PAR Laboratory Assignment

Lab 3: Embarrassingly parallelism with OpenMP:
Mandelbrot set

Aleix Boné
Alex Herrero
par2109

2020-04-17

Contents

1 Introduction

In this laboratory assignment we explored the tasking model in OpenMP to express iterative task decompositions. We started by exploring the most appropriate ones by using *Tareador*. The program that we used is the computation of the *Mandelbrot set*, a particular set of points, in the complex domain, whose boundary generates a distinctive and easily recognisable two-dimensional fractal shape.

For each point c in a delimited two-dimensional space, the complex quadratic polynomial recurrence $z_{n+1} = z_n^2 + c$ is iteratively applied in order to determine if it belongs or not to the Mandelbrot set. The point is part of the Mandelbrot set if, when starting with $z_0 = 0$ and applying the iteration repeatedly, the absolute value of z_n never exceeds a certain number however large n gets.

A plot of the Mandelbrot set is created by colouring each point c in the complex plane with the number of steps max for which $|z_{\max}| \geq 2$ (or simply $|z_{\max}|^2 \geq 2 * 2$ to avoid the computation of the square root in the modulus of a complex number). In order to make the problem doable, the maximum number of steps is also limited: if that number of steps is reached, then the point c is said to belong to the Mandelbrot set.

This means that when analyzing the parallelism of the program we have to keep in mind that the computation of the points will need different number of iterations ranging from 1 to the limit set when running the program.

We analyzed the potential parallelism for two possible task granularities that can be exploited in this program. Those two are:

a) *Point*: a task corresponds with the computation of a single point (`row`,`col`) of the Mandelbrot set.

b) *Row*: a task corresponds with the computation of a whole `row` of the Mandelbrot set.

2 Task decomposition analysis with *tareador*

We executed both, `mandel-tar`[1] and `mandeld-tar`[2], programs with the different granularities using the `./run-tareador.sh` script and we obtained the following results.

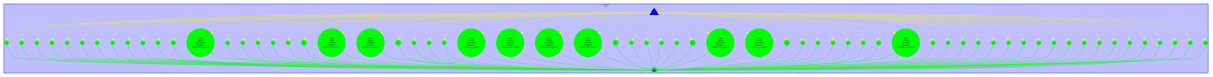


Figure 1: Task dependence graph of `mandel-tar.c` with point decomposition.

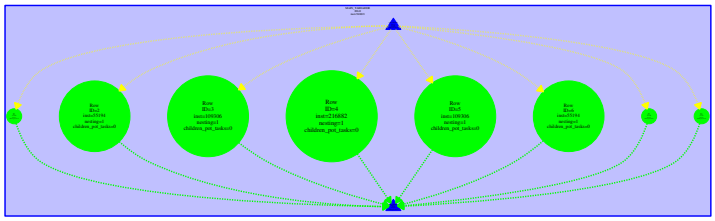


Figure 2: Task dependence graph of `mandel-tar.c` with row decomposition.

As we expected the point decomposition has a much more big granularity than the row one. But we obtained that the two decompositions generate similar task graphs for each executable.

We can observe in figures 1 and 2 that with the `mandel` executable all tasks can parallelize at the same level. But in figure 3 we see that with the `mandeld` executable we obtain a serialization of all tasks.

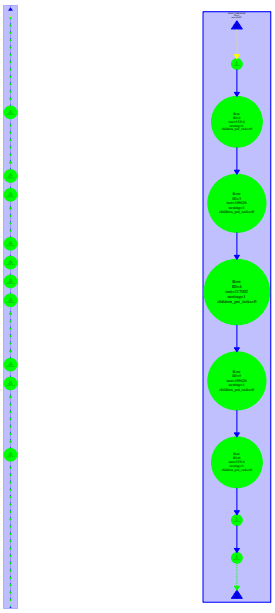


Figure 3: Task dependence graph of `mandeld-tar.c` with point and row decomposition.

[1] `mandel`: used for timing purposes and to check for the numerical validity of the output (non-graphical version).

[2] `mandeld`: generates a binary that visualizes the Mandelbrot set (graphical version).

This serialization in the graphical version is caused in the lines 120-130 of the code due to a data race when accessing the display, this implies full dependence between all tasks. Therefore when parallelizing the program with OpenMP, this section has to be protected with the `#pragma omp critical` clause as shown in the listing 1.

```
120 #if _DISPLAY_
121             /* Scale color and display point  */
122             long color = (long) ((k-1) * scale_color) + min_color;
123             #pragma omp critical
124             if (setup_return == EXIT_SUCCESS) {
125                 XSetForeground (display, gc, color);
126                 XDrawPoint (display, win, gc, col, row);
127             }
128 #else
129             output[row][col]=k;
130 #endif
```

Listing 1: Problematic section in `mandel-omp.c` protected with `#pragma omp critical`.

The program is embarrassingly parallelizable so to implement a parallel version of the Mandelbrot code the point granularity should be more appropriate. But with a low number of threads the overhead of task creation would increase and in that case the row granularity could have a better final execution time.

