# PAR Laboratory Assignment

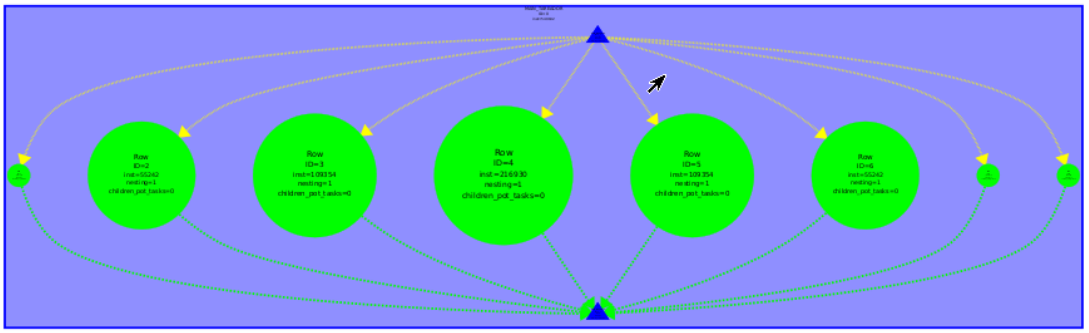
# Lab 3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

Alvaro Moreno Ribot, **PAR 2214**
Albert Escoté Alvarez, **PAR 2205**

# Task decomposition analysis with Tareador

## Row Strategy

We will be analyzing the code for the *mandel-tar.c* file. This code has the initialization of the Tareador API (start and stop) and we are asked to set the tasks for a *row* decomposition strategy. This means that a task will compute a row of the mandelbrot set. To do this we will be using the `tareador_start/end_task("task_name")` functions. We can note that this code is fully parallel and has no dependencies between tasks, we can also see that the work balance between tasks is not equal as there are tasks getting much more work than others. The task granularity is not homogeneous, so this is not the most efficient way to achieve parallelism.

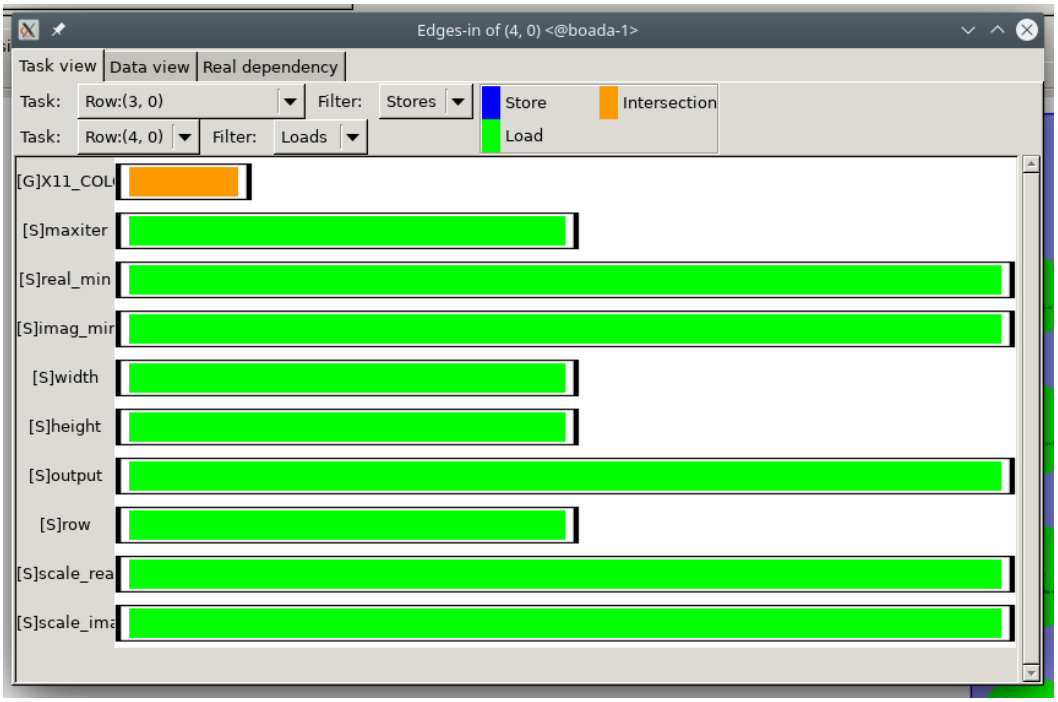


Next, we are asked to get the TDG for the same execution but using the *-d* option which will display the generated Mandelbrot set. We can see, in the image on the left, that this time the execution is completely sequential and has no parallelism.

