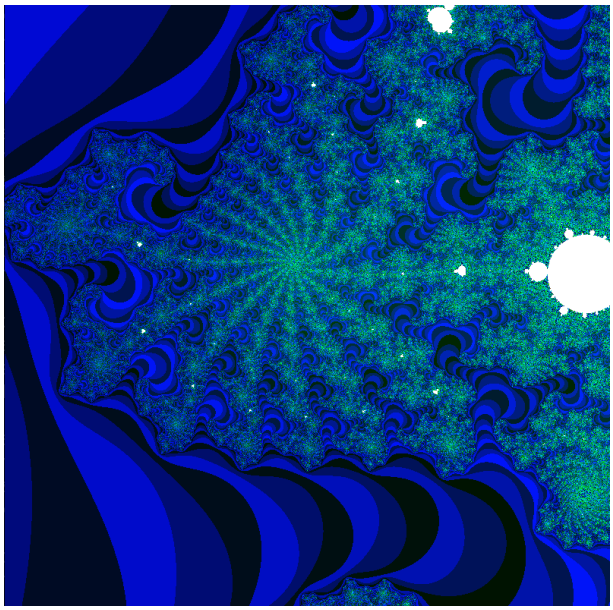
PAR Laboratory Assignment
Lab 3: Iterative task decomposition with OpenMP: the computation of the Mandelbrot set

Neus Mayol and Pol Verdura
group 11 - (par1117 and par1121)

Spring 2022-23



UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Index

Index **1**

1 Mandelbrot set computation **2**
1.1 Task decomposition analysis with *Tareador* 2
1.1.1 *Row* strategy 2
1.1.2 *Point* strategy 4
1.1.3 Comparision between *Row* and *Point* 5
1.2 Implementation and analysis of task decompositions in *OpenMP* 6
1.2.1 Implementation using `task` 6
1.2.2 Overall analysis with `Modelfactors` 7
1.2.3 Detailed analysis with *Paraver* 8
1.2.4 Optimization: granularity control using `taskloop` 9
1.3 Row strategy analysis 12

1

Mandelbrot set computation

1.1 Task decomposition analysis with *Tareador*

1.1.1 *Row* strategy

First parallellization strategy with *Tareador* consists on creating a new task for every executed row. As shown at the following piece of code, we added two *Tareador* commands.

```
void mandelbrot(int height, int width, double real_min, double imag_min,
    double scale_real, double scale_imag, int maxiter, int **output) {
    for (int row = 0; row < height; ++row) {
        tareador_start_task("Row");
        for (int col = 0; col < width; ++col) {
            ...
        }
        tareador_end_task("Row");
    }
}
```

Changes in code cause a different task distribution in the Task Decomposition Graph, where eight tasks are created (one for each row). Figure 1.1 shows TDG.

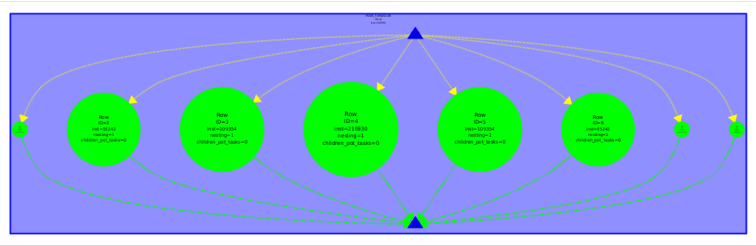


Figure 1.1: Mandelbrot TDG with *Row* Strategy

Next, following the assignement instructions, we repeated the execution adding `-d` flag at the initial command. The result, as we can see in Figure 1.2, continues dividing the execution into eight tasks (that correspond to the row dividing strategy that we programmed) but now, Figure 1.2 shows us that tasks have a sequential behaviour.

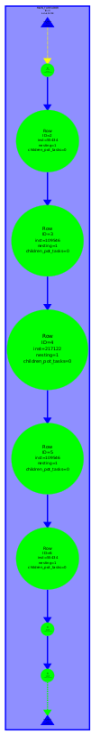


Figure 1.2: Mandelbrot TDG with *Row* Strategy executed using `-d` flag.

To know better what is the effect of `-d` flag, we executed the command `./mandel-tar -help`, that displays a simple explanation of all the binary possible flags. We found out that adding `-d` to our path makes our program display the final result in an image. The explanation itself does not give us any explicit clues about why do tasks run now sequentially but, checking `mandel-tar.c` we found a suspicious command which could be the cause of a different TDG. In the code, there's a boolean `output2display` that seems to be set to true when `-d` flag is activated. `output2display` makes the program accomplish a new condition, that will make the execution wait until the last task finishes successfully. The reason behind the wait could be to prevent problems in the image representation, which can also be repaired using an OpenMP clause such as `#pragma omp critical`.

