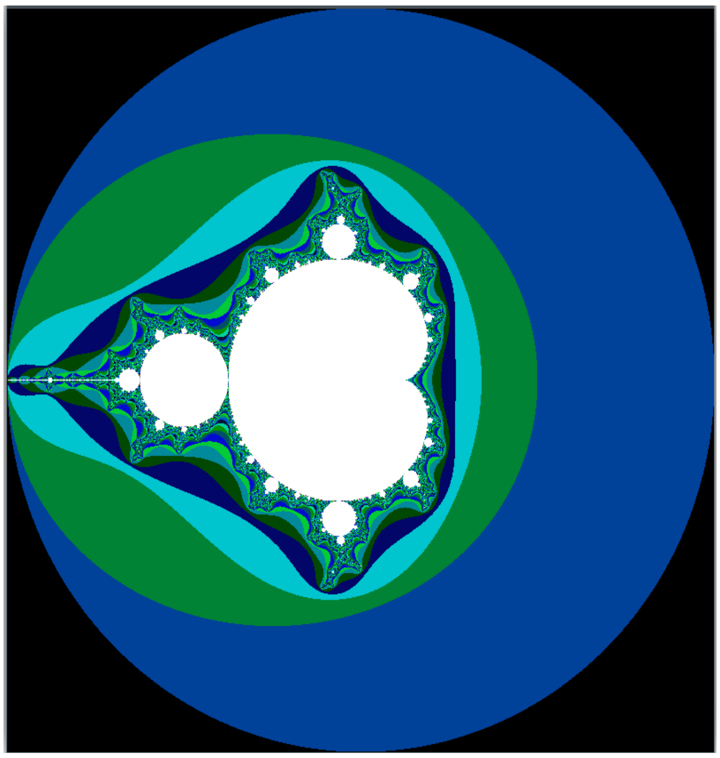
PAR Laboratory Assignment
Lab 3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

David Cañadas López and Igor Duran Gallart, group 42,
19 October 2023, par4204 and par4206 (respectively)

9 November 2023



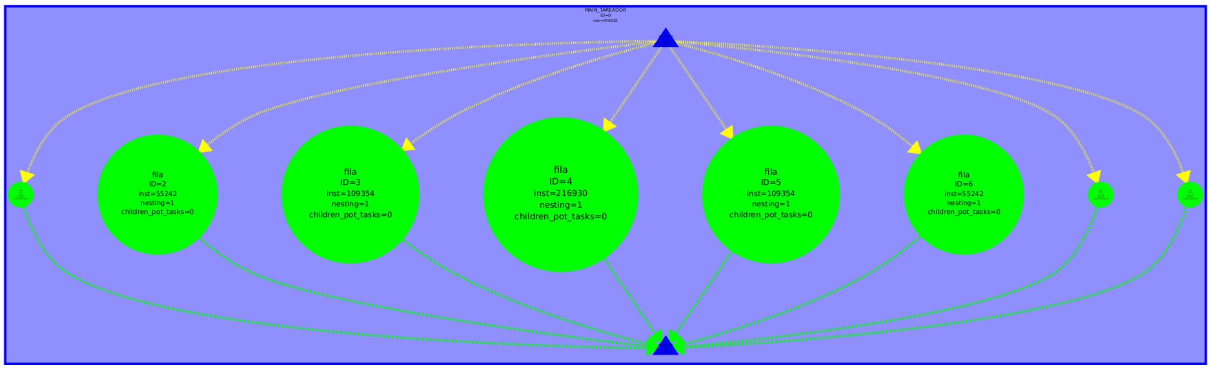
UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

1 Laboratory 3 notebook

1.1 Task decomposition analysis with Tareador

1.1.1 First analysis with Tareador

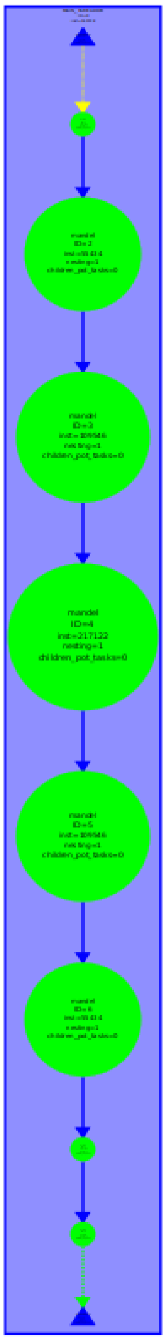
Which are the two most important characteristics for the task graph that is generated?



Out of the total 8 tasks, we can see that five of them make up most of the executed instructions, while the rest are very tiny in comparison. Additionally, the potential tasks do not have any dependencies between them, thus could be fully parallelized.

1.1.2 Second analysis with Tareador

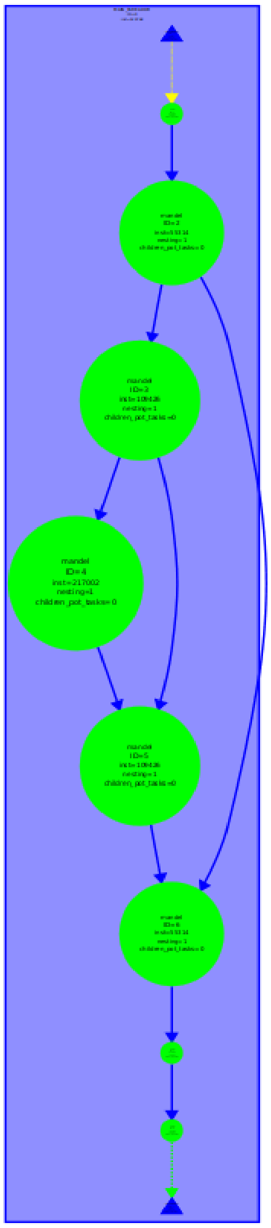
Which are the two most important characteristics for the task graph that is generated? Which part of the code is making a big difference with the previous case? How will you protect this section of code in the parallel OpenMP code?