In this case, when we use the ***-d*** option the following code is executed:

```
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);
    }
}
```

If we take a look at the Display > edges-in tool from Teareador, we will see the following information:




As we can see, the variable ***X11_COLOR_fake*** that is creating the dependency. When we use the ***XSetForeground()*** function (that sets the color of the pixel) it will set the global *XWindow* variable ***X11_COLOR_fake***. We have a clear datarace as all threads will try to modify this variable.

To solve this, we can use a ***#pragma omp critical*** for each call to *XWindow* functions to avoid this dataraces. But this wouldn't be the most efficient way to solve this as we know that using the critical directive is not always an efficient way to solve problems as it creates heavy overheads. To improve this, if we want to get more efficiency, we could make the computation of all colors in parallel and then use a single thread to set the color for every pixel (sequentially).

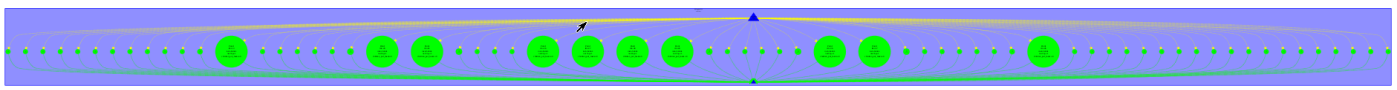
This time we have a global variable called **histogram** that is being accessed from all threads and creates a clear datarace.

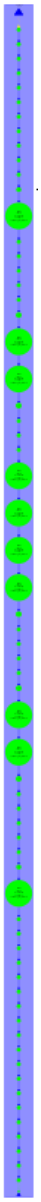
To solve this problem we could use an atomic directive or use a similar strategy as the explained before for the X11_COLOR_fake variable taking advantage of the fact that we have the k value stored on the output matrice.

## Point Strategy

Now we will do the same but this time using a point strategy. This means that a task corresponds to a single point (or pixel) of the mandelbrot set, so we will be moving the task generation to the most inner loop of the code.

For the execution without any options, we get a very similar TDG as we got with the row decomposition in terms of parallelism and granularity, but this time with a broader number of tasks. The workload (granularity) is also very irregular between tasks, but we have a fully parallel execution.





For the ***-d*** option (display) we can see (on the image on the left) that we also get the same behaviour as we got from the other decomposition strategy. The problem is caused by the very same global variable from XWindow and the solution would be the same.

On the right image we have the execution with the ***-h*** (histogram) option. As all the other executions, the behaviour is the same but with a bigger number of tasks due to the finer-grained tasks we are now using (a task per point).

