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

Walter J. Troiani Vargas (2318)

April 9, 2023

1 Task Decomposition With Tareador

1.1 Introduction

In this first section we are going to analyze deeply, for both strategies (Row and Point) using the Tareador interface, a series of parameters, such as the number of dependencies, the potential parallelism and the performance overall.

The two strategies mentioned above are:

- Row Strategy: each task corresponds to the computation of one or even more rows of the Mandelbrot.
- Point Strategy: each task corresponds to the computation of one or even more points of the Mandelbrot set.

Next section follows up with the discussion of the parallelism, studying each task dependency graphs for both strategies with different options of the code (no options, display option, histogram option...). Both source codes can be found at the attached compressed folder (mandel-tar-row and mandel-tar-point).

Figure 1: Point Strategy

Figure 2: Row Strategy

1.2 First Analysis

Judging by the dependency graph given by the Tareador interface, we can safely say for both strategies, which offer similar graphs, that this code is "embarrassingly parallel" due to no dependencies such as data races or other. However, the workload for each task is not evenly distributed, some tasks have to execute a lot of instructions compared to others and vice versa. Also, the point strategy has a lot more tasks, logically, due to that each task is related to each iteration of the innermost loop, so maybe in next chapter we could observe more overheads for task creation.

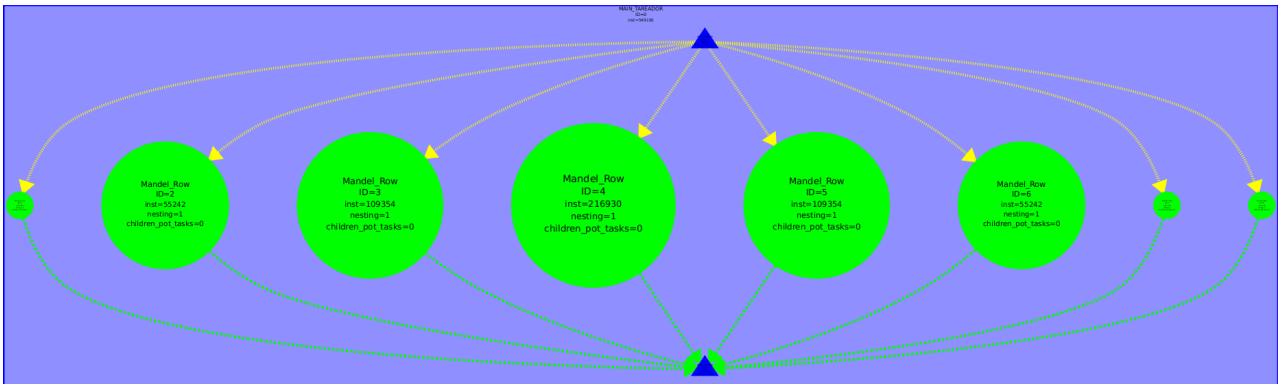


Figure 3: No option task decomposition (ROW)

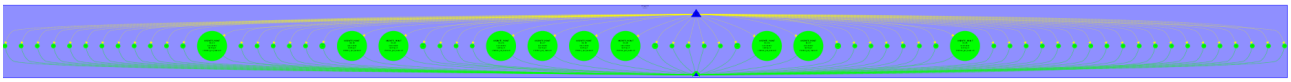


Figure 4: No option task decomposition (POINT)

1.3 Second Analysis

In this case, we are executing the same mandel-tar but now including the display option (-d), which by the way it's programmed, causes a data-race on a graphic variable that is used in order to display the Mandelbrot set after its computation, which leads to a sequential execution . Looking at the next two figures, it's no mystery that the size of the task remains the same, yet the execution it's done sequentially. Both again have the same difference, the point strategy has again more tasks, but in return those tasks have lighter workload.