Last, but not least, `-h` flag displays a different TDG, task dependencies change, as shown at Figure 1.3.

Figure 1.3: Mandelbrot TDG with *Row* Strategy executed using `-h` flag.

Based on `./mandel-tar -help` explanation, `-h` flag displays the histogram of values in computed image. That brings us to read the code again, where we find out that there is another boolean `output2histogram` that is only executed when `-h` flag is activated, provoking some accesses in the vector that are used by other tasks. Since not all task accesses overlap, TDG is not completely sequential, but it has some important dependencies. The problem can be treated using the clause `#pragma omp atomic` inside the if body, since we only need a fast way to protect a single line of the code from being accessed by different threads at the same time.

1.1.2 *Point* strategy

Now we reduce the task granularity by creating a task for each point of the matrix, we can do it moving *Tareador* statements inside the inner-loop, as we can see in the following piece of code:

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


    for (int row = 0; row < height; ++row) {
        for (int col = 0; col < width; ++col) {
            tareador_start_task("Point");
            ...
            tareador_end_task("Point");
        }
    }
}
```

If we generate the TDG of the last code (Figure 1.4), we can see that we have as many tasks as points in the matrix (8 col $\cdot$ 8 row = 64 points = 64 tasks), and not all tasks execute the same number of instructions.

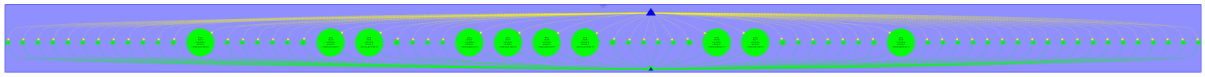


Figure 1.4: Mandelbrot TDG with *Point* Strategy

Using the `-d` flag, we obtain Figure 1.5. As we can see, the number of tasks and balance is still the same, but now, tasks are distributed in serial, same as in Row Strategy. Because we are using the same code as the Row Strategy, we must execute the task sequentially for the same reason.

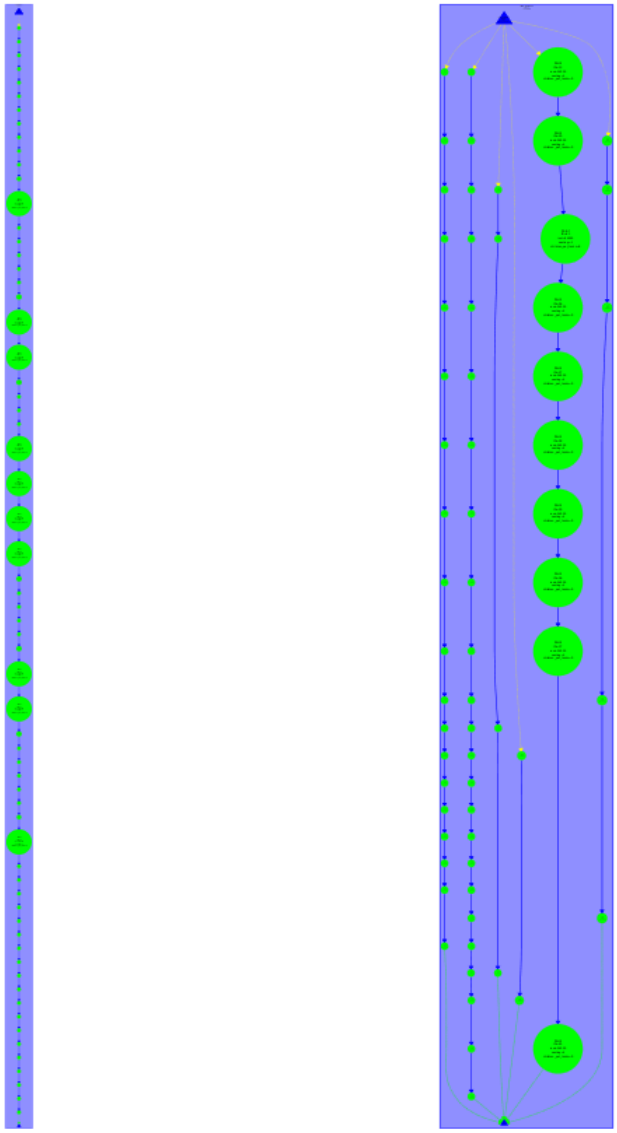


Figure 1.5: Mandelbrot TDG with `-d` flag (left) and `-h` flag (right) and *Point* Strategy

1.1.3 Comparision between *Row* and *Point*

The main difference between TDGs is that row decomposition creates one task every time a new row is executed, that makes the program create eight tasks. On the other hand, point strategy creates a task for each internal loop iteration, creating a total amount of 64 tasks. Moreover, we can see that task distribution is not balanced, that's because the tasks that have more white area, will have to execute

