LABORATORY 3

Group components: Albert Bausili Fernández, Noa-Yu Ventura Vila

Group identifier: par2301

10/11/2022

Index

Task decomposition analysis for the Mandelbrot set computation

1. The Mandelbrot set

The colors in the middle are white while the ones outside the biggest circumference are black, which means that the convergence times of the rows of points in black are extremely low (value of k might be 0 or 1) and the white ones have an extremely high time convergence (value of k is equal or close to the arbitrary value determined to stop the *do-while* loop). Points with higher convergence time match the tasks with a higher weight in the program, so the results from both executions match the expected output.

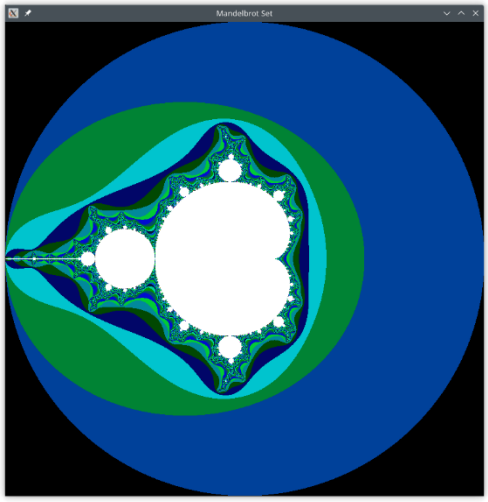

Figure 1. Display of the execution of the program mandel-seq without using any flags.

2. Row Strategy

Row strategy executes a task for every row in parallel. This is possible with the clause `#pragma omp task firstprivate(row)` inside of the first `for` loop (row iteration).

2.1 Unflagged execution of mendel-tar.c

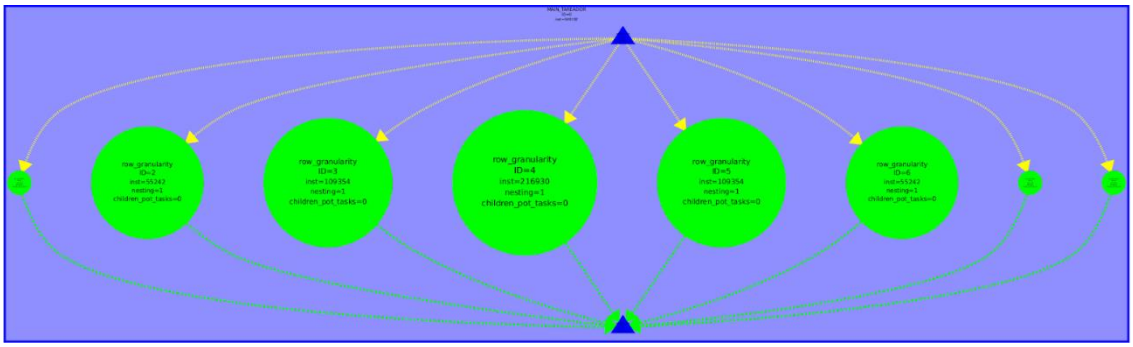

Figure 2. Screenshot of the execution for the program mandel-tar.c (row strategy) without using any flags.

2.1.1 Graph Analysis

We executed the program mandel-tar (row strategy) without flags using *Tareador* and got its task dependence graph shown in figure 2, from which we can deduce a few things. There are 8 tasks in total because there are 8 rows in the matrix, and it makes sense because we changed the code in order to get a task for each iteration of the row loop. Since they are all on the same level, we know the code is highly parallelizable, even though some tasks need to wait for the heaviest ones to finish their execution in order to proceed with the program.

From the size of each bubble that contains a task we can see that there is a big imbalance between the eight tasks, which is due to the different convergence times of each row of points, being the central ones the heaviest. This matches the results we get from the execution of mandel-seq (see figure 1), because the colors represent the convergence time for each of the points.

2.2 Execution of mandel-tar.c using flag -d

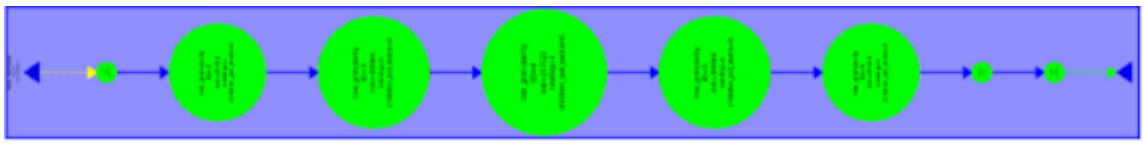

Figure 3. Screenshot of the execution for the program mandel-tar.c (row strategy) using the flag -d.

2.2.1 Graph Analysis

Figure 3 corresponds to the mandel-tar execution for *Tareador* using the flag -d. We can see that, unlike the previous task dependence graph, this one shows the tasks sequentially, but their size is still the same as before so the imbalance is unchanged.

2.2.2 Serialization discussion

In order to know what is causing this serialization we checked in the edges of the graph the real dependency. We found out a global variable, which *Tareador* named X11_COLOR_fake, is causing the dependencies between all the tasks, which makes them be executed sequentially even though they are parallelizable. This is shown in figure 4.