# Implementing task decompositions in OpenMP

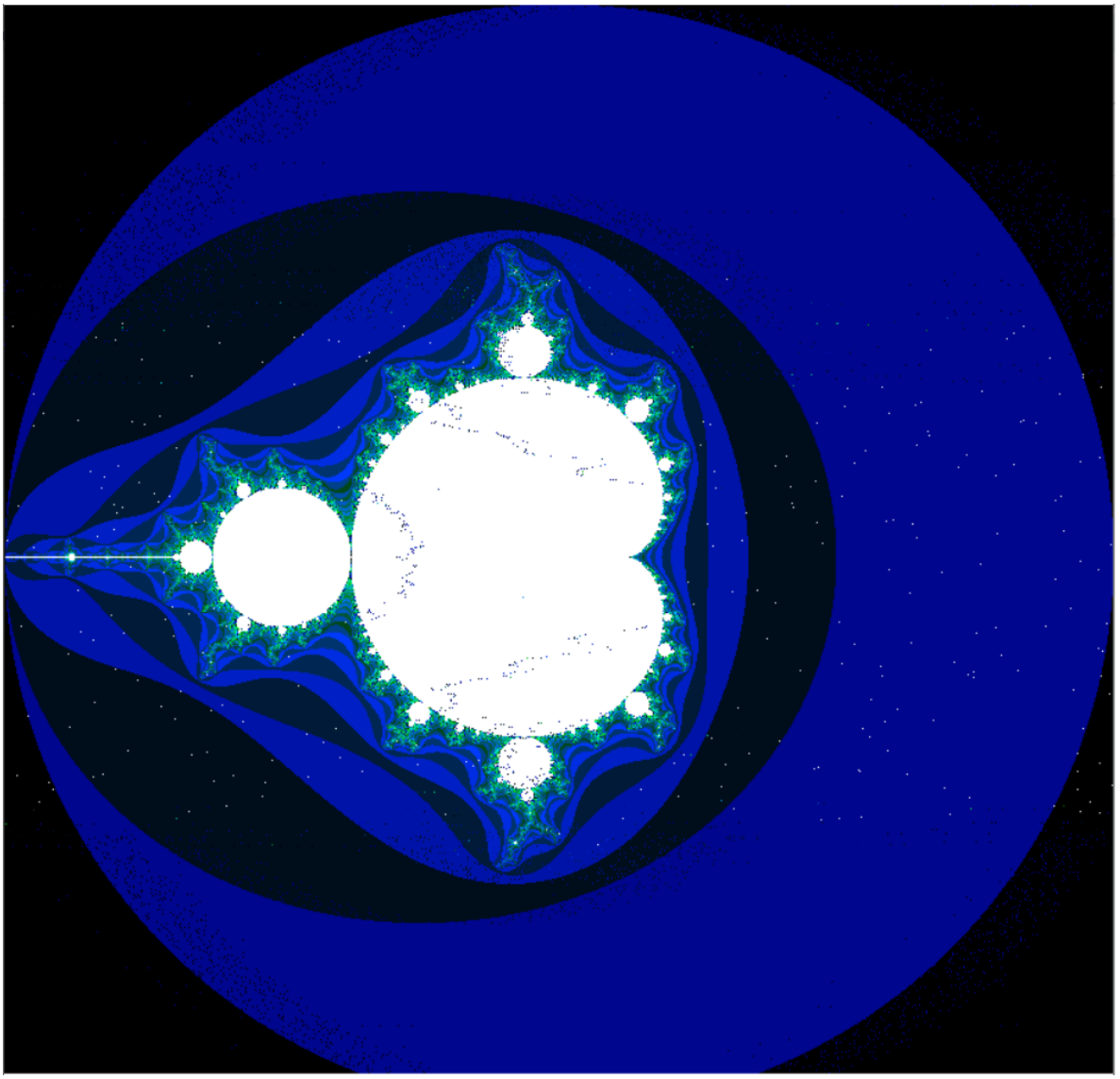
## Point decomposition in OpenMP

### Point strategy implementation using task

In this section we are told to insert the missing OpenMP directives to honour the dependencies detected in the analysis of Point Strategy we did before. So we add #pragma omp critical before both XWindow calls.

Then, we execute the binary generated with the Makefile with one and two threads, using this commands: OMP _NUM_THREADS=1 ./mandel-omp -d -h -i 10000 / OMP_NUM_THREADS=2 ./mandel-omp -d -h -i 10000.

Both are generated correctly but the one with 2 threads is not very defined. Here we insert the image generated with two threads, as we can see it is influenced by some type of interference.