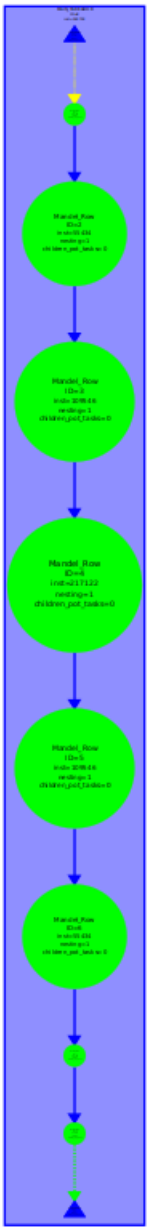


Figure 5: display task decomposition (ROW)

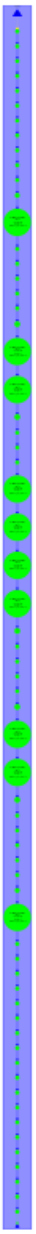


Figure 6: display task decomposition (POINT)

If we take a look at the code, we can observe that this dependency is caused by a graphic global variable called X11_COLOR used by the functions XSetForeGround and XDrawPoint that could be easily solved using the directive #pragma omp critical on the scope which both functions are called, thus avoiding that 2 or more threads can be executing that area at the same time.

```
            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);
                }
            }
        }
```

Figure 7: Code where the data race is produced

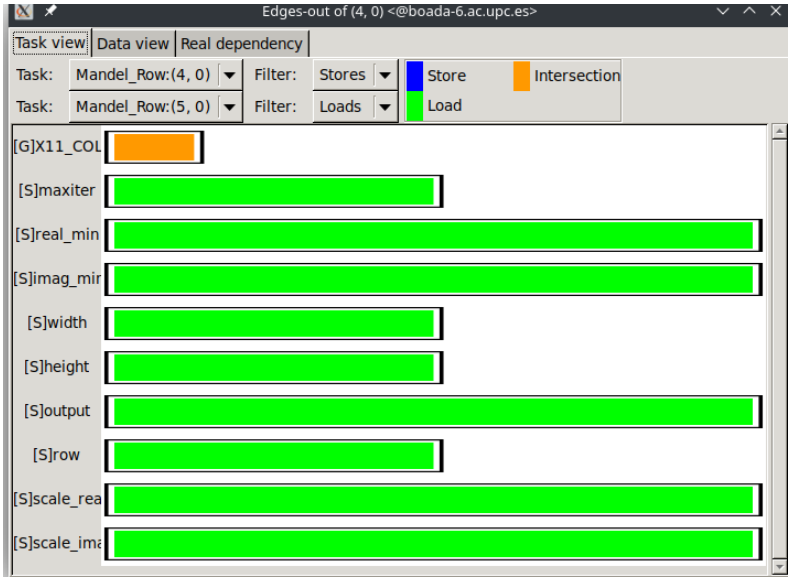


Figure 8: Tareador Dependency results

```
if (output2display) {
    /* Scale color and display point  */
    long color = (long) ((k-1) * scale_color) + min_color;
    if (setup_return == EXIT_SUCCESS) {
        #pragma omp critical
        {
        XSetForeground (display, gc, color);
        XDrawPoint (display, win, gc, col, row);
        }
    }
}
```

Figure 9: Dependency fixed using critical

1.4 Third Analysis

Discussion about the histogram option will be seen in this section. Executing the Tareador interface using the script we get as a result, these two dependency graphs:

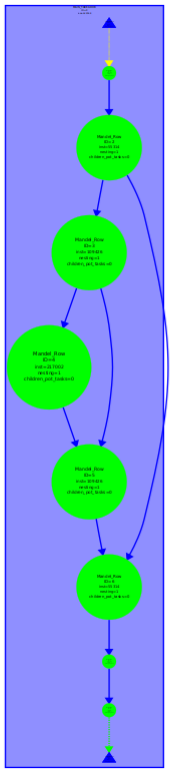


Figure 7: Histogram task decomposition (ROW)

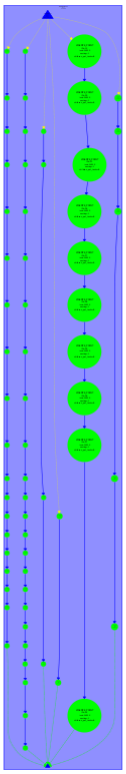


Figure 8: Histogram task decomposition (POINT)

Now we can certainly see a huge difference between both TDG's, and we can see there's some parallelism but not much at all, with some dependencies here and there. Judging the next code fragment, it's no mystery that the update of the histogram array causes all those dependencies. That's because a lot of points/rows could access the same array position (The k value will be known in execution time).

