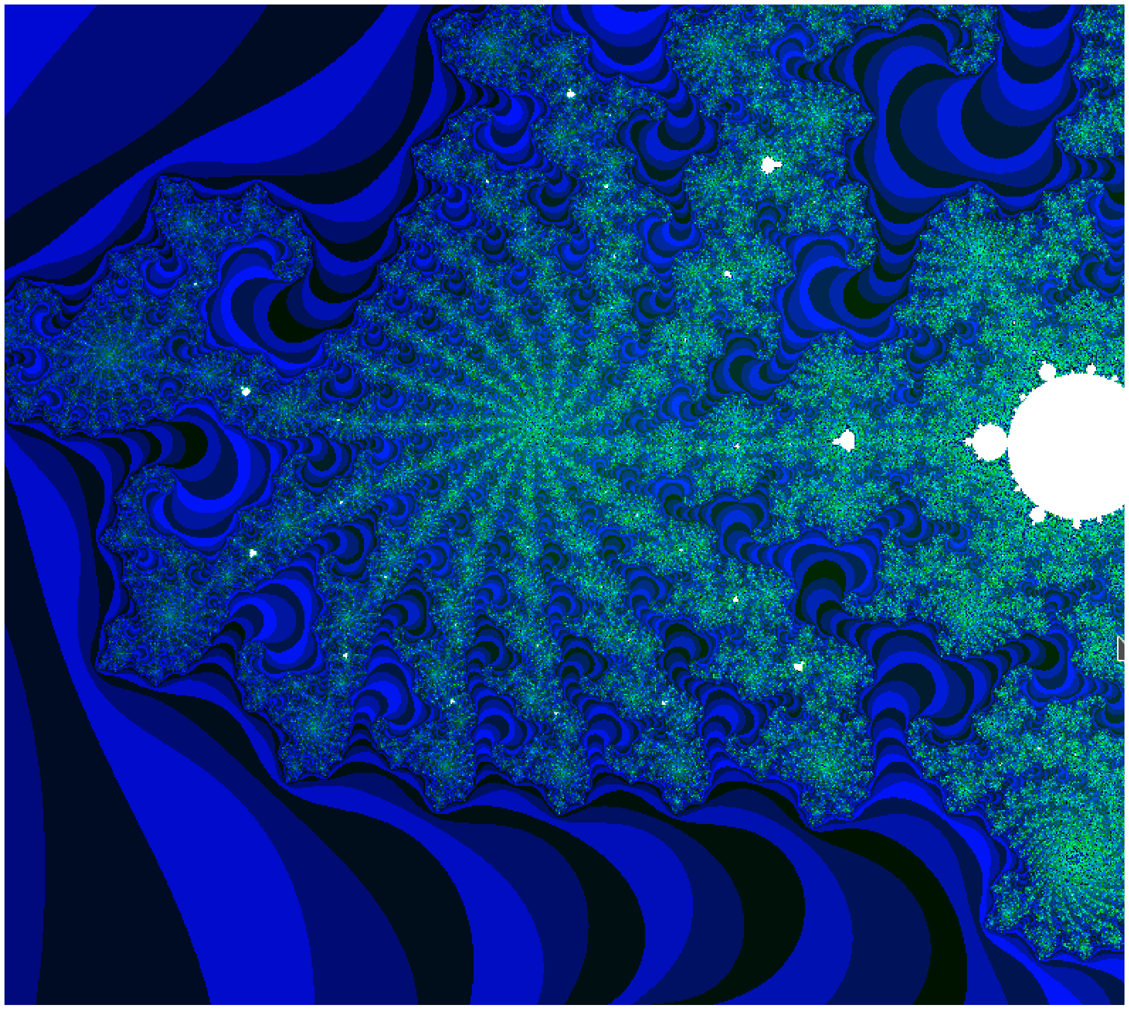
LAB 3

Embarrassingly parallelism with OpenMP: Mandelbrot set



Antonio J. Cabrera

Paul Gazel-Anthoine

PAR1401

Spring 2018-19

Table of contents

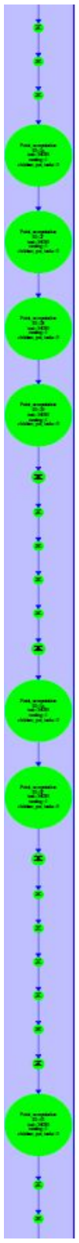
INTRODUCTION: The Mandelbrot set

In this report we explain the different approaches for parallelisation of the Mandelbrot set code we have been provided with.

For each of the experiments, we explain our predictions and the actual results, with references to multiple plots and Paraver traces.

In the end, we include our conclusions regarding the different decompositions of this problem and comment on the best solutions.

TASK DECOMPOSITION ANALYSIS



Point:

```
    for (row = 0; i < height; ++row){
        for (col = 0; col < width; ++col){
            tareador_start_task(“Point computation”);
            (...)
            tareador_end_task(“Point computation”);
        }
    }
```

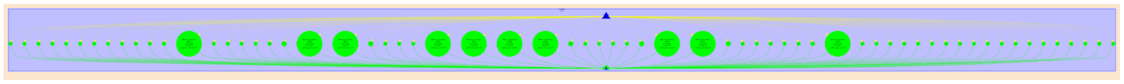


Figure 1.1: *Tareador* dependence graph for *mandel* (Point decomposition)

As we see in figure 1.1, when we execute without the visual flag, the Tareador execution for point decomposition gives us a graph with no dependencies. It projects a great situation for parallelism.

However, in figure 1.2 we see the complete opposite: a vertical strip full of dependencies which results in a very inefficient sequential execution. This is very bad news for parallelism.

Figure 1.2: *Very small crop* from *Tareador* dependence graph for visual *mandeld* (Point decomposition)

Row:

```
     for (row = 0; i < height; ++row){
         tareador_start_task(“Row computation”);
         for (col = 0; col < width; ++col){
             (...)
         }
         tareador_end_task(“Row computation”);
     }
```

Similarly, for row decomposition we maintain the same structure of the dependence graphs (visual or not).

The main difference is that the grain is coarse instead of fine and the amount of tasks is way smaller, even though each task performs more workload.

The serialization that we see in both decompositions is due to a part of the code that accesses the same variable. So, in order to do parallelism with this, we will need to address this part of the code, since we do not want to produce data races that will affect the output image.

We expect that row decomposition will work better if we can address the load balance issue.

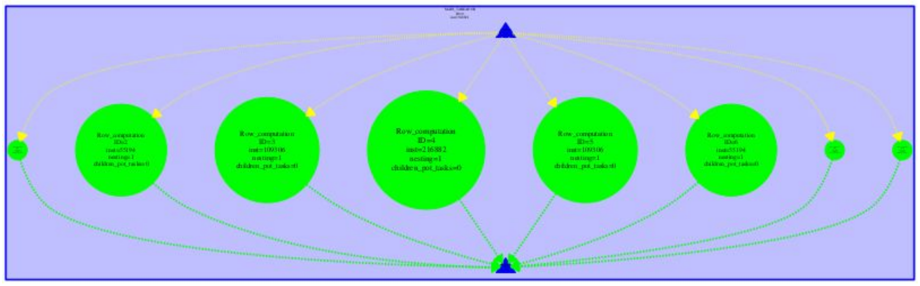


Figure 2.1: ***Tareador*** **dependence graph for** ***mandel*** **(Row decomp.)**

