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

GUILLEM GRÀCIA ANDREU, GERARD MADRID MIRÓ

TARDOR 2021-22

PAR3109

1. Task decomposition with Tareador

1.1 Row Strategy

Given a code that computes a Mandelbrot with the option to draw it or set the values in a histogram, we are given the task to parallelize it. First we will try a Row Strategy approach by adding the task to the first for loop.

```
// Calculate points and generate appropriate output
for (int row = 0; row < height; ++row) {
   tareador_start_task("RowStrategy");
   for (int col = 0; col < width; ++col) {
       complex z, c;
       ...
   }
   tareador_end_task("RowStrategy");
}
```

Figure 1: Row Strategy code

1.1.1 mandel-tar

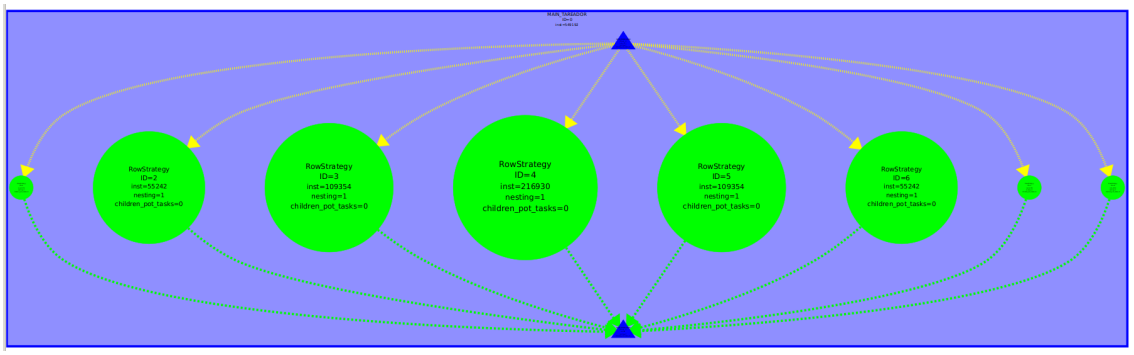


Figure 2: Task Dependence Graph of mandel-tar (Row Strategy)

After executing mandel-tar with the run-tareador script without any options, we can see the task dependency graph shown above (*Figure 2*). The most important characteristics are the fact that most of the program could be parallelized. The majority of the tasks (5 out of 8) have most of the program's weight. Knowing that, we will face no problems when parallelizing since threads will end on time. Also, we can assure that we will not need protection for any variable used within this zone.

1.1.2 mandel-tar -h

Figure 3: Task Dependence Graph of mandel-tar -h *(Row Strategy)*

Adding the "-h" option changes the program slightly in a way that interferes with the dependencies (*Figure 3*). Now we will have dependencies over the value "k" that is accessed when building the histogram. Knowing that, we will also have to protect that variable and get it inside an atomic, critical or even reduction pragma. We can find the access to protect in the code as: "`histogram[k-1]++`".

1.1.3 mandel-tar -d

Figure 4: Task Dependence Graph of mandel-tar -d *(Row Strategy)*

Adding the “-d” option we will find ourselves handling a totally sequential code (*Figure 4*). Every node now has dependencies with the “`Xlib`” library which is used to show the drawing on screen. Knowing that, we will have to protect all the following accesses to that library (`XSetForeground and XDrawPoint`) with a critical pragma section:

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

1.2 Point Strategy

Now that we have seen how the Row Strategy behaves, we have made a change to the code (*Figure 5*) to work with Point Strategy. The change has been to move the line of code to the innermost loop.

```
// Calculate points and generate appropriate output
for (int row = 0; row < height; ++row) {
   for (int col = 0; col < width; ++col) {
       tareador_start_task("PointStrategy");
       complex z, c;
       ...
       tareador_end_task("PointStrategy");
   }
}
```

Figure 5: Point Strategy code

1.2.1 mandel.tar

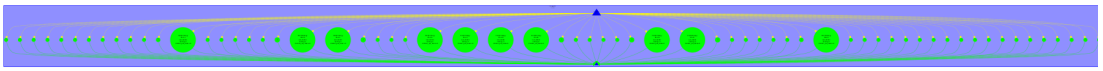


Figure 6: Task Dependency Graph of mandel-tar *(Point Strategy)*

Now that we have the new strategy implemented, we can see (*Figure 6*) a change in the graph above. We can see that it is really parallelizable and it has lots of small tasks and very few big ones (10 out of 64). Knowing this, we will face even less problems when parallelizing since threads will always end on time. Also, we can assure that we will not need protection for any variable used within this zone.