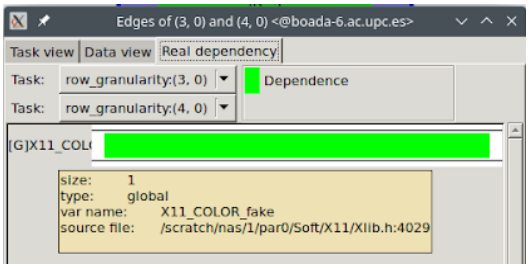

Figure 4. Real dependency of all edges of the tasks for mandel-tar.c (row strategy) using flag -d.

The fragment of the code below in section 1.2.3 is only executed when the option -d is chosen. The functions inside the if condition is from a library, one of which writes pixel for pixel the display. This is done sequentially, that's why it takes so much time to execute the program when using the -d option and the reason why *Tareador* shows us the parallelizable tasks depend on another one.

2.2.3 Solution to serialization with OpenMP

Writing `#pragma omp critical` is the solution to this problem, which is also shown in the code below. With this we are able to provide isolation to the critical section of the code. See the code below to check the solution.

```
…
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);
            }
      }
}
…
```

2.3 Execution of mandel-tar.c using flag -h

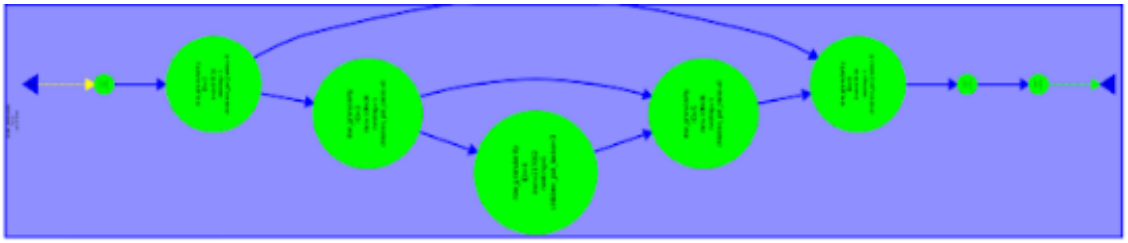

Figure 5. Screenshot of the execution for the program mandel-tar.c (row strategy) using the flag -h.

2.3.1 Graph Analysis

This is the same situation as using flag -d, there's a variable that makes parallelizable tasks depend on another one, making the execution of our code sequential. The difference now is that, even though the code is also executed sequentially, some of the heavier tasks depend on another heavy task. This is shown in figure 5, where each chain of tasks represents the dependencies between the access to histogram[k-1].

2.3.2 Serialization discussion

We followed the same logic for this task dependence graph for option -h and checked its edges. In this case, the variable that makes the code sequential is named uncast of type heap (see figure 6 below).

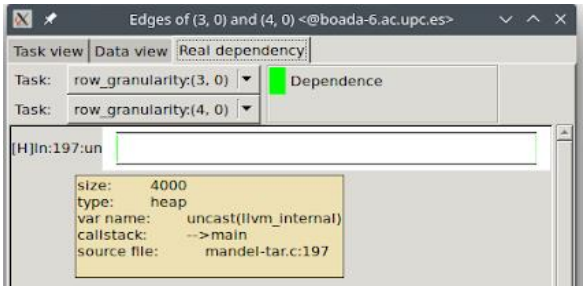

Figure 6. Screenshot of the edge for mandel-tar.c (row strategy) with option -h using Tareador.

Reading the code, we see a portion that is only executed when option -h is used, and this is verified with variable output2histogram. It makes the execution be sequential, because we access the vector histogram sequentially one position after the other.

2.3.3 Solution to serialization with OpenMP

Writing `#pragma omp atomic` is the solution to this problem, as the histogram vector suffers dataraces so with that clause we protect it from such problem. We have selected this option above the critical clause because that one is way more inefficient than the atomic one.

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

3. Point strategy

Point strategy executes a task for every column of each row in parallel. This is possible with the clause `#pragma omp task firstprivate(row,col)` inside of the inner loop.

3.1 Unflagged execution of mendel-tar.c

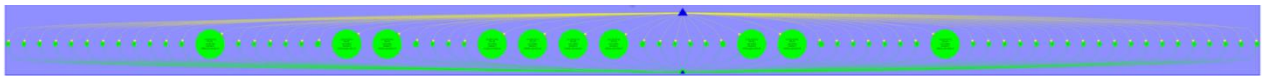

Figure 7. Screenshot of the execution for the program mandel-tar.c (point strategy) without using any flags.

3.1.1 Graph analysis

We executed the program mandel-tar (point strategy) without flags using Tareador and got its task dependence graph shown in figure 7, from which we can extract the following conclusions. There are as many tasks as points in the matrix. Since they are all on the same level, we know the code is higly parallelizable, even though some tasks need to wait need to wait for the heaviest ones to finish their execution in order to processw with the program.