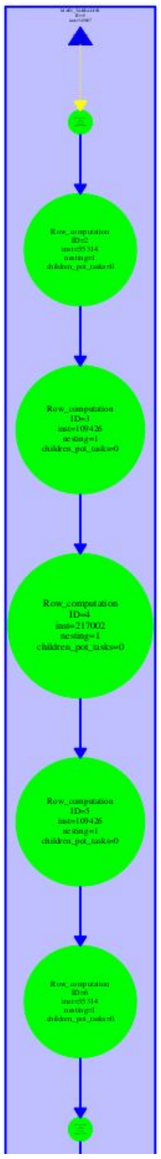


Figure 2.2: ***Tareador*** **dependence graph for visual** ***mandeld*** **(Row decomposition)**

POINT STRATEGY IMPLEMENTATION USING OMP TASK

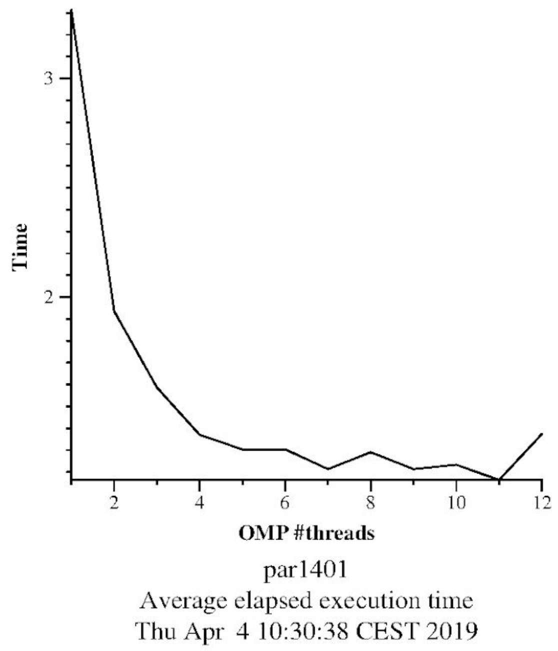
Our generated image for the execution with only 1 thread is correct.

We have seen that the sequential version is a little bit faster than this execution, because even though tasks can only be assigned to one thread, there is certainly a loss of time during these assignments. The difference is not huge, but we can see that task creation and deletion has a noticeable overhead.

When executing the same program (mandeld-omp) with 8 threads, the execution time drops to 1.24 seconds (around 2.7 in speedup compared to the one thread execution). We also get a correct image.

Execution times:

- 1Th mandel → 3.39 s
- 8Th mandel → 1.24 s
- Serial mandel → 3.05 s



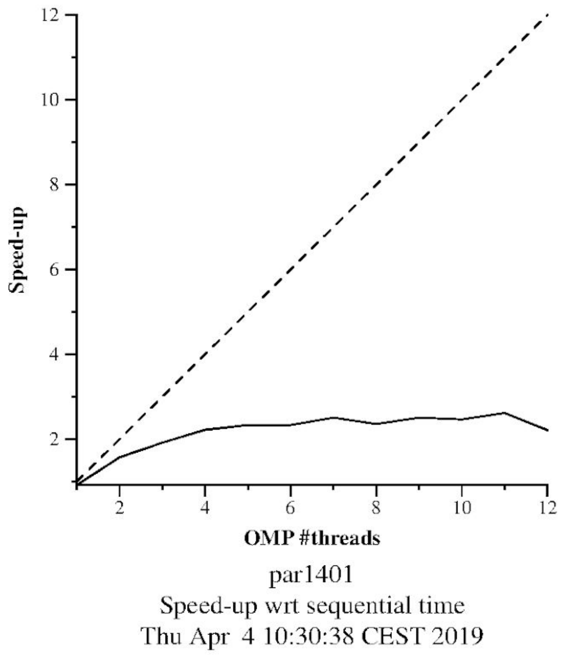


Figure 3: ***Mandel-omp strong scalability graphs (V1)***

By looking at figure 3, we can see that most of the potential speedup is accomplished with 4 threads. After that the execution time will not get any lower. This can happen because at this point the overheads of dealing with the extra threads are comparable to the computation task they need to perform.
In conclusion, we can say that the strong scalability is not appropriate, because it gives us little room for improvement by increasing the number of threads.

SIMPLEST TASKING CODE (V1)

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	95.231	200.000
THREAD 1.1.2	72.555	36.000
THREAD 1.1.3	82.313	135.200
THREAD 1.1.4	76.864	211.200
THREAD 1.1.5	78.703	25.600
THREAD 1.1.6	79.508	2.400
THREAD 1.1.7	77.771	8.000
THREAD 1.1.8	77.055	21.600

Table 1: ***Task profile for mandel-omp with 8 threads*** **(Point granularity V1)**

By looking at table 1, we see that tasks are not always created by the same thread. We are aware that this job is performed by the first thread available that reaches the beginning of the parallel region and reads the *single* directive.

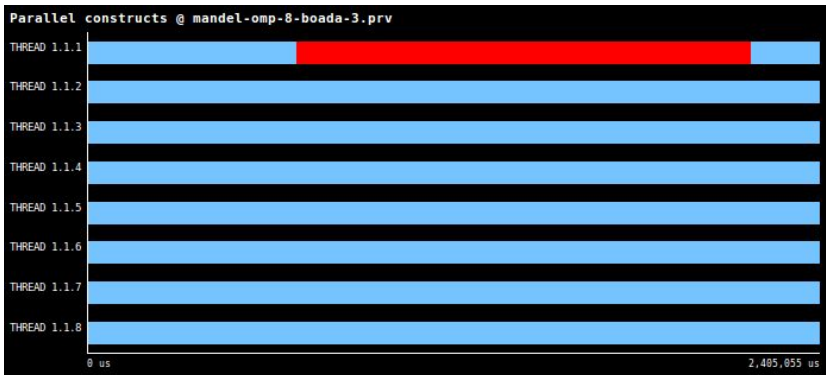


Figure 3: ***Paraver parallel constructs (Point granularity V1)***

As seen in figure 3, we can see that the main thread is the only one executing the parallel construct. If we look at the histogram, we see that there are 800 bursts, meaning that the main thread is the one in charge of each of the 800 row's parallel constructs.
The 'single' worksharing construct is invoked the same amount of times.

TASKWAIT (V2)

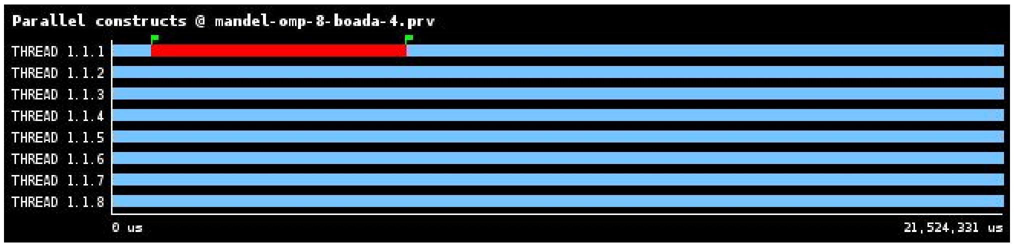


Figure 4: ***Paraver parallel constructs (Point granularity V2)***

In this case, we see that there is only one burst, which corresponds to the main thread. Thus, we verify that this main thread creates the only parallel region.
Furthermore, there are 640.000 tasks and 800 *taskwait* executions (figure 5), one for each row. The tasks maintain the granularity compared to V1.

