PAR Laboratory Assignment

Course 2018/19 (Fall semester)

Lab 2: Embarrassingly parallelism with OpenMP: Mandelbrot set

Pablo Vizcaino, Guillem Ramírez

User: Par2206

October 30, 2018

Contents

1 Introduction **2**

2 Parallelization strategies analysis **3**
2.1 Point decomposition 3
2.2 Row decomposition 4
2.3 Preliminary comparison between strategies 5
2.4 Graphical version analysis 5

3 Parallelization strategies implementation **7**
3.1 Point decomposition in *OpenMP* 7
3.2 Row decomposition in *openMP* 9

4 Performance evaluation **11**

5 Conclusions **12**

1 Introduction

In this laboratory we are going to study the task decomposition and the parallelization strategies of the computation of the Mandelbrot set.

The Mandelbrot set is a set of complex numbers C obtained from iterating the equation $Z = z^2 + C$. You can see a visualization of the result from the computation in the figure 1

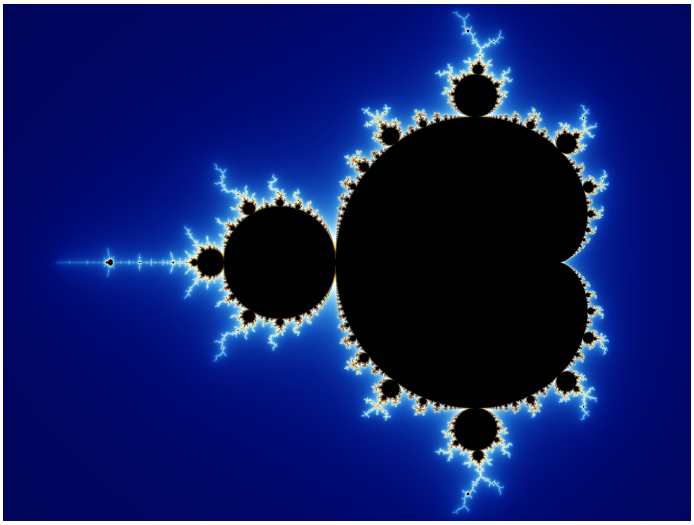

Figure 1: Mandelbrot visualization

2 Parallelization strategies analysis

Our first objective is to analyze the data dependency within the Mandelbrot computation. With this purpose we decided to explore row and point granularity. All executions with tareador use the option -w 8, which means 8 iterations per loop.

2.1 Point decomposition

In this section we are presenting the point decomposition strategy , where we are creating at each inner loop iteration a task.

Point decomposition analysis with *tareador*

We just needed to add tareador start task directive in the beginning of the inner for loop and a tareador end task at the end of the inner for loop. You can find the source code at *code/tareador/mandel-tar-point.c*. Listing 1 shows the structure.

Listing 1: Tareador instrumentation at point granularity

```
/* Calculate points and save/display */
for (row = 0; row < height; ++row) {
    for (col = 0; col < width; ++col) {
        tareador_start_task("point");
                    .
                    .
                    .
        tareador_end_task("point");
```

The result obtained from executing *run-tareador.sh* with *mandel-tar* objective (non-graphical version) is:

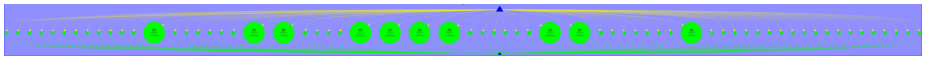

Figure 2: Tareador graph with point granularity

From the graph obtained we can observe that there are 64 non-dependant tasks, one for each point: $64 == 8 * 8$. Due to the big number of tasks the graph is in a long shape, you can view the source image at *img/tareador/mandel-tar-point-graph.png*. From the graph we can conclude that working with a big number of tasks can carry out to overheads.

2.2 Row decomposition

In this section we are presenting the row decomposition strategy, where we are creating a task for each row of points.

Row decomposition analysis with *tareador*

For this analysis we just need to add tareador directives in the middle of the for loops. You can find the source code at *code/tareador/mandel-tar-row.c*. Listing 2 shows the structure.

Listing 2: Tareador instrumentation at row granularity

```
/* Calculate points and save/display */
for (row = 0; row < height; ++row) {
    tareador_start_task("row");
    for (col = 0; col < width; ++col) {
                .
                .
                .
    }
    tareador_end_task("row");
}
```

The result from executing *run-tareador.sh* with *mandel-tar* objective (non-graphical version) is:

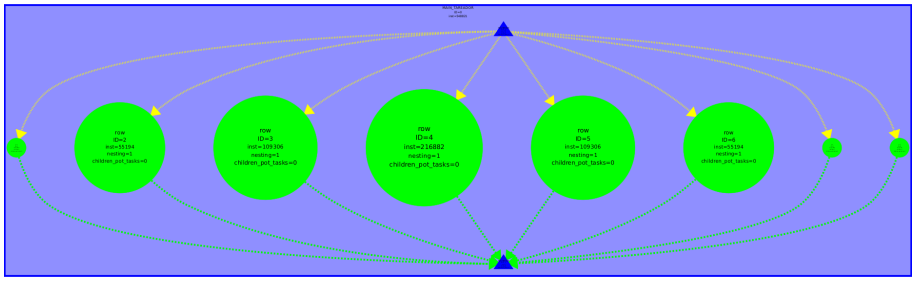

Figure 3: Tareador graph with row granularity

From the graph we can observe that we get 8 non-dependant tasks, one for each row, as expected. With less tasks we may have less overheads from the previous version but we may not be taking profit of all threads. This question will be evaluated in the following sections. The source image is at *img/tareador/mandel-tar-row-graph.png*.