This is fairly easy to solve though, we just need to check just one thread at a time is updating that variable , and that can be done with the use of the directive # pragma omp atomic , that will treat the update as a single instruction, so the data race will be solved.

```
if (output2histogram){
   #pragma omp atomic
   histogram[k-1]++;
 }
```

Figure 9: Solution for the histogram data race

1.5 Strategies Comparison

In this final section, we will give some light about which approach could be better, the row or the point strategy.

In the first and second analysis, we saw that there's no big difference between the task dependency graph's talking about dependencies. But there's always the same distinctions: The Row strategy has fewer tasks, but some of them are huge, and the Point strategy has a lot more tasks but a lighter workload for each task. So the second one has more task creation overheads for sure, but it has, potentially, more parallelism. Both have the same problem, and it's that the workload between tasks it's not evenly distributed, some are tiny and some are huge.

But nothing can be said yet before experimenting with both strategies and looking at the data. So next 2 sessions a lot of statistics will be thrown and compared between each other, trying to find the best strategy and the best implementation too.

2 Point Decomposition Strategy

2.1 Using Task

Firstly, we will implement the strategy simply by making each "col" iteration a task, using the firsprivate option, so each task has its own correct values that will be used for the computation of that specific point.

```
#pragma omp parallel
#pragma omp single
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)
        {
        complex z, c;
```

Figure 10: Task firstprivate implementation

Now we present the results of the scalability of this implementation, done by the submit-strong-omp-sh script. Afterwards, the table with all the statistics are presented and will be briefly commented.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.58	0.35	0.33	0.31	0.34
Speedup	1.00	1.65	1.76	1.86	1.69
Efficiency	1.00	0.41	0.22	0.15	0.11

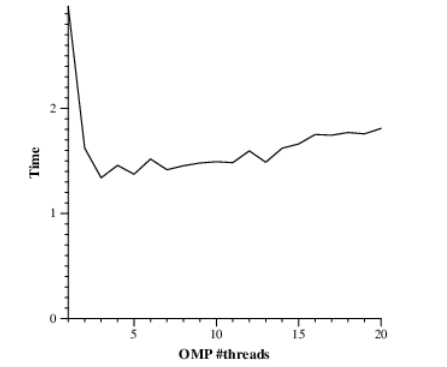
Table 1: Analysis done on Sun Apr 9 06:39:45 PM CEST 2023, par2318

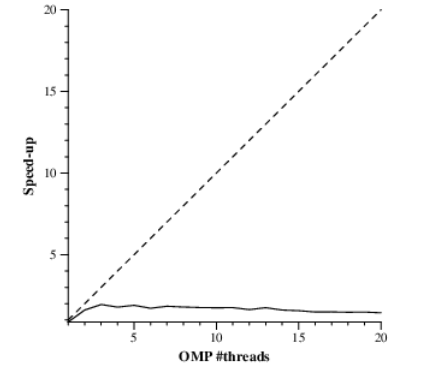
Overview of the Efficiency metrics in parallel fraction, ϕ=99.91%					
Number of processors	1	4	8	12	16
Global efficiency	95.28%	39.38%	20.94%	14.77%	10.08%
Parallelization strategy efficiency	95.28%	47.16%	27.77%	20.98%	15.12%
Load balancing	100.00%	92.39%	53.88%	38.63%	25.07%
In execution efficiency	95.28%	51.05%	51.55%	54.31%	60.29%
Scalability for computation tasks	100.00%	83.50%	75.39%	70.41%	66.70%
IPC scalability	100.00%	80.66%	75.34%	72.78%	69.12%
Instruction scalability	100.00%	103.97%	104.60%	104.44%	104.36%
Frequency scalability	100.00%	99.56%	95.66%	92.62%	92.47%

Table 2: Analysis done on Sun Apr 9 06:39:45 PM CEST 2023, par2318

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	0.79	0.82	0.85	0.84
LB (time executing explicit tasks)	1.0	0.89	0.9	0.89	0.9
Time per explicit task (average us)	4.9	5.66	5.81	6.08	6.07
Overhead per explicit task (synch %)	0.0	98.23	293.54	451.55	726.28
Overhead per explicit task (sched %)	5.46	29.61	26.87	22.87	21.89
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Sun Apr 9 06:39:45 PM CEST 2023, par2318





As we can clearly see, the speed-up grows poorly and when the number of cores is sufficiently high, it even gets worse. Then, we can say that the scalability is unacceptable due to the huge synchronization overheads of the # pragma omp critical that make the overhead per explicit task grow exponentially. Certainly the granularity is not appropriate, a bigger one would be better.