more instructions, as we can see in Figure 1.6.

The explanation above and Figure 1.6 helped us finding the reason behind the size of the tasks in the previous TDGs. For example: Figure 1.1 graph shows a big size difference between first and last rows and the other ones. This is due to the amount of computed iterations and consequently: the amount of color points in Figure 1.6.

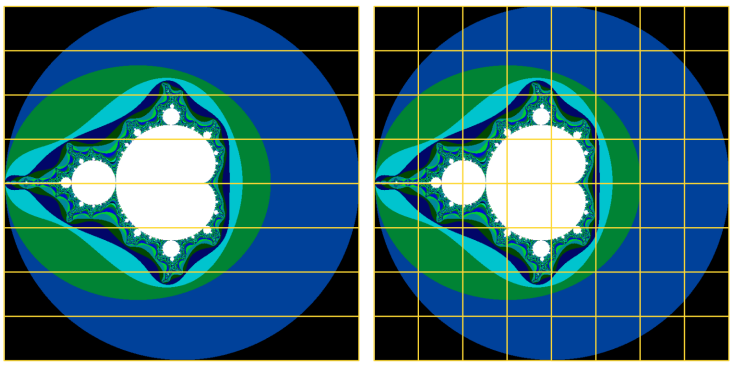


Figure 1.6: Mandelbrot set task distribution Row Strategy vs Point Strategy

1.2 Implementation and analysis of task decompositions in *OpenMP*

1.2.1 Implementation using `task`

Adding the clauses commented on the last section, we obtain the following information:

First, we ran the modified version using the commands `OMP NUM THREADS=1 ./mandel-omp -d -h -i 10000` and `OMP NUM THREADS=2 ./mandel-omp -d -h -i 10000`. Both commands give us the same result, which indicates us that the modifications are correct, and that now we can run the program using more than one thread. The results were checked using the histograms produced by the script.

Then, following the assessment statement, we submitted `submit-omp.sh` script indicating the `mandel-omp` binary with 1 and 8 threads, and we compared them with the original sequential program: which it didn't give us any difference too.

Next, we submitted `submit-strong-omp.sh` to compute the execution time and the speedup from one to twenty processors. The script results are shown in Figure 1.7.

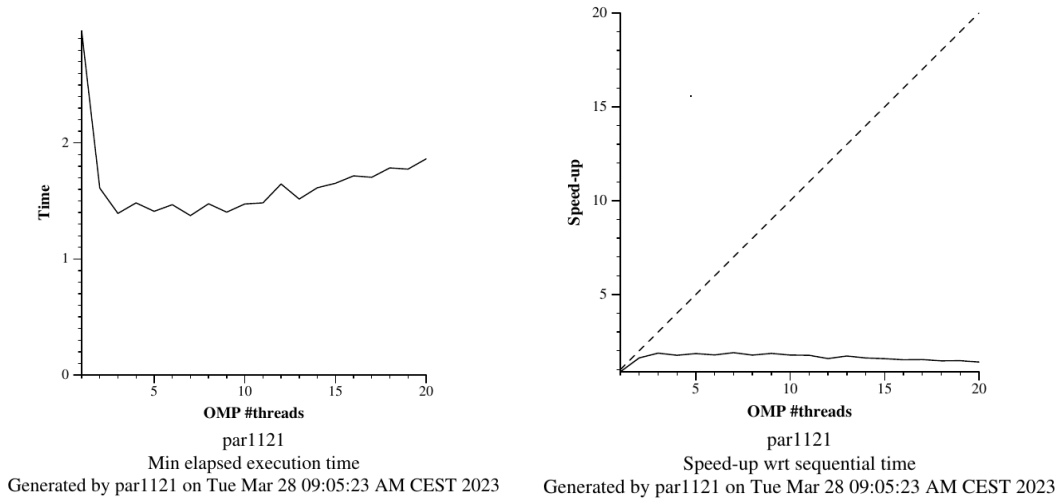


Figure 1.7: Reduction time and speedup using 1 to 20 threads in *Point* Strategy

As we can see in Figure 1.7 when we start adding more than 3 threads in our *Point* Strategy program, the time execution starts increasing causing a bad speed-up. This might occur because the processors are spending more time synchronizing and creating tasks than running code, as shown in Figure 1.8, where the red segments are synchronization time.

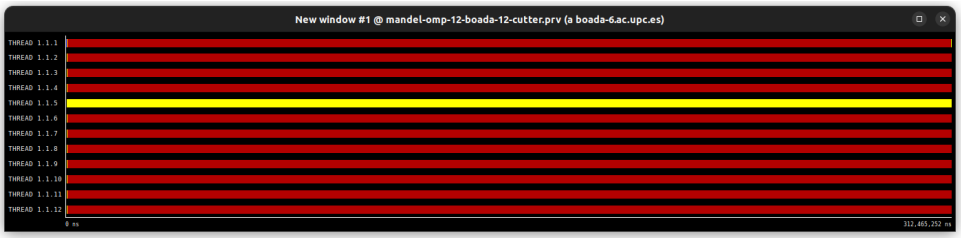


Figure 1.8: *Point* Strategy using 12 threads

1.2.2 Overall analysis with `Modelfactors`

Next point consists on analyzing the results of submitting `submit-strong-extrae.sh` script, that produces Table 1.1, Table 1.2 and Table 1.3.