From the size of each bubble that contains a task we can see that there is a big imbalance between the tasks, which is due to the different convergence times of each point, being the central ones the heaviest. This matches the results we get from the execution of mandel-seq (see figure 7), because the colors represent the convergence time for each of the points.

3.2 Execution of mandel-tar.c using flag -d

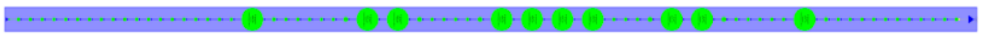

Figure 8. Screenshot of the execution for the program mandel-tar.c (point strategy) using the flag -d.

3.2.1 Graph Analysis

Figure 3 corresponds to the mandel-tar execution for *Tareador* using the flag -d. We can see that, unlike the previous task dependence graph, this one shows the tasks sequentially, but their size is still the same as before so the imbalance is unchanged.

3.2.2 Serialization discussion

In order to know what is causing this serialization we checked in the edges of the graph the real dependency. We found out a global variable, which *Tareador* named X11_COLOR_fake, is causing the dependencies between all the tasks, which makes them be executed sequentially even though they are parallelizable. This is shown in figure 4.

The fragment of the code below in section 1.2.3 is only executed when the option -d is chosen. The functions inside the if condition is from a library, one of which writes pixel for pixel the display. This is done sequentially, that's why it takes so much time to execute the program when using the -d option and the reason why *Tareador* shows us the parallelizable tasks depend on another one.

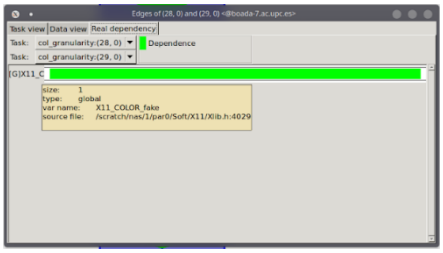

Figure 9. Real dependency of all edges of the tasks for mandel-tar.c (point strategy) using flag -d.

3.2.3 Solution to serialization with OpenMP

The solution in this case would be the same as in the row strategy, applying a critical to the XSetForeground and XDrawPoint.

3.3 Execution of mandel-tar.c using flag -h

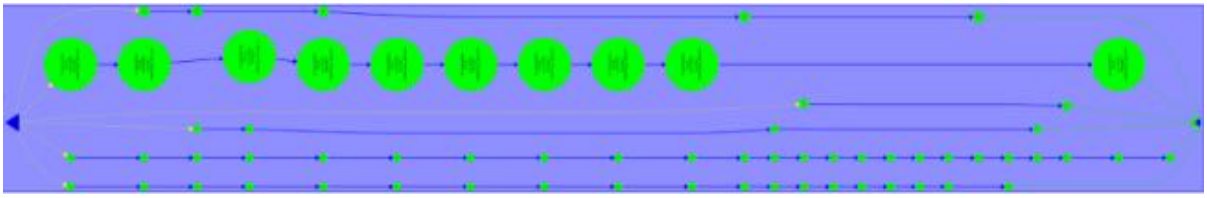

Figure 10. Screenshot of the execution for the program mandel-tar.c (point strategy) using the flag -h.

3.3.1 Graph Analysis

This is the same situation as using flag -d, there's a variable that makes parallelizable tasks depend on another one, making the execution of our code sequential. The difference now is that, even though the code is executed in parallel, some of the heavier tasks depend on another heavy task. This is shown in figure 10, where each chain of tasks represents the dependencies between the access to histogram[k-1].

3.3.2 Serialization discussion

We followed the same logic for this task dependence graph for option -h and checked its edges. In this case, the variable that makes the code sequential is named uncast of type heap (see figure 11 below).

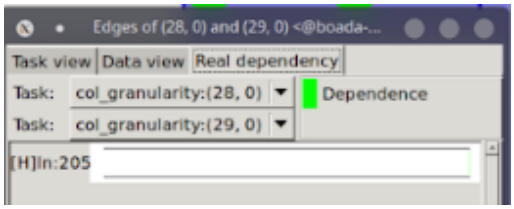

Figure 11. Screenshot of the edge for mandel-tar.c (point strategy) with option -h using Tareador.

Reading the code, we see a portion that is only executed when option -h is used, and this is verified with variable output2histogram. It makes the execution be sequential, because we access the vector histogram sequentially one position after the other.

3.3.3 Solution to serialization with OpenMP

The solution in this case would be the same as in the row strategy, applying an atomic to `histogram[k-1]++`.

Implementing task decompositions in OpenMP

1. Point strategy implementation using tasks

This scalability graphs come from the base version of the program, the original code with atomic and critical as optimitzations.