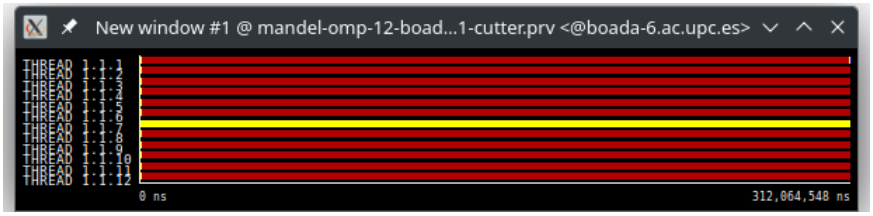


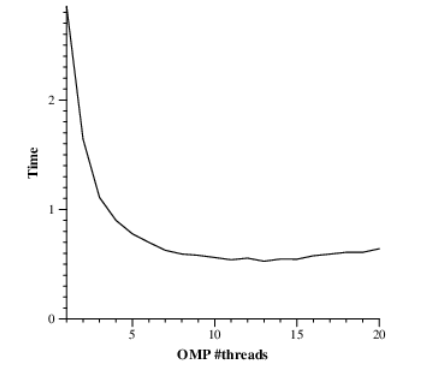
Figure 11: In this Paraver picture, we can observe that a thread is just creating tasks and scheduling them (Yellow), and the other ones are spending the vast majority of its time waiting for synchronization overheads (Red), so we can conclude that this execution it's terrific and shouldn't even be considered.

2.2 Controlling granularity with taskloop

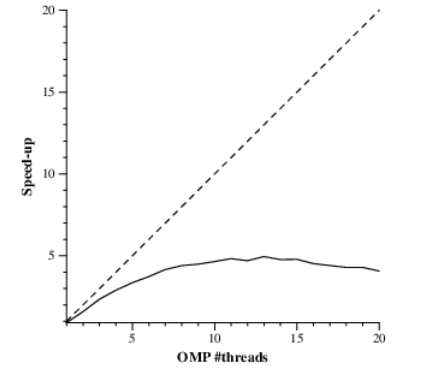
In this next implementation, we are going to see if implementing the same tactic with taskloop give us better results, leaving the granularity up to the OpenMP implementation to decide.

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

        z.real = z.imag = 0;