1.2.2 mandel-tar -h

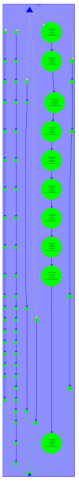


Figure 7: Task Dependency Graph of mandel-tar -h *(Point Strategy)*

As we can see (*Figure 7*), adding the "-h" option changes the graph into one with lots of small weighted tasks that have dependencies on each other. The dependencies are the same on this strategy than on the Row one: they are caused by the "k" value of the histogram. So once again, we should protect it with atomic, critical or reduction.

1.2.3 mandel-tar -d

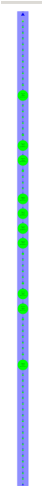


Figure 8: Task Dependency Graph of mandel-tar -d *(Point Strategy)*

As we can barely see on the graph above (*Figure 8*), the code is again totally sequential due to the dependencies with the "Xlib" library. So then again,

knowing that, we will have to protect all of the following accesses to that library (`XSetForeground and XDrawPoint`) with a critical pragma section:

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

2. Point decomposition in OpenMP

2.1 Task Point

```
void mandelbrot(int height, int width, double real_min,
                double imag_min, double scale_real, double scale_imag,
                int maxiter, int **output) {

   // Calculate points and generate appropriate output
   #pragma omp parallel
   #pragma omp single
   for (int row = 0; row < height; ++row) {
       for (int col = 0; col < width; ++col) {
           #pragma omp task firstprivate(row, col)
           {
               complex z, c;
               ...
```

Figure 9: Task Point strategy code

To get the task point strategy working, we have added a new line of code to set a region as parallelizable. The new line of code (seen above in *Figure 9*) is as follows:

```
#pragma omp task firstprivate(row, col)
```

After executing “OMP_NUM_THREADS=X ./mandel-omp -d -h- i” giving X values of 1 and 2, we are presented in both cases with a perfectly generated Mandelbrot without any noticeable changes when going from 1 thread to 2.

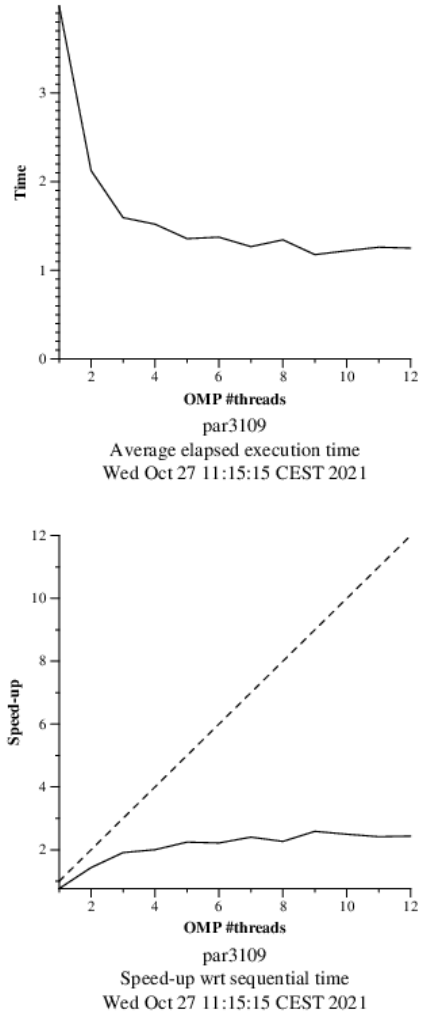


Figure 10: Task Point strategy scalability

As we can see in the plot above (*Figure 10*), the scalability in this case is really bad since you don't get more speed-up the more threads you assign to the program.

When executing the program we get different times in 1 and 8 threads. For the 1 thread execution, the program used a total of 4.11 seconds while the 8 thread execution used only 1.43 seconds. Even Though the 8 thread had a better performance, it is nowhere near what we would expect from using 7 more threads.

