Report

PAR Laboratory 3

Fall 2022-2023

Paula Barrachina Cáceres (*par4109*)
Marc Castro Chavez (*par4111*)

Índex

2. Task decomposition analysis for the Mandelbrot set computation

Completion date: 11/10/2022

In this laboratory assignment we explored the tasking model in OpenMP to express iterative decomposition. We used the program of the computation of the Mandelbrot set.

Procedure:

2.1 The Mandelbrot set

In this first section we executed the sequential version with ***.7madel-seq -h -i 1000 -o*** in order to have a reference and get its execution time and the output file. We need that reference to check the correctness of the different parallel versions we write.

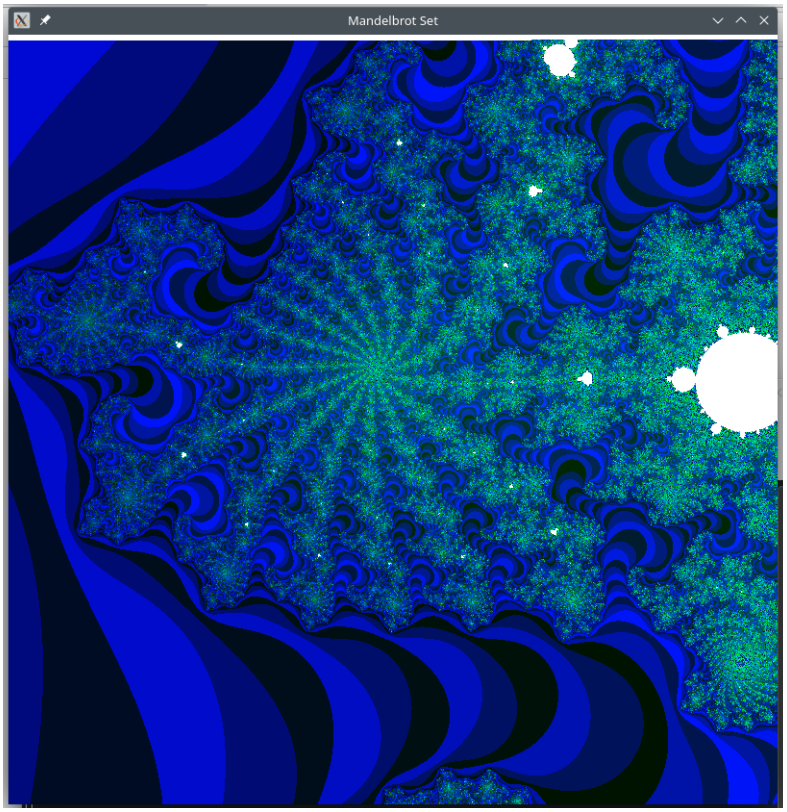


Image 1. Mandelbrot representation with stacked options as suggested

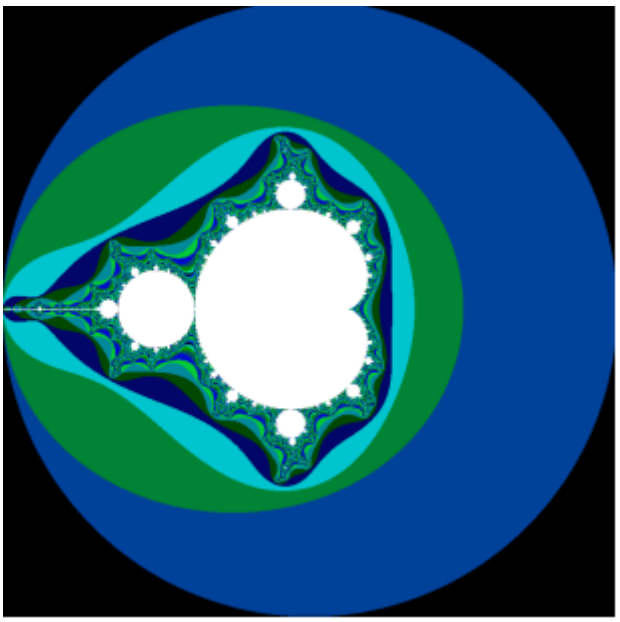


Image 2. First result generated by ./mandel-seq -h

2.2 Task decomposition analysis with Tareador

In this section we studied the main characteristics of Row task decomposition and Point task decomposition strategy, two different alternatives to generate tasks with different granularities.

Row Strategy vs Point Strategy

First we executed mandel-tar binary using the ***./run-tareador.sh*** script, with no additional options.

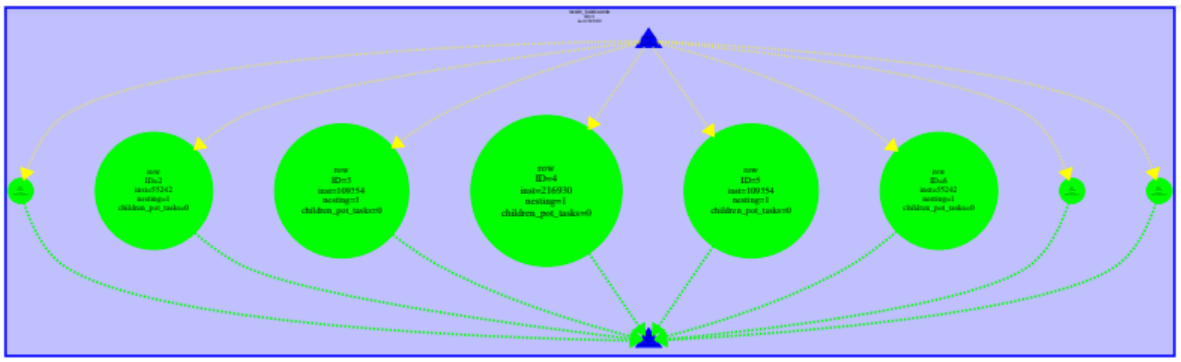


Image 3. Tareador representation of Row strategy with no additional options

As we could see in image 2 the two most important characteristics we can observe are that the work distribution behaves as a parabola, it starts with a small task and it grows until it reaches its maximum and then diminishes again. And also that there are no dependencies between tasks, which means that the program can be parallelised.

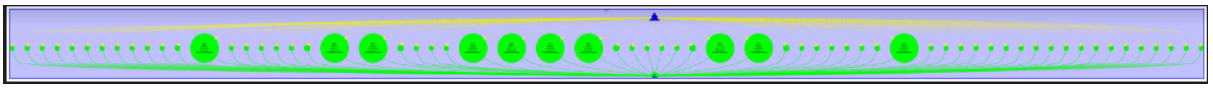


Image 4. Tareador representation of Point strategy with no additional options

This time as it shows image 3 there is a higher number of parallelizable tasks, most of them short tasks. As we can see, the workload is unevenly distributed. The change regarding Row strategy is a smaller number of instructions per task. However as the previous strategy the code is fully parallelizable as we can see in the representation from the *Tareador*.

Secondly we executed mandel-tar binary using the ***./run-tareador.sh***, but now indicating the name of the instrumented binary and the ***-d*** option.

Image 5. Tareador representation of Row strategy with -d option.

This time, image 3 shows that every task depends on the one previous to it, making the program essentially sequential, contrary to the first execution. However we can see that the workload for every task remains as a parabola.