We can also see that there is a big variation in the number of instructions per task, specially in the point decomposition. This is due to the recursive nature of the Mandelbrot set computation, where we can't know how many iterations it'll take to compute each point. The problematic loop is shown in listing 2 where we can see that for each point it could take from 1 to `maxiter` iterations of the loop to compute.

```
109             /* Calculate z0, z1, .... until divergence or maximum iterations */
110             int k = 0;
111             double lengthsq, temp;
112             do  {
113                 temp = z.real*z.real - z.imag*z.imag + c.real;
114                 z.imag = 2*z.real*z.imag + c.imag;
115                 z.real = temp;
116                 lengthsq = z.real*z.real + z.imag*z.imag;
117                 ++k;
118             } while (lengthsq < (N*N) && k < maxiter);
```

Listing 2: Problematic section in `mandel-omp.c`

3 Parallelization Strategies

3.1 *Point* strategy using *OpenMP* `task`

There are multiple ways to create a task for the computation of each point in the Mandelbrot set. We tried several options and analyzed the performance of each one.

The first version is shown in listing 3. We create a task for each iteration of the `col` for. The pool of threads is initialized at every row.

Listing 3: v1 of point task decomposition

```
91      /* Calculate points and save/display */
92      for (int row = 0; row < height; ++row) {
93          #pragma omp parallel
94          #pragma omp single
95          for (int col = 0; col < width; ++col) {
96              #pragma omp task firstprivate(col)
97              {
98              complex z, c;
```

In the second version we mode the parallel and single pragmas outside of the `row` loop. Therefore, there is only one parallel region in the program. Since we also add a taskwait, the program should behave the same as with version 1.

Listing 4: v2: point task decomposition with taskwait

```
91      /* Calculate points and save/display */
92      #pragma omp parallel
93      #pragma omp single
94      for (int row = 0; row < height; ++row) {
95          for (int col = 0; col < width; ++col) {
96              #pragma omp task firstprivate(row, col)
97              {
98              complex z, c;
```

```
132         }
133         #pragma omp taskwait // waiting point for all child tasks
134     }
```

In v3 we use `taskgroup` instead of `taskwait`. The main difference between `taskgroup` and `taskwait` is that the former waits for all tasks inside the group, including children tasks. `taskwait` only waits for the tasks generated on the same level, and not the children of those tasks. In our particular case, since there are no nested tasks generated, there should be no noticeable difference between v2 and v3.

Listing 5: v3: point task decomposition with taskgroup

```
91      /* Calculate points and save/display */
92      #pragma omp parallel
93      #pragma omp single
94      for (int row = 0; row < height; ++row)
95      #pragma omp taskgroup
96      {
97          for (int col = 0; col < width; ++col) {
98              #pragma omp task firstprivate(row, col)
```

```
133             }
134         }
135     } // waiting point for all tasks in taskgroup region
```

v4 removes the `taskwait` directive, since there is no need to wait for the previous row tasks to finish before generating the new ones.

Listing 6: v4: point task decomposition task without taskwait

```
91      /* Calculate points and save/display */
92      #pragma omp parallel
93      #pragma omp single
94      for (int row = 0; row < height; ++row) {
95          for (int col = 0; col < width; ++col) {
96              #pragma omp task firstprivate(row, col)
97              {
98              complex z, c;
```

```
131             }
132         }
133     }
```

v5 uses `taskloop` to explicitly generate the tasks. Since there is no granularity, OpenMP decides a value for it depending of the number of threads in the region.

Listing 7: v5: point task decomposition with taskloop

```
91      /* Calculate points and save/display */
92      #pragma omp parallel
93      #pragma omp single
94      for (int row = 0; row < height; ++row) {
95          #pragma omp taskloop firstprivate(row)
96          for (int col = 0; col < width; ++col) {
97              {
98              complex z, c;
```

In v6 we specify a granularity of 1, meaning that every iteration loop is assigned to a single task

Listing 8: v6: point task decomposition with taskloop grainsize(1)

```
91      /* Calculate points and save/display */
92      #pragma omp parallel
93      #pragma omp single
94      for (int row = 0; row < height; ++row) {
95          #pragma omp taskloop firstprivate(row) grainsize(1)
96          for (int col = 0; col < width; ++col) {
97              {
98              complex z, c;
```

In v7 we specify a granularity of 1 and `nogroup` that eliminates the implicit `taskwait` of the loop (meaning that they don't wait for the other tasks to finish before creating the next set).

Listing 9: v7: point task decomposition with taskloop grainsize(1) nogroup

```
91      /* Calculate points and save/display */
92      #pragma omp parallel
93      #pragma omp single
94      for (int row = 0; row < height; ++row) {
95          #pragma omp taskloop firstprivate(row) grainsize(1) nogroup
96          for (int col = 0; col < width; ++col) {
97              {
98              complex z, c;
```

3.2 *Row* strategy using *OpenMP* `task`

To implement the *row* parallelization strategy we defined a task for each iteration of the `row` loop as shown in listing 10.

Listing 10: v8: row task decomposition

```
91      /* Calculate points and save/display */
92      #pragma omp parallel
93      #pragma omp single
94      for (int row = 0; row < height; ++row) {
95          #pragma omp task firstprivate(row)
96          for (int col = 0; col < width; ++col) {
97              {
98              complex z, c;
```

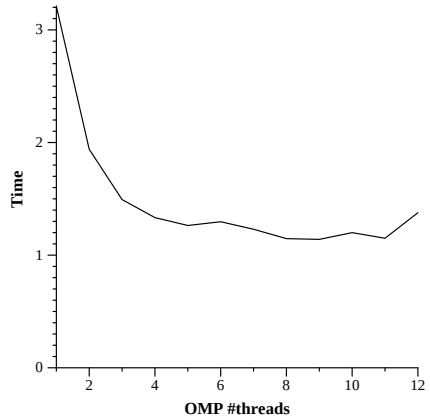
3.3 `for`-based parallelization

We used `#pragma omp parallel for collapse(2)` to collapse the two levels of the for loops.