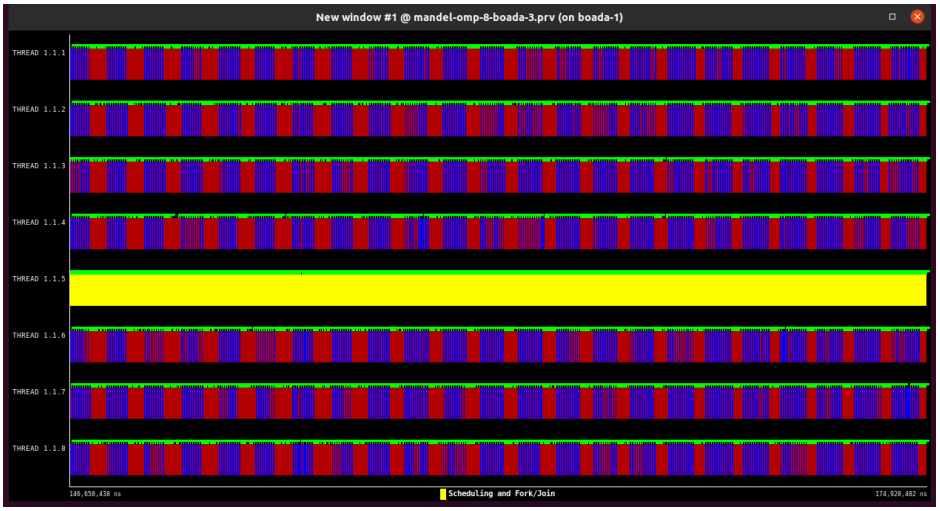


Figure 11: Task Point strategy implicit tasks

As we can see in the previous image (*Figure 11*), all the tasks are created by the same thread. In this case, the thread number 5 spends all of his time creating tasks (as it can be seen by the yellow section).

	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	30.92 %	69.08 %	0.01 %
THREAD 1.1.2	30.08 %	69.91 %	0.00 %
THREAD 1.1.3	30.39 %	69.61 %	0.00 %
THREAD 1.1.4	30.08 %	69.92 %	0.00 %
THREAD 1.1.5	30.34 %	69.66 %	0.00 %
THREAD 1.1.6	30.00 %	70.00 %	0.00 %
THREAD 1.1.7	47.49 %	0.00 %	52.51 %
THREAD 1.1.8	29.44 %	70.56 %	0.00 %
Total	258.73 %	488.74 %	52.52 %
Average	32.34 %	61.09 %	6.57 %
Maximum	47.49 %	70.56 %	52.51 %
Minimum	29.44 %	0.00 %	0.00 %
StDev	5.74 %	23.09 %	17.36 %
Avg/Max	0.68	0.87	0.13

Figure 12: Task Point strategy tasks executed

In the previous image (*Figure 12*), we can get a glimpse on the distribution of work for each thread in the three different states they can be in: Running, Synchronization and Scheduling. These states are visually represented in the same colours in Figure *11*.

Looking at the numbers, we can also see that most of the time is spent in Synchronization, which increases the execution time significantly.

Figure 14: Task Point strategy explicit tasks (middle section)

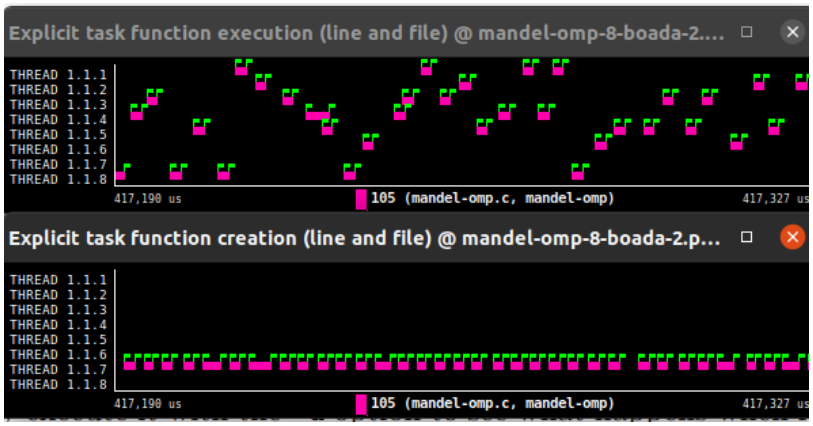


Figure 15: Task Point strategy explicit tasks (final section)

When taking a look at explicit tasks, we can see two very different sections. The middle section (seen in *Figure 14*) has longer tasks and a good load of parallelization. On the other hand, the final section of the program (seen in *Figure 15*) has no parallel tasks whatsoever and they are all really short in execution time. Meanwhile, thread number 7 creates all the tasks one after another.