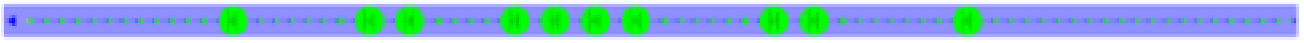


Image 6. Tareador representation of Point strategy with -d option.

From image 5 we observe that each task depends on the previous one, and has the same effect as in Row strategy. Moreover the workload distribution is equal to the first version of the Point strategy without any flag.

With the -d option, the *if(output2display)* clause is making a big difference in the execution, that is because of the functions *XSetForeground* and *XDrawPoint*. In order to protect the code we will use ***#pragma omp critical*** right before the two functions to prevent the different threads accessing the same variable that stores the color.

Finally we executed mandel-tar but now indicating the name of the instrumented binary and only the ***-h*** option.

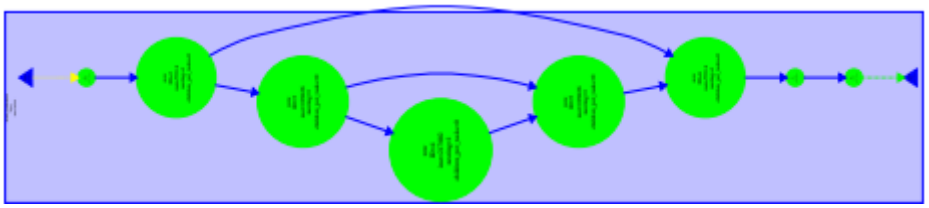


Image 8. Tareador representation of Row strategy with -h option.

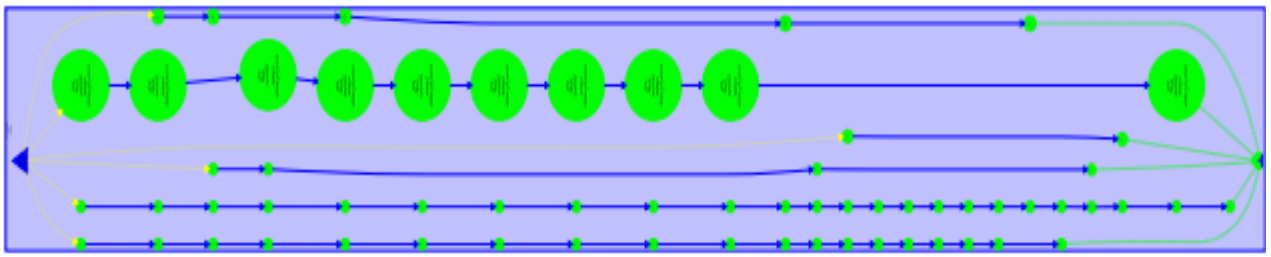


Image 9. Tareador representation of Point strategy with -h option.

Observing the two images we see that for Row strategy all tasks depend on each other, although the program is not completely sequentially whereas in Point strategy we see that we can execute a minor set of tasks in parallel.
The fragment of the code that is making the big difference between the two previous cases is the *if(output2histogram)historgram[k-1]*++; function.
In order to protect that part of the code we could use ***#pragma omp atomic*** before *histogram[k-1]*++.

In conclusion, according to the results we think that Point strategy is the one that feats more for a parallel version of Mandelbrote code. That is because the granularity and task decomposition is finer and the parallel execution would be more efficient. However this strategy may cause more overhead than the other but the execution time would be lower with the necessary processors and hardware.

3. Implementation and analysis of task decompositions in OpenMP

In this session we explored the different options in the OpenMP tasking model to express the iterative task decomposition strategies for the Mandelbrot computation program. Analyzing the scalability and behavior of our implementation.

3.1 Point decomposition strategy

First we edited the initial task version of ***mandel-omp.c*** adding ***#pragma omp atomic*** and ***#pragma omp critical*** we executed the code:

Execution with one thread

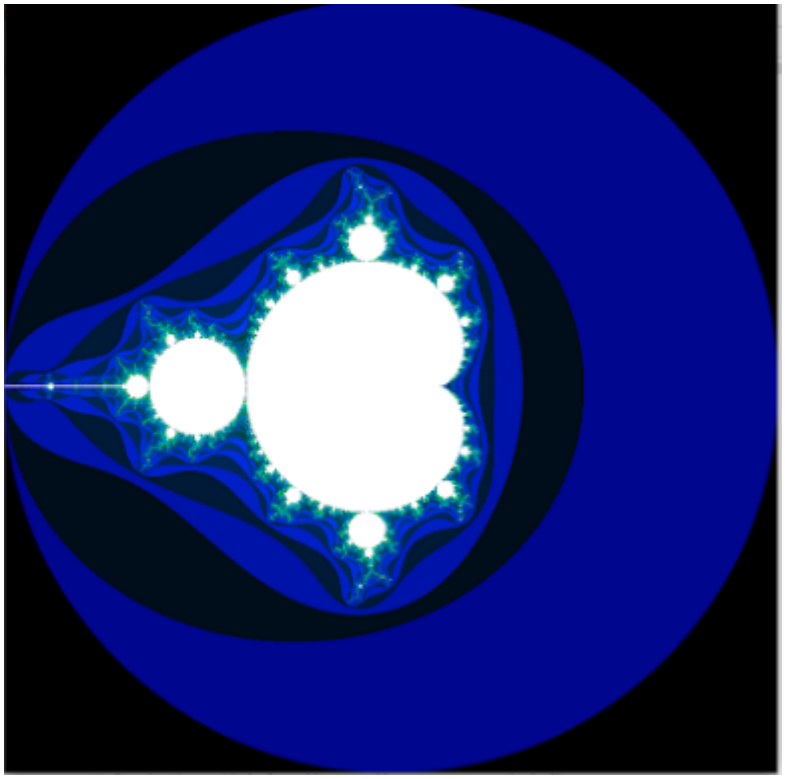


Image 10. Result of the execution OMP_NUM_THREADS=1 ./madel-omp -d -h -i 10000.

We can see in the result (image 9), that it is identical to the original image in shape, however the color display is not the same, which indicates that there is missing information.

Execution with two threads

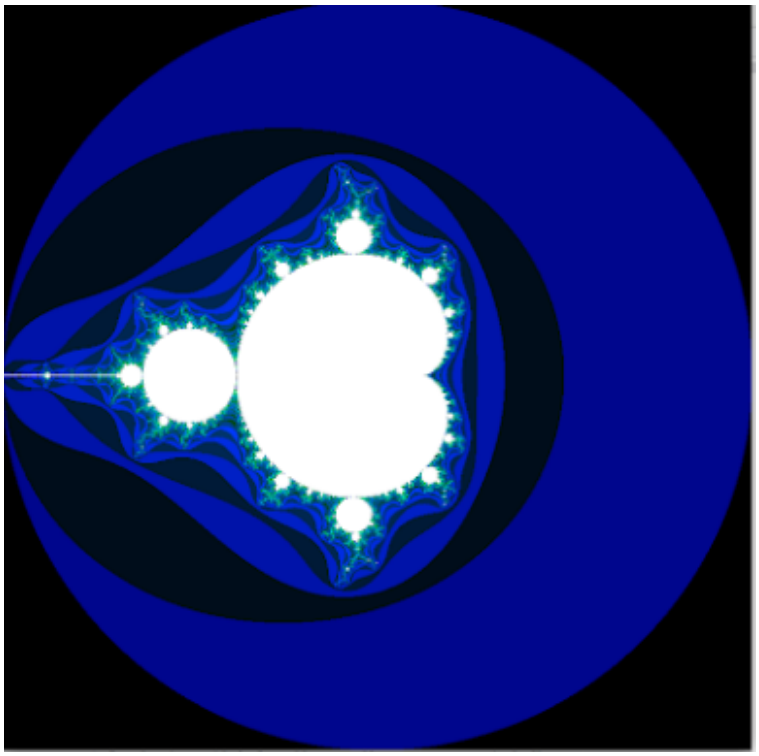


Image 11. Result of the execution OMP_NUM_THREADS=2 ./madel-omp -d -h -i 10000. explicació

In this case the image is identical but it generates fastly.