        /* Scale display coordinates to actual region  */
```

Figure 12: Taskloop implementation



par2318
Min elapsed execution time
Generated by par2318 on Sun Apr 9 08:30:59 PM CEST 2023



par2318
Speed-up wrt sequential time
Generated by par2318 on Sun Apr 9 08:30:59 PM CEST 2023

Judging by the plot, the scalability is much better, reaching almost a speedup of 5, but still far from the optimal, at the end it reaches almost no gain from additional threads. It's better, but there's still so much task synchronization even though the load balancing is much better (The execution efficiency it's terrible after 8 threads though). We can observe that the time executing explicit tasks goes down in a "logarithmic" manner, but it's much bigger than the past implementation and the number of tasks executed grows pretty fast compared to the other statistics. We can finally see that there's still a bad global efficiency , and now we have some additional time lost on scheduling and the implicit taskgroup.

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.16	0.14	0.16	0.20
Speedup	1.00	2.97	3.40	2.85	2.33
Efficiency	1.00	0.74	0.42	0.24	0.15

Table 4: Analysis done on Sun Apr 9 08:32:58 PM CEST 2023, par2318

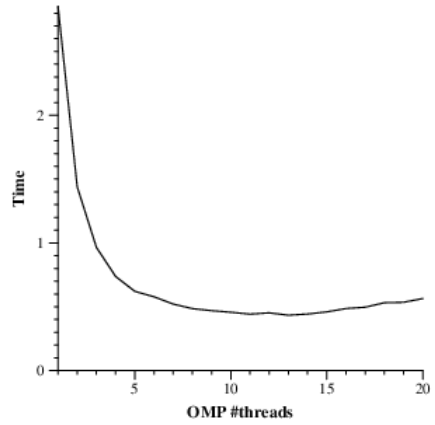
Overview of the Efficiency metrics in parallel fraction, ϕ=99.93%					
Number of processors	1	4	8	12	16
Global efficiency	99.61%	74.50%	42.31%	23.40%	14.06%
Parallelization strategy efficiency	99.61%	77.48%	47.81%	27.47%	16.78%
Load balancing	100.00%	95.24%	95.38%	94.73%	93.98%
In execution efficiency	99.61%	81.35%	50.12%	29.00%	17.86%
Scalability for computation tasks	100.00%	96.16%	88.50%	85.19%	83.77%
IPC scalability	100.00%	97.79%	97.45%	97.24%	96.70%
Instruction scalability	100.00%	99.45%	98.73%	98.02%	97.34%
Frequency scalability	100.00%	98.87%	91.98%	89.38%	88.99%

Table 5: Analysis done on Sun Apr 9 08:51:58 PM CEST 2023, par2318

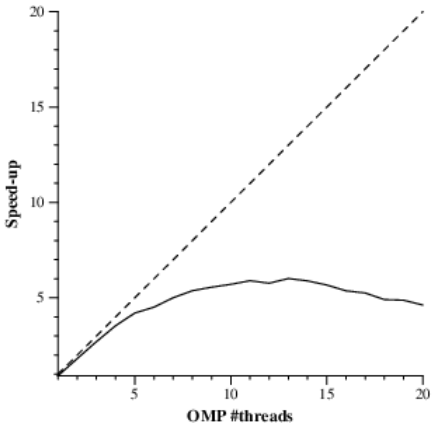
Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	3200.0	12800.0	25600.0	38400.0	51200.0
LB (number of explicit tasks executed)	1.0	0.94	0.83	0.53	0.47
LB (time executing explicit tasks)	1.0	0.95	0.95	0.95	0.94
Time per explicit task (average us)	143.1	37.21	20.21	14.0	10.67
Overhead per explicit task (synch %)	0.06	25.87	97.27	242.7	465.25
Overhead per explicit task (sched %)	0.33	3.21	11.95	21.48	31.15
Number of taskwait/taskgroup (total)	320.0	320.0	320.0	320.0	320.0

Table 6: Analysis done on Sun Apr 9 08:51:58 PM CEST 2023, par2318

As we have seen, we could improve even more the implementation, getting rid of those synchronization times caused by the dependencies of the implicit taskgroup. That's what our next implementation will consist of, just adding the clause "nogroup" to avoid that extra time wasted. Looking at the 2 following graphs we can certainly say that it's better than it's predecessor, but it has the same shape, so it suffers the same problems, except for the synchronization time of the taskgroups.



par2318
Min elapsed execution time
Generated by par2318 on Sun Apr 9 09:17:28 PM CEST 2023



par2318
Speed-up wrt sequential time
Generated by par2318 on Sun Apr 9 09:17:28 PM CEST 2023

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.46	0.13	0.11	0.14	0.19
Speedup	1.00	3.68	4.14	3.35	2.41
Efficiency	1.00	0.92	0.52	0.28	0.15

Table 7: Analysis done on Sun Apr 9 09:20:42 PM CEST 2023, par2318