To measure the reduction in time due to the parallel execution we submit the mandel-omp binary with the sbatch command with one and eight threads. We get the following results:

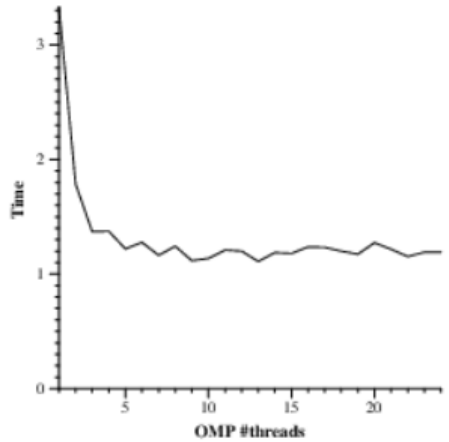
1 thread:

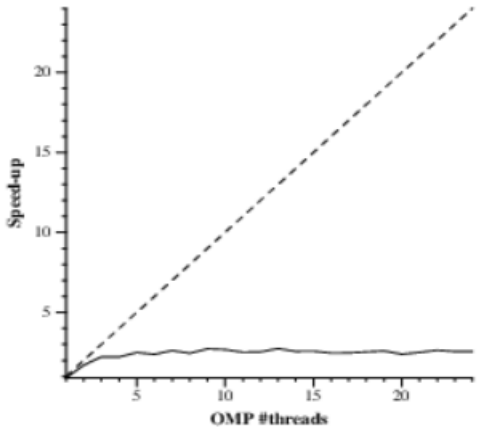
```
3.32user 0.00system 0:03.45elapsed 96%CPU (0avgtext+0avgdata 7092maxresident)k
144inputs+5088outputs (1major+1023minor)pagefaults 0swaps
```

2 threads:

```
9.59user 0.29system 0:01.38elapsed 716%CPU (0avgtext+0avgdata 8320maxresident)k
0inputs+5088outputs (0major+2614minor)pagefaults 0swaps
```

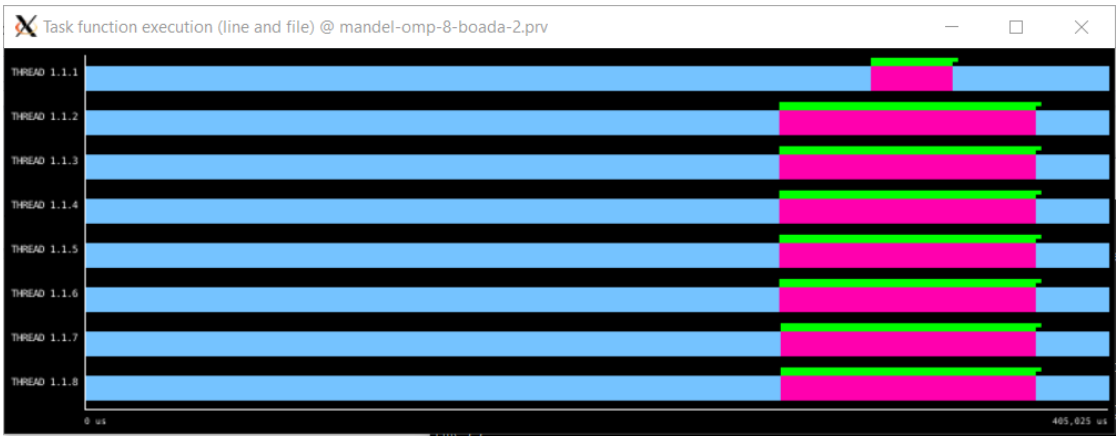
Once we get these results, we can submit the binary to the strong script to compare our results. We executed it with the following prompt command: sbatch ./submit-strong-omp.sh mandel-omp. And we get the following results:

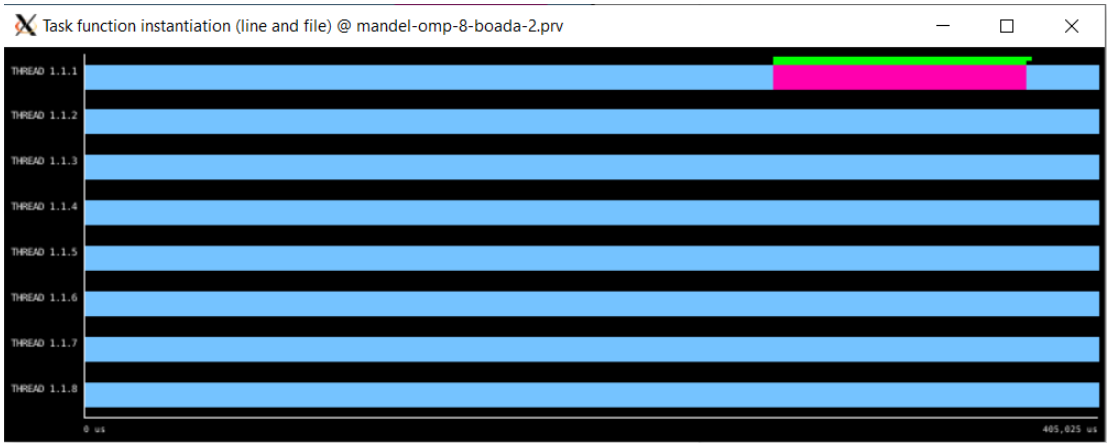



As we can see, the scalability is not appropriate. Time execution is approximately constant when we get 5 threads or more. Also we can see that the speed-up is not linear but remains practically constant.

Next step is about understanding better how the execution goes. We do an Extrae instrumented execution with 8 threads with this command: sbatch ./submit-extrae.sh mandel-omp 8. Then we open Paraver and open the trace generated before. We also open, as told, the configuration files OMP_tasks.cfg and OMP_tasks_profile.cfg and we obtain the following results:




Task profile (execution and instantiation) @ mandel-omp-8-boada-2.prv

| | Executed OpenMP task function | Instantiated OpenMP task function |
|---|---|---|
| THREAD 1.1.1 | 3,331 | 40,000 |
| THREAD 1.1.2 | 4,991 | - |
| THREAD 1.1.3 | 5,324 | - |
| THREAD 1.1.4 | 5,180 | - |
| THREAD 1.1.5 | 5,473 | - |
| THREAD 1.1.6 | 4,960 | - |
| THREAD 1.1.7 | 5,538 | - |
| THREAD 1.1.8 | 5,203 | - |
| | | |
| Total | 40,000 | 40,000 |
| Average | 5,000 | 40,000 |
| Maximum | 5,538 | 40,000 |
| Minimum | 3,331 | 40,000 |
| StDev | 659.45 | 0 |
| Avg/Max | 0.90 | 1 |

As we can see in the images we have shown, only one thread creates the tasks (Thread 1.1.1) and all 8 threads execute them. All threads have a similar time of execution with the exception of thread 1 that spends less time with the execution due to the time spent in instantiation. So as we can appreciate the granularity and is appropriate and for consequence we can say the load is well balanced.

## Point strategy with granularity control using taskloop

We will be controlling the granularity using the *taskloop* construct. We modified the code so it looks like this:

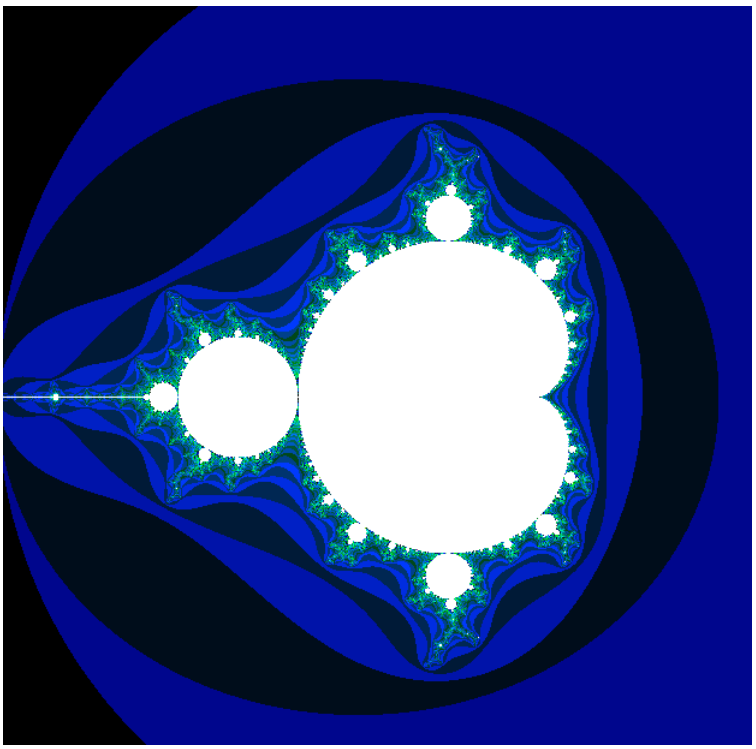
```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row)
    for (int col = 0; col < width; ++col) {
```

As it can be observed we did not specify the ***num_tasks*** nor the ***grainsize*** as we want OpenMP to take the decision on the granularity of the tasks.