Then we submit the execution with 1 and 8 threads.

Number of threads	Execution time
1	2.9675s
8	1.493260s

Table1: Table of execution of sbach ./submit.omp.sh mandel-omp with 1 and 8 threads

As we can see in table 1 the speed-up is not the ideal but we can observe considerable difference between 1 and 8 threads. Also we see that with 1 thread the execution time is worse than the sequential execution (2.09220s).

In order to study the scalability we executed the binary to obtain the execution time and speed-up plots for 1-20 processors range.

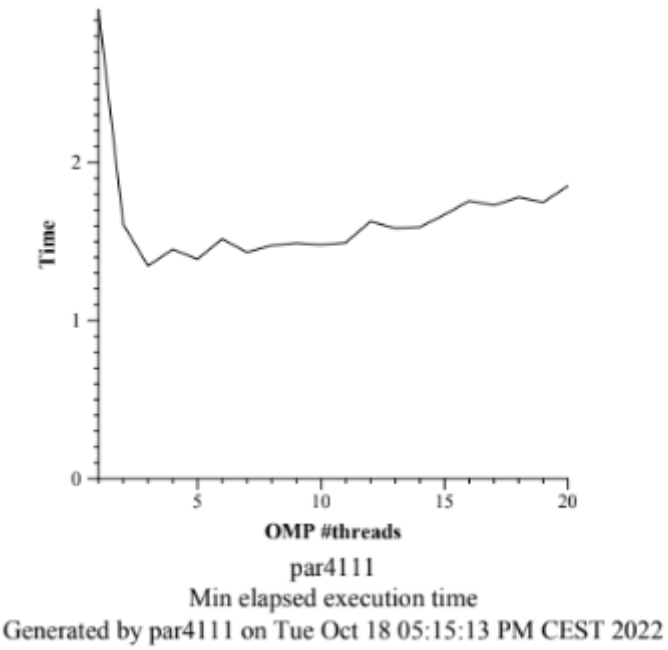


Image 12. Plot of the execution time in the 1-20 processor range.

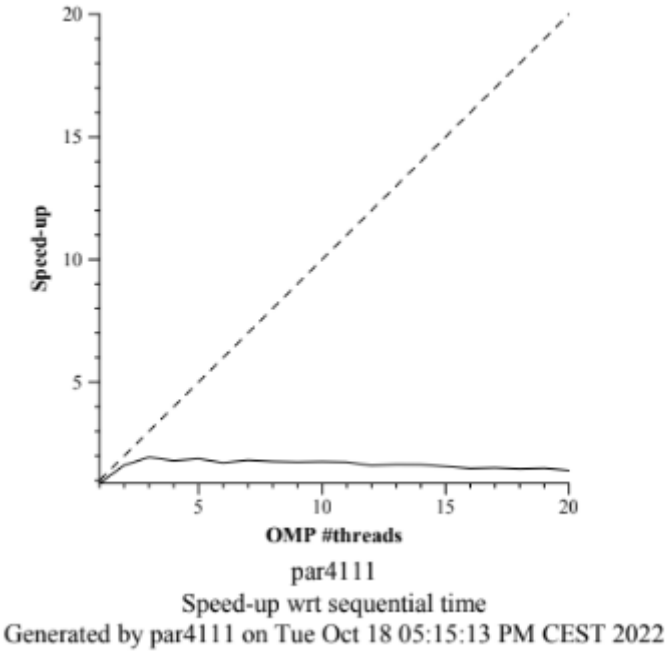


Image 13. Plot of the speed-up in the 1-20 processor range.

From the plots we analyzed their scalability. We can see that the scalability is not appropriate. In the first plot we see that the time is nearly constant and there is not considerable improvement in it. Moreover, if we observe the second plot we can see that the speed-up is practically constant.

In order to better understand how the execution goes, we submit the ***submit-strong-extrae.sh*** script to perform a first overall analysis of the parallel execution metrics with ***modelfactors.py***.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.58	0.36	0.31	0.31	0.35
Speedup	1.00	1.61	1.85	1.84	1.64
Efficiency	1.00	0.40	0.23	0.15	0.10

Table 1: Analysis done on Tue Oct 18 05:29:58 PM CEST 2022, par4111

Image 14. Table of execution metrics.

Observing the table we can see that in terms of speedup and execution time the results are nearly constant. Also, in terms of efficiency we see that as more processors we add it results in lower efficiency.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.95%					
Number of processors	1	4	8	12	16
Global efficiency	95.36%	38.34%	22.07%	14.64%	9.79%
Parallelization strategy efficiency	95.36%	46.43%	29.66%	20.18%	14.86%
Load balancing	100.00%	91.93%	55.69%	41.09%	23.85%
In execution efficiency	95.36%	50.51%	53.26%	49.11%	62.31%
Scalability for computation tasks	100.00%	82.57%	74.42%	72.53%	65.84%
IPC scalability	100.00%	79.78%	74.55%	74.88%	68.14%
Instruction scalability	100.00%	103.81%	104.31%	104.54%	104.11%
Frequency scalability	100.00%	99.70%	95.70%	92.65%	92.80%

Table 2: Analysis done on Tue Oct 18 05:29:58 PM CEST 2022, par4111

Image 15. Table of efficiency.

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	102400.0	102400.0	102400.0	102400.0	102400.0
LB (number of explicit tasks executed)	1.0	0.8	0.84	0.86	0.84
LB (time executing explicit tasks)	1.0	0.89	0.89	0.9	0.9
Time per explicit task (average us)	4.9	5.69	5.92	5.99	6.04
Overhead per explicit task (synch %)	0.0	101.66	265.89	463.99	754.72
Overhead per explicit task (sched %)	5.35	30.53	24.09	26.04	21.49
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Tue Oct 18 05:29:58 PM CEST 2022, par4111

Image 16.Table of the statistics about explicit tasks.

In conclusion we see that there is not a considerable improvement in the program by adding more processors and each time we add a processor the overhead time increases considerably.