2.2 Taskloop Point

```
void mandelbrot(int height, int width, double real_min,
                double imag_min, double scale_real,
                double scale_imag, int maxiter, int **output) {
    // Calculate points and generate appropriate output
    #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) {
           complex z, c;
           ...
```

Figure 16: Taskloop Point strategy code

To get this new strategy working, we have changed a few things inside the code (*Figure 16*). First, we have removed the existing task creation from the previous code (see *Figure 9*) and added a new line between the two for loops as follows:

```
#pragma omp taskloop firstprivate(row)
```

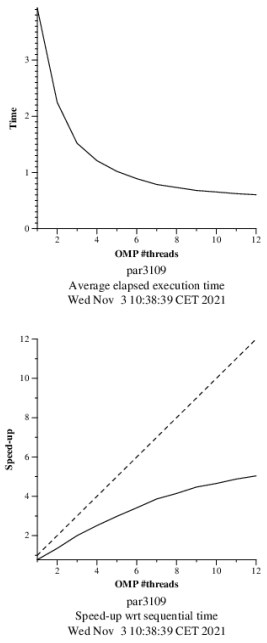
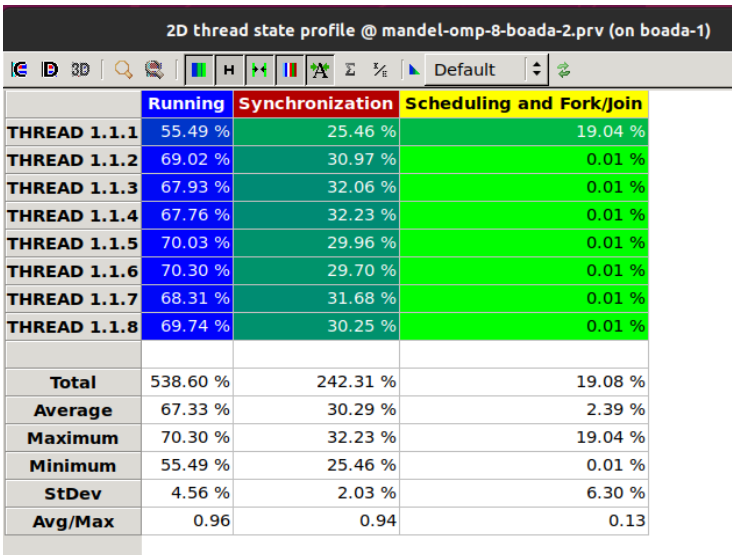


Figure 17: Taskloop Point strategy explicit tasks (final section)

As we can see in the plot above (*Figure 17*), the scalability in this case is a bit better than before (*Figure 10*), since it goes slightly up for each new processor used.

When executing the program we get different times in 1 and 8 threads. For the 1 thread execution, the program used a total of 3.92 seconds while the 8 thread execution used only 0.73 seconds. We can now see that the scalability is improving as we make more changes in the code, making us guess that the next strategy will have an even better performance.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	55.49 %	25.46 %	19.04 %
THREAD 1.1.2	69.02 %	30.97 %	0.01 %
THREAD 1.1.3	67.93 %	32.06 %	0.01 %
THREAD 1.1.4	67.76 %	32.23 %	0.01 %
THREAD 1.1.5	70.03 %	29.96 %	0.01 %
THREAD 1.1.6	70.30 %	29.70 %	0.01 %
THREAD 1.1.7	68.31 %	31.68 %	0.01 %
THREAD 1.1.8	69.74 %	30.25 %	0.01 %
Total	538.60 %	242.31 %	19.08 %
Average	67.33 %	30.29 %	2.39 %
Maximum	70.30 %	32.23 %	19.04 %
Minimum	55.49 %	25.46 %	0.01 %
StDev	4.56 %	2.03 %	6.30 %
Avg/Max	0.96	0.94	0.13

Figure 18: Taskloop Point strategy tasks executed

In this case, we can see (in *Figure 18*) how the time spent in synchronization has reduced to more than half, improving the performance of the program by consequently raising the time threads spend running rather than synchronizing to each other.