Listing 11: for-based task decomposition

```
91      /* Calculate points and save/display */
92      #pragma omp parallel for collapse(2) schedule(guided, 1)
93      for (int row = 0; row < height; ++row) {
94          for (int col = 0; col < width; ++col) {
95              {
96              complex z, c;
```

4 Performance evaluation



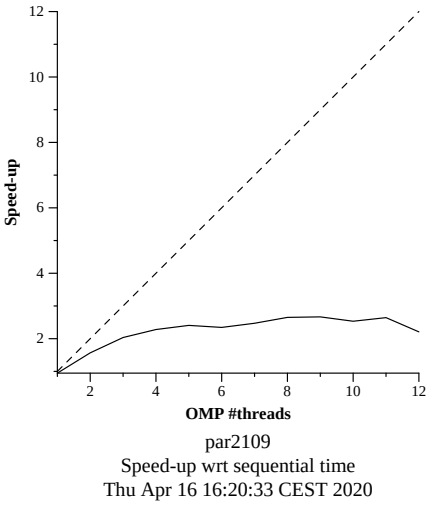
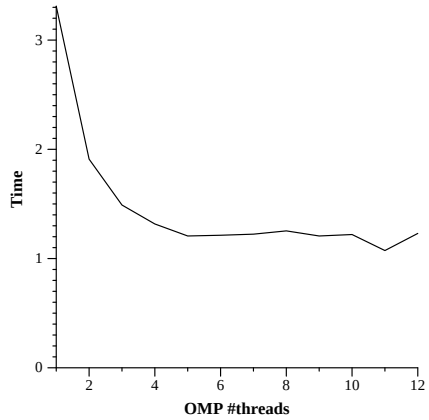


Figure 4: Strong scalability analysis v1 (parallel inside for)



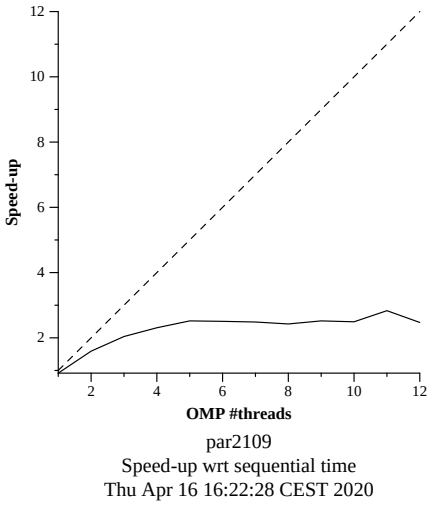
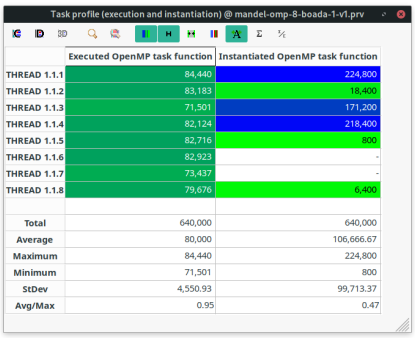


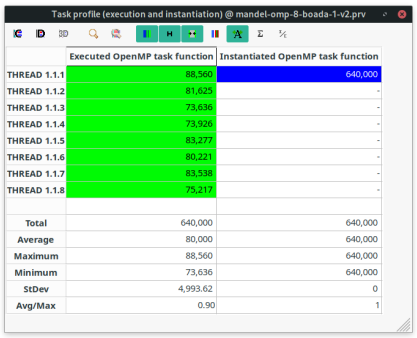
Figure 5: Strong scalability analysis v2 (taskwait)

In figures 4 and 5 we can see that there is a little improvement from version 1 to version 2, probably due to the reduced overhead of having the same thread create all the tasks. But the slope of the speedup behaves similarly and is quite flat from 5 threads on-wards.



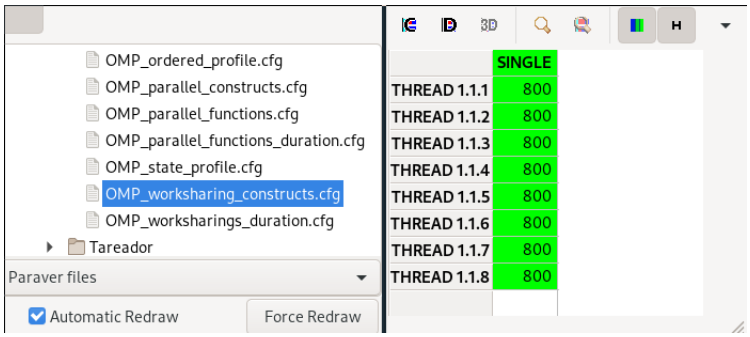
	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	84,440	224,800
THREAD 1.1.2	83,183	18,400
THREAD 1.1.3	71,501	171,200
THREAD 1.1.4	82,124	218,400
THREAD 1.1.5	82,716	800
THREAD 1.1.6	82,923	-
THREAD 1.1.7	73,437	-
THREAD 1.1.8	79,676	6,400
Total	640,000	640,000
Average	80,000	106,666.67
Maximum	84,440	224,800
Minimum	71,501	800
StDev	4,550.93	99,713.37
Avg/Max	0.95	0.47

