

Fig7 and Fig8 show the results of the plot of execution time and speed-up of our program with strong scalability. The execution time decreases significantly as more threads are used. In the second plot we can appreciate how the speed-up is quasi linear. It deviates from linearity when too much threads are used due to synchronization overheads.

We can therefore conclude that this decomposition is very appropriate.

Next we will look at the characteristics of the second strategy, which is going to define a task for the computation of each point of the mandelbrot set. Again, to see the changes made in the code, we added a file named “mandel-omp-point-task.c” in the source folder.

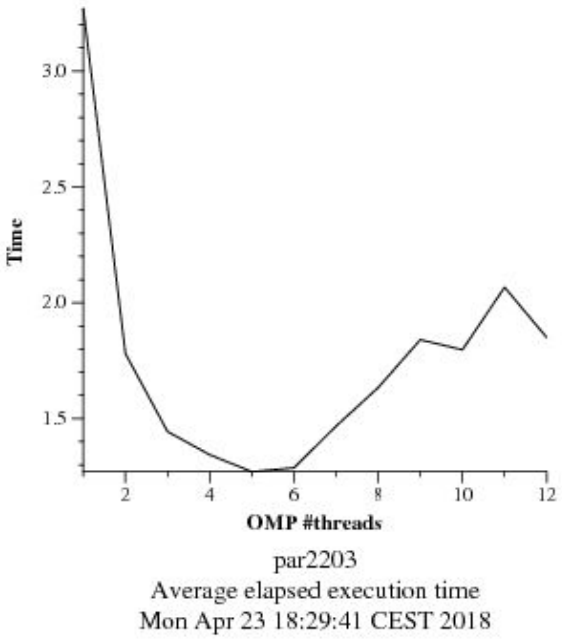

Fig9. Execution time plot using Point parallelization strategy

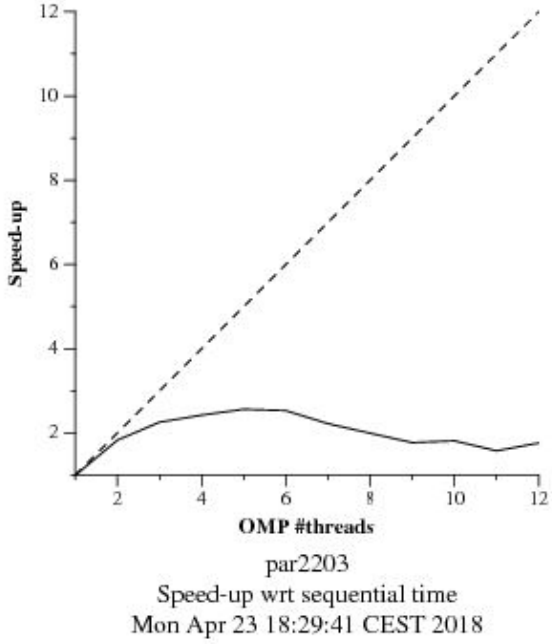

Fig10. Speed-up plot using Point parallelization strategy

Fig9 and Fig10 show the plots for the execution time of our program when using the point parallelization strategy. We can see that there is some improvement when using up to 4 threads, but any more threads than that and it starts to be a hindrance to performance.

There may be different reasons for this. First of all, the number of tasks defined is very large. There is a task for every point in the image. This is going to increase the impact of synchronization overheads significantly. Also, when we defined a critical region we caused a certain sequentiality in the part of the code that prints the output. When we have only as many tasks as rows this is not a problem, but when there is a bigger number of tasks (quadratically as

many) there is a significant decrease in performance. Finally, there is another cause for the difference of performance between the two strategies which has to do with an uneven workload. Not every point in the set is going to need the same computation effort. There are some point which quickly converge and thus their color is quickly defined and others that do not end and their computation is stopped by the maximum step limit imposed in the code. This means that there is going to be a colossal difference between the workload of different tasks and as we have seen in other laboratory assignments, this is a drawback to performance.

Therefore, dividing our program into a task for each point is not beneficial. The task granularity is excessive and causes many problems that result in a performance less than ideal.

In conclusion, the best way to decompose our program to exploit its potential parallelism is to define a task for each row of points in the mandelbrot set.

OpenMP taskloop–based parallelization

In this section, we will devise a way to implement our code using the taskloop directive. We will do so with the same two decomposition strategies, per row and per point.

We will begin using a row decomposition.

The taskloop directive allows us to define the grain size, which determines how many iterations will constitute a single task. We will test the behaviour of the program when using 8, 16, 32 and 64 as grain size. Smaller granularity will mean more potential parallelization, but also a bigger impact of synchronization overheads. We will try to determine which has best performance.

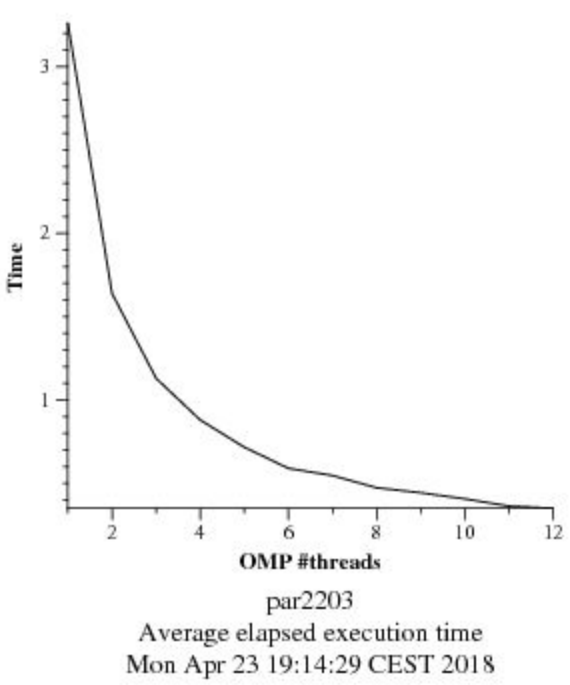

Fig11. Execution time plot using Row decomposition with task-loop parallelization and grainsize(8)