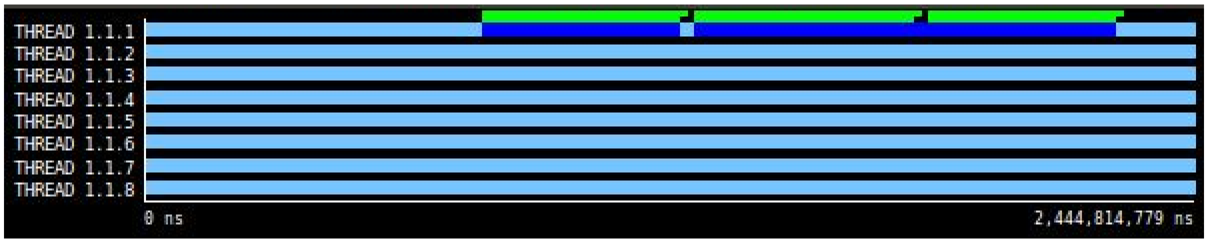


Figure 5: ***Paraver taskwait constructs (Point granularity V2)***

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	103.461	640.000
THREAD 1.1.2	72.729	0
THREAD 1.1.3	69.428	0
THREAD 1.1.4	82.661	0
THREAD 1.1.5	76.321	0
THREAD 1.1.6	82.728	0
THREAD 1.1.7	70.526	0
THREAD 1.1.8	82.146	0

Table 2: ***Task profile for mandel-omp with 8 threads (Point granularity V2)***

TASKGROUP (V3)

There is no difference in the amount of tasks created, the distribution of executed tasks among threads or timing. It is obvious why there is no difference: in version 2, we are creating a *taskgroup* manually (using *taskwait*), whereas in version 3 we use the directive *taskgroup* (which implicitly uses *taskwait*).
So, we have the same amount of *taskgroups* than *taskwaits* in the previous version (800).

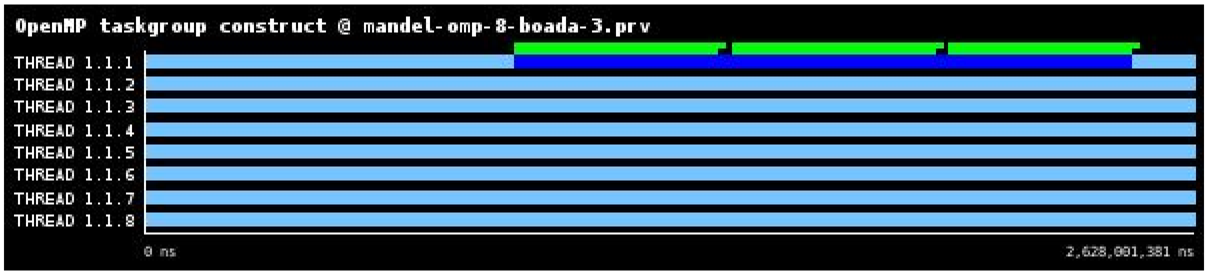


Figure 6: ***Paraver taskgroup constructs (Point granularity V3)***

	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	89.031	640.000
THREAD 1.1.2	81.079	0
THREAD 1.1.3	76.606	0
THREAD 1.1.4	75.184	0
THREAD 1.1.5	81.928	0
THREAD 1.1.6	82.877	0
THREAD 1.1.7	82.110	0
THREAD 1.1.8	71.185	0

Table 3: ***Task profile for mandel-omp with 8 threads (Point granularity V3)***

We see that it is not necessary to use taskwait (or taskgroup, for that matter), because there is no dependence other than the one inside the critical region. In fact, we are slowing down the program with this barriers.

(OPTIONAL: *taskgroup* vs *taskwait*)

The main difference between the two is that with *taskwait* the current task waits only for its child tasks, whereas with *taskgroup* the current task waits until all child tasks and their descendant tasks complete their execution. *Taskgroup* waits for tasks in a deeper level than *taskwait*.
In this case, we have no descendants of the child tasks, so there should be no difference in that sense.

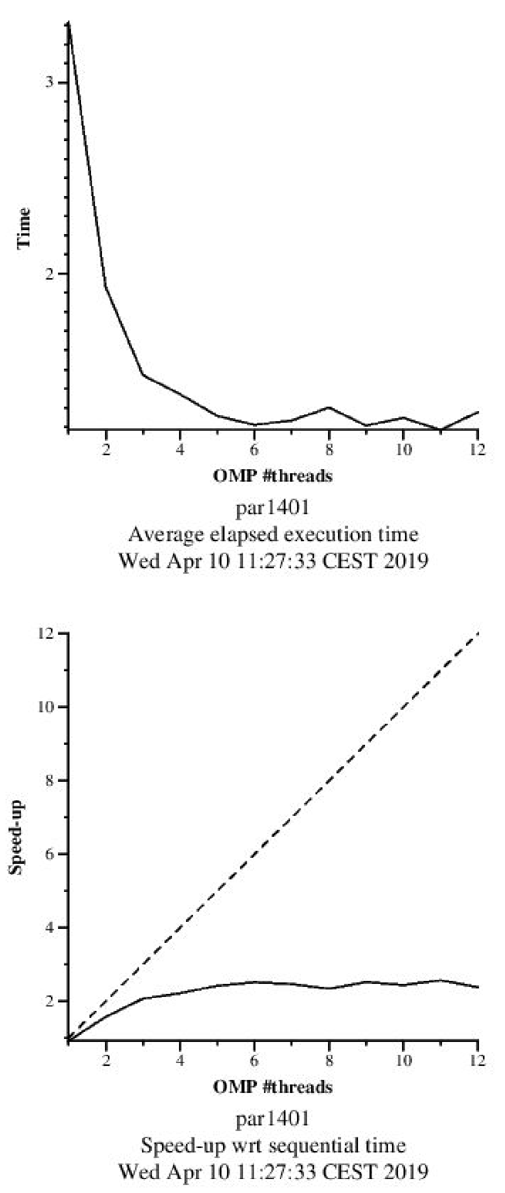


Figure 7: ***Mandel-omp strong scalability graphs for TaskWait***

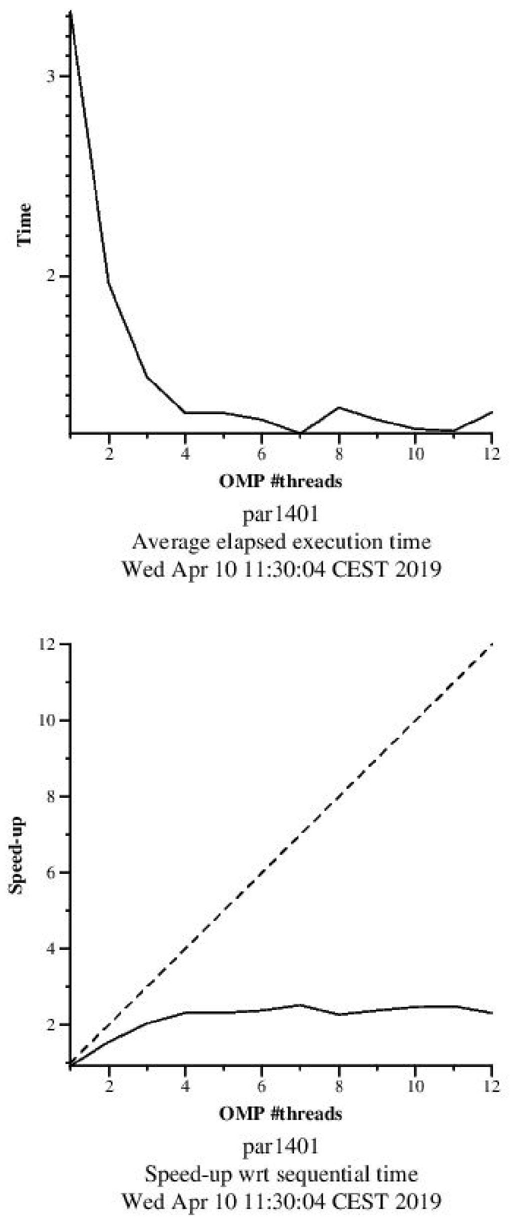


Figure 8: ***Mandel-omp strong scalability graphs for TaskGroup***

As we can see comparing figures 7 and 8, the strong scalability plots do not show noticeable differences. Both implementations have a bad strong scalability.