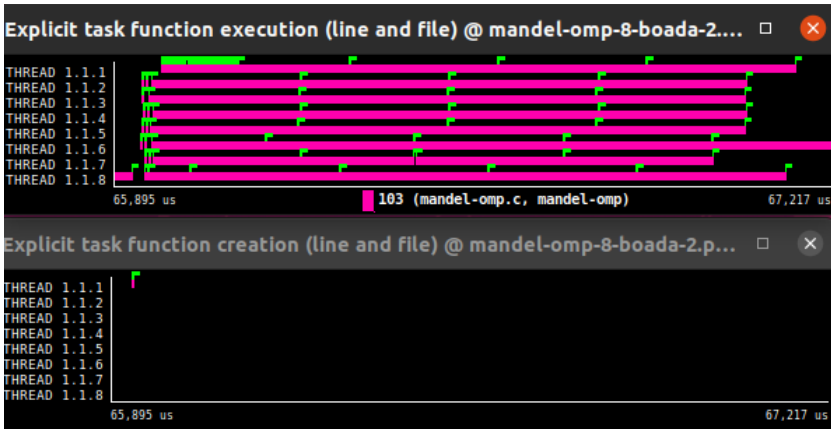


Figure 19: Taskloop Point strategy explicit tasks (middle section)

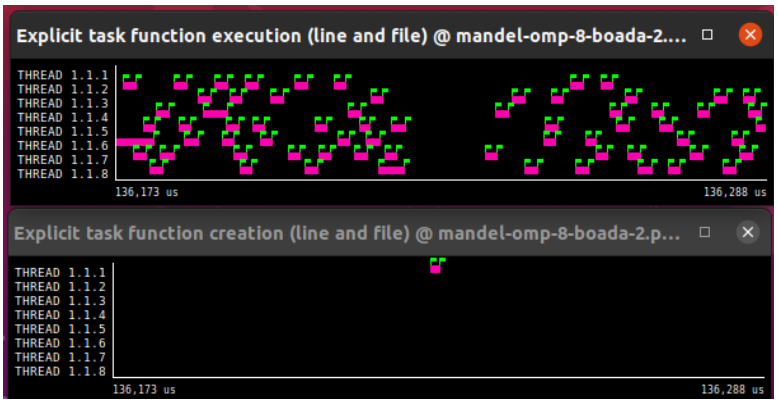


Figure 20: Taskloop Point strategy explicit tasks (final section)

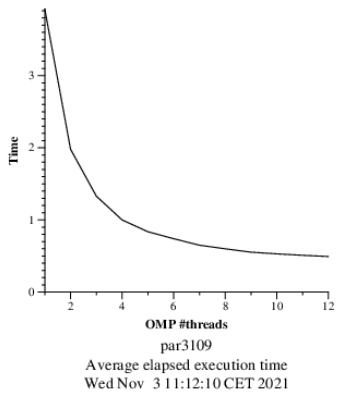
In this case, in comparison with the one before, we can see how in the middle section (*Figure 19*), all threads are working after a really small time creating tasks. We can also notice that the final section (*Figure 20*) has more parallelized tasks than before and only a really small amount of the time is spent creating tasks, which differs a lot from the one in *Figure 15*.

2.3 Taskloop nogroup Point

```
void mandelbrot(int height, int width, double real_min,
                double imag_min, double scale_real,
                double scale_imag, int maxiter, int **output) {
    // Calculate points and generate appropriate output
    #pragma omp parallel
    #pragma omp single
    for (int row = 0; row < height; ++row) {
        #pragma omp taskloop firstprivate(row) nogroup
        for (int col = 0; col < width; ++col) {
            #pragma omp task firstprivate(row, col)
            {
                complex z, c;
                ...
```

Figure 21: Taskloop Point strategy explicit tasks (final section)

This new strategy needed now adds a nogroup clause to the taskloop in the row for loop as seen in the code above (*Figure 21*)