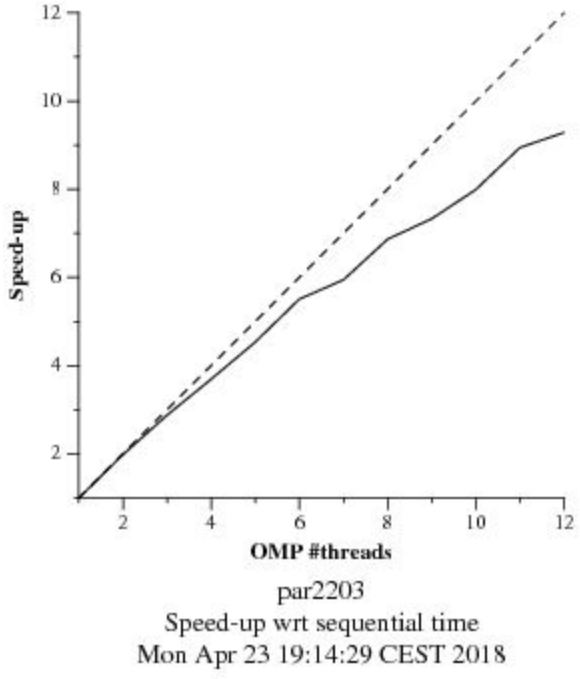

Fig12. Speed-up plot using Row decomposition with task-loop parallelization and grainsize(8)

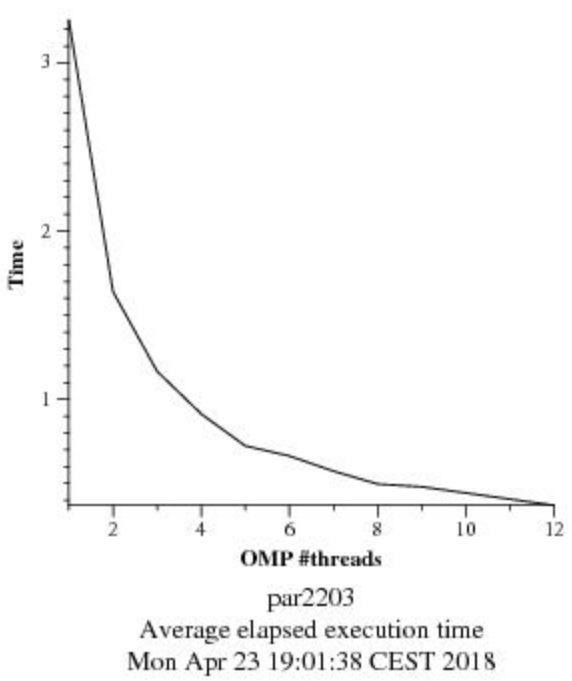

Fig13. Execution time plot using Row decomposition with task-loop parallelization and grainsize(16)

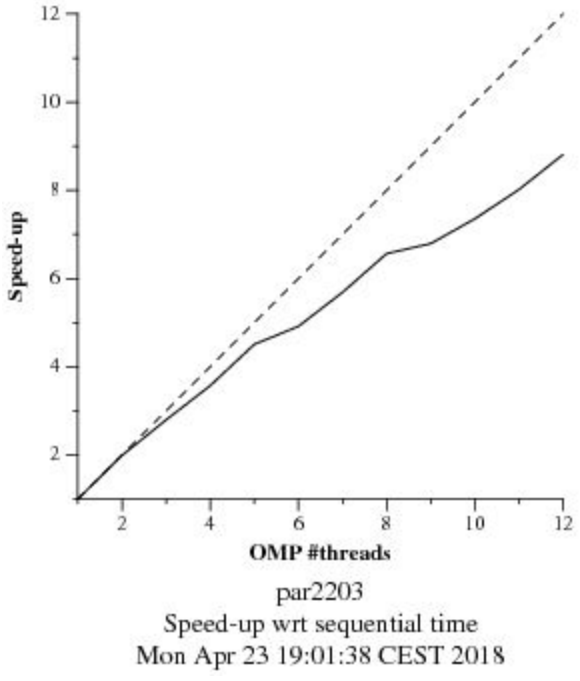

Fig14. Speed-up plot using Row decomposition with task-loop parallelization and grainsize(16)

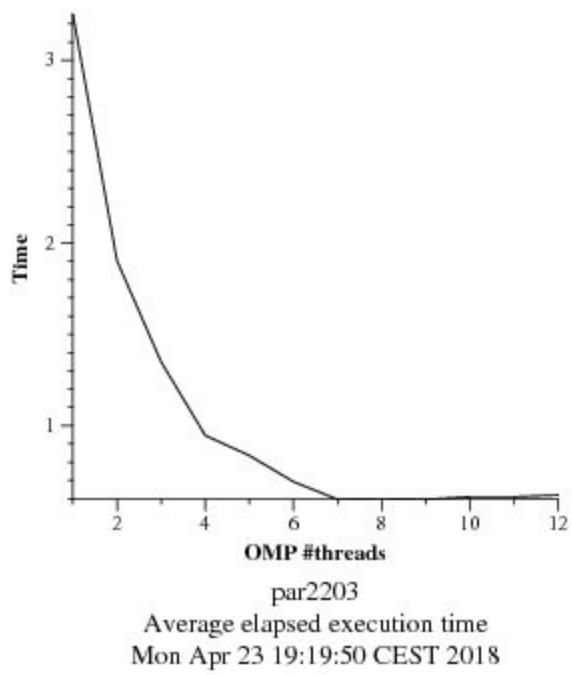

Fig15. Execution time plot using Row decomposition with task-loop parallelization and grainsize(32)

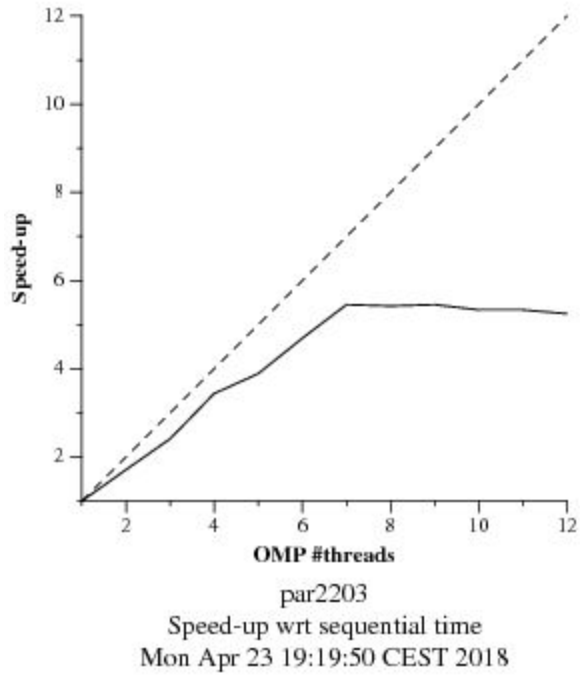

Fig16. Speed-up plot using Row decomposition with task-loop parallelization and grainsize(32)