WITHOUT SYNCHRONISATION (V4)

The amount of tasks, as we see in table 4, is the same as the previous versions. The main difference is a smaller execution time, which comes from a decrease in unnecessary synchronisation time.

As per the behaviour of the main thread, we can guess by looking at figures 9 and 10. We know this thread is the one in charge of creating tasks. So, at first, it creates all the tasks that fit in the pool. Then, similarly to regular threads, it executes tasks until the pool can accept a certain amount of new tasks. At that point, the main thread will start creating tasks again.

	Executed OpenMP task function	**Instantiated OpenMP task function**
THREAD 1.1.1	110.298	640.000
THREAD 1.1.2	75.224	0
THREAD 1.1.3	71.432	0
THREAD 1.1.4	73.787	0
THREAD 1.1.5	80.433	0
THREAD 1.1.6	79.330	0
THREAD 1.1.7	69.643	0
THREAD 1.1.8	79.853	0

Table 4: ***Task profile for mandel-omp with 8 threads (Point granularity V4)***

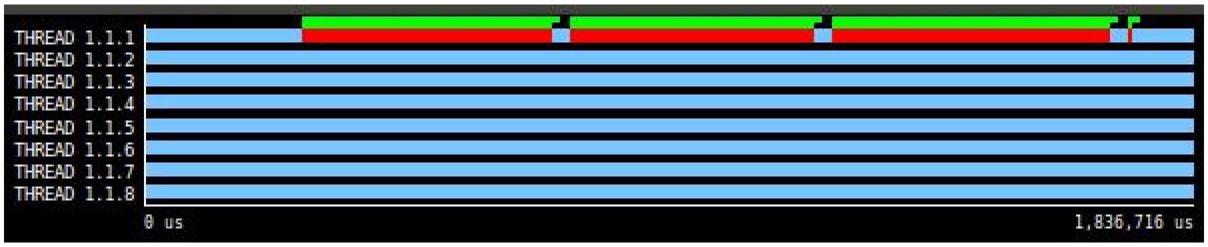


Figure 9: ***Paraver task instantiation (Point granularity V4)***

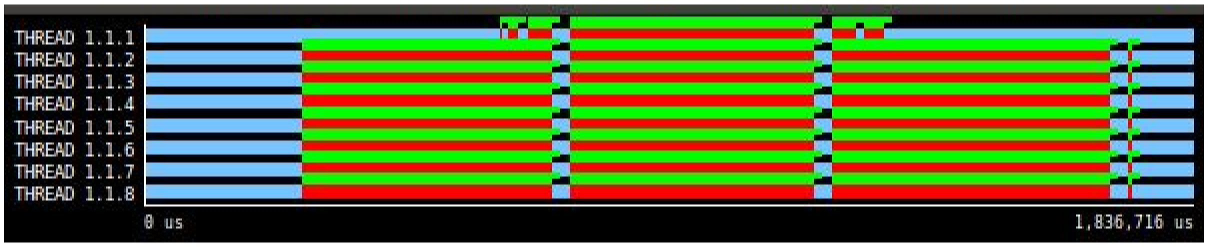


Figure 10: ***Paraver task execution (Point granularity V4)***

By comparing figures 3 and 11, we can see that strong scalability is very similar. The improvement is hardly noticeable. Maybe the speedup we can accomplish is a little higher, but the graph flattens very quickly anyway.

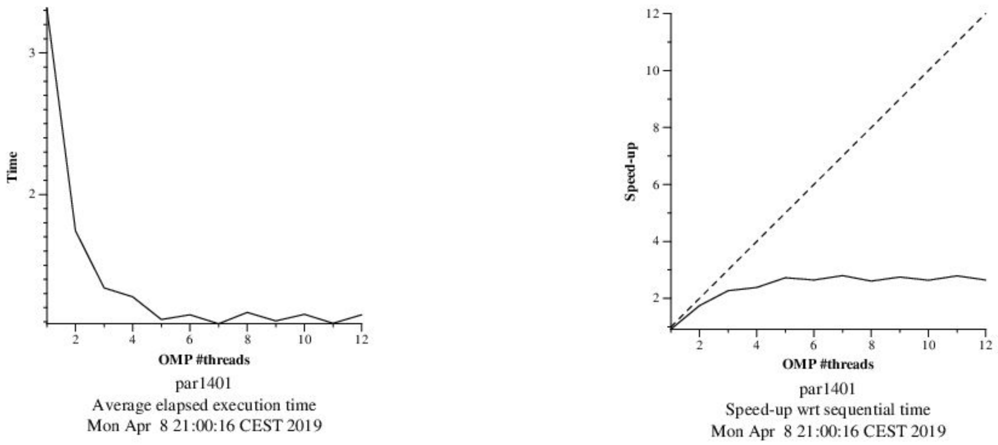


Figure 11: ***Mandel-omp strong scalability graphs (V4)***

TASKLOOP (V5)

By using the implementation with *taskloop*, we obtain an unexpected result. For some reason, as can be seen in table 5, the total amount of instantiated/created tasks, according to the trace, is much smaller than the number of executed tasks. We commented with our professor and we agreed that it could be a problem with the event capture of the *extrae* library.

In terms of speed, this program executed faster with *num_task(800)* than any previous execution.
Since the instrumented execution gives us a strange result, we are not comfortable extracting conclusions, but we think the reason why *taskloop* is faster than the previous methods is that the way it creates the tasks inside an implicit *taskgroup* is much faster than manually creating the tasks inside a *taskgroup*.

	Executed OpenMP task function	**Instantiated OpenMP task function**
THREAD 1.1.1	81.321	4.429
THREAD 1.1.2	82.092	1.147
THREAD 1.1.3	80.799	1.240
THREAD 1.1.4	81.671	1.257
THREAD 1.1.5	83.749	1.156
THREAD 1.1.6	78.210	1.160
THREAD 1.1.7	82.703	1.247
THREAD 1.1.8	81.455	1.164
TOTAL	**652.000**	**12.800**

Table 5: ***Task profile for mandel-omp with 8 threads (Point granularity V5)***

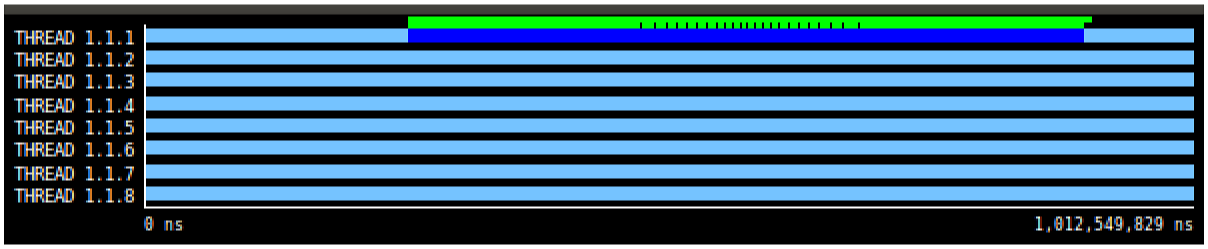
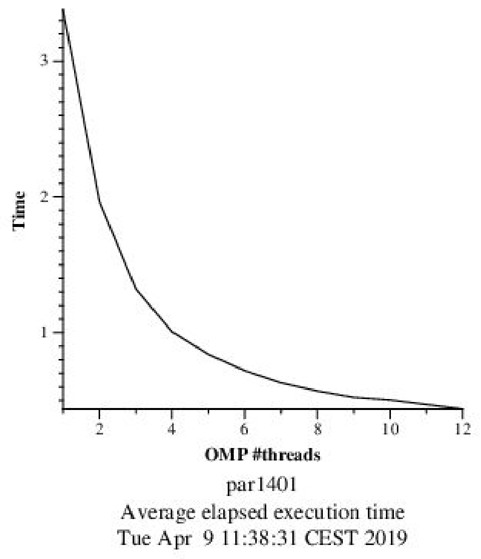