Figure 6: Paraver `tasks_profile` v1



	Executed OpenMP task function	Instantiated OpenMP task function
THREAD 1.1.1	88,560	640,000
THREAD 1.1.2	81,625	-
THREAD 1.1.3	73,636	-
THREAD 1.1.4	73,926	-
THREAD 1.1.5	83,277	-
THREAD 1.1.6	80,221	-
THREAD 1.1.7	83,538	-
THREAD 1.1.8	75,217	-
Total	640,000	640,000
Average	80,000	640,000
Maximum	88,560	640,000
Minimum	73,636	640,000
StDev	4,993.62	0
Avg/Max	0.90	1

Figure 7: Paraver `tasks_profile` v2

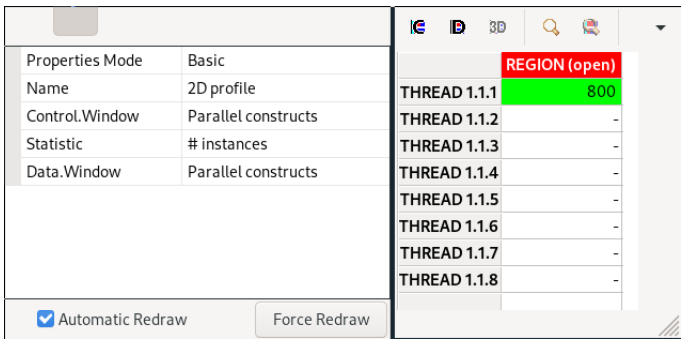


	SINGLE
THREAD 1.1.1	800
THREAD 1.1.2	800
THREAD 1.1.3	800
THREAD 1.1.4	800
THREAD 1.1.5	800
THREAD 1.1.6	800
THREAD 1.1.7	800
THREAD 1.1.8	800

Figure 8: Paraver `SINGLE` v1

	SINGLE
THREAD 1.1.1	1
THREAD 1.1.2	1
THREAD 1.1.3	1
THREAD 1.1.4	1
THREAD 1.1.5	1
THREAD 1.1.6	1
THREAD 1.1.7	1
THREAD 1.1.8	1
Total	8
Average	1
Maximum	1
Minimum	1
StDev	0
Avg/Max	1

Figure 9: Paraver `SINGLE` v2



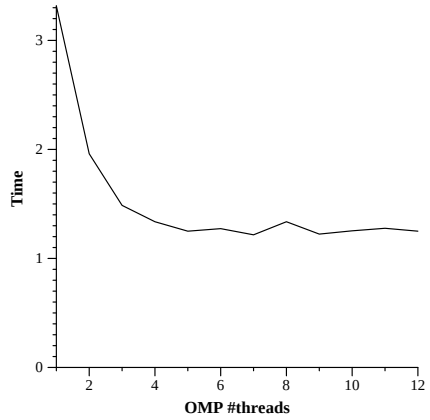
	REGION (open)
THREAD 1.1.1	800
THREAD 1.1.2	-
THREAD 1.1.3	-
THREAD 1.1.4	-
THREAD 1.1.5	-
THREAD 1.1.6	-
THREAD 1.1.7	-
THREAD 1.1.8	-

Figure 10: Paraver `PARALLEL` v1

	REGION (open)
THREAD 1.1.1	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	-
Total	1
Average	1
Maximum	1
Minimum	1
StDev	0
Avg/Max	1

Figure 11: Paraver `PARALLEL` v2

In figures 6 to 11 we can see how for version 1 there are 800 parallel constructs and 800 singles for each thread while for v1 there is one parallel construct and 1 single for each thread. In v2, all tasks are created by the same thread.



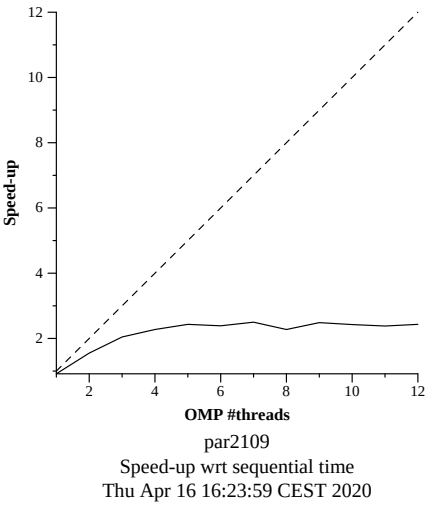
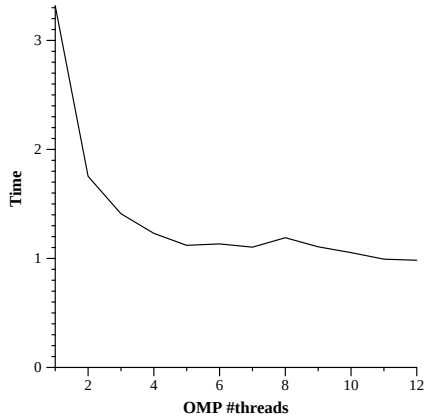


Figure 12: Strong scalability analysis v3 (taskgroup)