Now the execution is fully sequential, the (little) parallelism that we had now is gone, every task needs to be executed one after the other. The big difference is found in the block inside the mandelbrot function that prints to the display, which uses two functions that modify the window’s pixels and is done sequentially. The display variable could be private to each task in order to reduce dependencies.

1.1.3 Third analysis with Tareador

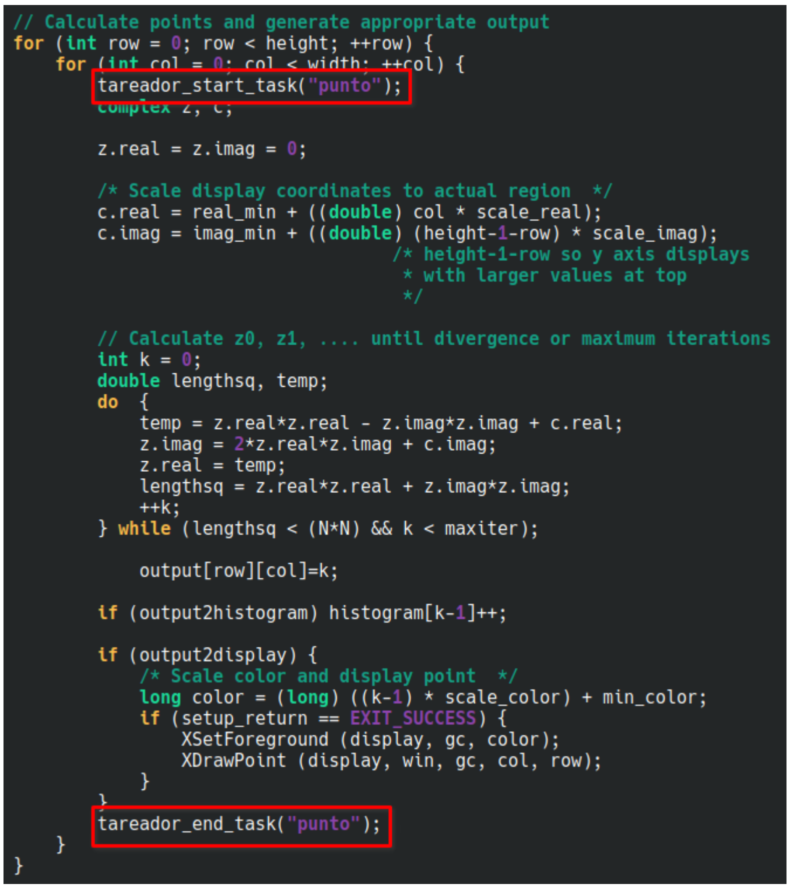
What does each chain of tasks in the task graph represent? Which part of the code is making the big difference with the two previous cases? How will you protect this section of code in the parallel OpenMP code?



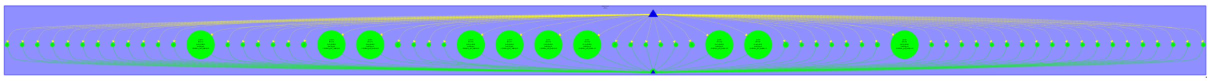
Each chain of tasks represents the dependency caused by modifying the same histogram array positions. The part that causes this TDG is the line in the mandelbrot function where the histogram array is modified. Since the points with the same size modify the same array positions, and some of them are processed in different tasks, dependencies are created. A local variable could be used instead of modifying the array, so that a reduction can be performed on it to prevent these dependency chains.

1.1.4 Point Strategy analysis with Tareador

Which is the main change that you observe with respect to the Row strategy?



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

        z.real = z.imag = 0;

        /* Scale display coordinates to actual region  */
        c.real = real_min + ((double) col * scale_real);
        c.imag = imag_min + ((double) (height-1-row) * scale_imag);
                                    /* height-1-row so y axis displays
                                     * with larger values at top
                                     */

        // Calculate z0, z1, .... until divergence or maximum iterations
        int k = 0;
        double lengthsq, temp;
        do  {
            temp = z.real*z.real - z.imag*z.imag + c.real;
            z.imag = 2*z.real*z.imag + c.imag;
            z.real = temp;
            lengthsq = z.real*z.real + z.imag*z.imag;
            ++k;
        } while (lengthsq < (N*N) && k < maxiter);

            output[row][col]=k;

        if (output2histogram) histogram[k-1]++;