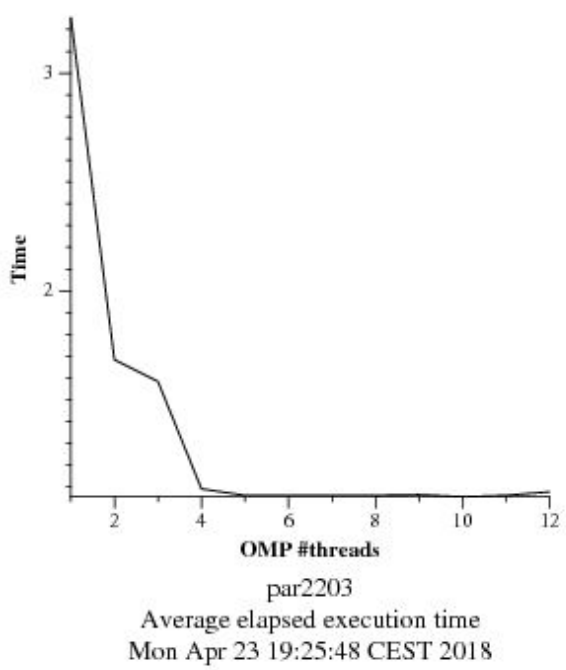

Fig17. Execution time plot using Row decomposition with task-loop parallelization and grainsize(64)

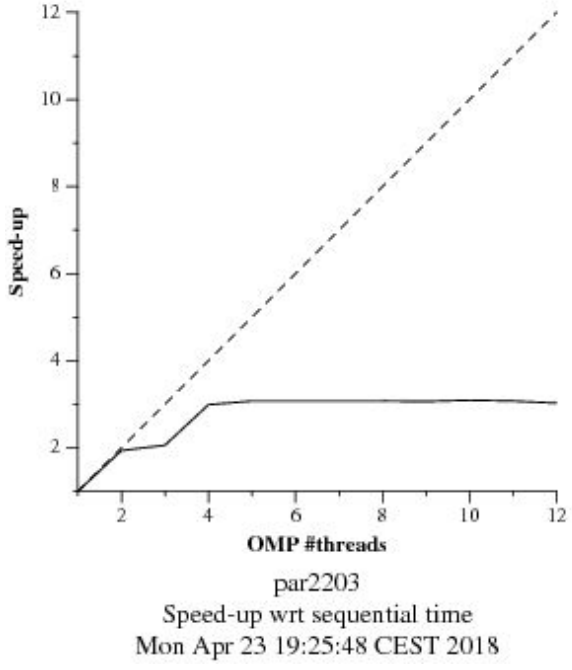

Fig18. Speed-up plot using Row decomposition with task-loop parallelization and grainsize(64)

The previous plots, fig11-18, show how the behaviour of our program changes with different grain sizes. The grain size is the number of iterations of the loop that constitute a task.

Looking at the speed-up plots, we see that after a certain number of threads, there is barely any speed-up for the bigger sizes of grain. However, with a smaller grain size, the speed-up is quasi lineal. If we take in consideration the penalization for overheads, we can ignore the deviation from linearity and conclude the best value for grain size is eight.

This can be explained as follows; the smaller the grain size, the more our program can be parallelized, because there is going to be more balance between tasks. Given that the total number of iterations is proportionally not big, with a bigger grain size, (for example 64) there is not much difference of speed-up in respect to the sequential version, because there are only so many tasks of 64 iterations that can be defined.

Therefore, we conclude that a smaller grain size is better, but we have to be careful to not over do it, because after a certain point, the penalization for creating extra tasks is going to have a bigger toll than benefit.

Now, we will take a look at the point decomposition. As we saw in previous sections, this strategy might not be ideal, given the number of tasks is so abysmal. However, we might be able to exploit the tools given by the taskloop directive which let as tweak the size of the tasks, and therefore define a more reasonable number of tasks.

We will execute our program with the following grain sizes: 8, 16, 32, 64.

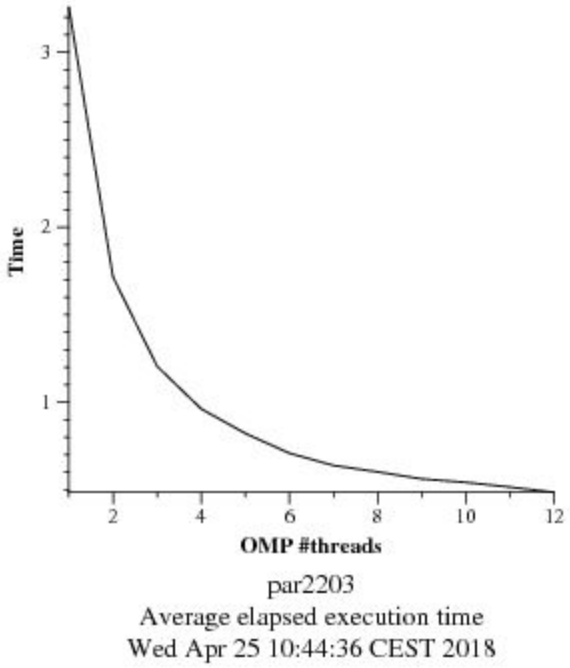

Fig19. Execution time plot using Point decomposition with task-loop parallelization and grainsize(8)

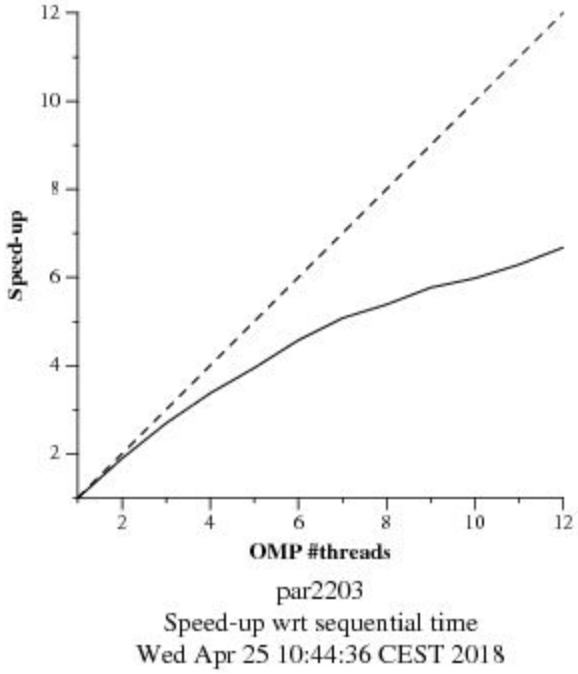

Fig20. Speed-up plot using Point decomposition with task-loop parallelization and grainsize(8)