As shown in Table 1.1, we can see that the speed-up is not proportional to the number of processors, if we take a look at the "Efficiency" row, we see that the efficiency is never 1 if we use more than one processor, this is why our program does not scale correctly. If it had appropriate scalability, using 16 processors would create a speed-up of 16 instead of 1.63 ($\simeq 16 \cdot 0.1$).

Overview of whole program execution metrics					
Number of processors	**1**	**4**	**8**	**12**	**16**
Elapsed time (sec)	0.58	0.36	0.31	0.33	0.36
Speedup	1.00	1.62	1.85	1.73	1.63
Efficiency	1.00	0.40	0.23	0.14	0.10

Table 1.1: Analysis done on Tue Mar 28 09:33:12 AM CEST 2023, par1117

Table 1.2 displays data about the efficiency, where we can appreciate how efficiency and load balancing decrease as we increase the number of processors. This means that a bad load balancing occurs and, as a consequence, global efficiency decreases.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.91%					
Number of processors	**1**	**4**	**8**	**12**	**16**
Global efficiency	95.31%	38.56%	22.06%	13.75%	9.68%
Parallelization strategy efficiency	95.31%	46.47%	29.70%	20.20%	14.76%
Load balancing	100.00%	91.49%	54.45%	35.01%	24.19%
In execution efficiency	95.31%	50.79%	54.54%	57.70%	61.01%
Scalability for computation tasks	100.00%	82.99%	74.28%	68.08%	65.60%
IPC scalability	100.00%	80.41%	74.38%	70.30%	67.78%
Instruction scalability	100.00%	103.92%	104.33%	104.28%	104.15%
Frequency scalability	100.00%	99.32%	95.71%	92.87%	92.93%

Table 1.2: Analysis done on Tue Mar 28 09:33:12 AM CEST 2023, par1117

As we see in Table 1.3, the load balance of the number and time of explicit tasks is smaller if we have more than one processor. On the other hand, we see that the most relevant overhead is synchronization,

which increases proportionally to the number of processors, which spends more time synchronizing data than executing the task itself.

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	0.83	0.84	0.87
LB (time executing explicit tasks)	1.0	0.89	0.89	0.89	0.9
Time per explicit task (average us)	4.9	5.65	5.88	6.08	6.07
Overhead per explicit task (synch %)	0.0	101.68	267.63	490.0	756.71
Overhead per explicit task (sched %)	5.4	30.34	23.54	22.65	22.28
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 1.3: Analysis done on Tue Mar 28 09:33:12 AM CEST 2023, par1117

1.2.3 Detailed analysis with *Paraver*

As a way to analyze deeply how threads distribute the program work, we visualize the data provided by *Paraver*. The resulting data, shown in Figure 1.9, shows us that there's one thread whose main role consists on task creation (in Figure 1.9 it is thread 1.1.5). Consequently, thread 1.1.5 is now executing much less of the program.