        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);
            }
        }
        tareador_end_task("punto");
    }
}
```



With this strategy, many more tasks are created, and we can clearly see that some points require the same computation time. When not passing the "-d" nor "-h" parameters, the parallelism is increased since the points with the most cost can be calculated in parallel. In the "-d" case the tasks are executed sequentially since each one modifies the display variable (just like in the Row strategy part), while in the "-h" case the dependency chains between points of the same size are now clearly visible.

1.2 Point decomposition strategy

Is the speedup appropriate? Is the scalability appropriate? Which threads are creating and executing the explicit tasks? Is one of the threads devoted to create them, and therefore executing much less? If yes, this should be a clear contributor to load unbalance, right? Do you think the granularity of the tasks is appropriate for this parallelization strategy?

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.52	0.35	0.31	0.31	0.31
Speedup	1.00	1.48	1.68	1.69	1.68
Efficiency	1.00	0.37	0.21	0.14	0.10

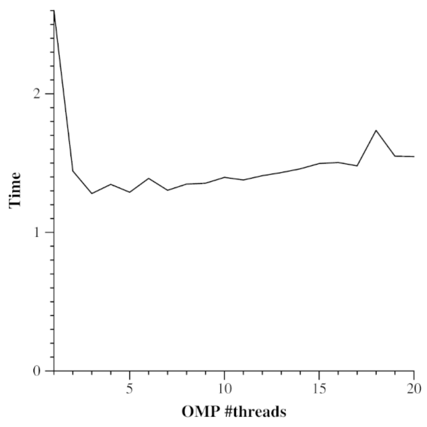
Table 1: Analysis done on Mon Nov 6 03:01:56 PM CET 2023, par4204

Overview of the Efficiency metrics in parallel fraction, ϕ=99.89%					
Number of processors	1	4	8	12	16
Global efficiency	94.93%	35.01%	19.89%	13.34%	9.95%
Parallelization strategy efficiency	94.93%	44.14%	26.65%	19.13%	14.69%
Load balancing	100.00%	82.08%	56.66%	39.22%	28.32%
In execution efficiency	94.93%	53.78%	47.03%	48.78%	51.87%
Scalability for computation tasks	100.00%	79.31%	74.63%	69.72%	67.74%
IPC scalability	100.00%	76.32%	73.68%	71.76%	70.85%
Instruction scalability	100.00%	105.35%	106.27%	106.06%	106.31%
Frequency scalability	100.00%	98.63%	95.30%	91.60%	89.94%

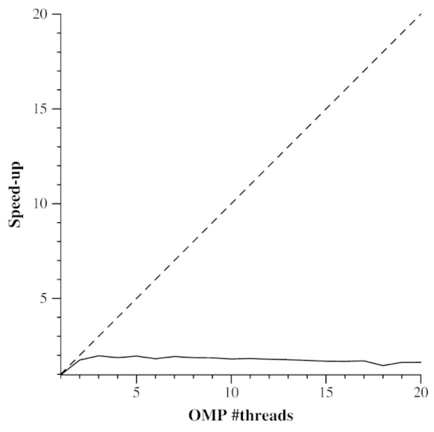
Table 2: Analysis done on Mon Nov 6 03:01:56 PM CET 2023, par4204

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	0.84	0.82	0.84	0.89
LB (time executing explicit tasks)	1.0	0.89	0.9	0.89	0.97
Time per explicit task (average us)	4.35	5.01	5.23	5.38	5.35
Overhead per explicit task (synch %)	0.0	121.91	309.0	513.95	742.91
Overhead per explicit task (sched %)	5.94	31.83	30.61	28.56	27.03