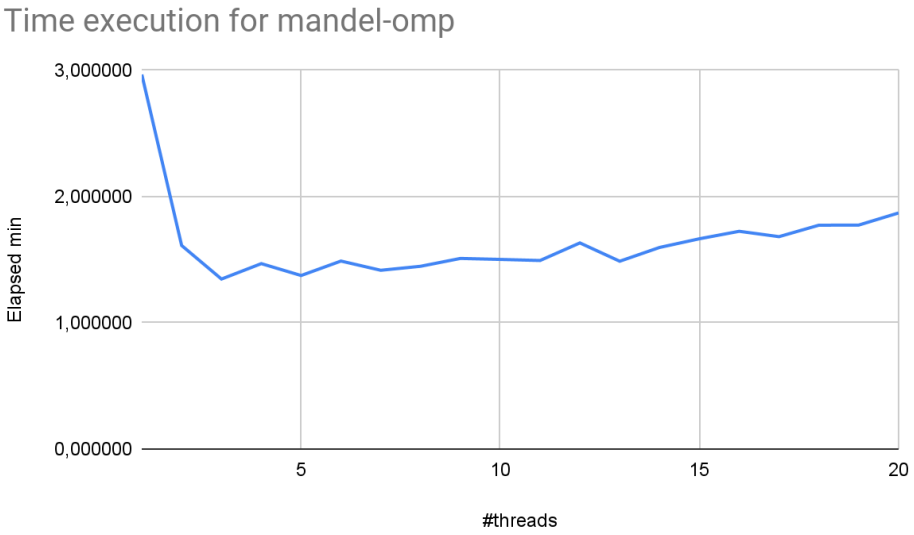

Figure 12. Graph representing the amount of execution time compared with the number of threads.

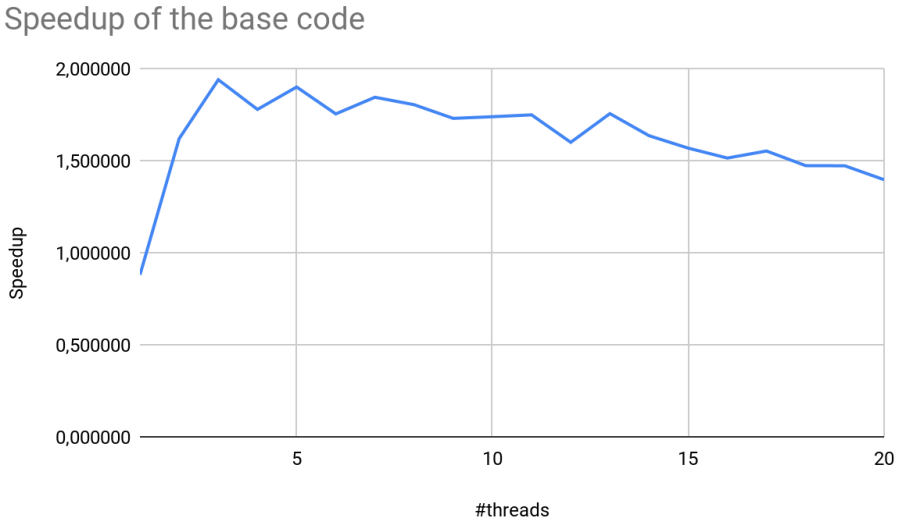

Figure 13. Graph representing the speedup compared with the number of threads.

The conclusion we extract from this is that the code does not get any gains past the 3-core mark (even it goes worse and worse), that means that the parallelization is quite poor with this code.

This is the post script file content, note that the added tables are just 2 columns.

```
make: 'mandel-seq' is up to date.
make: 'mandel-omp' is up to date.
Executing mandel-seq sequentially
Run 0...Elapsed time = 2.607449
Run 1...Elapsed time = 2.612795
Run 2...Elapsed time = 2.612108
ELAPSED TIME MIN OF 3 EXECUTIONS =2.607449

mandel-omp 1 20 3
Starting OpenMP executions...
Executing mandel-omp with 1 threads
Run 0...Elapsed time = 2.967796
Run 1...Elapsed time = 2.970036
Run 2...Elapsed time = 2.967594
ELAPSED TIME MIN OF 3 EXECUTIONS =2.967594

Executing mandel-omp with 2 threads
Run 0...Elapsed time = 1.629440
Run 1...Elapsed time = 1.632965
Run 2...Elapsed time = 1.620811
ELAPSED TIME MIN OF 3 EXECUTIONS =1.620811

Executing mandel-omp with 3 threads
Run 0...Elapsed time = 1.380806
Run 1...Elapsed time = 1.432131
Run 2...Elapsed time = 1.463838
ELAPSED TIME MIN OF 3 EXECUTIONS =1.380806

Executing mandel-omp with 4 threads
Run 0...Elapsed time = 1.514774
Run 1...Elapsed time = 1.442650
Run 2...Elapsed time = 1.521002
ELAPSED TIME MIN OF 3 EXECUTIONS =1.442650

Executing mandel-omp with 5 threads
Run 0...Elapsed time = 1.406101
Run 1...Elapsed time = 1.373618
Run 2...Elapsed time = 1.422241
ELAPSED TIME MIN OF 3 EXECUTIONS =1.373618

Executing mandel-omp with 6 threads
Run 0...Elapsed time = 1.524149
Run 1...Elapsed time = 1.540987
Run 2...Elapsed time = 1.483137
ELAPSED TIME MIN OF 3 EXECUTIONS =1.483137

Executing mandel-omp with 7 threads
Run 0...Elapsed time = 1.424452
Run 1...Elapsed time = 1.423655
Run 2...Elapsed time = 1.445816
ELAPSED TIME MIN OF 3 EXECUTIONS =1.423655

Executing mandel-omp with 8 threads
Run 0...Elapsed time = 1.453293
Run 1...Elapsed time = 1.490221
Run 2...Elapsed time = 1.475250
ELAPSED TIME MIN OF 3 EXECUTIONS =1.453293
```