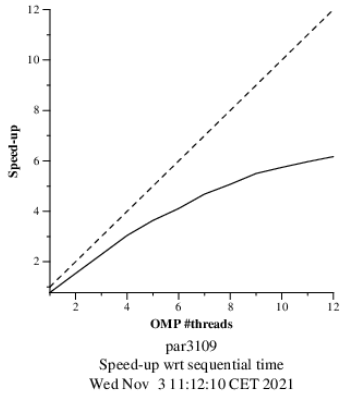
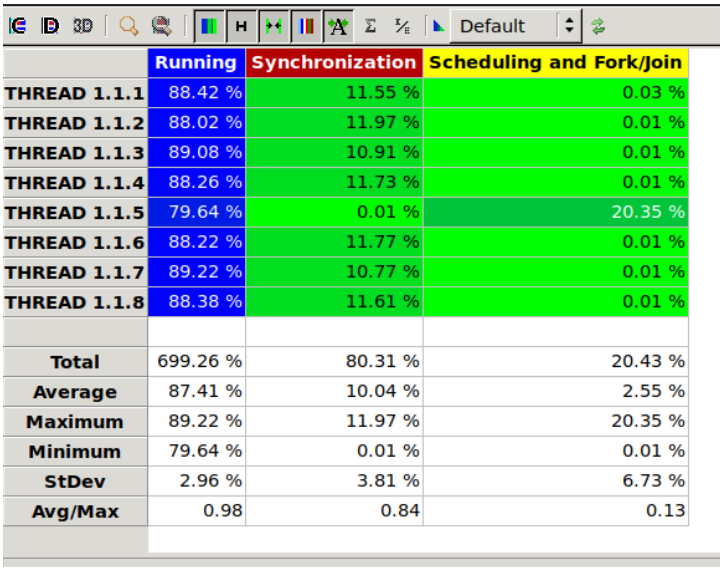


Figure 22: Taskloop nogroup Point strategy scalability

As we can see in the plot above (*Figure 22*), the scalability in this case is a bit better than before (*Figure 17*) and way better than the first time (*Figure 10*)

When executing the program we get different times in 1 and 8 threads. For the 1 thread execution, the program used a total of 3.92 seconds while the 8 thread execution used only 0.60 seconds. The value for the 8 thread execution is the best one we have seen so far, so we were right with the prediction earlier.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	88.42 %	11.55 %	0.03 %
THREAD 1.1.2	88.02 %	11.97 %	0.01 %
THREAD 1.1.3	89.08 %	10.91 %	0.01 %
THREAD 1.1.4	88.26 %	11.73 %	0.01 %
THREAD 1.1.5	79.64 %	0.01 %	20.35 %
THREAD 1.1.6	88.22 %	11.77 %	0.01 %
THREAD 1.1.7	89.22 %	10.77 %	0.01 %
THREAD 1.1.8	88.38 %	11.61 %	0.01 %
Total	699.26 %	80.31 %	20.43 %
Average	87.41 %	10.04 %	2.55 %
Maximum	89.22 %	11.97 %	20.35 %
Minimum	79.64 %	0.01 %	0.01 %
StDev	2.96 %	3.81 %	6.73 %
Avg/Max	0.98	0.84	0.13

Figure 24: Taskloop nogroup Point strategy tasks executed

We can clearly see (*Figure 24*) the reason why the performance has increased and the execution time has gone down to minimum values: Synchronization times are lower than ever and don't even reach 12% of the whole program's execution time.

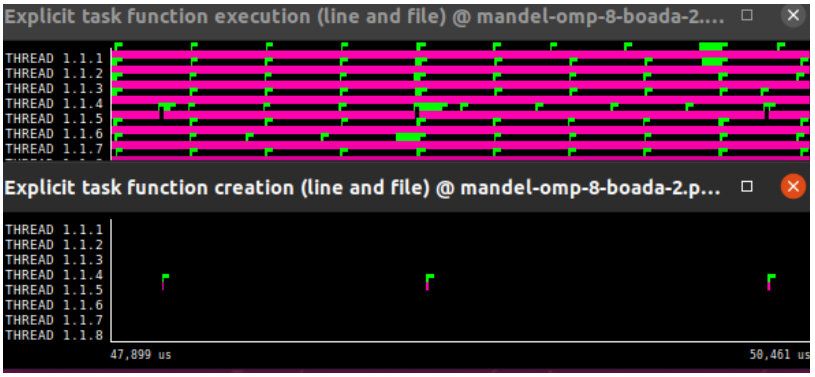


Figure 25: Taskloop nogroup Point strategy explicit tasks (middle section)

Figure 26: Taskloop nogroup Point strategy explicit tasks (final section)

In these two figures, we can see a great improvement in synchronization. The middle section (*Figure 25*) shows how little time is spent creating tasks and how all of the threads are working on tasks almost all of the time. We can also see that in the final section (*Figure 26*), small tasks have now reached a new level of parallelization, with almost all threads always working in parallel to each other.