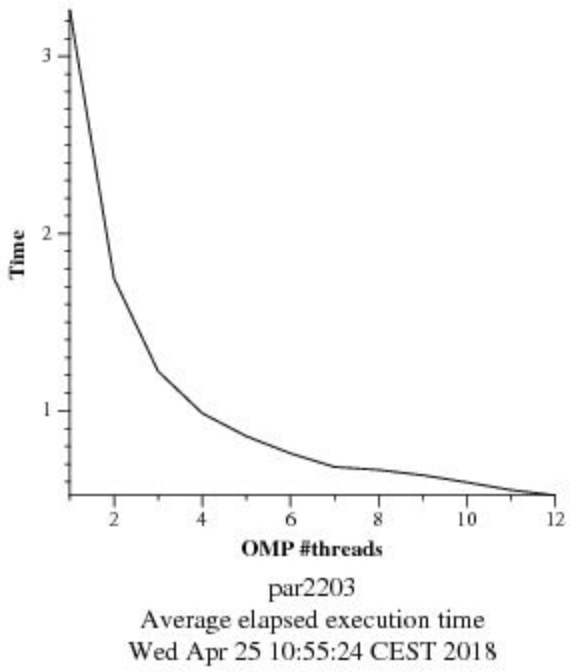

Fig21. Execution time plot using Point decomposition with task-loop parallelization and grainsize(16)

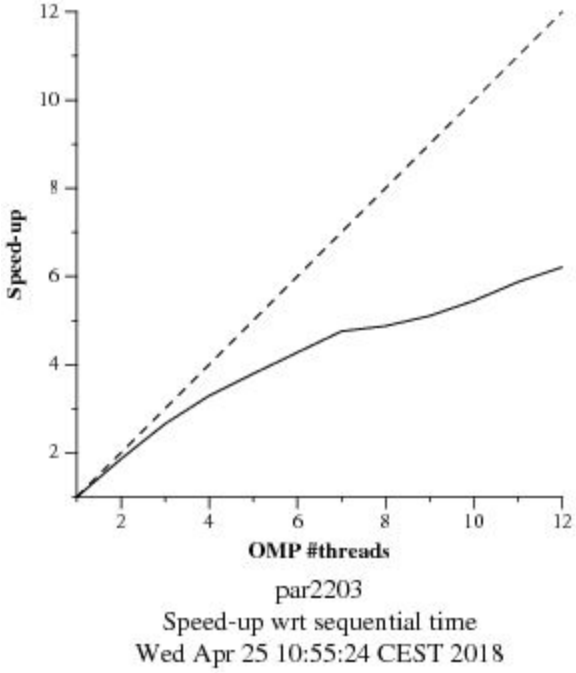

Fig22. Speed-up plot using Point decomposition with task-loop parallelization and grainsize(16)

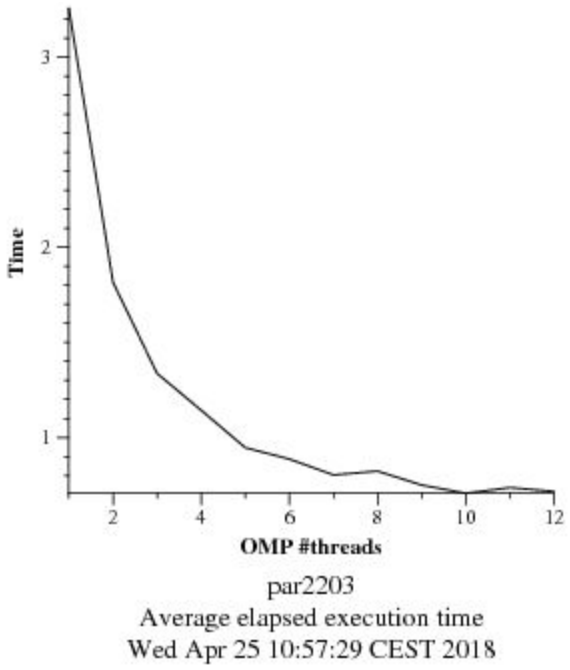

Fig23. Execution time plot using Point decomposition with task-loop parallelization and grainsize(32)

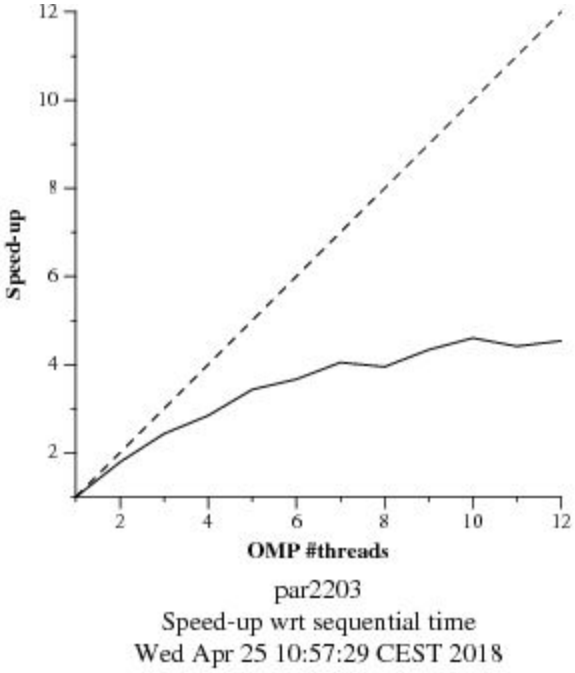

Fig24. Speed-up plot using Point decomposition with task-loop parallelization and grainsize(32)

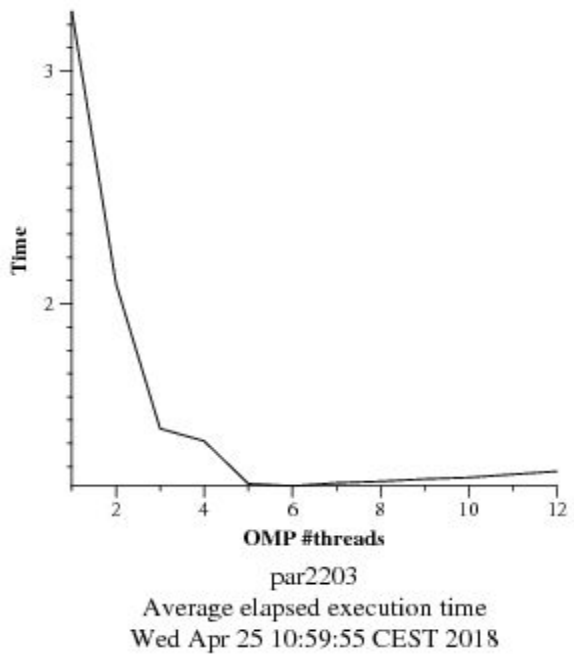

Fig25. Execution time graph using Point decomposition with task-loop parallelization and grainsize(64)