Executing mandel-omp with 9 threads
Run 0...Elapsed time = 1.516040
Run 1...Elapsed time = 1.478456
Run 2...Elapsed time = 1.520367
ELAPSED TIME MIN OF 3 EXECUTIONS =1.478456

Executing mandel-omp with 10 threads
Run 0...Elapsed time = 1.514485
Run 1...Elapsed time = 1.504713
Run 2...Elapsed time = 1.528736
ELAPSED TIME MIN OF 3 EXECUTIONS =1.504713
Executing mandel-omp with 11 threads
Run 0...Elapsed time = 1.533935
Run 1...Elapsed time = 1.525739
Run 2...Elapsed time = 1.539707
ELAPSED TIME MIN OF 3 EXECUTIONS =1.525739

Executing mandel-omp with 12 threads
Run 0...Elapsed time = 1.516490
Run 1...Elapsed time = 1.652558
Run 2...Elapsed time = 1.688536
ELAPSED TIME MIN OF 3 EXECUTIONS =1.516490

Executing mandel-omp with 13 threads
Run 0...Elapsed time = 1.545798
Run 1...Elapsed time = 1.524688
Run 2...Elapsed time = 1.609556
ELAPSED TIME MIN OF 3 EXECUTIONS =1.524688

Executing mandel-omp with 14 threads
Run 0...Elapsed time = 1.718136
Run 1...Elapsed time = 1.670105
Run 2...Elapsed time = 1.643928
ELAPSED TIME MIN OF 3 EXECUTIONS =1.643928

Executing mandel-omp with 15 threads
Run 0...Elapsed time = 1.690923
Run 1...Elapsed time = 1.689535
Run 2...Elapsed time = 1.694433
ELAPSED TIME MIN OF 3 EXECUTIONS =1.689535

Executing mandel-omp with 16 threads
Run 0...Elapsed time = 1.716095
Run 1...Elapsed time = 1.761670
Run 2...Elapsed time = 1.820508
ELAPSED TIME MIN OF 3 EXECUTIONS =1.716095

Executing mandel-omp with 17 threads
Run 0...Elapsed time = 1.729341
Run 1...Elapsed time = 1.760967
Run 2...Elapsed time = 1.687576
ELAPSED TIME MIN OF 3 EXECUTIONS =1.687576

Executing mandel-omp with 18 threads
Run 0...Elapsed time = 1.970642
Run 1...Elapsed time = 1.779902
Run 2...Elapsed time = 1.824226
ELAPSED TIME MIN OF 3 EXECUTIONS =1.779902

Executing mandel-omp with 19 threads
Run 0...Elapsed time = 1.821544
Run 1...Elapsed time = 1.761749
Run 2...Elapsed time = 1.807641
ELAPSED TIME MIN OF 3 EXECUTIONS =1.761749

Executing mandel-omp with 20 threads
Run 0...Elapsed time = 1.980055
Run 1...Elapsed time = 1.862589
Run 2...Elapsed time = 1.916657
ELAPSED TIME MIN OF 3 EXECUTIONS =1.862589

#threads	Elapsed min
1	2,966506
2	1,612094
3	1,346258
4	1,468426
5	1,374311
6	1,488779
7	1,415782
8	1,447476
9	1,509646
10	1,501862
11	1,493176
12	1,632445
13	1,487451
14	1,596570
15	1,665312
16	1,724529
17	1,682151
18	1,772812
19	1,773767
20	1,869757

#threads	Speedup
1	0,879753
2	1,618884
3	1,938553
4	1,777272
5	1,898983
6	1,752975
7	1,843358
8	1,802996
9	1,728745
10	1,737705
11	1,747813
12	1,598702
13	1,754540
14	1,634625
15	1,567150
16	1,513337
17	1,551462
18	1,472121
19	1,471328
20	1,395793

1.1 Overall analysis with Modelfactors

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.58	0.37	0.32	0.32	0.35
Speedup	1.00	1.58	1.82	1.83	1.66
Efficiency	1.00	0.40	0.23	0.15	0.10

Table 1: Analysis done on Thu Nov 10 01:47:25 AM CET 2022, par2301

Figure 14. Modelfactors table 1 of the base code

Overview of the Efficiency metrics in parallel fraction, ϕ=99.94%					
Number of processors	1	4	8	12	16
Global efficiency	95.21%	37.67%	21.67%	14.56%	9.86%
Parallelization strategy efficiency	95.21%	45.21%	29.26%	20.95%	14.92%
Load balancing	100.00%	92.48%	54.12%	37.45%	24.16%
In execution efficiency	95.21%	48.89%	54.06%	55.95%	61.76%
Scalability for computation tasks	100.00%	83.32%	74.09%	69.51%	66.05%
IPC scalability	100.00%	80.06%	74.15%	71.72%	68.32%
Instruction scalability	100.00%	103.90%	104.29%	104.35%	104.19%
Frequency scalability	100.00%	100.16%	95.81%	92.87%	92.79%