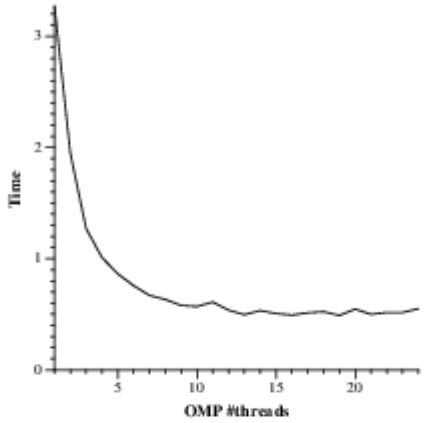
Now we compiled this new code using and run it using just 1 processor and the ***-d*** option with the following command:

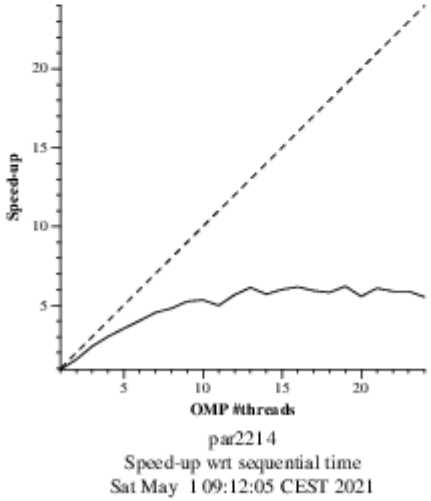
```
OMP_NUM_THREADS=1 ./mandel-omp -d -h -i 10000
```

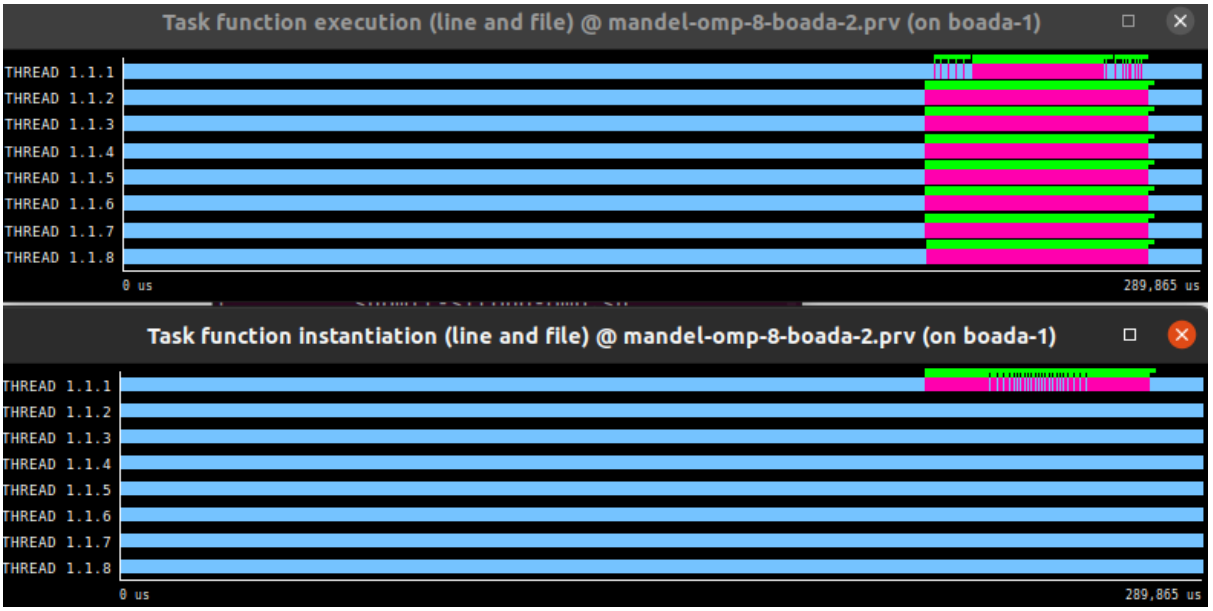
With the execution of the above command we got the visualization of the correct image for the Mandelbrot set:



Now we sent the ***./submit-strong-omp.sh*** script with the ***sbatch*** command for execution and we got the following scalability plot. We can see that the average elapsed execution time is slightly better for this version when compared to the previous (using task) when the number of threads is increased. We can also see that the speedup is not linear and increases (slowly) when we add more threads (note that when reaching 20 threads it seems to start decreasing). So we can confirm that this strategy is better than the previous one.






Task profile (execution and instantiation) @ mandel-omp-8-boada-2.prv (on boada-1)

| | Executed OpenMP task function | Instantiated OpenMP task function |
|---|---|---|
| THREAD 1.1.1 | 2,741 | 200 |
| THREAD 1.1.2 | 1,835 | - |
| THREAD 1.1.3 | 1,975 | - |
| THREAD 1.1.4 | 1,910 | - |
| THREAD 1.1.5 | 1,985 | - |
| THREAD 1.1.6 | 1,858 | - |
| THREAD 1.1.7 | 1,868 | - |
| THREAD 1.1.8 | 1,828 | - |
| | | |
| Total | 16,000 | 200 |
| Average | 2,000 | 200 |
| Maximum | 2,741 | 200 |
| Minimum | 1,828 | 200 |
| StDev | 285.58 | 0 |
| Avg/Max | 0.73 | 1 |

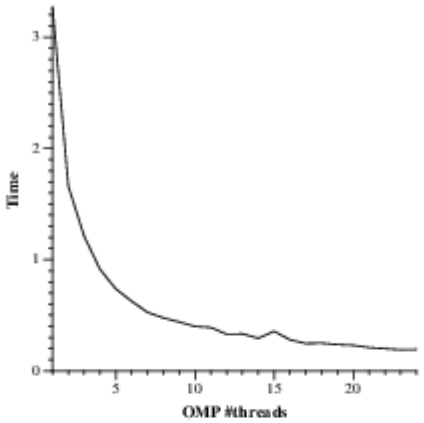
## Row strategy implementation

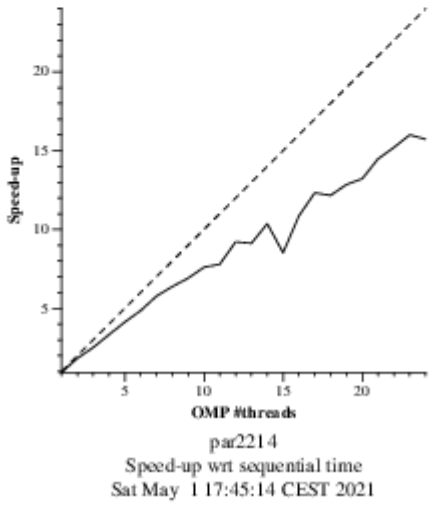
Now we will be testing a new parallelization strategy. We will create less tasks but we will have a heavier workload. To do this, we will be creating the tasks at the row level of the two for loops that calculate each point of the matrix.

This is the code that we used for this strategy:

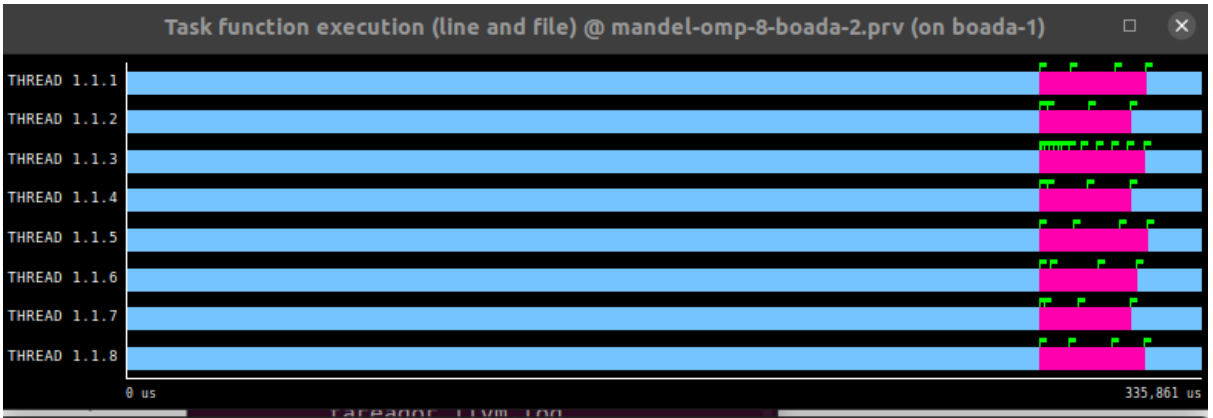
```
    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop nogroup
    for (int row = 0; row < height; ++row) {
          for (int col = 0; col < width; ++col) {
. . .
```

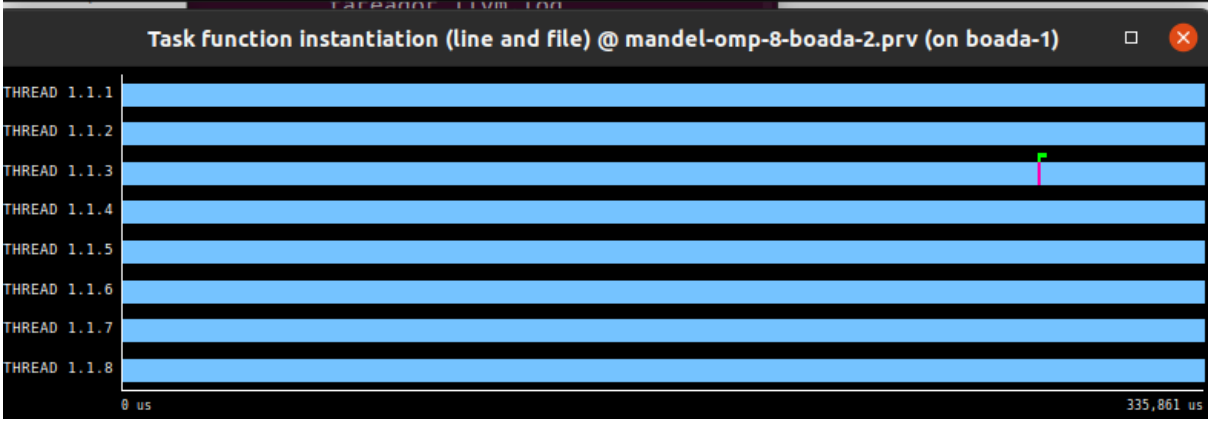
Here we can see the plot for the row strategy:




We can see that this strategy is significantly faster than the previous two versions. This may be because as we are creating tasks at a row level, we have less taks and thus less overhead, but we still have enough tasks so the threads take a sufficient amount of work. This is the best method out of the tested.




Task profile (execution and instantiation) @ mandel-omp-8-boada-2.prv (on boada-1)

| | Executed OpenMP task function | Instantiated OpenMP task function |
|---|---|---|
| THREAD 1.1.1 | 7 | - |
| THREAD 1.1.2 | 7 | - |
| THREAD 1.1.3 | 40 | 1 |
| THREAD 1.1.4 | 7 | - |
| THREAD 1.1.5 | 7 | - |
| THREAD 1.1.6 | 3 | - |
| THREAD 1.1.7 | 6 | - |
| THREAD 1.1.8 | 3 | - |
| | | |
| Total | 80 | 1 |
| Average | 10 | 1 |
| Maximum | 40 | 1 |
| Minimum | 3 | 1 |
| StDev | 11.46 | 0 |
| Avg/Max | 0.25 | 1 |