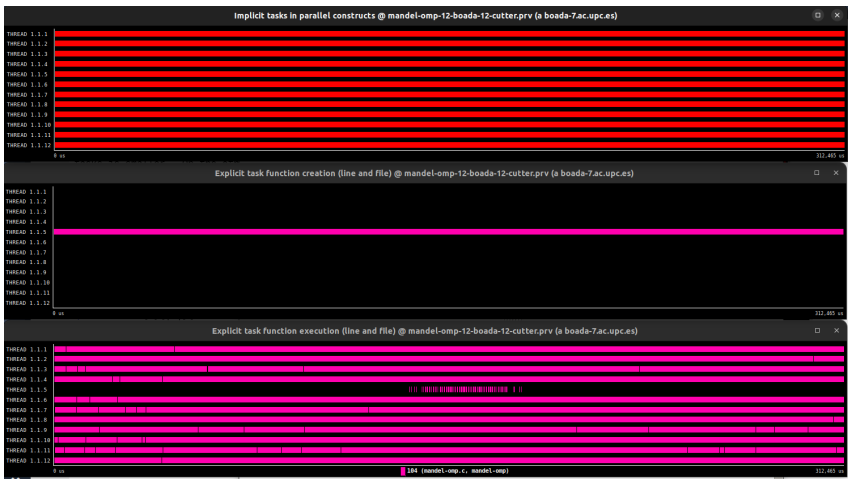
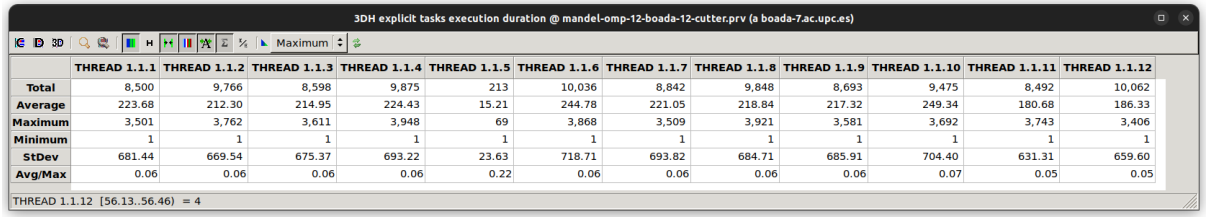


Figure 1.9: Program statistics using 12 threads. From top to bottom: Implicit task distribution, explicit task creation and explicit task execution.

Another way of visualizing data is through the histogram shown in Figure 1.10, where we can appreciate how Thread 1.1.5 does not contribute to the execution of explicit tasks as much as the other threads seeing the total row.



	THREAD 1.1.1	THREAD 1.1.2	THREAD 1.1.3	THREAD 1.1.4	THREAD 1.1.5	THREAD 1.1.6	THREAD 1.1.7	THREAD 1.1.8	THREAD 1.1.9	THREAD 1.1.10	THREAD 1.1.11	THREAD 1.1.12
Total	8,500	9,766	8,598	9,875	213	10,036	8,842	9,848	8,693	9,475	8,492	10,062
Average	223.68	212.30	214.95	224.43	15.21	244.78	221.05	218.84	217.32	249.34	180.68	186.33
Maximum	3,501	3,762	3,611	3,948	69	3,868	3,509	3,921	3,581	3,692	3,743	3,406
Minimum	1	1	1	1	1	1	1	1	1	1	1	1
StDev	681.44	669.54	675.37	693.22	23.63	718.71	693.82	684.71	685.91	704.40	631.31	659.60
Avg/Max	0.06	0.06	0.06	0.06	0.22	0.06	0.06	0.06	0.06	0.07	0.05	0.05

Figure 1.10: Distribution of explicit tasks between threads

Moreover, aside from Thread 1.1.5 executing fewer tasks than the other threads, in Figure 1.11 we can appreciate that the longest tasks (in blue) are distributed around all threads except Thread 1.1.5, which contributes more to a lower load balance. Additionally, we can appreciate how grey

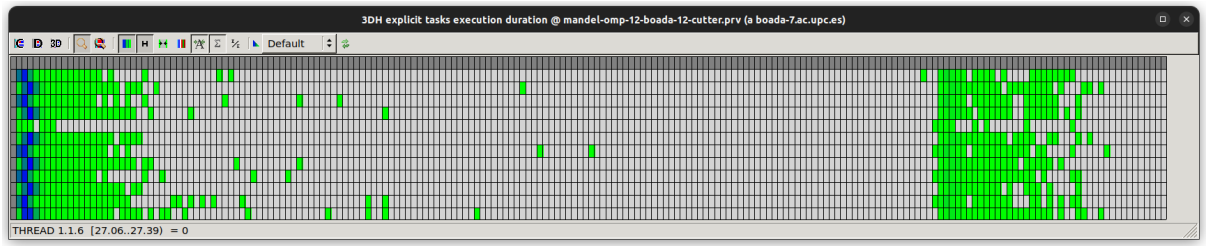


Figure 1.11: Explicit task distribution (columns) between 12 threads (rows)

From what we have seen, having a small granularity, in this case, adds an enormous penalty creating a bad efficiency, telling us that maybe we should increment it.

1.2.4 Optimization: granularity control using `taskloop`

As a way to increase granularity, we wil now add a `#pragma omp taskloop` at the loop. The change, as we can see in Figure 1.12, causes a better performance than the program above (Figure 1.7).

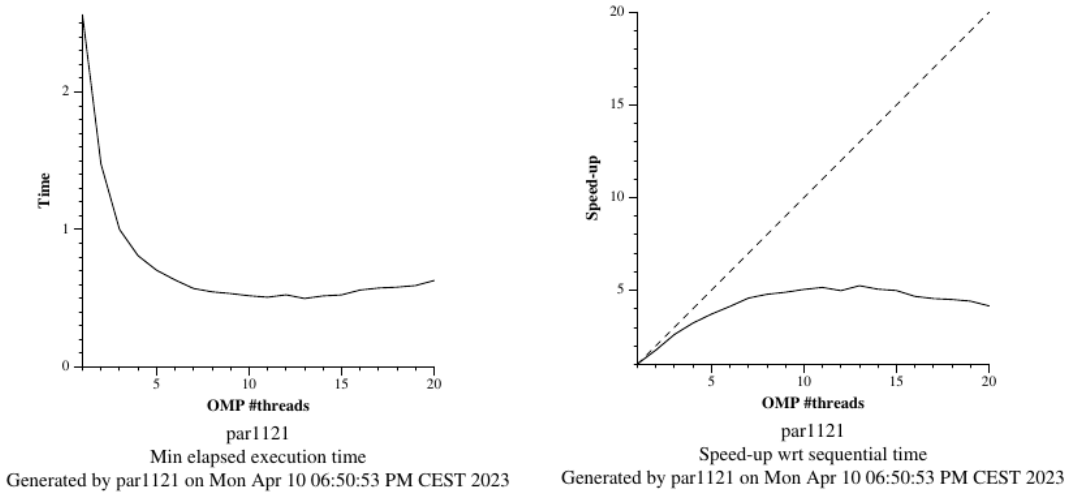


Figure 1.12: Reduction time and speedup using 1 to 20 threads in *Point* Strategy with `taskloop`

However, speedup is not as ideal as we wanted to, since it starts decreasing once the processor number is greater than 8, as shown in Table 1.4.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.41	0.14	0.13	0.14	0.20