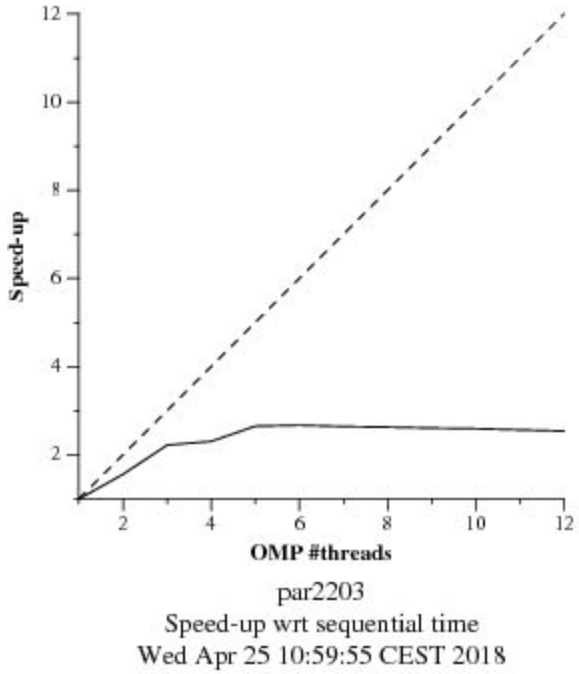

Fig26. Speed-up graph using Point decomposition with task-loop parallelization and grainsize(64)

The previous plots show the average elapsed execution time and speed-up as function of the number of threads, for the different grain sizes we discussed, using the taskloop directive and the point decomposition strategy.

First of all, we will focus our attention on Fig20. We see that the speed-up is way better in this version compared with the previous one that used the same point decomposition but the task directive, instead of taskloop. The reason behind this improvement is that a more reasonable number of tasks is defined and thus there is a way better workload balance between tasks (as explained in the previous section, there might be an enormous difference of computation needs between two different points). However, if we start increasing the grain size (Fig21-26) we run into the same problems as with the row decomposition, i.e. the task size is too big and we lose the capacity to parallelize and our program begins to behave more “sequentially”.

We arrive at the conclusion that the *taskloop* gives us an advantage with respect to *task* when decomposing with the point strategy. However, the performance of the program is highly dependant on the grain size value defined, for both decomposition strategies. The best value is the one that is small enough to allows to take advantage of the potential parallelization, but big enough so that there is not an excessive overhead caused by creating and managing tasks.

OpenMP for–based parallelization

We will now try to analyze the for-based parallelization. We will begin focusing on the average execution time and the speed-up in respect to the sequential version. With the same problem decompositions as before, row and point, we will execute the program using four different loop schedules (always using 8 threads and 10000 iterations) and reason about the results. The schedules are: “static”, “static,10”, “dynamic, 10”, “guided, 10”.

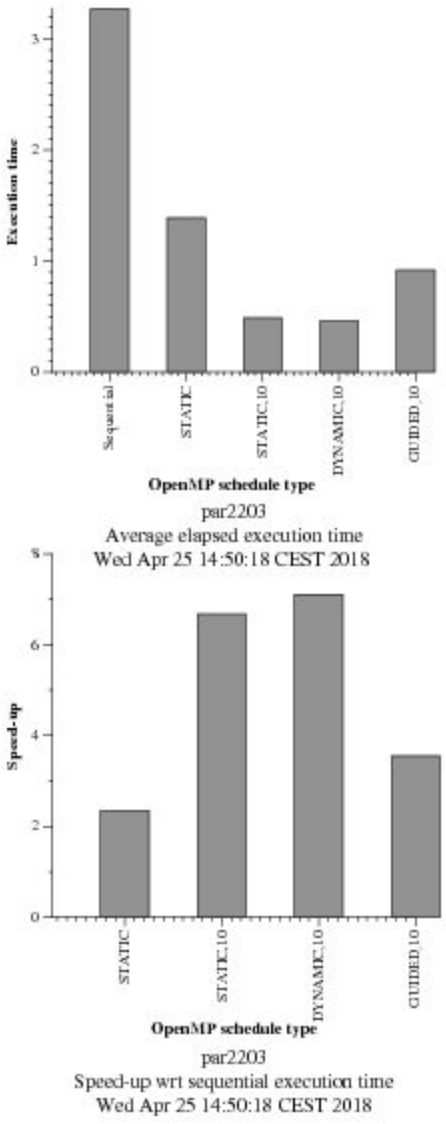

Fig27. Execution time and speed-up graphs using Row decomposition with for parallelization

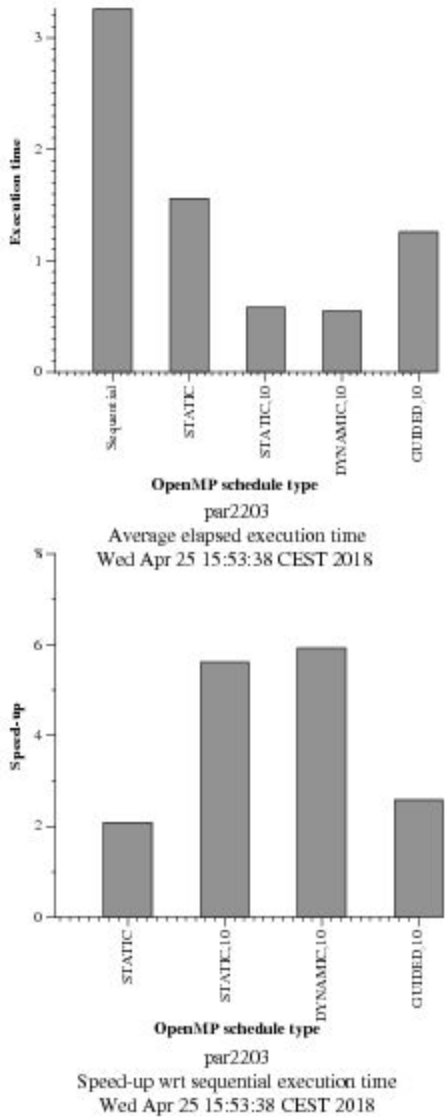

Fig28. Execution time and speed-up graphs using Point decomposition with for parallelization

The previous plots, show the results of our experiments. At first glance, we see that the row and point decomposition look very similar. Upon further inspection, however, we see that the point decomposition has a worse speed-up in every schedule. As in previous sections, we see that the best task decomposition strategy is to divide by rows and not points.