Then we did a detailed analysis with Paraver:

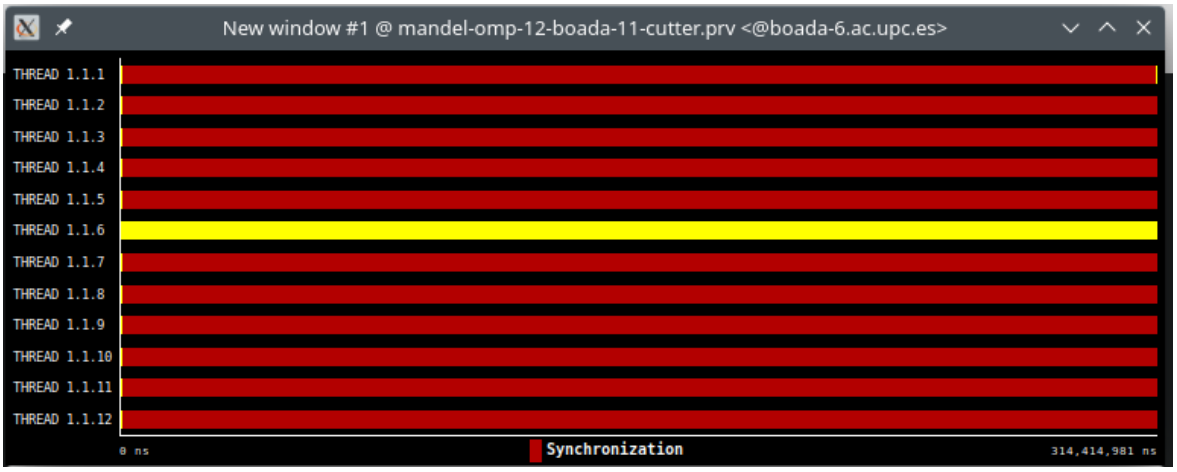


Image 17. Timeline of the execution of implicit task of mandel-omp-12

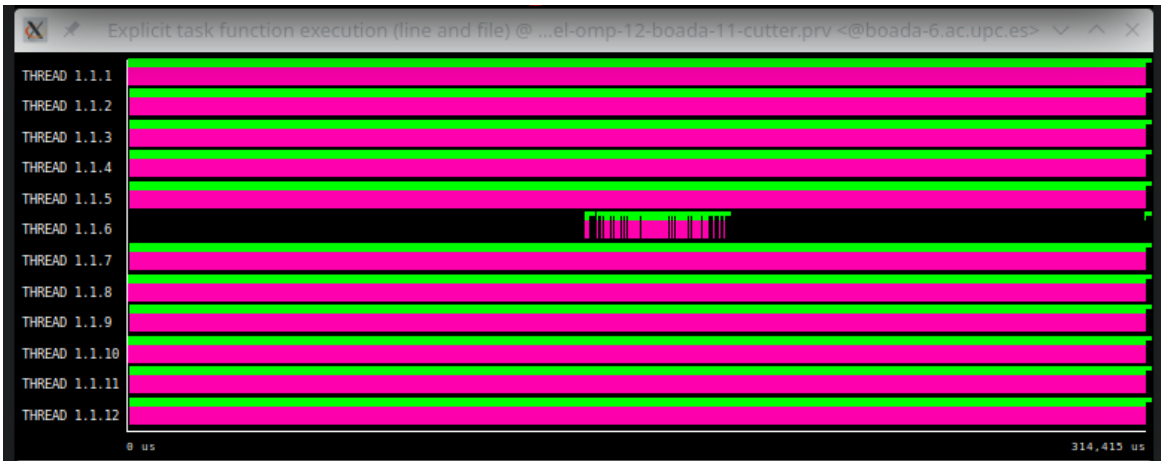


Image 18.Timeline of the execution of explicit task of mandel-omp-12

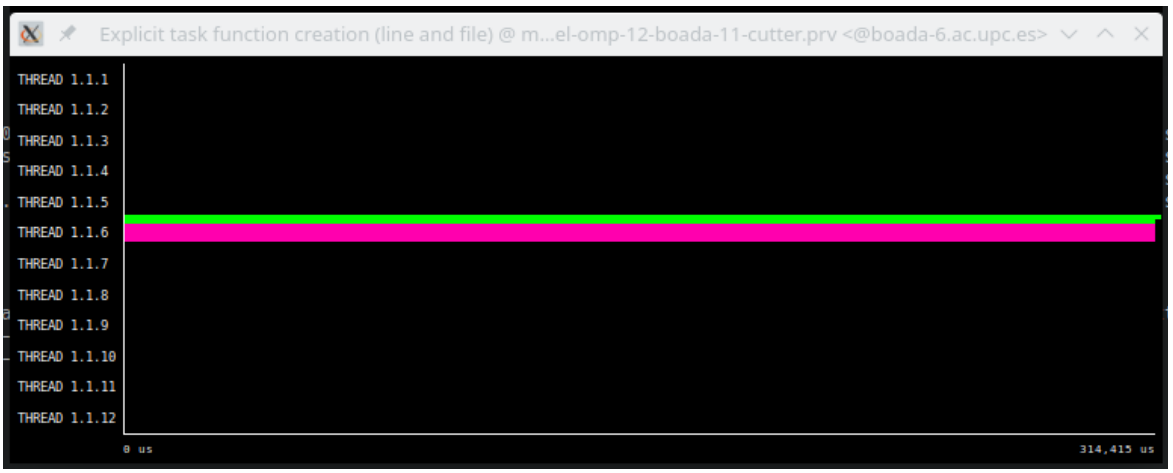
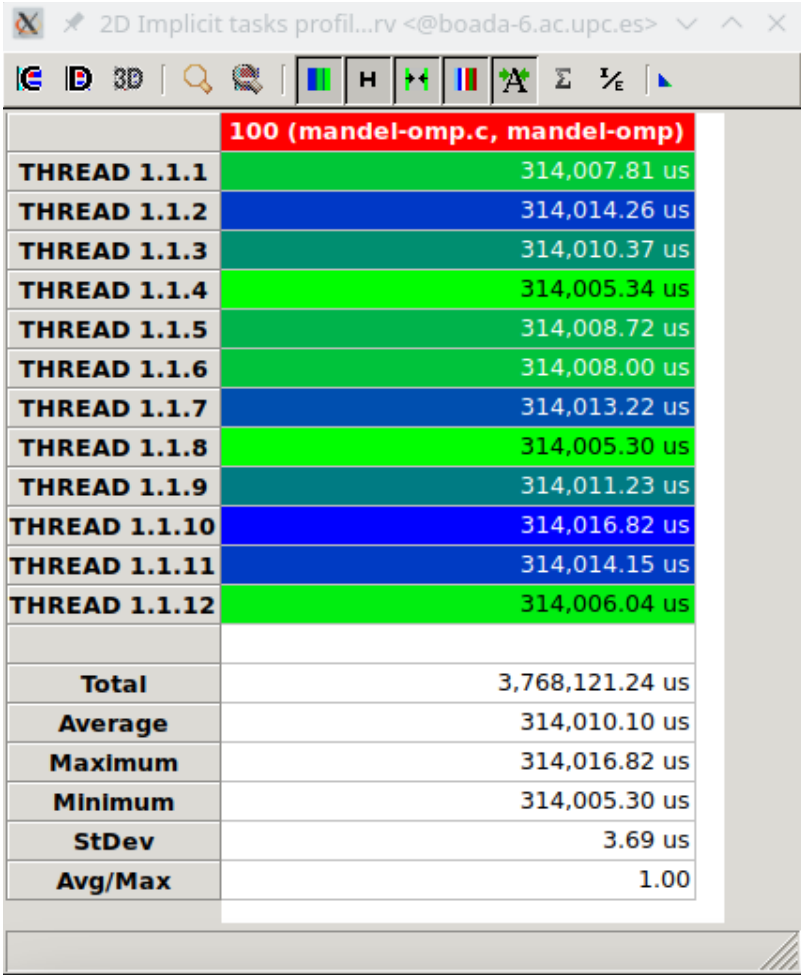


Image 19. Timeline of the creation of explicit tasks of mandel-omp

From these three timelines we concluded that thread 1.1.6 is the one creating the tasks and that only one thread creates all the tasks.