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

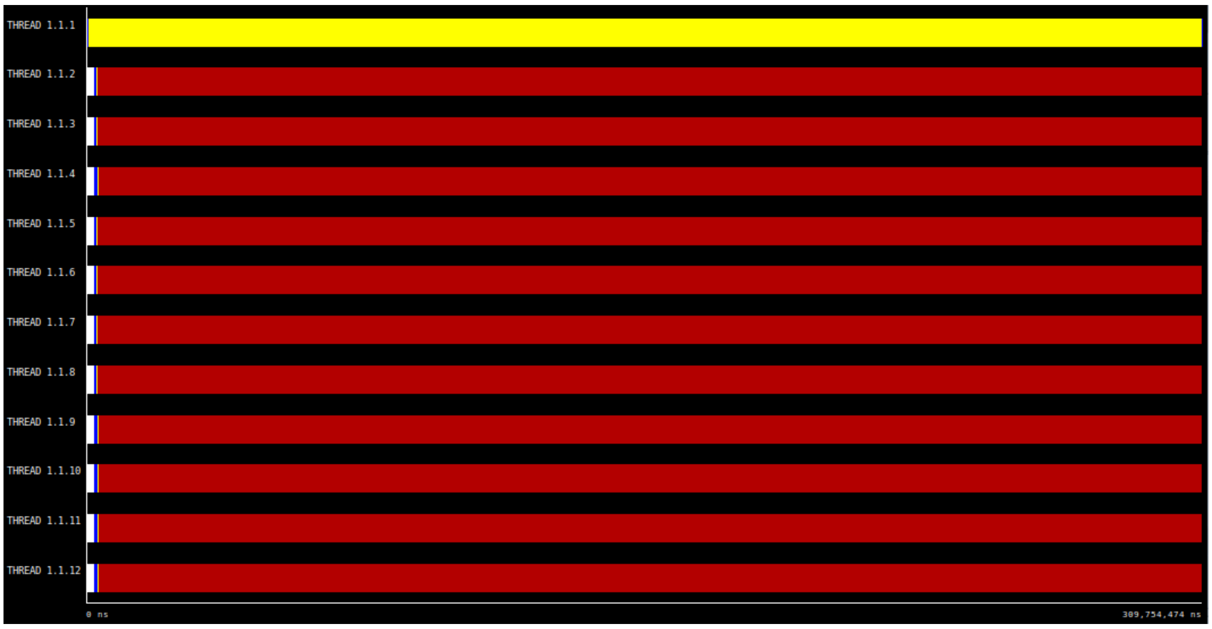
Table 3: Analysis done on Mon Nov 6 03:01:56 PM CET 2023, par4204



par4204
Min elapsed execution time
Generated by par4204 on Thu Oct 26 04:32:58 PM CEST 2023



par4204
Speed-up wrt sequential time
Generated by par4204 on Thu Oct 26 04:32:58 PM CEST 2023



It seems like the speedup is not appropriate since it stays more or less constant in executions with more threads, as well as the scalability, which only increases a little when the number of threads is duplicated. In the Paraver timelines we observe that clearly the task management is executed by only one thread. However, the granularity of this case is not the most appropriate since too many tasks are being generated and the load balance is far from great, causing the efficiency to drop significantly. This is caused by the synchronization overhead, which increases exponentially with the amount of threads used to execute tasks.

1.2.2 Taskloops without using neither num tasks nor grainsize

Do you think the granularity of the tasks is appropriate for this parallelization strategy? Is the version with taskloop performing better than the last version based on the use of task? Do you notice a relevant change in the two metrics that contribute to the parallelization strategy efficiency? Should you blame "load balancing" or "in execution efficiency"? Remember to scan the information provided in the third table that is included in the modelfactor-tables.pdf le. What can you tell now from the total number of tasks generated and the new average execution time for explicit tasks and the synchronization and scheduling overheads percentage? How many tasks are executed per taskloop? Is the task granularity better? But notice that task synchronization still takes a big %. Why are there now task synchronizations in the execution? Where are these task synchronisations happening?

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.42	0.14	0.13	0.14	0.18
Speedup	1.00	2.89	3.22	2.94	2.34
Efficiency	1.00	0.72	0.40	0.25	0.15

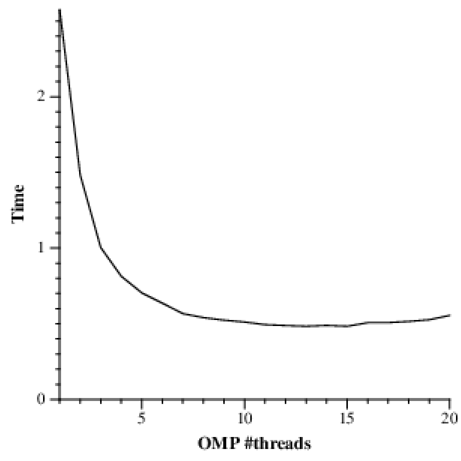
Table 1: Analysis done on Mon Nov 6 03:59:36 PM CET 2023, par4204

Overview of the Efficiency metrics in parallel fraction, ϕ=99.87%					
Number of processors	1	4	8	12	16
Global efficiency	99.63%	71.94%	40.16%	24.48%	14.55%