Figure 12: *Paraver taskloop construct (Point granularity V5)*



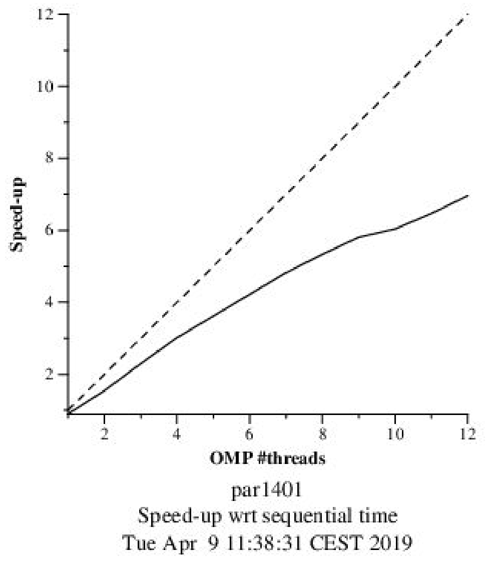
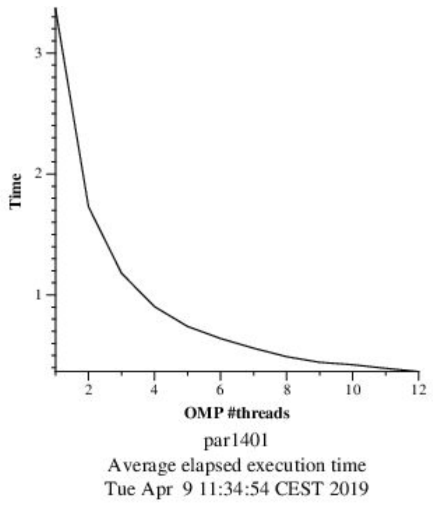


Figure 13: *Mandel-omp strong scalability graphs (V5)*

As we can see in figure 13, with the *taskloop* directive the strong scalability is massively improved. We can see that after 8 threads, the speed-up grows a little slower, but still has a decent slope compared to what we could accomplish in previous implementations.

TASKLOOP NOGROUP (V6)

After adding the *nogroup* clause, we have seen a slight improvement in execution time while keeping the correctness of the program (confirmed by obtaining the same output image with *mandeld*). As we can see in figure 14, the strong scalability accomplished is slightly superior to version 5 (figure 13).
Of course, we predicted this would happen, because there is no dependence between rows and it is unnecessary to synchronise after the computation of each row. This also happened in version 4 of the code.



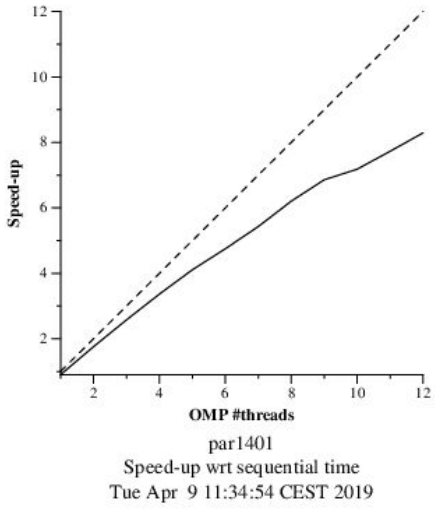


Figure 14: ***Mandel-omp strong scalability graphs (V6)***

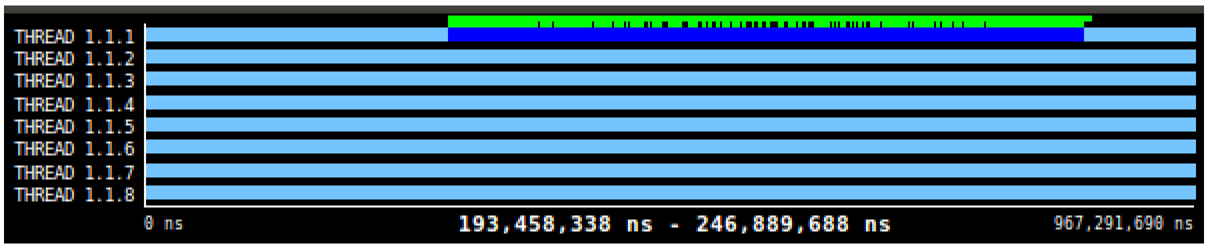


Figure 15: ***Paraver taskloop construct (Point granularity V6)***

TASKLOOP NOGROUP PERFORMANCE

Using version 6 of the code, we have studied the behavior with different task granularities (see results in table 7). There is not a big difference between the times obtained, although after analysing it we can see a small increase in time with finer grain tasks because of the overhead that comes with generating many more tasks.

This tasks are very small (computationally speaking), so there is no need to do finer grain tasks. Quite the opposite, we can already imagine that the row decomposition (which is close to this test with 1 task per iteration) will work better than point decomposition.

# tasks / iteration	Time (s)
1	0,4555
2	0,4425
5	0,4474
10	0,4587
25	0,4616
50	0,4804
100	0,4580
200	0,4692
400	0,4714
800	0,4898

Table 7: Performance vs number of tasks per iteration *(Point granularity V6)*

ROW DECOMPOSITION IN OMP

For row decomposition, we adapted the best approach obtained for point decomposition (V6), which used *taskloop* and *nogroup*, but with *num_tasks*(height/#threads). This way we generate a lot less tasks, but with coarse grain. Expectedly, this produces an increased performance compared to version 6, because each thread will compute complete rows, without any synchronisation.
By looking at figure 16 we see that almost half of the chart corresponds to serial execution. For versions 1 and 2, the serial part of the code looked much smaller, which means we have reduced a lot the parallel part of our program.

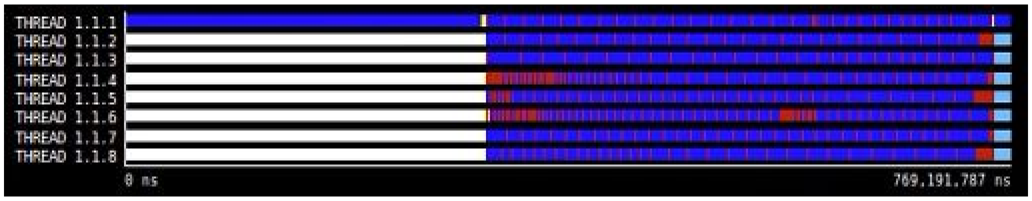
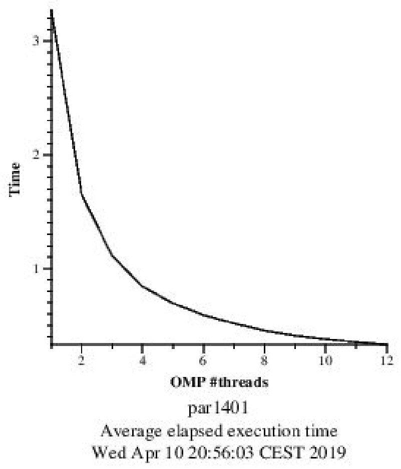


Figure 16: ***Paraver of mandel-omp (Row decomposition)***

	Executed OpenMP task function	**Instantiated OpenMP task function**
THREAD 1.1.1	66	5
THREAD 1.1.2	27	3
THREAD 1.1.3	53	0
THREAD 1.1.4	198	1
THREAD 1.1.5	43	2
THREAD 1.1.6	339	4
THREAD 1.1.7	40	1
THREAD 1.1.8	49	0
TOTAL	**815**	**16**

Table 8: ***Task profile for mandel-omp with 8 threads (Row decomposition)***

For row decomposition we finally obtain a very nice strong scalability. It is not optimal, but it is by far the best parallelism that we have accomplished up to this point.