	100 (mandel-omp.c, mandel-omp)
THREAD 1.1.1	314,007.81 us
THREAD 1.1.2	314,014.26 us
THREAD 1.1.3	314,010.37 us
THREAD 1.1.4	314,005.34 us
THREAD 1.1.5	314,008.72 us
THREAD 1.1.6	314,008.00 us
THREAD 1.1.7	314,013.22 us
THREAD 1.1.8	314,005.30 us
THREAD 1.1.9	314,011.23 us
THREAD 1.1.10	314,016.82 us
THREAD 1.1.11	314,014.15 us
THREAD 1.1.12	314,006.04 us
Total	3,768,121.24 us
Average	314,010.10 us
Maximum	314,016.82 us
Minimum	314,005.30 us
StDev	3.69 us
Avg/Max	1.00

Image 20. Implicit task table of paraver.

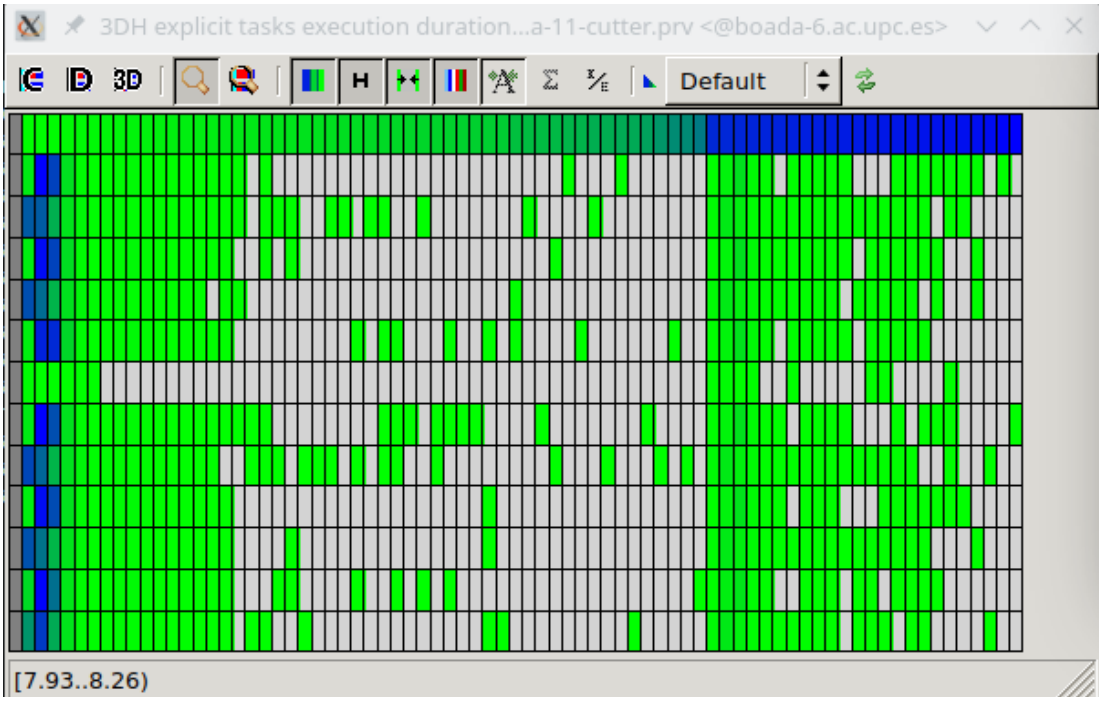


Image 21. Histogram of the duration of the explicit tasks of mandel-omp-12.

As we can see in the images, only one thread creates the tasks (Thread 1.1.6) and all 12 threads execute them. However all threads have a similar execution time, from that, we can concluded that the granularity and the workload is well balanced between the threads.

It becomes clear how inefficient the program is by looking at the tables generated by the ***modelfactor.py***. Looking at the paravers results we concluded that however the granularity of the tasks is balanced, the increment of overheads and the decrement of the efficiency is not insignificant.

Next we use the taskloop construct in order to control the granularity, ***#pragma omp taskloop num_tasks(n).*** That clause generates tasks out of iteration of the loop so it allows a better control of them.

To study the scalability we executed the submit-strong.sh and we obtain the strong scalability graphs and the modelfactor tables:

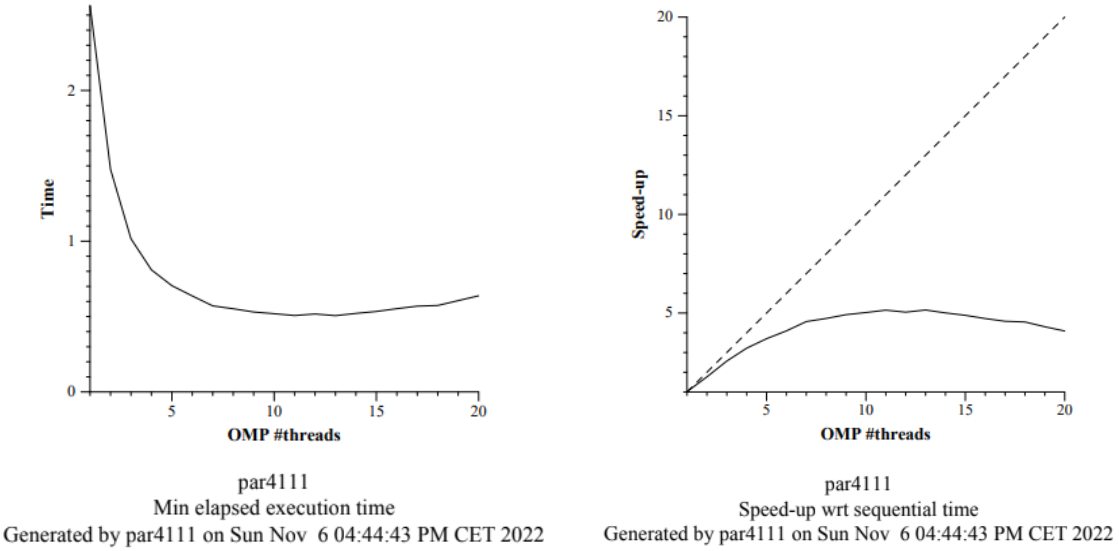


Image 22. Scalability plots of the taskloop version.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.41	0.14	0.13	0.15	0.20
Speedup	1.00	2.95	3.19	2.71	2.06
Efficiency	1.00	0.74	0.40	0.23	0.13

Table 1: Analysis done on Sun Nov 6 04:20:39 PM CET 2022, par4111

Image 23. Execution metrics table of the taskloop version.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.88%					
Number of processors	1	4	8	12	16
Global efficiency	99.53%	73.49%	39.72%	22.47%	12.81%
Parallelization strategy efficiency	99.53%	76.70%	45.01%	26.49%	15.42%
Load balancing	100.00%	96.01%	96.01%	94.99%	95.25%
In execution efficiency	99.53%	79.89%	46.88%	27.89%	16.18%
Scalability for computation tasks	100.00%	95.81%	88.24%	84.83%	83.11%
IPC scalability	100.00%	97.51%	97.19%	96.91%	96.14%
Instruction scalability	100.00%	99.42%	98.67%	97.91%	97.18%
Frequency scalability	100.00%	98.83%	92.02%	89.41%	88.96%

Table 2: Analysis done on Sun Nov 6 04:20:39 PM CET 2022, par4111

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	3200.0	12800.0	25600.0	38400.0	51200.0
LB (number of explicit tasks executed)	1.0	0.94	0.83	0.53	0.48
LB (time executing explicit tasks)	1.0	0.96	0.96	0.95	0.95
Time per explicit task (average us)	128.71	33.59	18.23	12.64	9.68
Overhead per explicit task (synch %)	0.07	26.75	108.52	254.97	514.46
Overhead per explicit task (sched %)	0.4	3.64	13.68	22.71	34.48
Number of taskwait/taskgroup (total)	320.0	320.0	320.0	320.0	320.0

Table 3: Analysis done on Sun Nov 6 04:20:39 PM CET 2022, par4111

Image 24. Efficiency and explicit tasks table of the taskloop version.