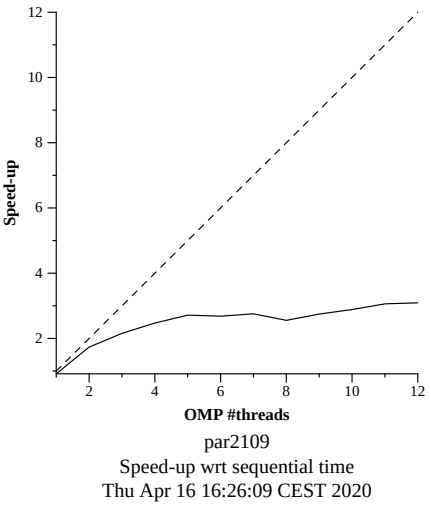
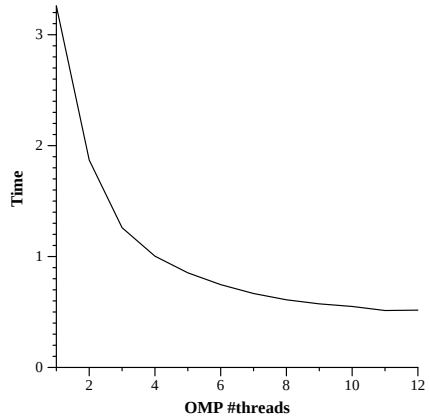


Figure 13: Strong scalability analysis v4 (no taskwait)

We can see that there is no difference in the plots between v2 (figure 5) and v3 (figure 12), just as we expected given that there are no nested tasks created and therefore `taskwait` and `taskgroup` should behave the same. we can see that removing `taskwait` (v4 in figure 13) offers a slight increase in performance more noticeably in from 8 threads on-wards.



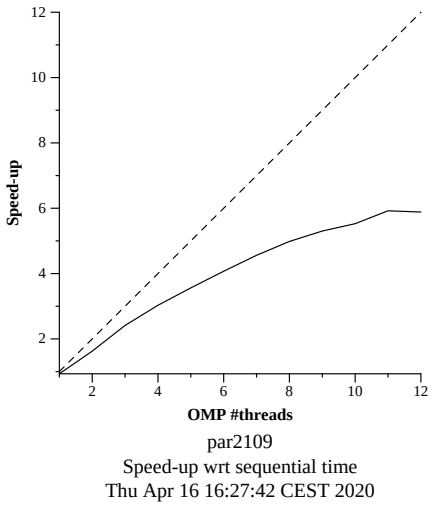
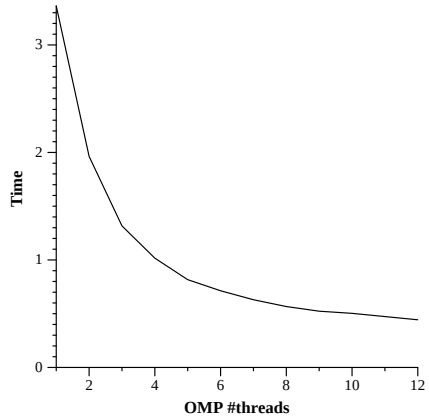


Figure 14: Strong scalability analysis v5 (taskloop)



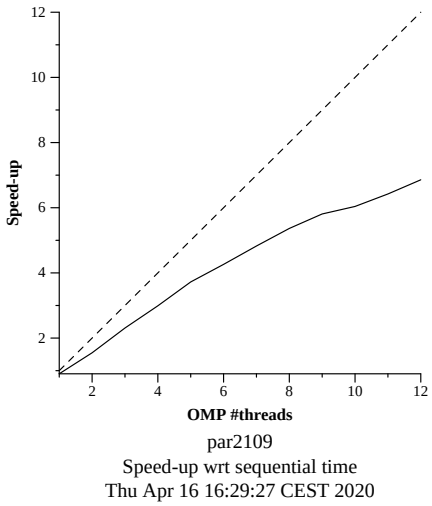
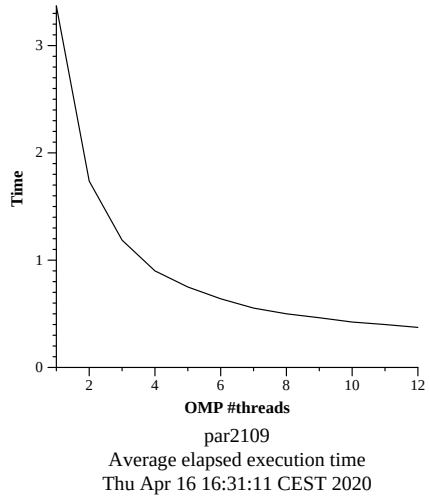


Figure 15: Strong scalability analysis v6 (taskloop grainsize(1))

Figure 14 shows that the usage of `taskloop` improves the performance significantly giving a speed up much closer to the ideal when compared to the previous versions. Using a grainsize of 1 be obtain even better results as shown in figure 15.



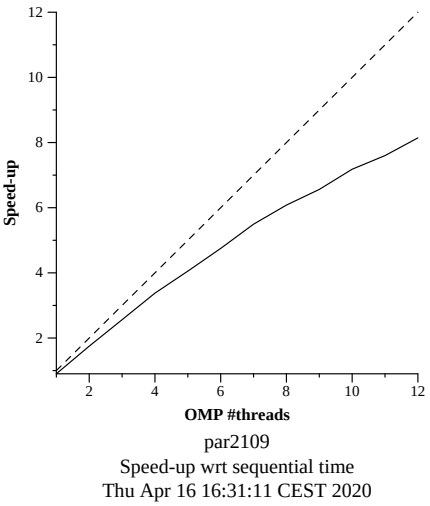
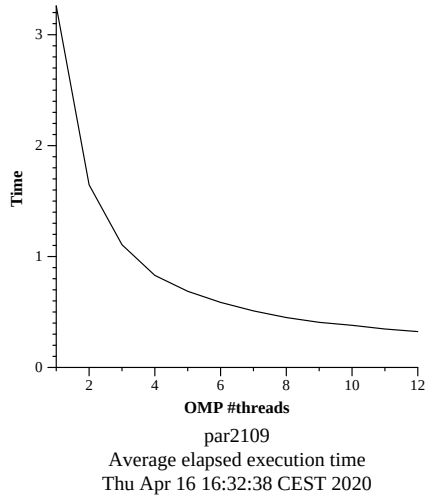