Parallelization strategy efficiency	99.63%	75.82%	45.64%	29.57%	18.18%
Load balancing	100.00%	96.15%	95.65%	95.34%	94.15%
In execution efficiency	99.63%	78.85%	47.71%	31.01%	19.31%
Scalability for computation tasks	100.00%	94.89%	87.98%	82.80%	80.02%
IPC scalability	100.00%	97.92%	97.53%	97.56%	97.06%
Instruction scalability	100.00%	99.25%	98.26%	97.29%	96.34%
Frequency scalability	100.00%	97.63%	91.81%	87.24%	85.58%

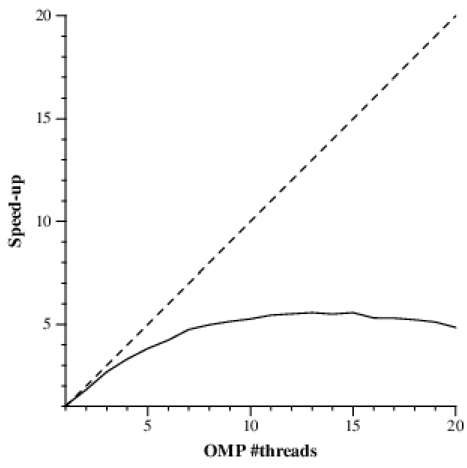
Table 2: Analysis done on Mon Nov 6 03:59:36 PM CET 2023, par4204

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	0.92	0.84	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)	129.24	33.88	18.14	12.68	9.69
Overhead per explicit task (synch %)	0.04	27.51	105.74	221.19	434.25
Overhead per explicit task (sched %)	0.33	4.13	13.39	19.8	29.36
Number of taskwait/taskgroup (total)	320.0	320.0	320.0	320.0	320.0

Table 3: Analysis done on Mon Nov 6 03:59:36 PM CET 2023, par4204



par4204
Min elapsed execution time
Generated by par4204 on Mon Nov 6 03:55:54 PM CET 2023



par4204
Speed-up wrt sequential time
Generated by par4204 on Mon Nov 6 03:55:54 PM CET 2023

This new version using the *taskloop* directive is performing better than the version using explicit tasks. The overall speedup is increased and also its in-execution efficiency, but the latter still drops significantly when increasing the number of threads: this is what causes the problem.
In this version the number of total tasks is significantly lower, and scales with the number of threads used in the execution thanks to the taskloop clause. However, it is because of the implicit *taskgroup* of this directive, which creates dependencies when it waits for the completion of each *taskloop*'s tasks and the tasks they create (if they did).

1.2.3 Taskloops without nogroup

Has the scalability improved? What about task granularity and overheads?

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.42	0.11	0.11	0.13	0.18
Speedup	1.00	3.61	3.93	3.30	2.37
Efficiency	1.00	0.90	0.49	0.28	0.15

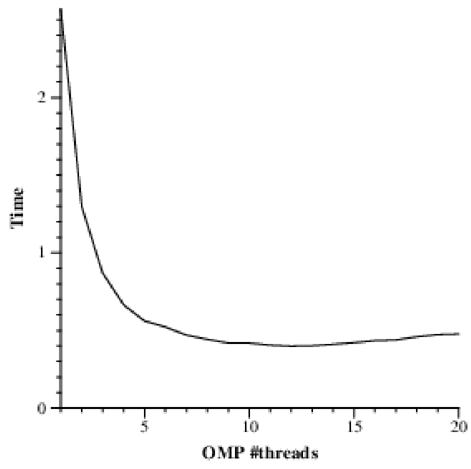
Table 1: Analysis done on Mon Nov 6 09:10:40 PM CET 2023, par4204

Overview of the Efficiency metrics in parallel fraction, ϕ=99.92%					
Number of processors	1	4	8	12	16
Global efficiency	99.68%	90.17%	49.20%	27.54%	14.78%
Parallelization strategy efficiency	99.68%	94.29%	55.85%	33.40%	18.54%