3. Row decomposition in OpenMP

Now, for this section of the laboratory assignment, we had to apply a taskloop to a Row Strategy.

```
void mandelbrot(int height, int width, double real_min,
                double imag_min, double scale_real,
                double scale_imag, int maxiter, int **output) {
    // Calculate points and generate appropriate output
    #pragma omp parallel
    #pragma omp single
    #pragma omp taskloop
    for (int row = 0; row < height; ++row) {
        for (int col = 0; col < width; ++col) {
            complex z, c;
            ...
```

Figure 27: Taskloop Row strategy code

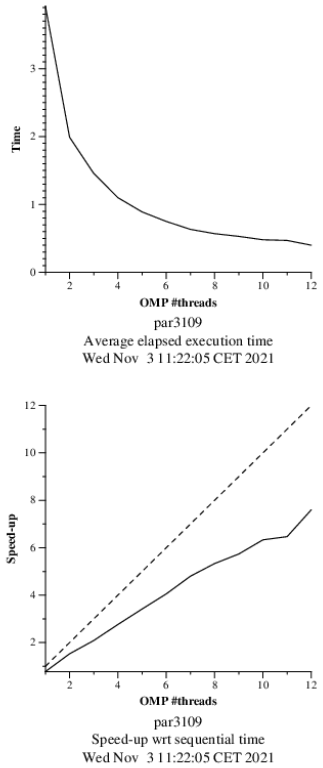
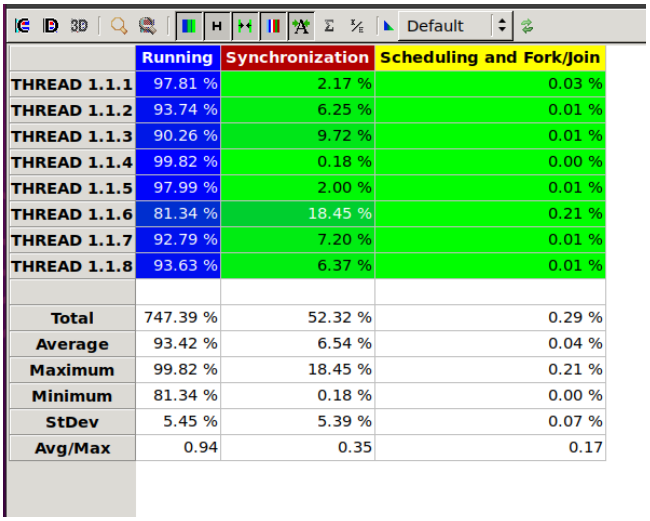


Figure 28: Taskloop Row strategy scalability

As we can see on *Figure 28*, the speed-up function is the closest that it has been to the 45° line. This means that the more processors we add, the better the

performance will be. Looking back at the previous plots, we can observe an increase given that the first speed-up computed was almost a constant function at y = 2.

When executing the program we get different times in 1 and 8 threads. For the 1 thread execution, the program used a total of 3.92 seconds while the 8 thread execution used only 0.57 seconds. The value of the 8 thread execution is better that the last one got, but the change is so slight that we can guess that there is not much more room for improvement, as the code must take some time to execute the computations themselves.



	Running	Synchronization	Scheduling and Fork/Join
THREAD 1.1.1	97.81 %	2.17 %	0.03 %
THREAD 1.1.2	93.74 %	6.25 %	0.01 %
THREAD 1.1.3	90.26 %	9.72 %	0.01 %
THREAD 1.1.4	99.82 %	0.18 %	0.00 %
THREAD 1.1.5	97.99 %	2.00 %	0.01 %
THREAD 1.1.6	81.34 %	18.45 %	0.21 %
THREAD 1.1.7	92.79 %	7.20 %	0.01 %
THREAD 1.1.8	93.63 %	6.37 %	0.01 %
Total	747.39 %	52.32 %	0.29 %
Average	93.42 %	6.54 %	0.04 %
Maximum	99.82 %	18.45 %	0.21 %
Minimum	81.34 %	0.18 %	0.00 %
StDev	5.45 %	5.39 %	0.07 %
Avg/Max	0.94	0.35	0.17

Figure 29: Taskloop Row strategy tasks executed

These speed-up improvements come together with the lowest value of synchronization, with threads not even getting a 1% of their execution time and almost 100% of runtime.