Looking at the tables, we see that contrary to the previous execution without taskloop this execution has less overhead and more speed-up and efficiency as we could see comparing image 23 with image 14.
Overall this version of the program is better than the last one but it still has a lot of overhead and the efficiency could be improved. We should blame that to the execution efficiency. Moreover, the scalability plots are better than the previous ones, the speed up is not as constant as the first execution and the execution time decreases when more processors are added.

Talking in terms of parallelization strategy efficiency we see a huge change with 4 and 8 threads but with more than that it is nearly the same as the previous version. Also we can see an improvement with the granularity, there are 320 tasks executed per taskloop and in the previous execution there were 0.
In terms of total number of tasks generated we see that there is a huge difference with 16 processors and with 4 and 8 it is similar.

To continue studying this version with the taskloop we used Paraver to extract the following timelines:

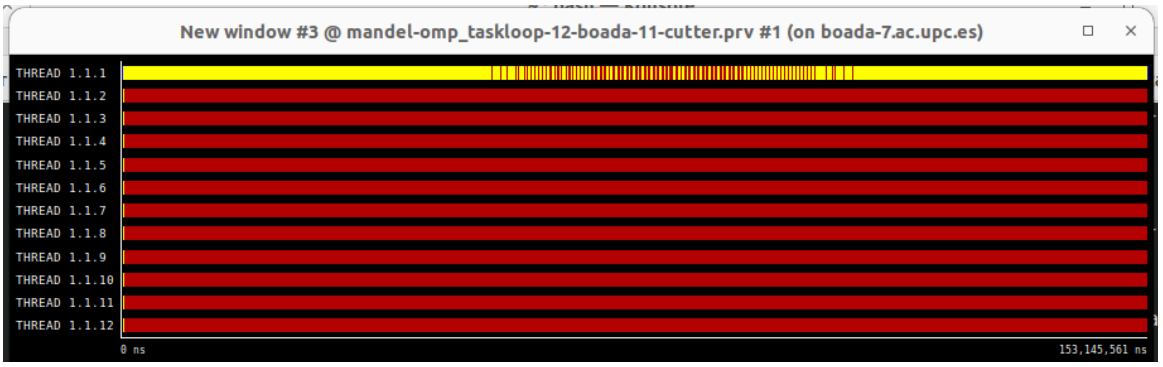


Image 25. Efficiency and explicit tasks table of the taskloop version.

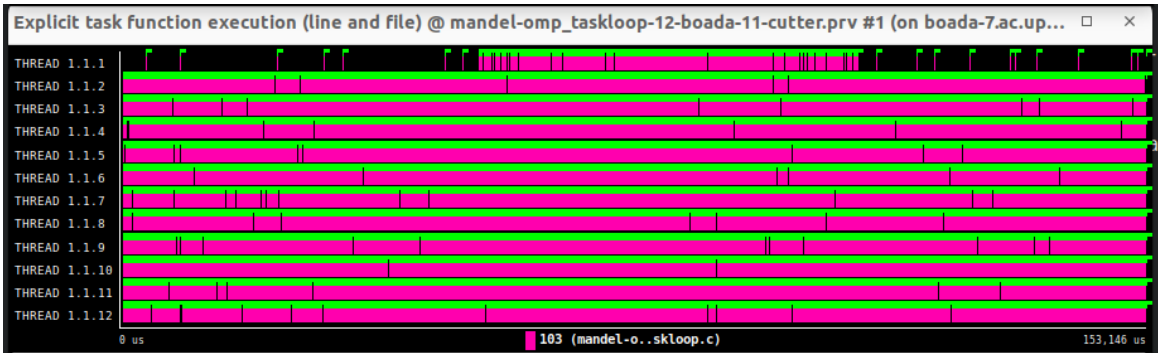


Image 26. Efficiency and explicit tasks table of the taskloop version.

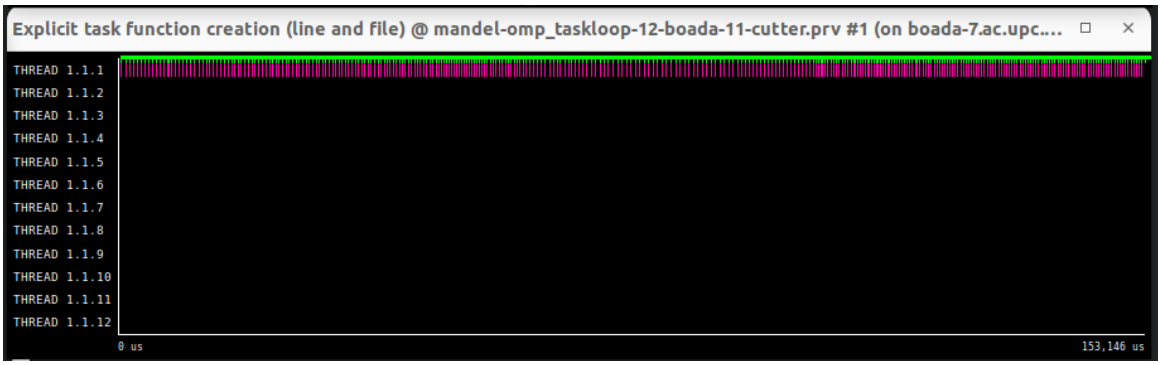


Image 27. Efficiency and explicit tasks table of the taskloop version.

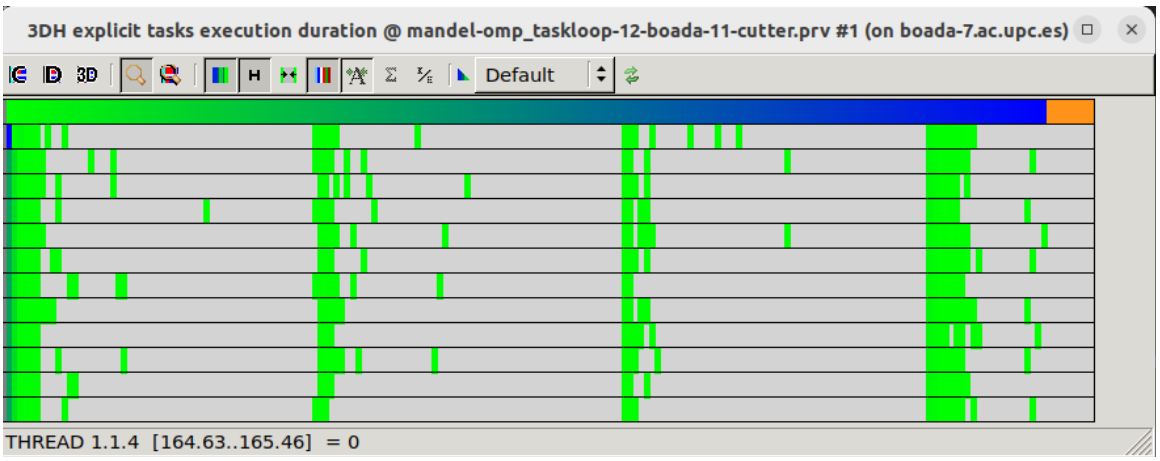


Image 28. Efficiency and explicit tasks table of the taskloop version.