Speedup	1.00	2.90	3.16	2.87	2.06
Efficiency	1.00	0.73	0.40	0.24	0.13

Table 1.4: Analysis done on Mon Apr 10 08:12:32 PM CEST 2023, par1121

Table 1.5 shows that now load balancing has improved significantly. However, global efficiency decreases as we increase number of processors. So now load unbalance does not cause inefficiencies.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.87%					
Number of processors	1	4	8	12	16
Global efficiency	99.54%	72.24%	39.38%	23.84%	12.79%
Parallelization strategy efficiency	99.54%	76.23%	44.77%	28.21%	15.36%
Load balancing	100.00%	95.56%	96.08%	93.81%	95.53%
In execution efficiency	99.54%	79.77%	46.59%	30.07%	16.07%
Scalability for computation tasks	100.00%	94.76%	87.96%	84.51%	83.31%
IPC scalability	100.00%	97.57%	97.18%	96.54%	96.30%
Instruction scalability	100.00%	99.42%	98.64%	97.89%	97.13%
Frequency scalability	100.00%	97.70%	91.76%	89.42%	89.06%

Table 1.5: Analysis done on Mon Apr 10 08:12:32 PM CEST 2023, par1121

Finally, Table 1.6 shows how there exists a synchronisation overhead that occurs when we need to wait for all the tasks in a taskgroup to finish before starting a new taskgroup. Comparing the results with the ones obtained in Table 1.3, we decreased the number of explicit tasks, which consequently increased the time for explicit task. Granularity increased causing an improvement.

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	0.95	0.84	0.53	0.48
LB (time executing explicit tasks)	1.0	0.96	0.96	0.94	0.95
Time per explicit task (average us)	128.85	33.99	18.31	12.7	9.66
Overhead per explicit task (synch %)	0.07	27.26	109.56	233.96	516.91
Overhead per explicit task (sched %)	0.39	3.93	13.87	20.74	34.69
Number of taskwait/taskgroup (total)	320.0	320.0	320.0	320.0	320.0

Table 1.6: Analysis done on Mon Apr 10 08:12:32 PM CEST 2023, par1121

`nogroup` clause addition

When we tried to add the `nogroup` clause it turned out that the program resulted in a `core dumped` message, which we've been able to avoid using another taskloop in the external loop. We decided to proceed adding the `nogroup` clause because we think taskgroup barriers are not necessary: we are waiting for all the threads to finish, and then we create another taskgroup, which is not necessary since all loop iterations are totally parallel.

Results are in Table 1.7, Table 1.8, Table 1.9.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.12	0.07	0.05	0.04
Speedup	1.00	3.69	6.43	8.66	10.61