Figure 30: Taskloop Row strategy explicit tasks (initial section)

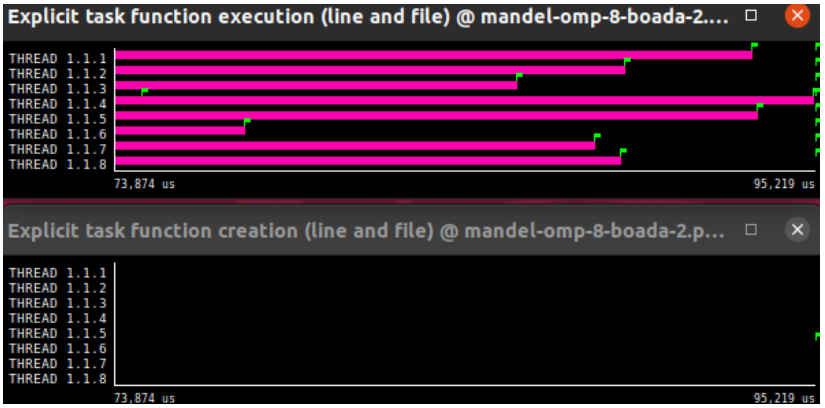


Figure 31: Taskloop Row explicit tasks (final section)

Unlike the last sections, we can see that this time we have divided the explanation into the initial section (*Figure 30*) and the final section (*Figure 31*). The reason behind this decision is mainly the lack of information provided by the middle section that we used to analyze.

We can see how in the initial section, thread 6 creates all tasks and all the other threads get to work instantly, followed shortly by thread 6. We can also see that in the final section, there is a slight disequilibrium in the threads when finishing their tasks as some tasks are lighter than others or may have started earlier.

4. Optional

Lastly, for this section of the deliverable, we had to apply user input to the code to be able to manage the number of tasks (user_param). To do this, we had to first take a look at the script that managed the input, and then review and update our code to fulfill the demand.

```
void mandelbrot(int height, int width, double real_min,
                double imag_min, double scale_real,
                double scale_imag, int maxiter, int **output) {
   // Calculate points and generate appropriate output
   #pragma omp parallel
   #pragma omp single
   #pragma omp taskloop num_tasks(user_param)
   for (int row = 0; row < height; ++row) {
       for (int col = 0; col < width; ++col) {
           complex z, c;
```

Figure 32: Adaptation of the code for the script inputs

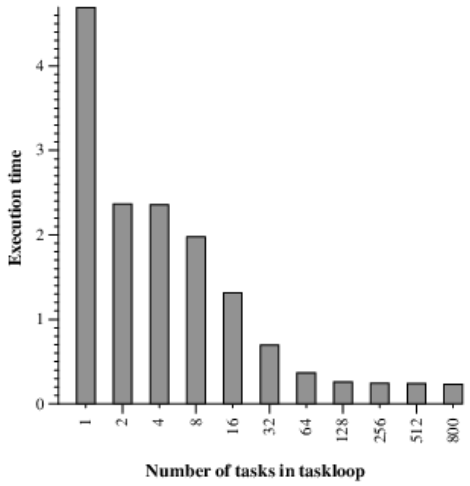


Figure 33: Histogram of execution time per number of tasks

In this histogram (*Figure 33*), we can see how the number of tasks in the taskloop directly influences the execution time of the program. However, giving the program a million tasks would not make it better since we can see that the execution time gets it's better value between 64 and 128 tasks.

5. Conclusion

This laboratory assignment has been a challenging piece of work that has mainly made us realize how hard it can be to properly parallelize an application without causing it to go slower due to synchronization times. We also learned how to interpret different Hints for Paraver that we didn't know existed and how useful it can be to zoom into a thread’s line of work to see how the tasks are created and executed. To sum it all up, we have taken in these new learned concepts and are ready to apply them to future projects both inside and outside the Parallelism course.