Finally we open with Paraver thee trace generated with 12 threads
We can observe that the execution does not continue until all tasks generated up to each one of these points are finished, introducing a certain load unbalance that is because of the task synchronization caused by the taskloop.

In conclusion we think that the task barriers are necessary in order to obtain the correct result and reduce the overheads and improve the program efficiency. However, we think that the overall results could be better with other parallel strategy.

3.2 Row decomposition strategy

In this section we studied the row strategy and compared to the point strategy granularity. The execution give us the correct result:

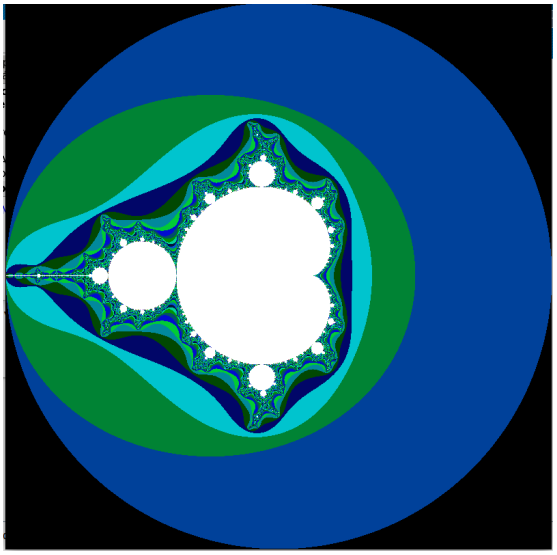


Image 29. Result with row execution.

Then we executed modelfactor in order to obtain the tables and the plots to compare it to the point strategy.

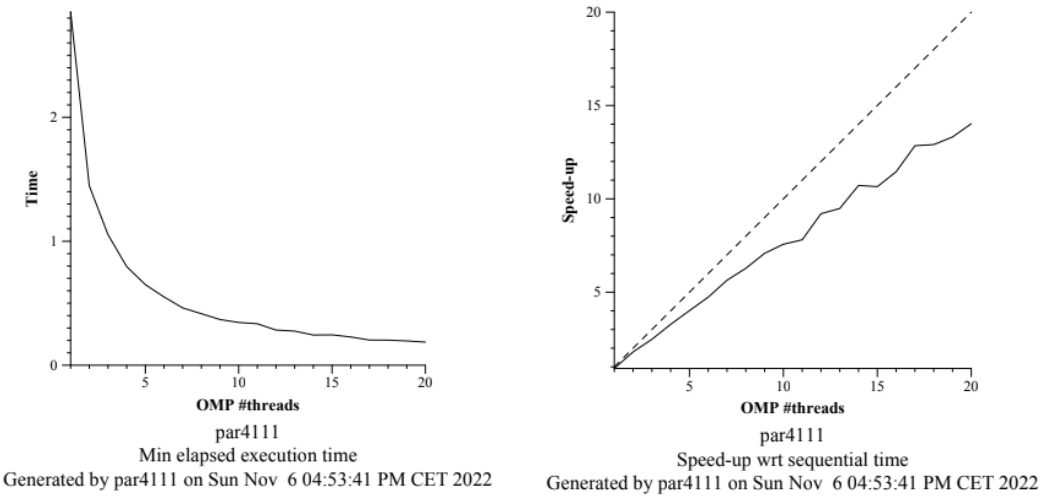


Image 30. Scalability plots of row strategy.

As we can see in the plots the speed-up tends to the ideal speed-up gain, opposed to point-strategy versions of the code, which were nearly constant.
Furthermore we observe that the time decreases as we add more threads as we expected when we parallelize. From the plots we clearly see a huge improvement versus the first version and a lot of potential for more threads.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.13	0.07	0.05	0.04
Speedup	1.00	3.52	6.65	9.50	12.29
Efficiency	1.00	0.88	0.83	0.79	0.77

Table 1: Analysis done on Sun Nov 6 03:41:52 PM CET 2022, par4111

Image 31.Execution metrics of row strategy.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.93%					
Number of processors	1	4	8	12	16
Global efficiency	99.98%	87.96%	83.47%	79.52%	77.31%
Parallelization strategy efficiency	99.98%	90.46%	92.10%	90.74%	88.91%
Load balancing	100.00%	90.53%	92.22%	90.93%	89.26%
In execution efficiency	99.98%	99.93%	99.87%	99.80%	99.61%
Scalability for computation tasks	100.00%	97.24%	90.63%	87.63%	86.95%
IPC scalability	100.00%	98.46%	97.41%	96.69%	95.98%
Instruction scalability	100.00%	100.00%	99.99%	99.99%	99.99%
Frequency scalability	100.00%	98.76%	93.05%	90.64%	90.60%

Table 2: Analysis done on Sun Nov 6 03:41:52 PM CET 2022, par4111

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	10.0	40.0	80.0	120.0	160.0
LB (number of explicit tasks executed)	1.0	0.59	0.32	0.19	0.17
LB (time executing explicit tasks)	1.0	0.91	0.92	0.91	0.89
Time per explicit task (average us)	45779.17	11769.62	6313.3	4352.91	3289.82
Overhead per explicit task (synch %)	0.0	10.51	8.53	10.14	12.38
Overhead per explicit task (sched %)	0.01	0.02	0.04	0.05	0.06
Number of taskwait/taskgroup (total)	1.0	1.0	1.0	1.0	1.0

Table 3: Analysis done on Sun Nov 6 03:41:52 PM CET 2022, par4111

Image 32.Efficiency and explicit tasks table of row strategy.

Comparing the table to the images 14,15,16 we can see a huge overall improvement.
The elapsed time is reduced drastically and the speed-up increases considerably.Moreover, the global efficiency is higher thanks to the parallel strategy chosen and also the overheads are reduced. As we can see the granularity is coarser than the point strategy and it covers much more iterations per task.

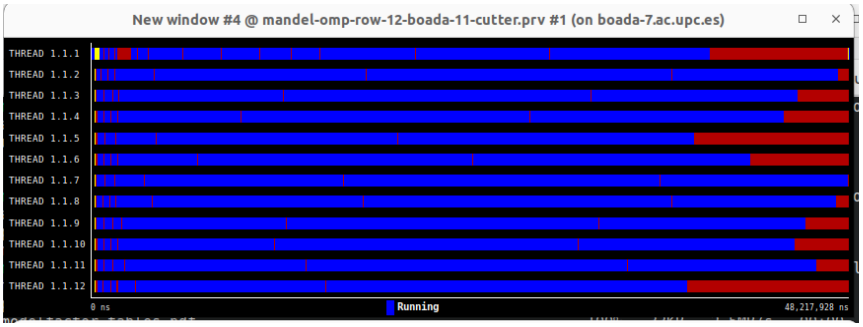


Image 33.Efficiency and explicit tasks table of row startegy.