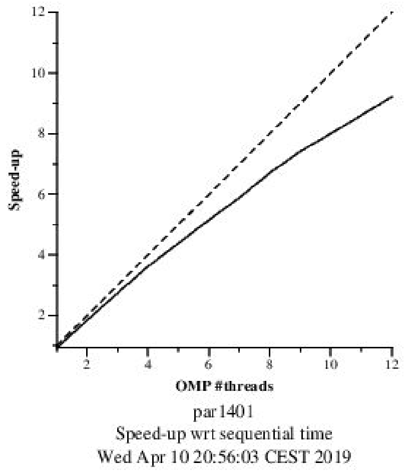


Figure 17: ***Mandel-omp strong scalability graphs (Row decomposition)***

(OPTIONAL: *For*-based parallelization)

```
#pragma omp parallel for schedule(static)
for (row = 0; i < height; ++row){
    for (col = 0; col < width; ++col){
        (...)
    }
}
```

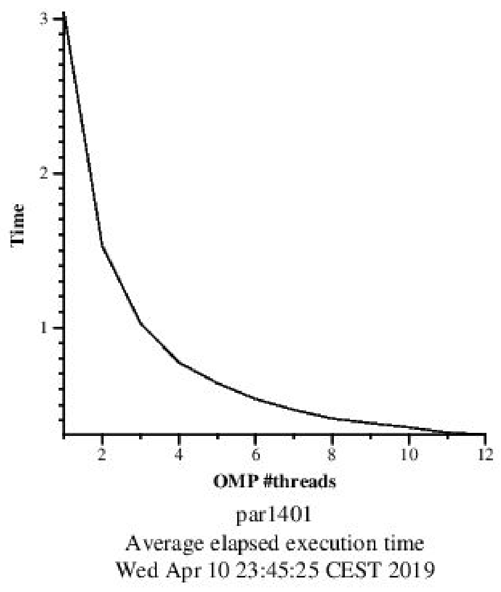
As we can see in this study (figures 18-26 and table 9), static scheduling performs is the worst with default chunk (figure 24), but it increases when reducing the chunk (figures 25 and 26). When the chunk is 1, we obtain the best execution time of all experiments.

Regarding dynamic scheduling, we don't see a big difference. This is because it doesn't the computation work is very small, so the amount of rows to perform is not that important, but the fact that it balances the workload is.

For that reason, the guided scheduling does not bring anything new to the table. It is a dynamic scheduling with a first assignment that is very big (similarly to the static + default case), which makes it slower than the dynamic scheduling.

Schedule mode	**Chunk**	**Time (s)**
Static	Default(100)	1.2940
Static	10	0.4444
Static	1	0,4199
Dynamic	Default(100)	0,4180
Dynamic	10	0.4334
Dynamic	1	0.4266
Guided	Default(100)	0.4724
Guided	10	0.4694
Guided	1	0.4653

Table 9: Time execution for the different schedulings mode*(For-based parallelization)*