Table 2: Analysis done on Thu Nov 10 01:47:25 AM CET 2022, par2301

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	0.81	0.83	0.85	0.85
LB (time executing explicit tasks)	1.0	0.89	0.89	0.89	0.88
Time per explicit task (average us)	4.91	5.65	5.91	6.11	6.05
Overhead per explicit task (synch %)	0.0	106.38	273.77	456.77	747.59
Overhead per explicit task (sched %)	5.52	32.36	24.19	22.25	21.62
Number of taskwait/taskgroup (total)	0.0	0.0	0.0	0.0	0.0

Table 3: Analysis done on Thu Nov 10 01:47:25 AM CET 2022, par2301

Figure 15. Modelfactors table 1and 2 of the base code

From this tables we can deduce the following conclusions. The first one is that the paralitzation achieved is almost 100%, that means that we haven't left almost any sequential code. We can also see that we have a huge balancing problem, as the lowest efficiency is achieved by that. The last conclusion is that we have a lot of tasks for only 16 threads, too much of them (10000). In the 16 thread we have almost up to 800% of overheads.

The recommended solution in order to solved this is reducing the amount of tasks and increasing the amount of work done by each of them.

1.2 Detailed analysis with Paraver

Figure 16. Paraver output

From this paraver we can clearly see that there is jsu tone thread that does all the task creation, while the others are most of the time in synchroitzation. This really contributes to the unvalance of tasks.

1.3 Optimitzation: point strategy using OpenMP clause taskloop

```
2   ...
3   // Calculate points and generate appropriate output
4     #pragma omp parallel
5     #pragma omp single
6     for (int row = 0; row < height; ++row) {
7       #pragma omp taskloop
8       for (int col = 0; col < width; ++col) {
9           #pragma omp task firstprivate(row, col)
10        {
11        complex z, c;
12        z.real = z.imag = 0;
13  ...
```

Figure 17. Code with the taskloop optimization

With this optimitzation we are able to control the granularity of the loop, in this case as we don't especify any granularity OpenMP will decide a fitting amount for us.

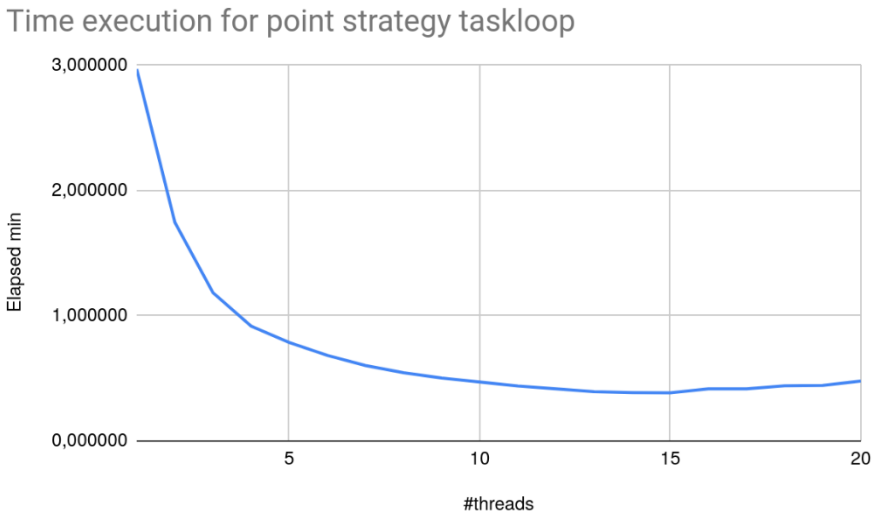

Figure 18. Graph representing the amount of execution time compared with the number of threads.

Figure 19. Graph representing the speedup compared with the number of threads.

1.3.1 Overall analysis with Modelfactors

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.58	0.32	0.24	0.22	0.22
Speedup	1.00	1.82	2.41	2.61	2.66
Efficiency	1.00	0.45	0.30	0.22	0.17

Table 1: Analysis done on Thu Nov 10 03:49:14 AM CET 2022, par2301

Figure 20. Modelfactors table 1 of the base code

Overview of the Efficiency metrics in parallel fraction, ϕ=99.95%					
Number of processors	1	4	8	12	16
Global efficiency	95.10%	43.21%	28.66%	20.68%	15.82%
Parallelization strategy efficiency	95.10%	51.97%	36.79%	27.87%	21.90%
Load balancing	100.00%	96.19%	94.33%	92.72%	92.19%
In execution efficiency	95.10%	54.03%	39.00%	30.06%	23.75%
Scalability for computation tasks	100.00%	83.14%	77.91%	74.19%	72.23%
IPC scalability	100.00%	81.65%	80.87%	80.48%	79.27%
Instruction scalability	100.00%	104.64%	104.00%	103.37%	102.78%
Frequency scalability	100.00%	97.31%	92.65%	89.18%	88.66%