Let us look at the behaviour of the program when different schedules are used. The best performance is achieved with the dynamic schedule. This makes sense because it can dynamically grab chunks of the right size, which counteracts the inherent workload imbalance of the program.

Now that we have determined that the best strategy is to decompose the problem by rows, let us analyze even further the behaviour of the program with different schedules, using the tools provided by paraver. We will execute the program using eight threads and 10000 iterations, once for each type of schedule: “static”, “static,10”, “dynamic,10” and “guided,10”.

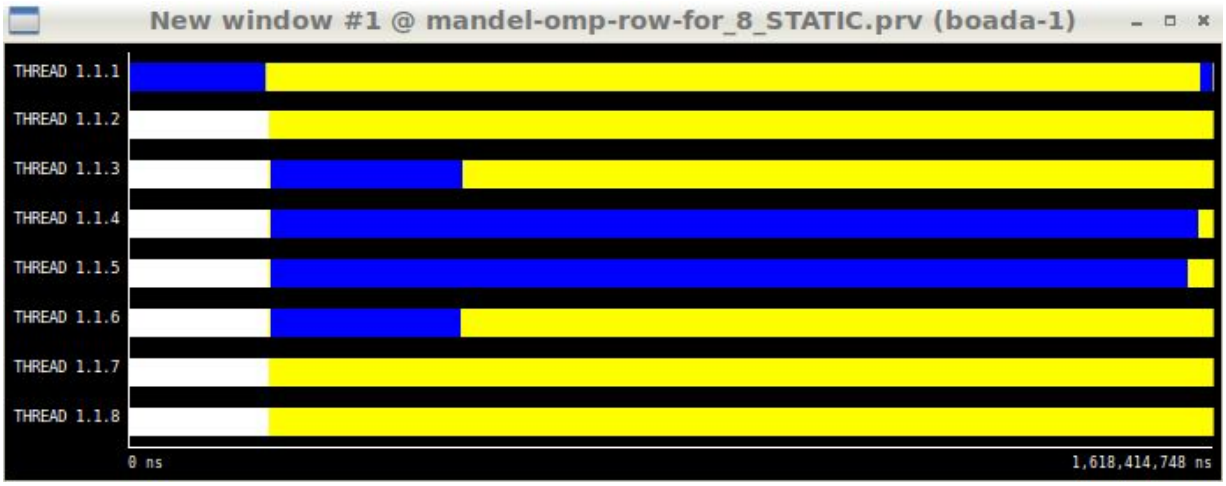

Fig29. Paraver trace using Row decomposition. Schedule = “static” with for parallelization

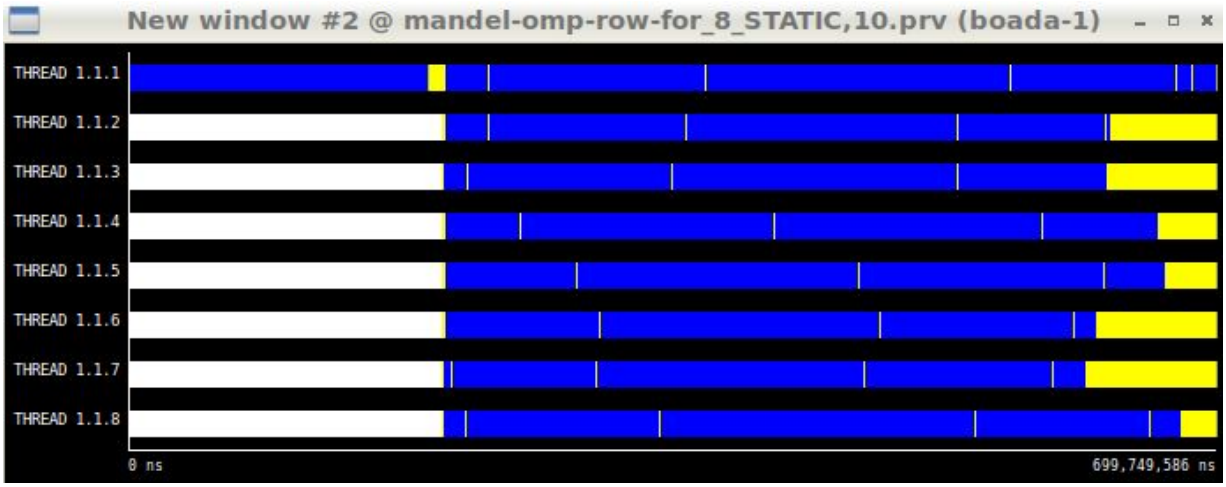

Fig30. Paraver trace using Row decomposition. Schedule = “static,10” with for parallelization

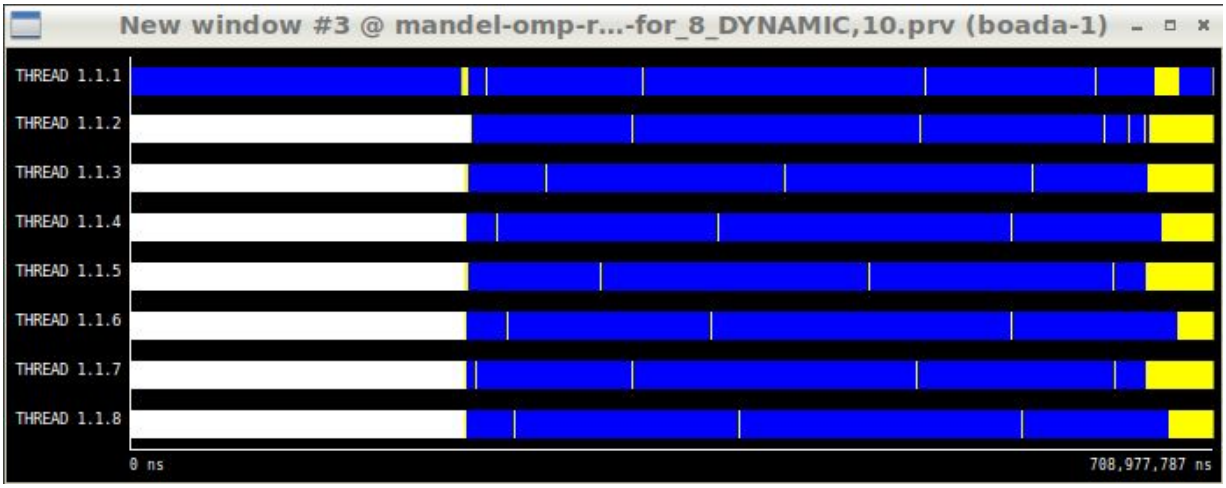

Fig31. Paraver trace using Row decomposition. Schedule = “dynamic,10” with for parallelization