Overview of the Efficiency metrics in parallel fraction, ϕ=99.93%					
Number of processors	1	4	8	12	16
Global efficiency	99.72%	91.78%	51.71%	27.91%	15.04%
Parallelization strategy efficiency	99.72%	95.75%	58.64%	32.86%	18.06%
Load balancing	100.00%	98.84%	98.66%	96.97%	97.06%
In execution efficiency	99.72%	96.87%	59.43%	33.88%	18.60%
Scalability for computation tasks	100.00%	95.86%	88.19%	84.95%	83.31%
IPC scalability	100.00%	98.54%	97.50%	96.92%	96.15%
Instruction scalability	100.00%	99.45%	98.73%	98.04%	97.34%
Frequency scalability	100.00%	97.81%	91.62%	89.40%	89.01%

Table 8: Analysis done on Sun Apr 9 09:20:42 PM CEST 2023, par2318

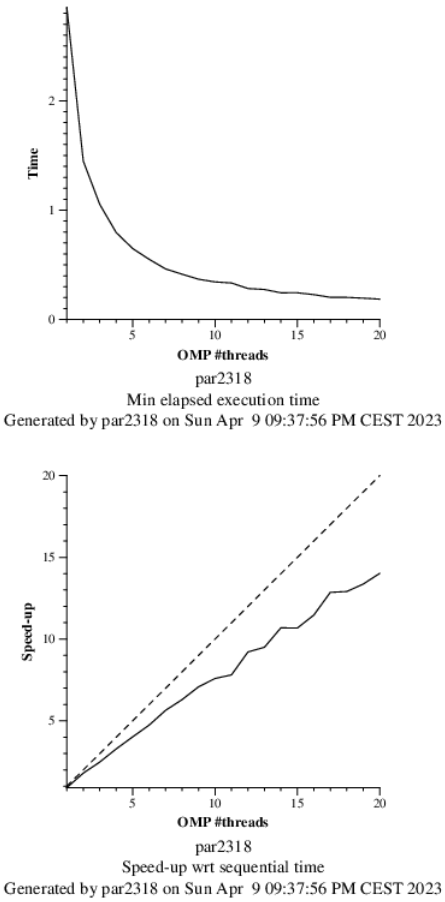
Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	3200.0	12800.0	25600.0	38400.0	51200.0
LB (number of explicit tasks executed)	1.0	0.5	0.91	0.75	0.9
LB (time executing explicit tasks)	1.0	0.99	0.99	0.97	0.97
Time per explicit task (average us)	143.15	37.32	20.28	14.04	10.73
Overhead per explicit task (synch %)	0.0	2.79	60.75	186.04	424.29
Overhead per explicit task (sched %)	0.28	1.65	9.83	18.45	29.91
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 9: Analysis done on Sun Apr 9 09:20:42 PM CEST 2023, par2318

As we can see, the results are practically the same except for the speedup, minor improves to the efficiency and the null number of taskgroups, achieving a few extra scalability.

3 Row Decomposition Strategy

Finally, we are discussing the Row strategy that will be simple but effective enough. It's just a taskloop in the outermost loop, in order to divide the tasks such that a task is a row or more. Again, we will let the OpenMP taskloop implementation decide the granularity for us. We will additionally write a second implementation without the taskloop construct, just to have another implementation to compare it against. Firstly, let's start with the taskloop implementation:



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	6.66	9.53	12.32
Efficiency	1.00	0.88	0.83	0.79	0.77

Table 10: Analysis done on Sun Apr 9 09:38:29 PM CEST 2023, par2318

Overview of the Efficiency metrics in parallel fraction, ϕ=99.88%					
Number of processors	1	4	8	12	16
Global efficiency	99.97%	88.02%	83.45%	79.76%	77.45%
Parallelization strategy efficiency	99.97%	91.00%	91.98%	90.78%	88.98%
Load balancing	100.00%	91.06%	92.12%	91.01%	89.38%
In execution efficiency	99.97%	99.93%	99.86%	99.74%	99.55%
Scalability for computation tasks	100.00%	96.72%	90.72%	87.86%	87.04%
IPC scalability	100.00%	98.55%	97.64%	96.83%	95.98%
Instruction scalability	100.00%	100.00%	99.99%	99.99%	99.99%
Frequency scalability	100.00%	98.15%	92.92%	90.74%	90.70%

Table 11: Analysis done on Sun Apr 9 09:38:29 PM CEST 2023, par2318

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)	45792.91	11836.21	6309.1	4342.71	3287.04
Overhead per explicit task (synch %)	0.0	9.86	8.66	10.08	12.28
Overhead per explicit task (sched %)	0.02	0.02	0.04	0.05	0.07
Number of taskwait/taskgroup (total)	1.0	1.0	1.0	1.0	1.0

Table 12: Analysis done on Sun Apr 9 09:38:29 PM CEST 2023, par2318