Table 2: Analysis done on Thu Nov 10 03:49:14 AM CET 2022, par2301

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	105600.0	115200.0	128000.0	140800.0	153600.0
LB (number of explicit tasks executed)	1.0	0.76	0.57	0.49	0.57
LB (time executing explicit tasks)	1.0	0.98	0.98	0.96	0.96
Time per explicit task (average us)	4.77	7.98	9.54	11.13	12.48
Overhead per explicit task (synch %)	0.05	37.72	56.04	68.37	79.65
Overhead per explicit task (sched %)	5.58	28.75	43.27	54.06	62.1
Number of taskwait/taskgroup (total)	320.0	320.0	320.0	320.0	320.0

Table 3: Analysis done on Thu Nov 10 03:49:14 AM CET 2022, par2301

Figure 21. Modelfactors table 2 of the base code

From the scability graphs we can see the huge difference that there is of this code compared to the base one. Now we deduce that the number of tasks have been reduced a lot, and we can check that in the modelfactors tables confirming our hipotesis.

Now as we see in Table 2 load balancing is not a problem anymore, now the only concern Is execution efficiency.

Now there are more tasks generated and the average of execution times are better. The overheads in sync and sched have reduced in a huge amount.

0.57 tasks are executed per taskloop, the granularity is better, but the sync still takes a considerable amount of resources because of the type of problem that it cannot be completely paralized.

1.3.2 Detailed analysis and possible optimization: Paraver

Figure 22. Paraver output taskloop code

Yes, they are necessary because if we are executing code in more than one thread then it takes a lot of time to execute the program. Because of the table obtained using taskloop nogroup.

2. Row strategy implementation

To implement the row strategy we moved the clause #pragma omp taskloop to the line before the for loop that iterates the rows of the matrix. This was, for every row we perform a task in parallel. The modified code is:

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

As we can see in the elapsed time of row strategy the more threads, we use the faster the program is executed. This matches the results obtained on second graph which shows the speedup increases lineally.