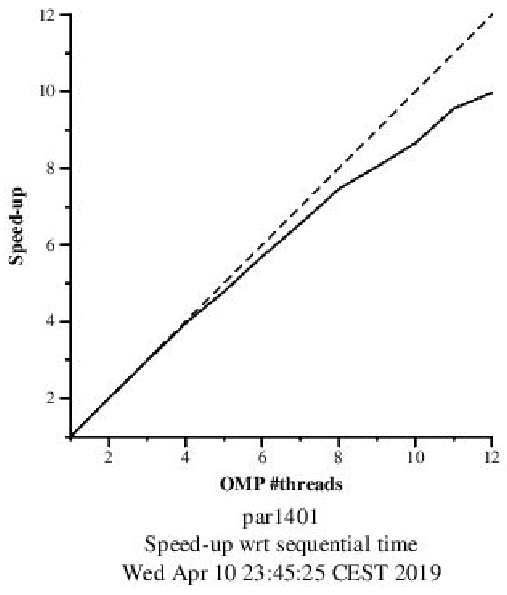


Figure 18: ***Mandel-omp strong scalability graphs (Dynamic with Default)***

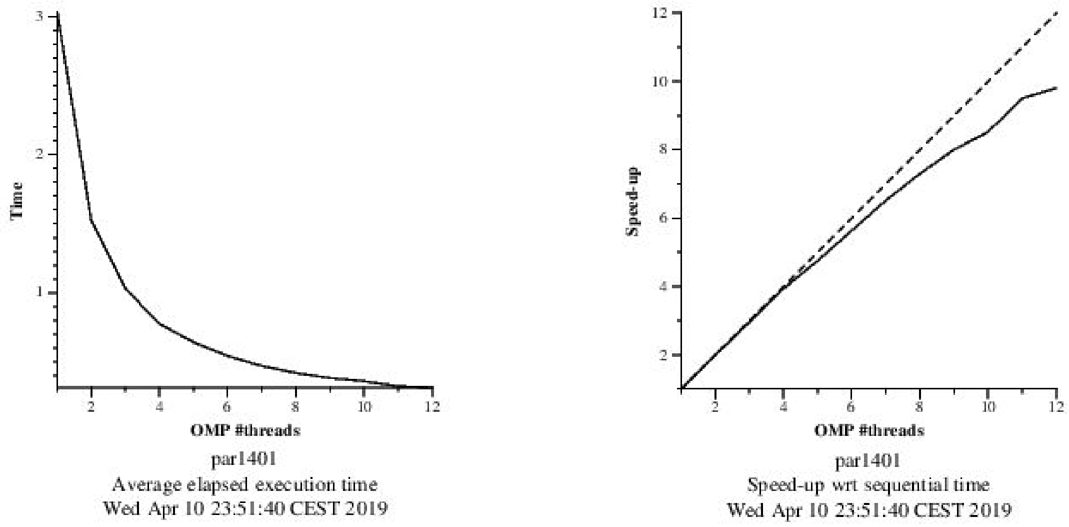


Figure 19: ***Mandel-omp strong scalability graphs (Dynamic with chunk 1)***

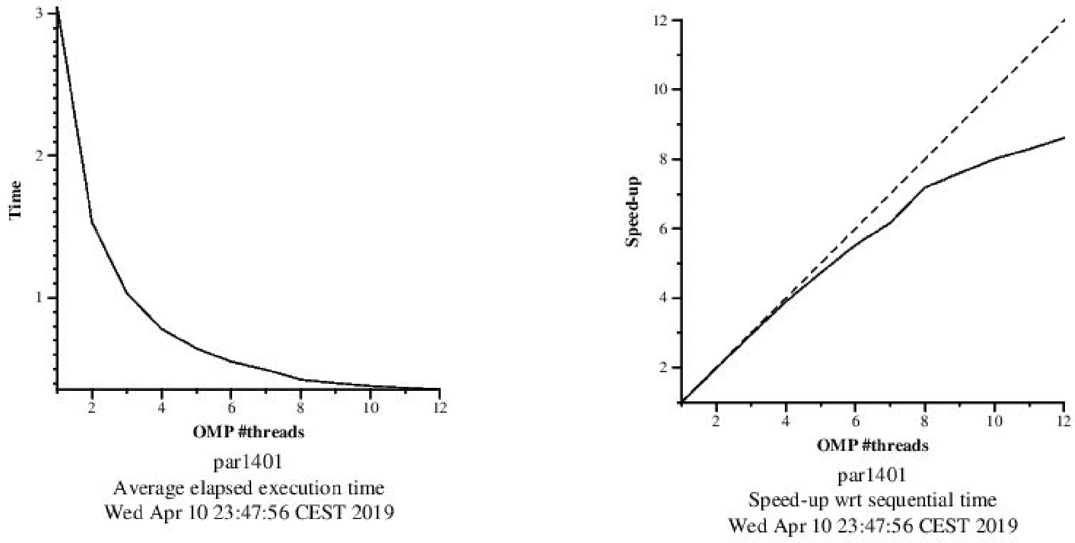
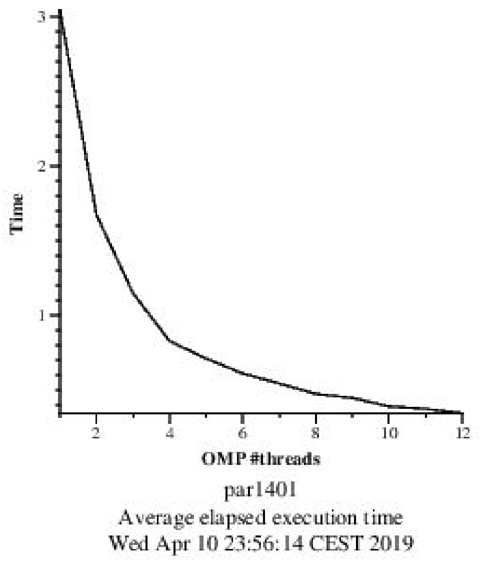


Figure 20: ***Mandel-omp strong scalability graphs (Dynamic with chunk 10)***



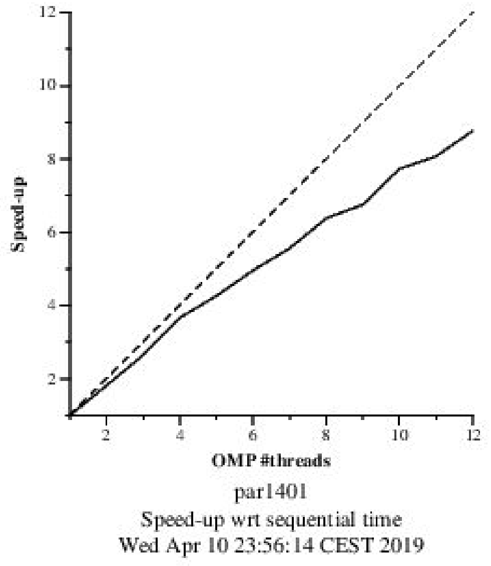
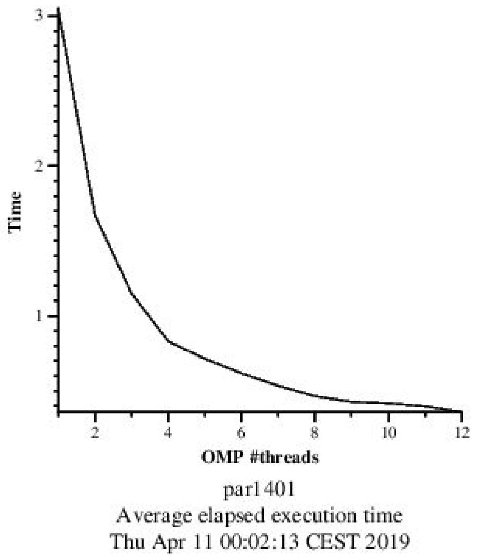