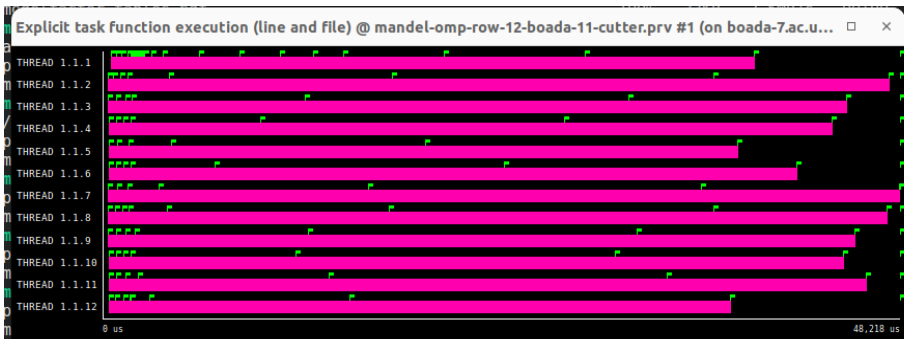


Image 34.Explicit task execution of row strategy.

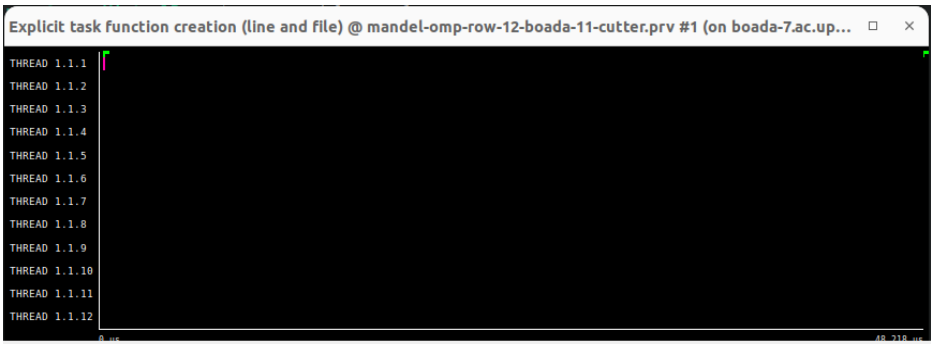


Image 35.Explicit task creation of row strategy.

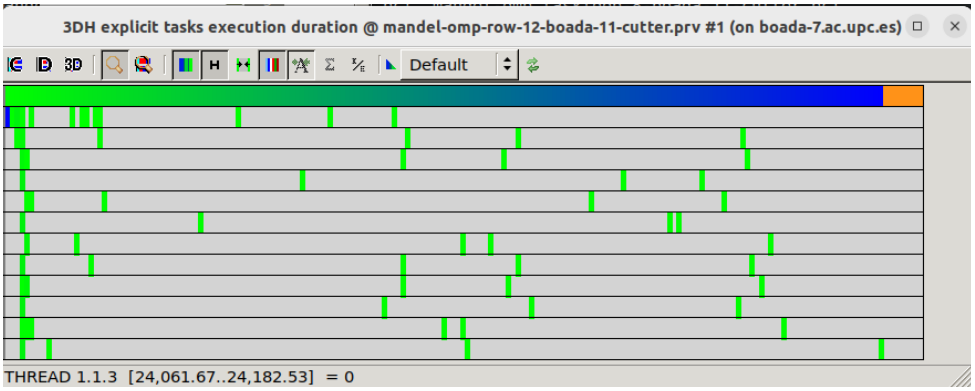


Image 36.Explicit tasks execution duration.

With the paraver results we see how the task creation is faster than the point version and we also can observe that workload is distributed differently than in point strategy.

In conclusion, after all the results we obtained we conclude that the row strategy is the best option for our program.
That is because the two strategies are basically opposites; one has fine granularity (Point) and the other coarser granularity (Row). This explains the difference between the results we obtained and why our program works much better in Row strategy.

3.3 Optional: task granularity tune

In order to understand better the effects of different task granularities we used the provided ***submit-numtasks-omp.sh*** script.
We modified the code introducing the command in both strategies:
#pragma omp taskloop num_task(user_param)

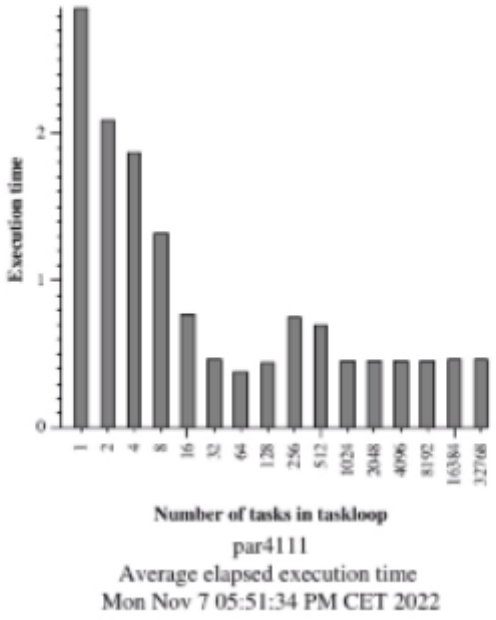


Image 37.Plot of number of tasks in taskloop in point strategy.

The point strategy presented a clear descent on execution time, however we can observe that with 256 and 512 threads it increases and does not follow the decreasing tendency. Furthermore we see that the behavior of the task in taskloop is as we should expect because by adding more tasks does not decrease the time, in fact the better time is with 64 tasks in taskloop and not with 32768.

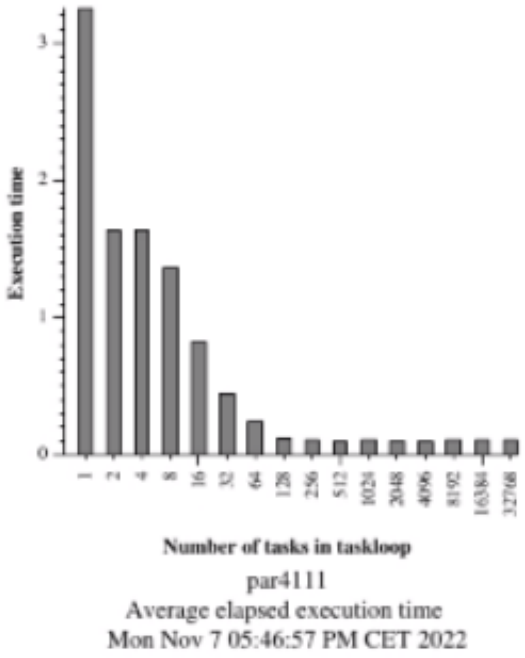


Image 38.Plot of number of tasks in taskloop in row strategy.

The row strategy shows a big improvement in time execution as we add tasks per taskloop, in addition we can observe that from 128 tasks to 32768 tasks in the taskloop it maintains the lowest value. Moreover in contrast to the first plot we see that there are no ups and downs in time execution.

Seeing both plots, we conclude that as the previous sections have shown us, in this code Row strategy is the best choice.