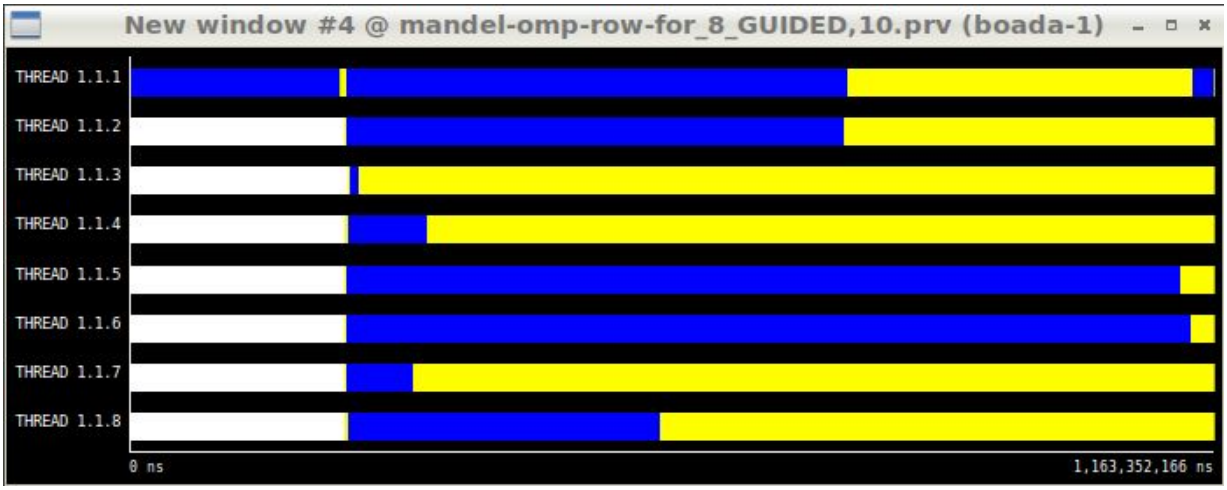

Fig32. Paraver trace using Row decomposition. Schedule = “guided,10” with for parallelization

We have decided to include the traces of the execution obtained using paraver because they provide a visual representation of the differences observed. The zones in blue, represent the time any given thread is actually running the program. The ones in yellow represent time dedicated to scheduling forks and joins, and when a thread is not yet created its trace is white. With a simple look at the traces we can see which schedules perform better. This results agree with the ones we obtained in the previous section (see Fig. 27).

In the following table we will include the numerical results for a more precise comparison.

	static	**static, 10**	**dynamic, 10**	**guided, 10**
Running average time per thread	445,942 µs	473,070 µs	482,515 µs	454,997 µs
Execution unbalance	0.3214	0.6842	0.6966	0.5012
SchedForkJoin	989,199 µs	50,423 µs	34,336 ns	506,053 µs

Fig33. Table with numerical data for comparison of loop schedules

First of all, it can be observed how the average running time per thread is very similar in all four schedules. However if we measure the execution unbalance, which is the average time divided by the maximum, we get a better picture. A lower value is better, because it will mean a more equitative distribution of work among threads. Therefore, in par with what was said before, we observe that the best execution time is achieved when “dynamic,10” is used as loop schedule, with “static,10” being second best. The next line in the table gives similar information. More time spend scheduling forks and joins means less time actually running the program.

Optional

Optional 1

As an extra research for this session, we will explore a different task decomposition. In the previous sections, we tried dividing the program in different tasks either by rows or by points, and then, using different directives we grouped this tasks to distribute their computation among threads. In this section, we will use the collapse directive to break our program into smaller parts regardless of their meaning (i.e. neither by rows nor points) to exploit its potential parallelism. This way, the image will be computed in a different manner.

Later we will compare the results with the strategies from previous sections and determine which schedules are best suited for the aforementioned directive.