Figure 16: Strong scalability analysis v7 (taskloop grainsize(1) nogroup)



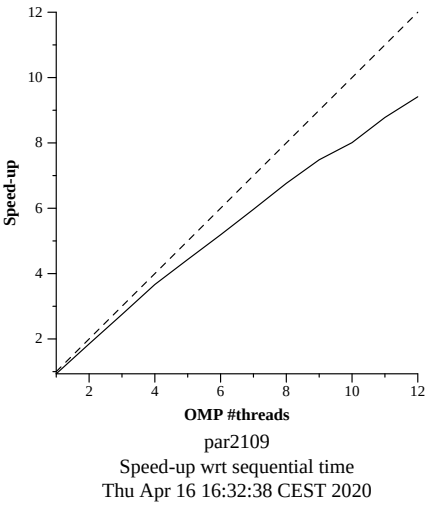


Figure 17: Strong scalability analysis v8 (row decomposition)

The addition of `nogroup` doesn't seem to have much effect on the execution time of the program as we can see in figure 16. However the row decomposition offers the best results as seen in figure 17.

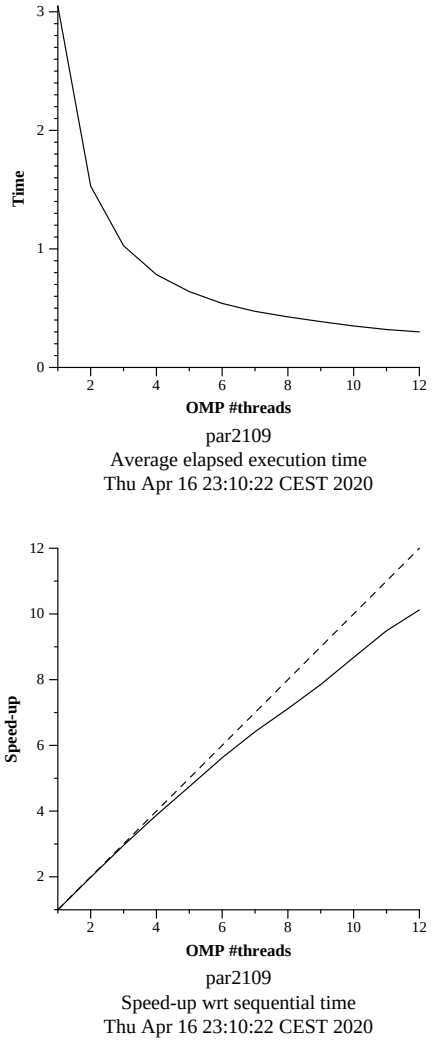


Figure 18: Strong scalability analysis
(for schedule(dynamic, 25))

In figure 18 we show the strong scalability analysis of the `for`-based decomposition with dynamic scheduling and a chunk size of 25. This approach gives even better results that the row based decomposition. In section 4.2 we discuss how we settled on these parameters.

4.1 Taskloop grainsize analysis

We tried different grainsize value for the taskloop decomposition. To obtain the results we ran 10 repetitions of `./mandel-omp -i 10000` for each grainsize value. The mean results of execution times are shown in table 1.

Table 1: Execution times with different grainsizes

`grainsize`	Execution time (s)
1	0.317474
2	0.361412
5	0.372336
10	0.256939
25	0.208336
50	0.196701
100	0.183566
200	0.181784
400	0.185489
800	0.185322

Figures 19 and 20 show the data of table 1 on a linear scale and logarithmic scale respectively. We can see that the best results are obtained with grainsizes above 100. The best result was with a grainsize of 200, although the difference between the execution times with other grainsize above 100 is not significant.

This results show that a small granularity affects negatively the execution time, this is probably due to the load imbalance we saw when analyzing the program with tareador. This load imbalance is evened out between the tasks when we have bigger granularity. Given that a grainsize of 800 seems to give similar execution times than the best cases, it is worth considering *row* based task decomposition instead of *point* based (since 1 row = 800).