Efficiency	1.00	0.92	0.80	0.72	0.66

Table 1.7: Analysis done on Mon Apr 10 08:48:53 PM CEST 2023, par1117

Overview of the Efficiency metrics in parallel fraction, ϕ=99.87%					
Number of processors	1	4	8	12	16
Global efficiency	99.70%	92.10%	80.30%	72.30%	66.52%
Parallelization strategy efficiency	99.70%	96.60%	93.35%	89.71%	86.00%
Load balancing	100.00%	99.34%	98.42%	97.50%	96.85%
In execution efficiency	99.70%	97.25%	94.85%	92.02%	88.80%
Scalability for computation tasks	100.00%	95.34%	86.02%	80.59%	77.35%
IPC scalability	100.00%	97.71%	95.45%	92.36%	89.91%
Instruction scalability	100.00%	99.46%	98.73%	98.01%	97.32%
Frequency scalability	100.00%	98.10%	91.28%	89.02%	88.40%

Table 1.8: Analysis done on Mon Apr 10 08:48:53 PM CEST 2023, par1117

Since we we considered that is unnecessary to put barriers, the program spends less time synchronising tasks. This is reflected on the overhead data in Table 1.9, which is smaller than data in Table 1.5.

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	3210.0	12840.0	25680.0	38520.0	51360.0
LB (number of explicit tasks executed)	1.0	0.57	0.32	0.23	0.19
LB (time executing explicit tasks)	1.0	0.99	0.99	0.98	0.97
Time per explicit task (average us)	142.63	37.65	20.99	15.08	11.91
Overhead per explicit task (synch %)	0.0	2.36	4.79	7.04	10.06
Overhead per explicit task (sched %)	0.3	1.13	2.23	4.16	5.66
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 1.9: Analysis done on Mon Apr 10 08:48:53 PM CEST 2023, par1117

Or through the data displayed in Figure 1.13, where we can see an increment in the speedup compared with the same graph at Figure 1.12

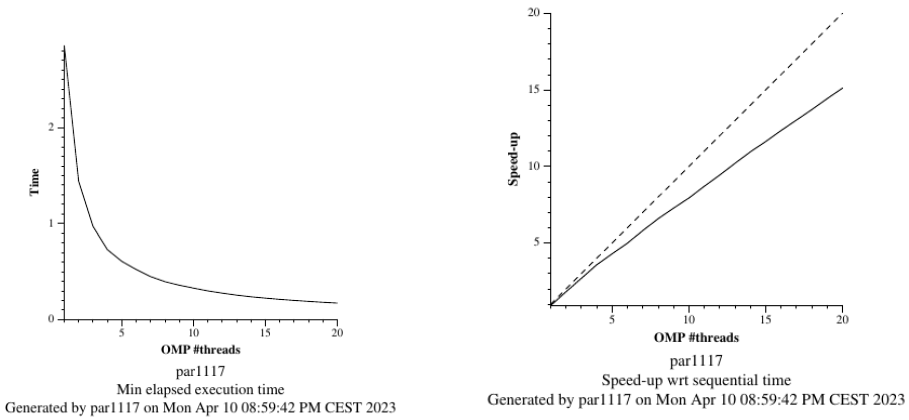


Figure 1.13: Time and Speedup plots using *Point* strategy and `taskloop` with `nogroup` clause

Granularity is almost the same, since we continue assigning a task for each loop iteration. Scalability improves.

1.3 Row strategy analysis

With the following figures, we can see how *Row* Strategy is by far more salable and appropriate than *Point* Strategy Figure 1.7 if we use the same clauses. On the other side, we see on Table 1.10 and Figure 1.14 how speedup doesn't decrement when we increase the number of threads.

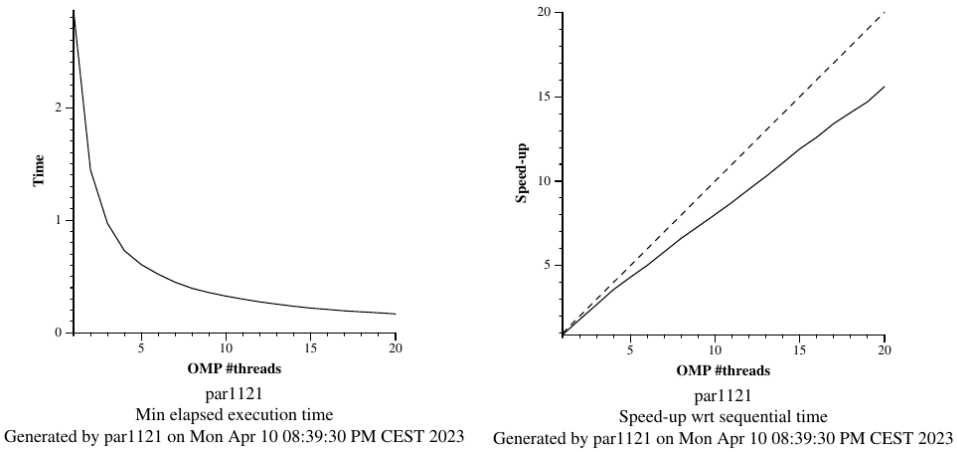


Figure 1.14: Reduction time and speedup using 1 to 20 threads in *Row* Strategy

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.12	0.07	0.05	0.04
Speedup	1.00	3.83	7.07	10.04	12.81
Efficiency	1.00	0.96	0.88	0.84	0.80