If we look closely, we can see this implementation by far the best. The average time spent on a single time is the biggest, but decreases as the number of thread increases. Has the best global efficiency and a good load balancing (Which has the best "In execution efficiency"). Time spent in scheduling is almost none and the % spent in synchronization grows linearly but in small values. It has the best scalability, better than the 3 implementations of the point strategy and has potential for even using more threads (Maybe speedup would be even higher, up to a certain number of threads). Notice that the number of tasks increases perfectly as the number of threads increases, balancing the load of each task in a huge amount (Notice also, that looking at the second table you can guess that the load balance of each task is also fantastic).

Now let's take a look at the second implementation of the row strategy, which each task correspond to a row, having the finest granularity possible for this implementation:

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	3.84	7.02	9.99	12.83
Efficiency	1.00	0.96	0.88	0.83	0.80

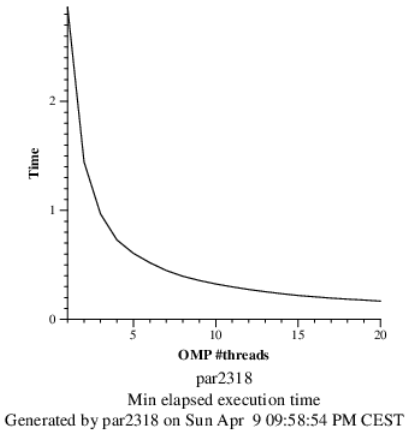
Table 13: Analysis done on Sun Apr 9 09:59:12 PM CEST 2023, par2318

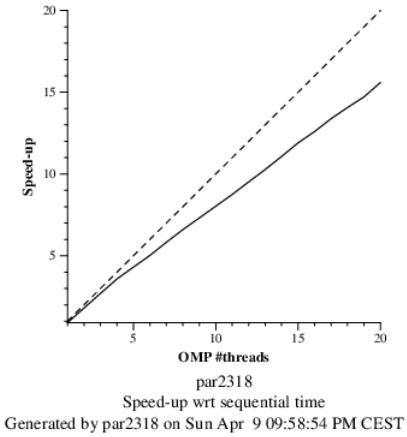
Overview of the Efficiency metrics in parallel fraction, ϕ=99.93%					
Number of processors	1	4	8	12	16
Global efficiency	99.97%	96.03%	88.00%	83.70%	80.72%
Parallelization strategy efficiency	99.97%	98.61%	97.68%	95.42%	92.44%
Load balancing	100.00%	99.06%	97.88%	95.68%	92.90%
In execution efficiency	99.97%	99.54%	99.79%	99.73%	99.50%
Scalability for computation tasks	100.00%	97.39%	90.09%	87.72%	87.33%
IPC scalability	100.00%	98.96%	97.71%	96.85%	96.45%
Instruction scalability	100.00%	100.02%	100.02%	100.02%	100.02%
Frequency scalability	100.00%	98.40%	92.19%	90.56%	90.53%

Table 14: Analysis done on Sun Apr 9 09:59:12 PM CEST 2023, par2318

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	0.98	0.96	0.93
Time per explicit task (average us)	1435.03	1473.54	1592.83	1635.75	1643.06
Overhead per explicit task (synch %)	0.0	1.27	2.26	4.65	7.99
Overhead per explicit task (sched %)	0.03	0.14	0.11	0.14	0.13
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 15: Analysis done on Sun Apr 9 09:59:12 PM CEST 2023, par2318





It's crystal clear that this implementation it's even better than the last one: the Number of explicit tasks remain constant but as the number of processor increases, the execution time gets much better, achieving an optimal performance and parallelism. The % of synchronization time is even lower and the number of taskgroups is none. The global efficiency is slightly better than the last one and also its load balancing.

Comparing this strategy with the point one, we can certainly say that the fine-grain of the point strategy causes too much overhead to be viable and thus, the scalability is deficient. Also, the workload of each task in the point strategy is not evenly distributed, due to the nature of the problem we are facing (Some points of the Mandelbrot set require much more computation than others). Furthermore, their percentage of time spent in synchronization overheads in the Point strategy is ridiculous and exponential, getting with 16 processors values as high as 726% which is just unacceptable and not scalable. In conclusion, Row strategy is far superior from the point one, doesn't matter which implementation you pick, the scalability is much better and has a lot of potential for even increasing the number of threads to higher values such as 32,64... Yet, the point strategy would be even worse if the number of threads rises.