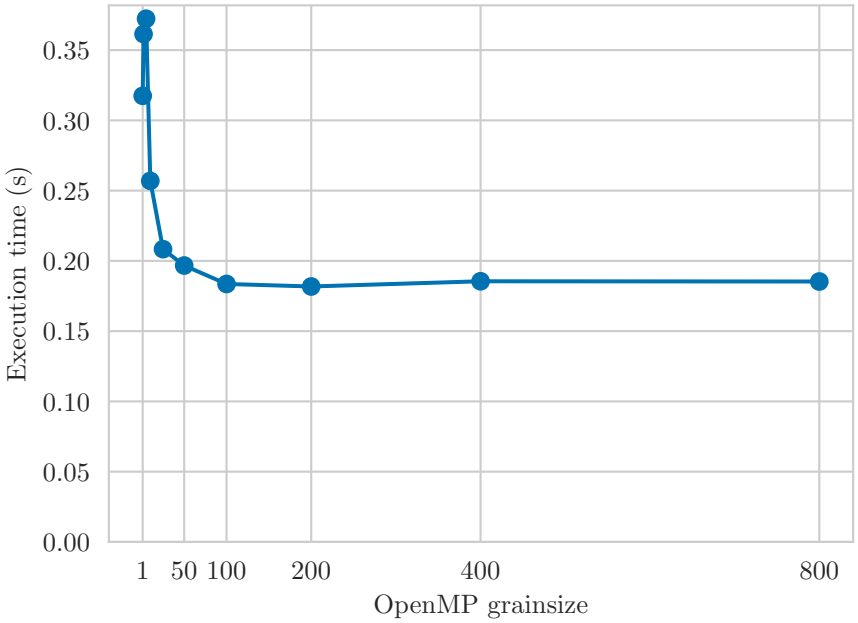


Figure 19: Plot of execution time for different grainsizes

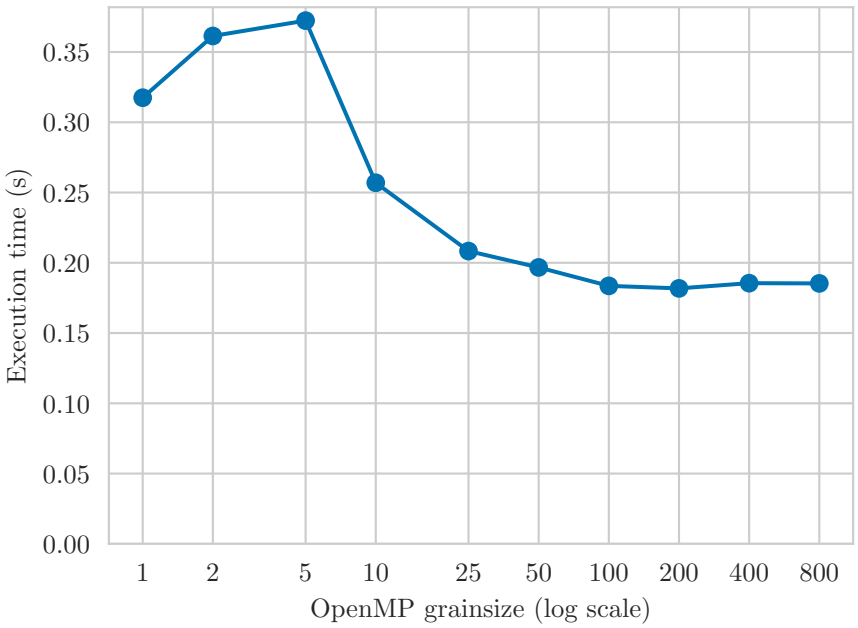


Figure 20: Plot of execution time for different grainsizes (log scale)

4.2 for scheduling analysis

We ran the for task decomposition of the program with different scheduling schemes and chunk-sizes. The results obtained can be seen in table 2.

Table 2: Execution times with different scheduling and chunk-sizes

	Scheduling		
chunk-size	dynamic	guided	static
1	0.198096	0.221012	0.186173
2	0.191246	0.220252	0.185386
5	0.186887	0.222712	0.201038
10	0.185859	0.222406	0.230312
25	0.185232	0.220492	0.234002
50	0.186607	0.219147	0.341946
100	0.187840	0.222233	0.557168
200	0.185515	0.220431	0.416928
400	0.189663	0.221099	0.256937
800	0.187154	0.219909	0.183334

Figures 21 and 22 show the data of table 2 on a linear scale and logarithmic scale respectively. We can see that chunk-size does not affect execution time with dynamic or guided scheduling and that it severely affects the execution time with static scheduling. Guided scheduling provides no benefits, which makes sense given that all tasks have the same cost. The best results are obtained with static scheduling and a chunk size of 800, although similar execution times are obtained with static 1, 2 and dynamic scheduling.