Load balancing	100.00%	98.19%	98.09%	96.26%	95.17%
In execution efficiency	99.68%	96.03%	56.93%	34.69%	19.48%
Scalability for computation tasks	100.00%	95.63%	88.10%	82.46%	79.73%
IPC scalability	100.00%	98.48%	97.52%	97.52%	96.83%
Instruction scalability	100.00%	99.25%	98.27%	97.29%	96.38%
Frequency scalability	100.00%	97.84%	91.94%	86.91%	85.43%

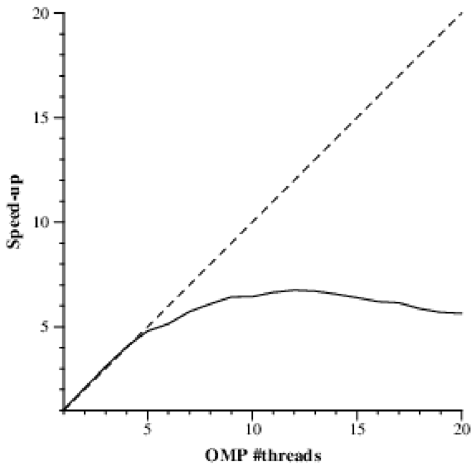
Table 2: Analysis done on Mon Nov 6 09:10:40 PM CET 2023, par4204

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	0.51	0.93	0.75	0.9
LB (time executing explicit tasks)	1.0	0.98	0.98	0.96	0.95
Time per explicit task (average us)	129.13	33.61	18.11	12.68	9.7
Overhead per explicit task (synch %)	0.0	3.42	67.34	182.88	422.75
Overhead per explicit task (sched %)	0.32	2.24	11.14	18.55	30.28
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Mon Nov 6 09:10:40 PM CET 2023, par4204



par4204
Min elapsed execution time
Generated by par4204 on Mon Nov 6 09:10:07 PM CET 2023



par4204
Speed-up wrt sequential time
Generated by par4204 on Mon Nov 6 09:10:07 PM CET 2023

This version using *nogroup* achieves better efficiency and speedup compared to the version without *nogroup*. The overheads due to scheduling and synchronization are reduced on executions with less threads available, but on the ones where more threads

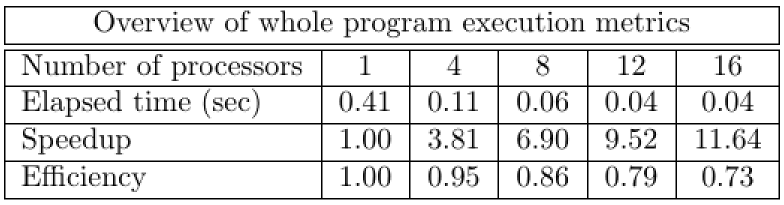
are used it remains high due to the atomic and critical parts. With this strategy we reduce the total number of taskwait/taskgroup to 0, as expected.
As for the granularity, we see that the total number of tasks per execution is the same as the previous version. All in all, the scalability is the same as the previous version of the program.

1.3 Row decomposition strategy

Which are the main differences that you observe when comparing the Row and Point strategies? What is the number of tasks created for Row and Point strategies? Is there any load unbalance? What is the task synchronization cost now?

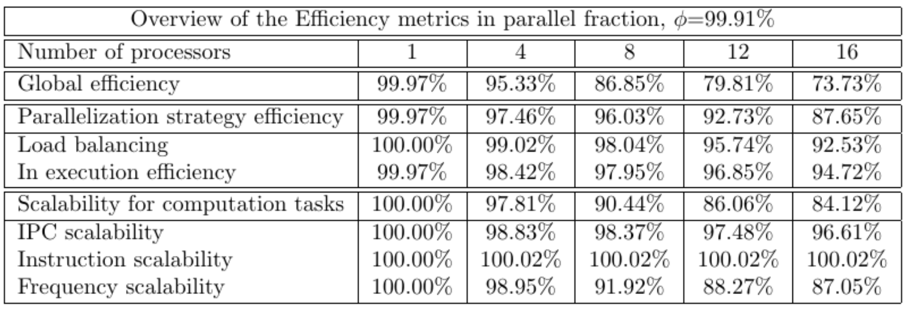
For this strategy, we will first analyze the use of explicits tasks with the *task* directive, and then through the *taskloop* directive to compare how it performs in both cases, just as we did with the point decomposition strategy.

- Using the *task* directive:



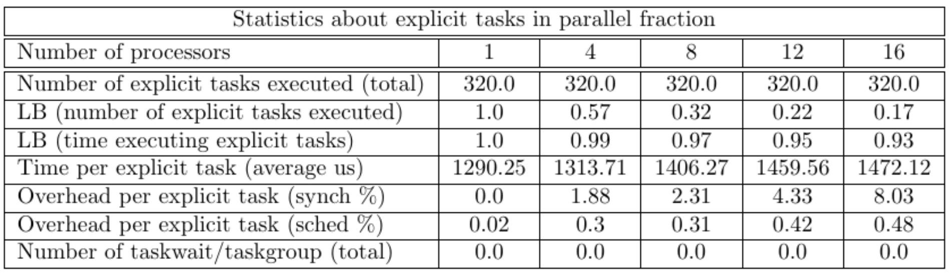
Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.41	0.11	0.06	0.04	0.04
Speedup	1.00	3.81	6.90	9.52	11.64
Efficiency	1.00	0.95	0.86	0.79	0.73

Table 1: Analysis done on Mon Nov 6 10:39:48 PM CET 2023, par4204



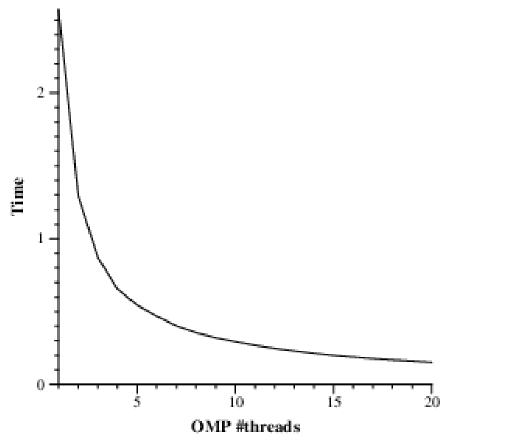
Overview of the Efficiency metrics in parallel fraction, ϕ=99.91%					
Number of processors	1	4	8	12	16
Global efficiency	99.97%	95.33%	86.85%	79.81%	73.73%
Parallelization strategy efficiency	99.97%	97.46%	96.03%	92.73%	87.65%
Load balancing	100.00%	99.02%	98.04%	95.74%	92.53%
In execution efficiency	99.97%	98.42%	97.95%	96.85%	94.72%
Scalability for computation tasks	100.00%	97.81%	90.44%	86.06%	84.12%
IPC scalability	100.00%	98.83%	98.37%	97.48%	96.61%
Instruction scalability	100.00%	100.02%	100.02%	100.02%	100.02%
Frequency scalability	100.00%	98.95%	91.92%	88.27%	87.05%

Table 2: Analysis done on Mon Nov 6 10:39:48 PM CET 2023, par4204

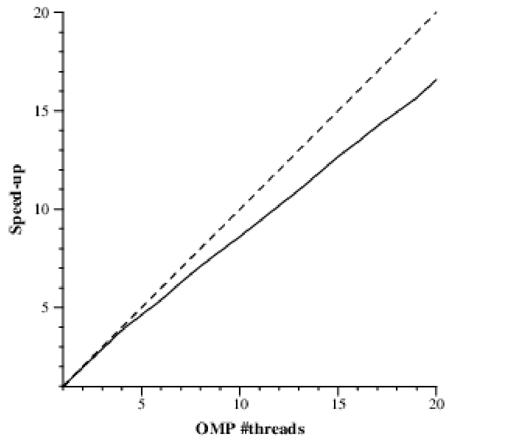


Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	320.0	320.0	320.0	320.0	320.0
LB (number of explicit tasks executed)	1.0	0.57	0.32	0.22	0.17
LB (time executing explicit tasks)	1.0	0.99	0.97	0.95	0.93
Time per explicit task (average us)	1290.25	1313.71	1406.27	1459.56	1472.12
Overhead per explicit task (synch %)	0.0	1.88	2.31	4.33	8.03
Overhead per explicit task (sched %)	0.02	0.3	0.31	0.42	0.48
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Mon Nov 6 10:39:48 PM CET 2023, par4204

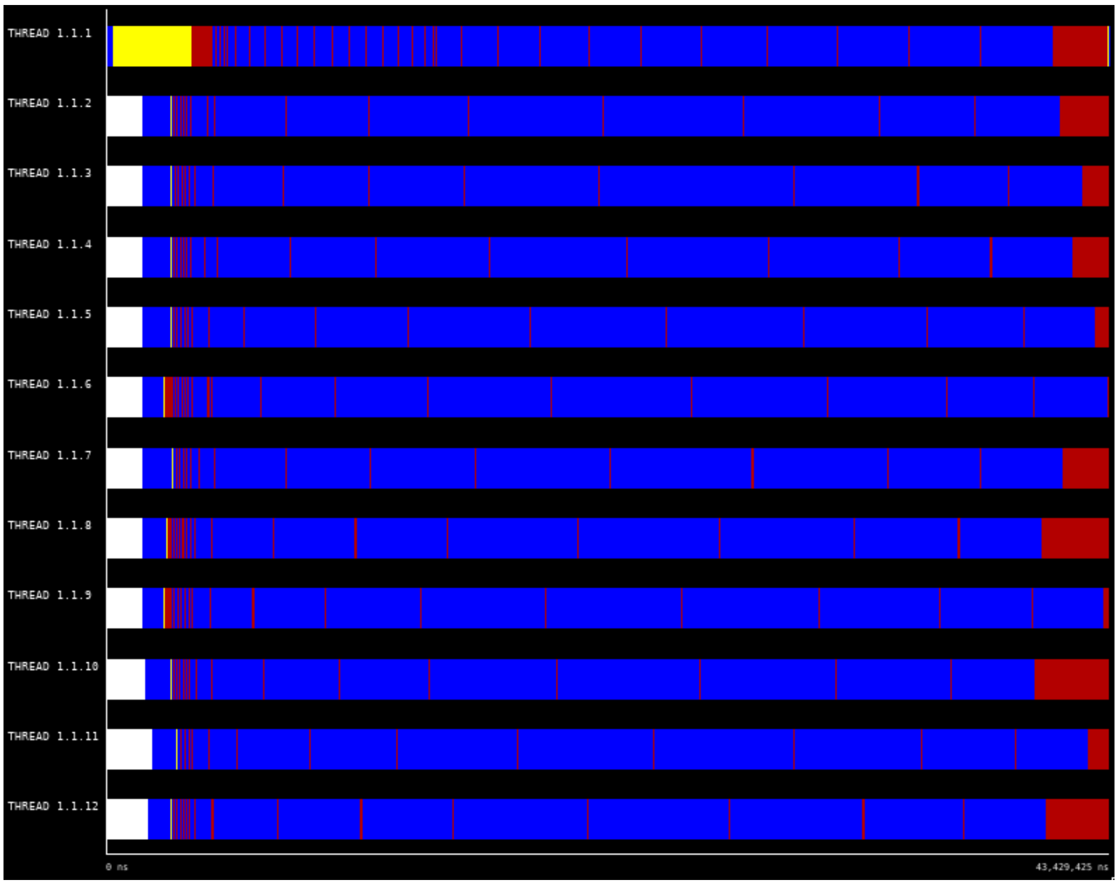


par4204
Min elapsed execution time
Generated by par4204 on Mon Nov 6 10:39:40 PM CET 202



par4204
Speed-up wrt sequential time
Generated by par4204 on Mon Nov 6 10:39:40 PM CET 202

A massive improvement in terms of scalability is achieved with this strategy: both the efficiency and speedup have been improved, the latter increasing almost linearly with the number of threads used. Regarding the overheads, they take much less time compared to the previous versions, since the code marked as *atomic* and *critical* is already sequentialized for each row, as we can see in this screenshot of the status timeline for the execution with 12 threads:



A further analysis also reveals that, since not many tasks are created with this strategy, the task creation and join overhead only amounts to a small fraction of the total execution time, as seen in the yellow fragment of the previous screenshot, on THREAD 1.1.1.