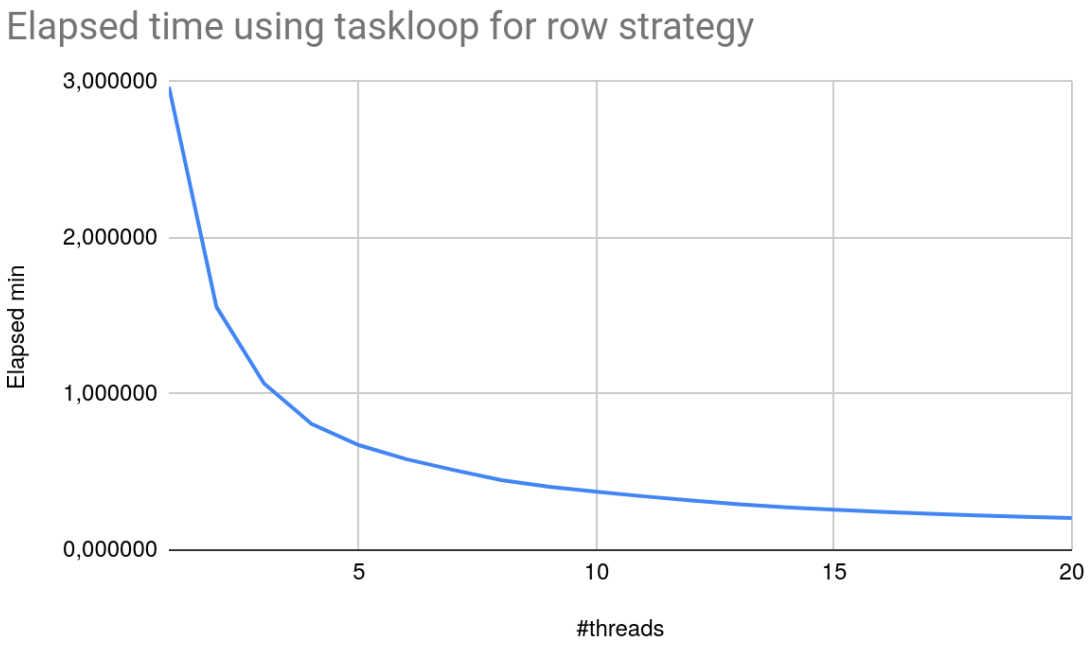

Figure 24. Elapsed time compared with the amount of processors in row strategy

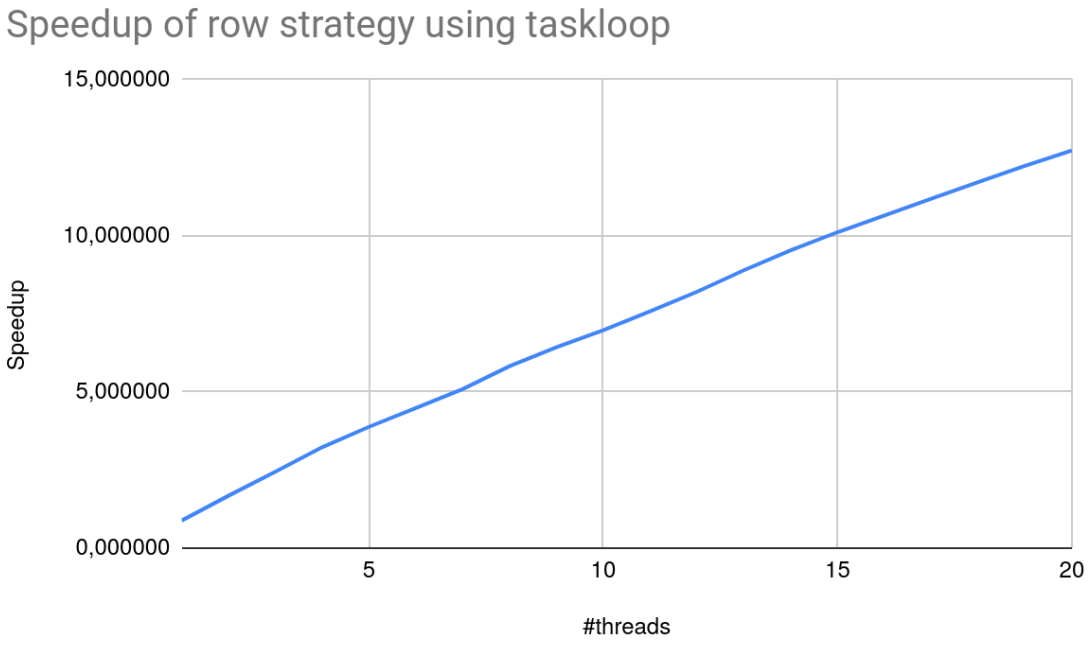

Figure 23. Speedup compared to the number of threads of row strategy

There's a big difference between point strategy scalability graphs and row strategy scalability graphs. In the first one there's no speedup at all and the elapsed time stays more or less the same, independently of the number of threads used. On the row strategy the speedup increases linearly and the more threads used the less time it takes for the rogram to finish. This is due to the fact that the number of tasks of point strategy

is very big, which wastes a lot of time synchronizing thread, unlike row strategy that has only eight tasks.

To sum up, there's a very big load imbalance on the point strategy (there are a lot of tasks and they do only one iteration) and waste a lot of time on synchronization, that's why using more threads doesn't decrease the execution time. On the other hand, row strategy is more balanced having 8 tasks with a reasonable number of iterations each.

The results in this tables reinforce our conclusions; those were that row strategy is a lot more parallelizable. And in the paraver we see that there are very few synchronization.

2.1 Modelfactors

Overview of whole program execution metrics					
Number of processors	1	4	8	12	16
Elapsed time (sec)	0.58	0.38	0.44	0.49	0.70
Speedup	1.00	1.53	1.32	1.17	0.82
Efficiency	1.00	0.38	0.17	0.10	0.05

Table 1: Analysis done on Thu Nov 10 05:07:38 AM CET 2022, par2301

Figure 24. First table Modelfactors of row strategy

Overview of the Efficiency metrics in parallel fraction, ϕ=99.95%					
Number of processors	1	4	8	12	16
Global efficiency	95.41%	36.47%	15.76%	9.34%	4.91%
Parallelization strategy efficiency	95.41%	66.88%	51.85%	44.77%	37.37%
Load balancing	100.00%	96.05%	95.96%	93.03%	91.35%
In execution efficiency	95.41%	69.62%	54.03%	48.13%	40.91%
Scalability for computation tasks	100.00%	54.53%	30.39%	20.87%	13.13%
IPC scalability	100.00%	85.05%	82.10%	83.39%	88.48%
Instruction scalability	100.00%	89.39%	85.21%	85.37%	80.28%
Frequency scalability	100.00%	71.73%	43.44%	29.31%	18.49%

Table 2: Analysis done on Thu Nov 10 05:07:38 AM CET 2022, par2301

Statistics about explicit tasks in parallel fraction					
Number of processors	1	4	8	12	16
Number of explicit tasks executed (total)	102410.0	102440.0	102480.0	102520.0	102560.0
LB (number of explicit tasks executed)	1.0	0.63	0.7	0.85	0.88
LB (time executing explicit tasks)	1.0	0.88	0.85	0.8	0.74
Time per explicit task (average us)	4.9	5.72	8.46	13.25	24.79
Overhead per explicit task (synch %)	0.0	6.44	27.18	43.88	53.39
Overhead per explicit task (sched %)	5.28	78.87	166.96	195.78	223.08
Number of taskwait/taskgroup (total)	1.0	1.0	1.0	1.0	1.0

Table 3: Analysis done on Thu Nov 10 05:07:38 AM CET 2022, par2301

Figure 25. Second and third table Modelfactors of row strategy

2.2 Paraver

Figure 26. Paraver output of row strategy