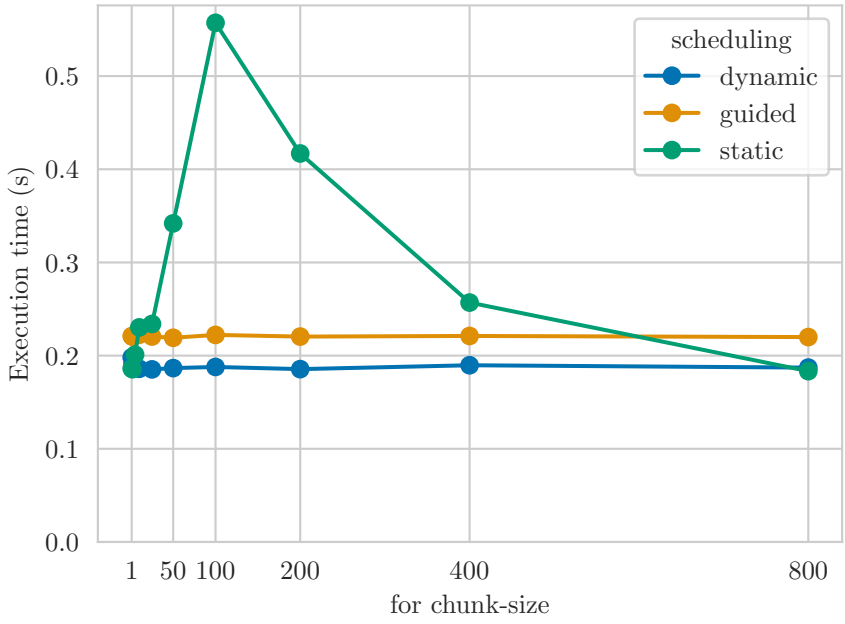


Figure 21: Plot of execution time for different chunk-sizes and scheduling

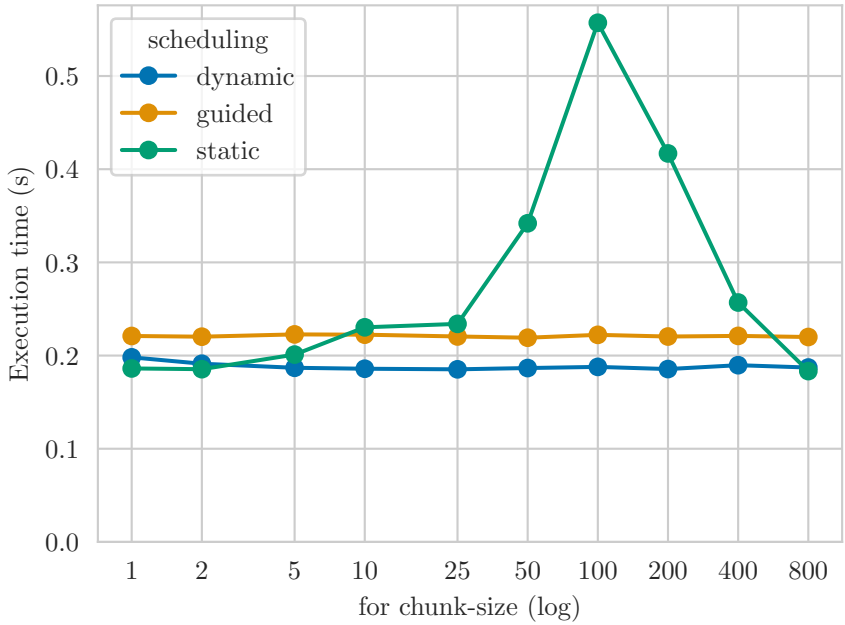


Figure 22: Plot of execution time for different chunk-sizes and scheduling (log scale)

The fact that static scheduling has much longer execution times for certain-chunk sizes is due to the load imbalance we mentioned in the *tareador* analysis (section 2) and in the previous section when discussing the taskloop gransize. One thing to notice is that the computation of a point takes longer in the *edges* of the set, therefore, if we use a static scheduling and the chunk size makes it so that the one thread gets assigned a lot of *edge* points, the execution time will be affected as it happened with our chunk size of 100.

In our case we used always the same image of the set (shown in figure 23), but given that the program can be used to compute other sections of the set the best option is dynamic scheduling with small chunks, but not too small as to minimize the load imbalance of the tasks.

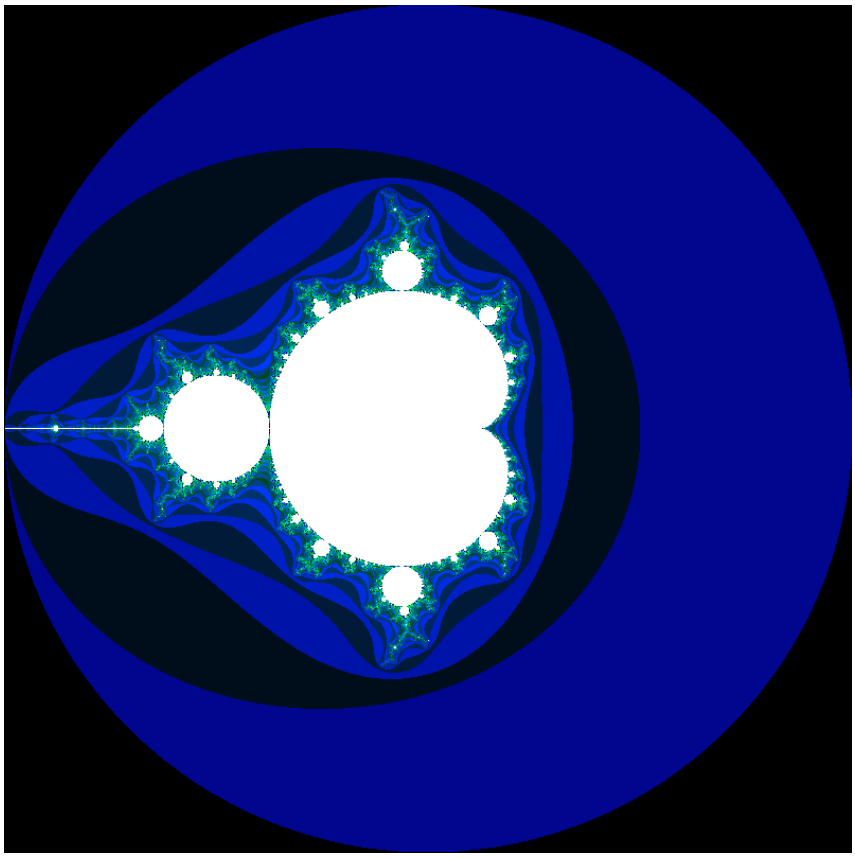


Figure 23: Mandelbrot set with maxiter=10000

5 Conclusions

As we have seen, the Mandelbrot computation is embarrassingly parallel since every point can be computed independently from the others. However, there is a load imbalance in the tasks due to the nature of the computation which makes higher granularity approaches unpractical.

Although we can achieve pretty good speedup values with a taskloop with granularity 1 and nogroup, the row decomposition approach gives better results. The best results we obtained were using `for`-based decomposition with dynamic scheduling and a chunk-size of 25. This chunk size is the one that gave the best results in our experiments, but it may no be ideal for other regions of the image, we think however that this size minimizes the load imbalance of the tasks in most cases.