- Using the *taskloop* directive:

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.41	0.12	0.06	0.05	0.04
Speedup	1.00	3.49	6.55	8.90	11.23
Efficiency	1.00	0.87	0.82	0.74	0.70

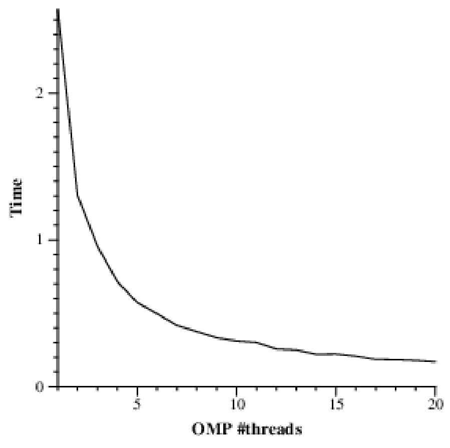
Table 1: Analysis done on Mon Nov 6 10:24:03 PM CET 2023, par4204

Overview of the Efficiency metrics in parallel fraction, ϕ=99.91%					
Number of processors	1	4	8	12	16
Global efficiency	99.99%	87.58%	82.48%	75.02%	70.73%
Parallelization strategy efficiency	99.99%	90.72%	91.74%	87.82%	85.14%
Load balancing	100.00%	91.31%	93.52%	91.58%	89.18%
In execution efficiency	99.99%	99.36%	98.10%	95.89%	95.46%
Scalability for computation tasks	100.00%	96.54%	89.90%	85.43%	83.08%
IPC scalability	100.00%	98.37%	97.85%	96.96%	96.03%
Instruction scalability	100.00%	100.00%	99.99%	99.99%	99.98%
Frequency scalability	100.00%	98.14%	91.89%	88.11%	86.53%

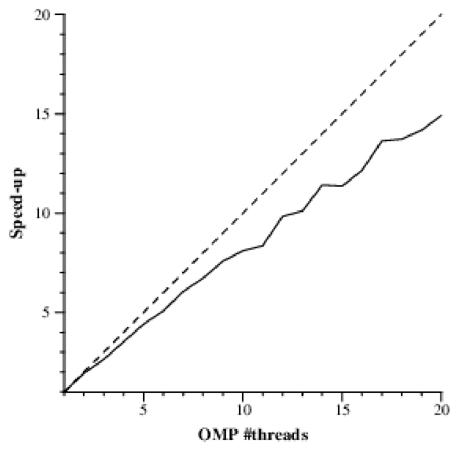
Table 2: Analysis done on Mon Nov 6 10:24:03 PM CET 2023, par4204

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	0.93	0.91	0.9
Time per explicit task (average us)	41159.21	10623.87	5653.54	3908.05	2961.09
Overhead per explicit task (synch %)	0.0	9.58	7.26	10.01	12.27
Overhead per explicit task (sched %)	0.01	0.08	0.21	0.29	0.4
Number of taskwait/taskgroup (total)	1.0	1.0	1.0	1.0	1.0

Table 3: Analysis done on Mon Nov 6 10:24:03 PM CET 2023, par4204

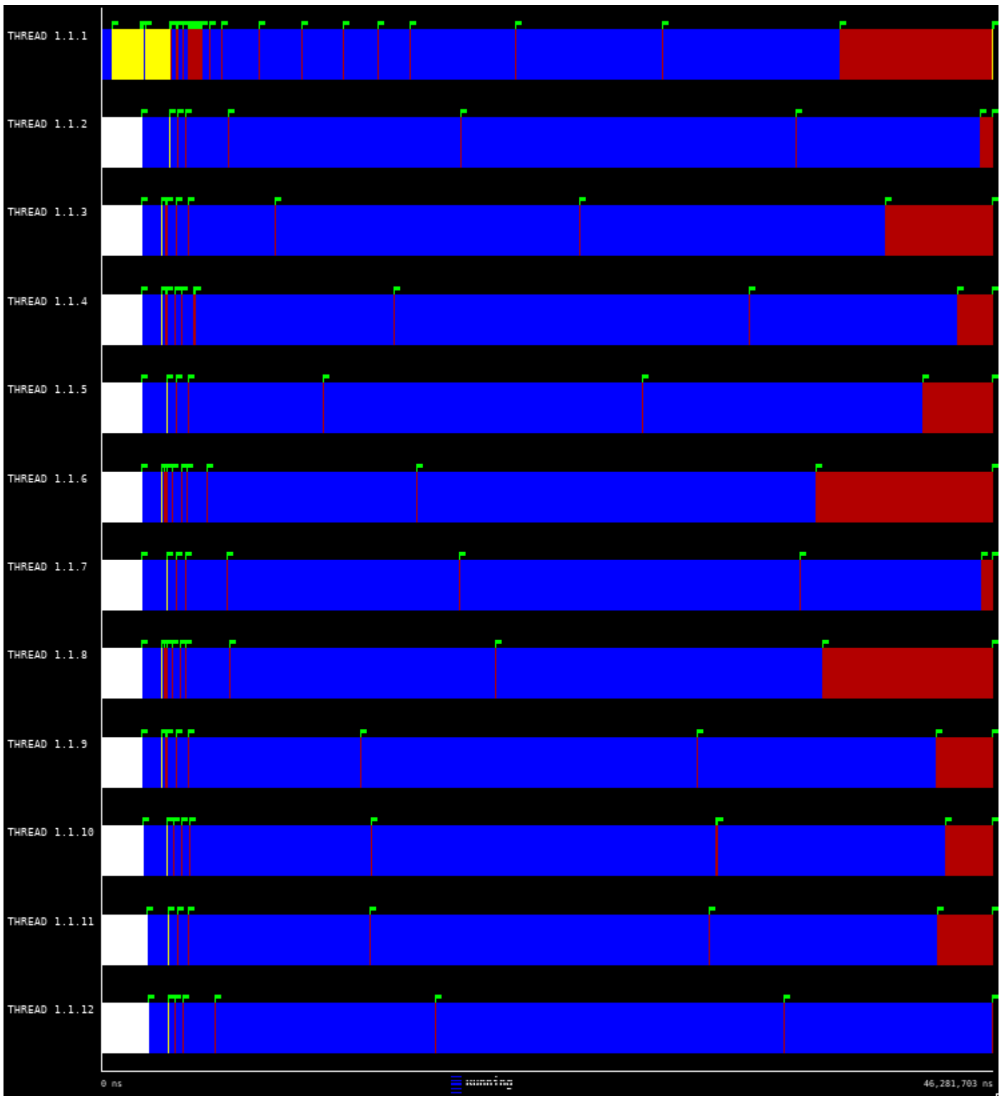


par4204
Min elapsed execution time
Generated by par4204 on Mon Nov 6 10:23:44 PM CET 2023



par4204
Speed-up wrt sequential time
Generated by par4204 on Mon Nov 6 10:23:44 PM CET 2023

Using *taskloop* only seems to worsen the performance of the parallel program due to the unnecessary synchronization that the implicit *taskgroup* clause introduces. Furthermore, since the number of tasks generated by the *taskloop* depends on the number of threads, the grain size of each task is less proportional to one another, and thus the load balance on threads is negatively affected. We can see this on a screenshot of the status timeline of the execution with 12 threads:



Overall, using *taskloop* is not the best strategy, but still performs much better than both iterations of the point decomposition strategy that we previously analyzed.