Table 1.10: Analysis done on Mon Apr 10 12:09:45 AM CEST 2023, par1121

The most remarkable thing of Table 1.11 is that Load Balance is very similar if we use a different number of processors in comparison of Table 1.2 where it decrements quickly.

Overview of the Efficiency metrics in parallel fraction, ϕ=99.89%					
Number of processors	1	4	8	12	16
Global efficiency	99.97%	95.95%	88.54%	84.01%	80.57%
Parallelization strategy efficiency	99.97%	98.85%	97.29%	95.62%	92.56%
Load balancing	100.00%	99.10%	97.46%	95.89%	92.84%
In execution efficiency	99.97%	99.76%	99.82%	99.72%	99.70%
Scalability for computation tasks	100.00%	97.06%	91.00%	87.87%	87.05%
IPC scalability	100.00%	98.86%	97.82%	96.76%	95.91%
Instruction scalability	100.00%	100.02%	100.02%	100.02%	100.01%
Frequency scalability	100.00%	98.16%	93.01%	90.80%	90.75%

Table 1.11: Analysis done on Mon Apr 10 12:09:45 AM CEST 2023, par1121

If we take a look at Table 1.12, we see that the time of each explicit task is increased in respect of *Point* Strategy, but at the same time, the number of explicit tasks is decreased, meaning that we won't have to waste that much time with the extra overhead (where using 16 CPU's we spend only 7.89% per explicit task synchronizing, whereas using *Point* Strategy we spend 756.71% per explicit task, Table

1.2). On the other hand, we see that the Load Balance of tasks is shared between processors when we increase them.

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	0.96	0.93
Time per explicit task (average us)	1438.47	1482.03	1580.62	1636.86	1652.02
Overhead per explicit task (synch %)	0.0	1.01	2.65	4.44	7.89
Overhead per explicit task (sched %)	0.02	0.14	0.13	0.13	0.12
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 1.12: Analysis done on Mon Apr 10 12:09:45 AM CEST 2023, par1121

Finally, seeing Figure 1.15, we can appreciate the execution timeline of the different strategies. The most impressive thought might be how using the same strategy if we use different clauses the execution time is halved. On the other hand, we can see that in this case, using a bigger granularity, is more beneficial because the processors spend more time executing the instructions of our program (blue) than synchronizing (red) or creating tasks (yellow).

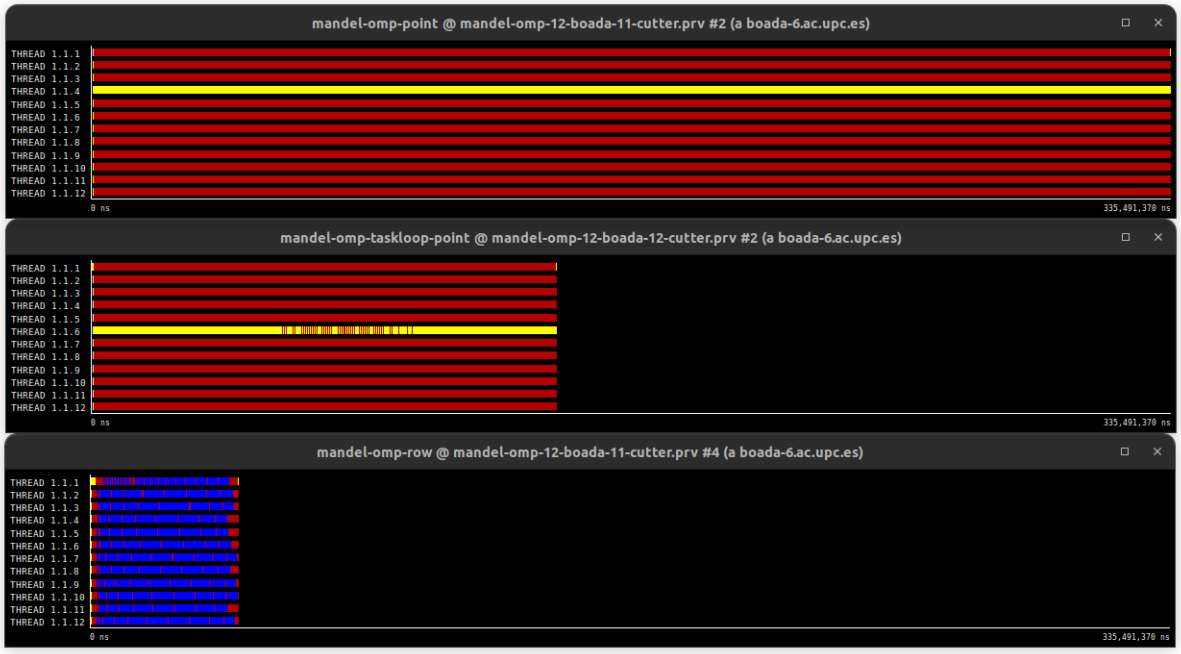


Figure 1.15: Execution timeline using a *Point* Strategy without (top) and with (middle) `taskloop`, and using a *Row* Strategy (bottom)