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

Fig34. Section of the code with the for and collapse directives

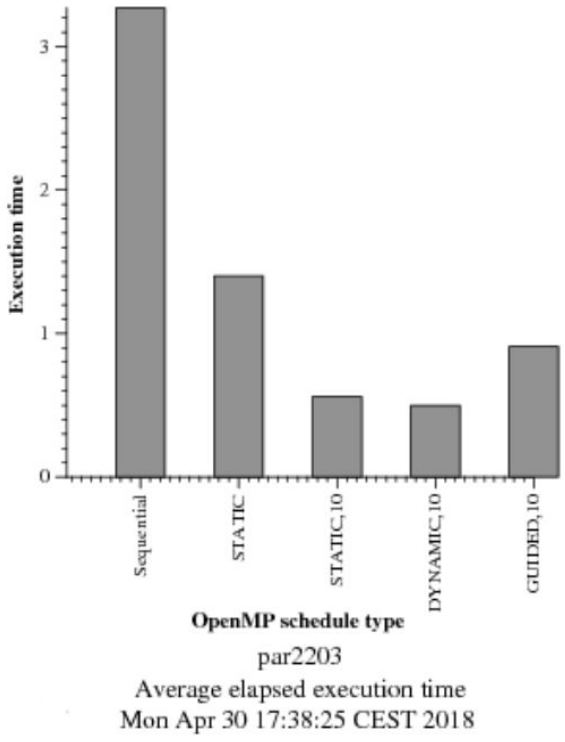

Fig34. Execution time graph using the Fig34 clauses

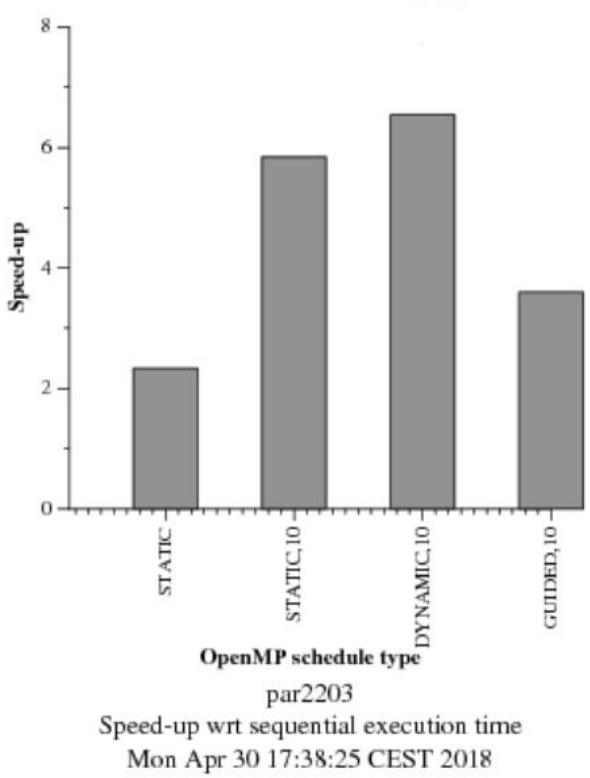

Fig35. Speed-up graph using the Fig34 clauses

Comparing this results with the ones we obtained using the Row and Point decomposition strategy using the for directive, we notice they are very similar (see Fig27 and Fig 28). This makes sense because what determines the chunk size is the schedule used. Similar to the previous strategies, the best schedule is “dynamic,10”. This can be explained by the same arguments we used in the previous sections, because the same task characteristics apply. In fact, if we think about how the division is performed we realize it is very similar in both cases. In all cases the chunk size will be the same, the only difference being that this new strategy does not care if a computation belongs to a certain row or point.

Therefore we conclude that this strategy is perfectly valid but not any better than using the for directive with the same schedules.

Optional 2

In this section we will try another strategy. We will try to combine the for directive with the task directive to try to exploit the advantages of each one.

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

Fig35. Section of the code with the for directive

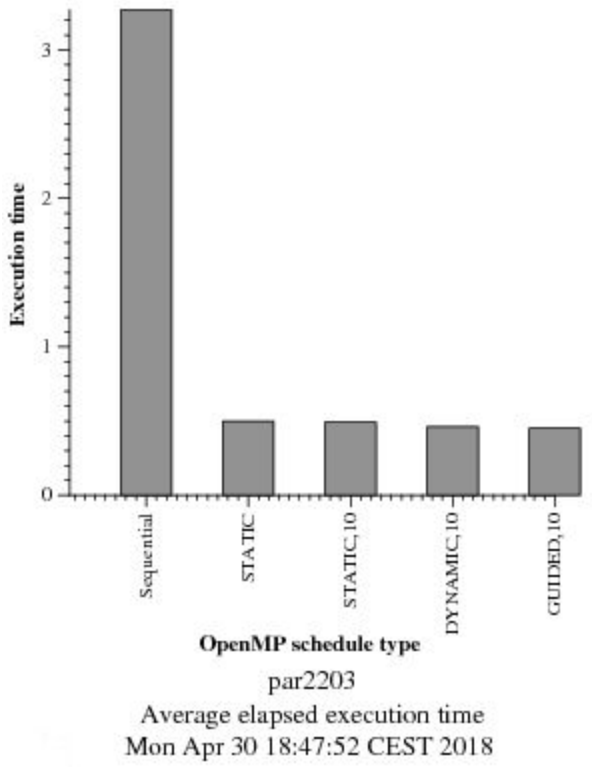

Fig36. Execution time graph using the Fig35 clauses

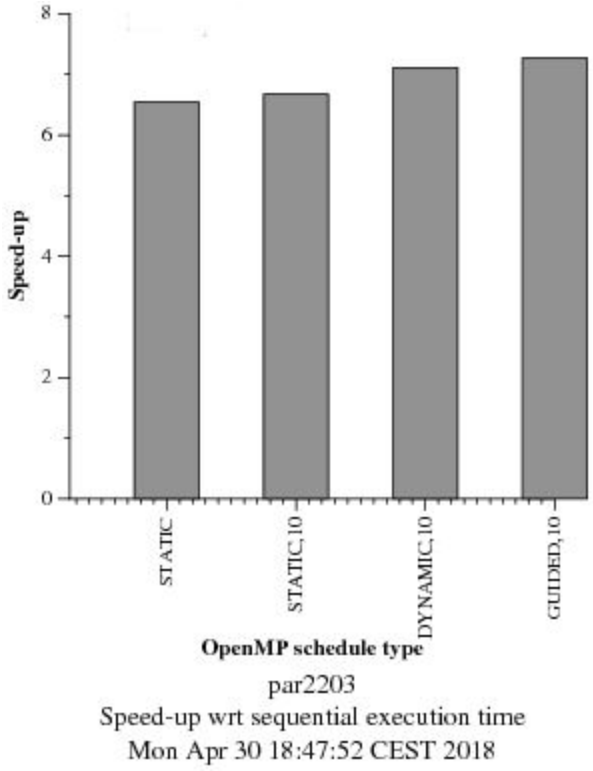

Fig37. Execution time graph using the Fig35 clauses

The results show a very similar result among the different schedules, with the best one being “guided,10”. The explanation for this may be that when we use the task directive we are dividing the problem in a given set of lesser tasks. When using the different schedules, the tasks are assigned to threads but are all very similar, therefore it matters little in how they are being distributed. The execution times are, in the best case, very similar to the best of the previous version, only slightly better. With this information we conclude that this strategy is as valid as the previous ones and slightly better if we consider the average times of the different schedules.

Conclusions

After having finished this session, we have discovered and learned new strategies and tools to exploit the potential parallelism of our programs. We have analyzed different tools and understood their behaviour, their strong points and their weaknesses.

We began focusing only on the size of the grain; how can we divide our program into smaller tasks and how the size of this tasks affects its potential parallelism. Later we discovered that the grain size is not the only factor in performance. There are things such as synchronization overheads that can heavily tamper our execution time.

We learned that the best performance is achieved when we know how to balance the grain size with the respective overheads and also distribute the workload evenly between the many tasks. To do so we looked at different directives provided by the OpenMP API such as task, task-loop and for. With the latter we experimented with different scheduling options and discovered which were the best for our program and what characteristics of it make that so.

This way, when we face the problem of parallelising a different program we are going to be able to reason which strategy is going to be best suited for that particular program.

Finally we can say that we also learned things not directly related to OpenMP or parallelism. We learned different experimentation strategies and how to organize a document to report the data gathered in an experimental environment.