Figure 21: ***Mandel-omp strong scalability graphs (Guided default)***



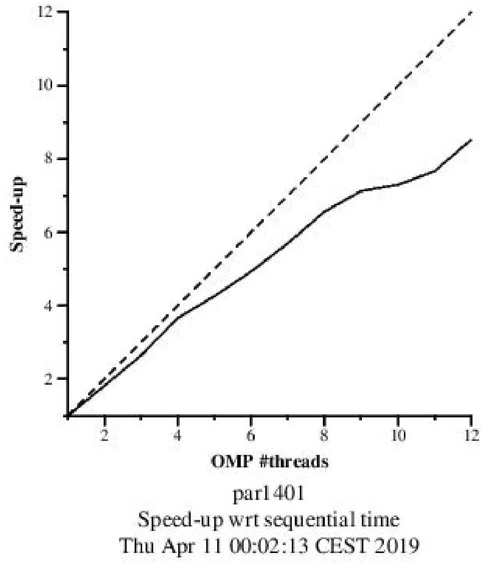


Figure 22: ***Mandel-omp strong scalability graphs (Guided with chunk 1)***

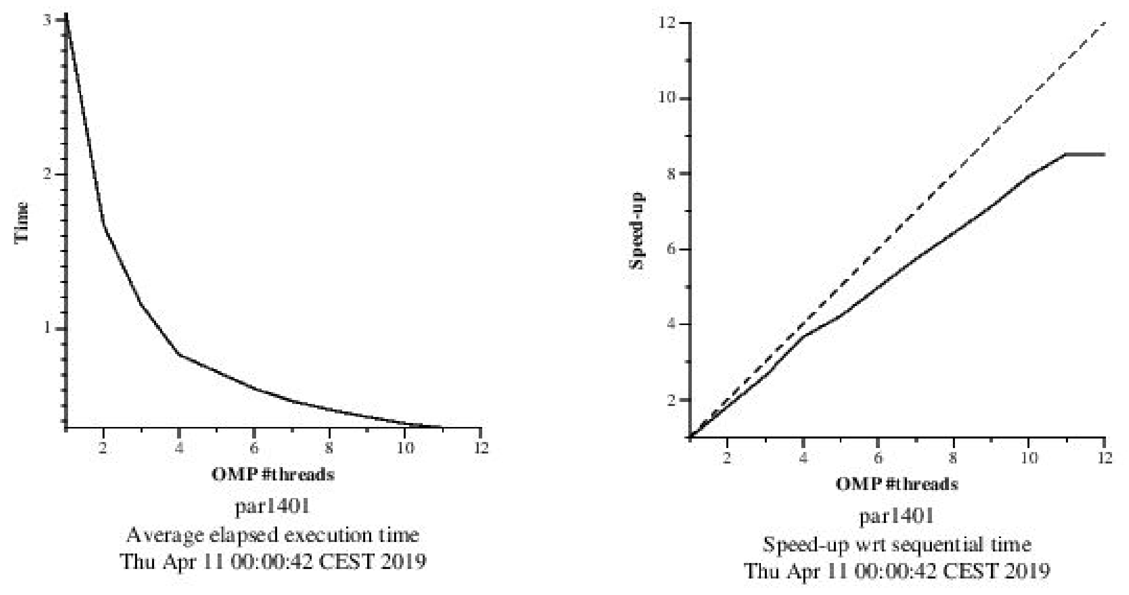


Figure 23: ***Mandel-omp strong scalability graphs (Guided with chunk 10)***

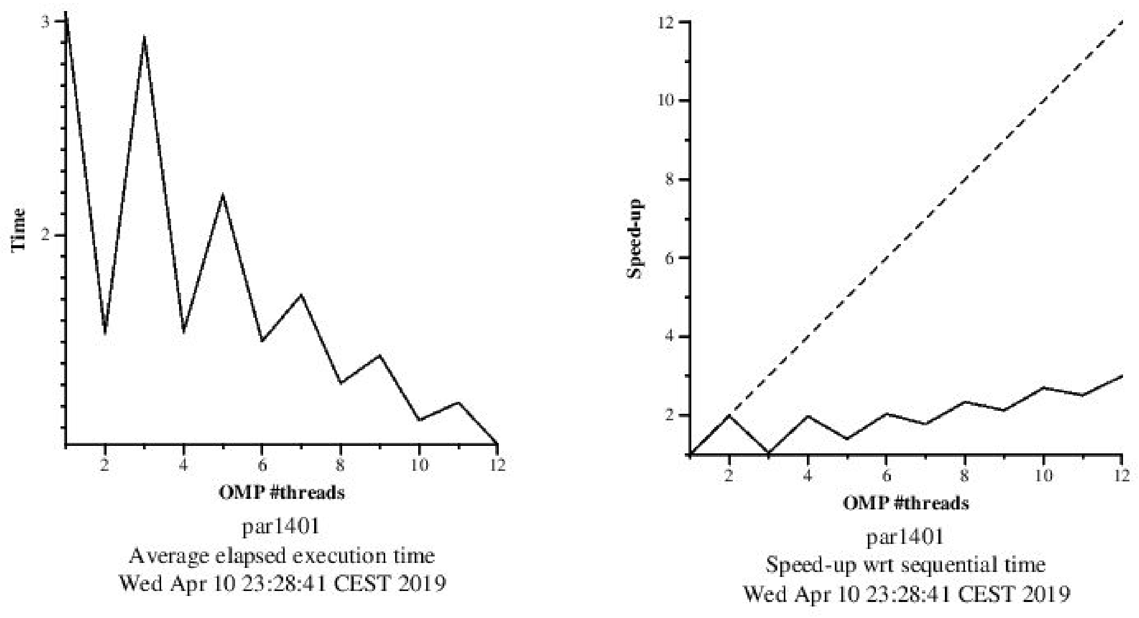


Figure 24: ***Mandel-omp strong scalability graphs (Static default)***

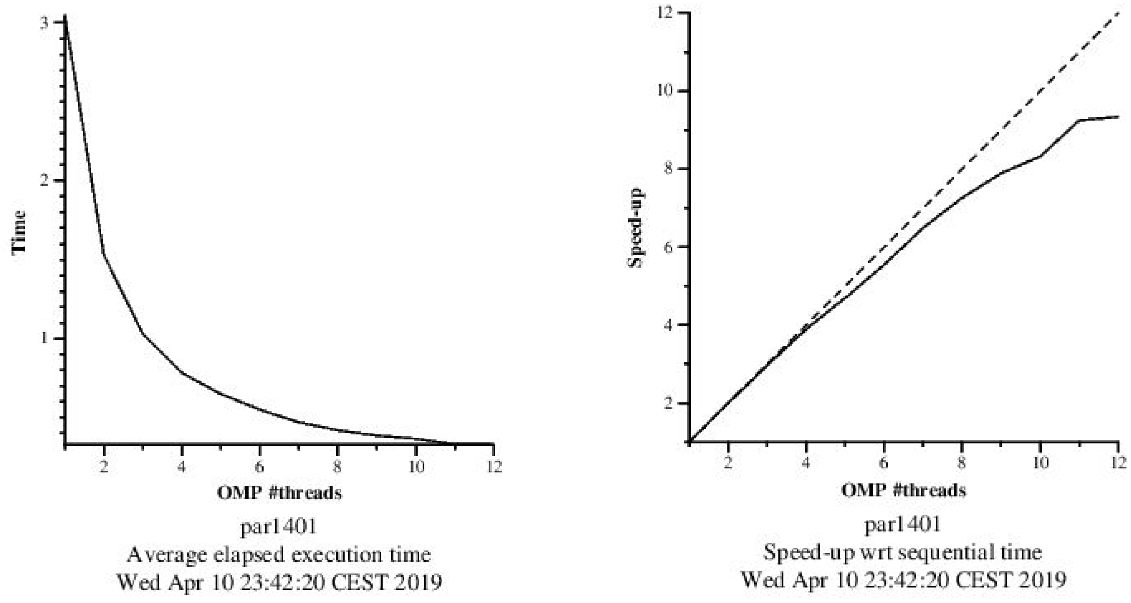


Figure 25: ***Mandel-omp strong scalability graphs (Static with chunk 1)***

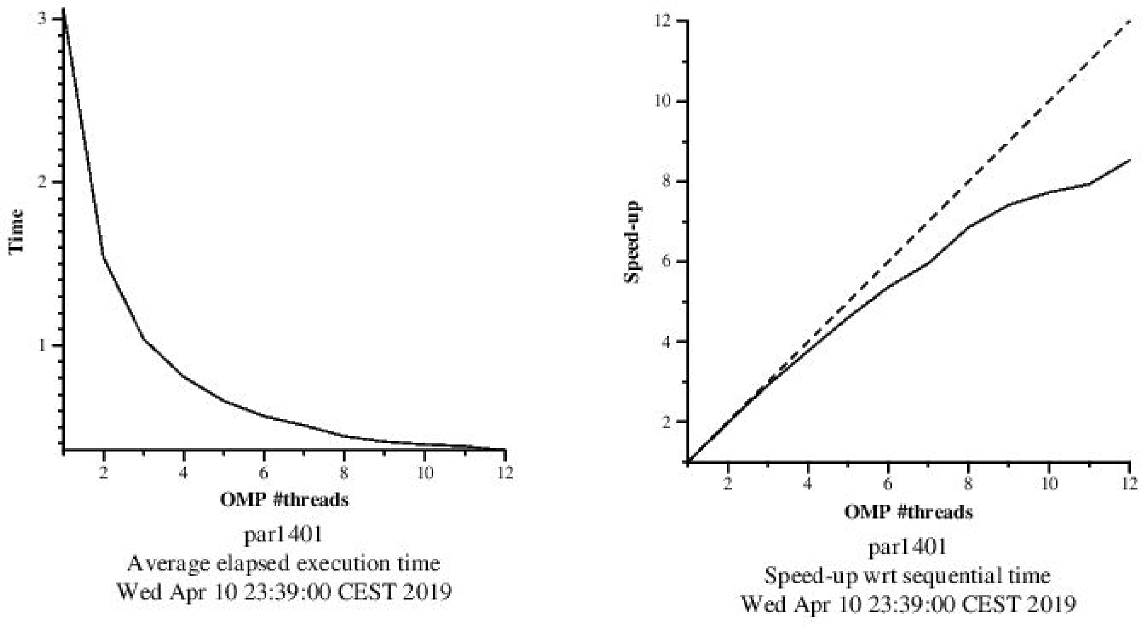


Figure 26: ***Mandel-omp strong scalability graphs (Static with chunk 10)***

CONCLUSIONS

In this study we have been improving the performance of the computation of the Mandelbrot set in small steps until we have obtained a very decent execution time based on a high parallelisation of our code, and a highly improved strong scalability, which is more important if we have access to hardware with more threads to work with.

We have learned about the similarities and differences between directives that we first saw in the previous laboratory, but now we can further explain why some clauses or complete approaches to parallelisation work better than others.
This will obviously give us more tools for future parallelism projects.