2.3 Preliminary comparison between strategies

After analyzing the two versions we find out that with a point based strategy the number of tasks is: w^2 . In our case w is 8 so the number of tasks is 64, this number can increase to really high values in default executions and can be really problematic due to task creation and work distribution overheads. In the other hand we find out that row granularity the number of task linearly increase with w, which may be a more affordable value. This general guidelines and ideas will be taken into account for the next sections.

2.4 Graphical version analysis

Once the granularity of the computation itself is evaluated, we will be analyzing the graphical version. For this purpose we will be using the same source code as in the previous subsection (*code/tareador/mandel-tar-row.c*) but we will be compiling using the *mandeld-tar* Makefile target.

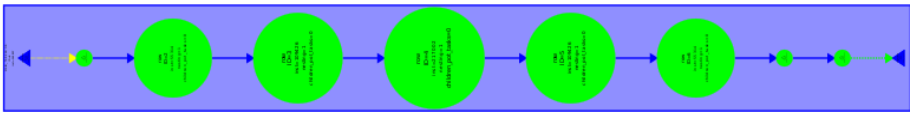

Figure 4: Tareador graph with mandeld target

The image is rotated for document structuring purpose. The full non-rotated image can be found in *img/tareador/mandeld-tar.png*.

We can observe that due to dependencies the graph is now serialized, if we look into the edges between tasks in tareador we can see that a graphics library variable is now needed between iterations. This is causing the serialization of the graph.

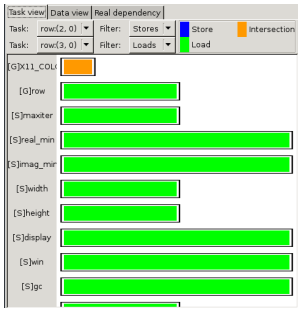

Figure 5: Variable usage between tasks

If we look into the code we can find that the the part causing serialization of the task graph is:

```
#if _DISPLAY_
        /* 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);
        }
#else
```

Specially what is causing serialization is the inner if statement. In the following sections we will be parallelizing the code, for protecting this part we will be adding the following clause before the if statement so that we are telling openMP that only one thread can be working at any time.

```
#pragman omp critical
if (setup_return == EXIT_SUCCESS) {
```

3 Parallelization strategies implementation

In this section we are going to present the implementation of both strategies in *OpenMP*

3.1 Point decomposition in *OpenMP*

For this strategy we went through a few versions. Starting with the initial version and making improvements.

Initial version

The initial version was given in the assignment and its:

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

In this version we introduced the firsprivate pragma which tells the compiler that the col variable inside the construct is a new variable initialized to the same original variable value.

Also notate that in the full code *code/omp/mandel-omp-initial.c* we added the *#pragma omp critical* clause in the region that generated dependencies, see: *graphical version analysis*.

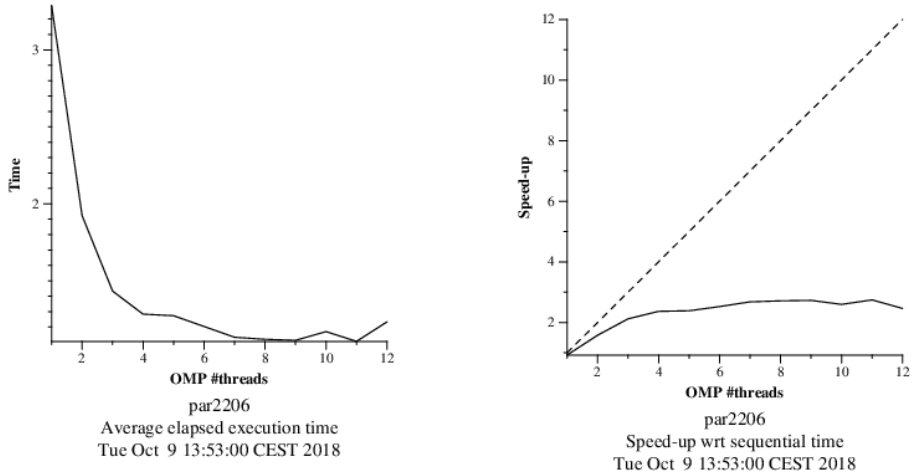

Figure 6: Strong scalability plots for initial version

We can observe that the speedup poorly improves with the increase of #threads that's due to the version not being optimized as we create a lot of tasks and the work distribution takes into serious overheads. We can evaluate the overheads measuring in paraver the number of bursts (in this case, 800)

Improved version

In this version we move out the loop *parallel and single pragmas* and in consequence we also privatize the row variable.

```
#pragma omp parallel
#pragma omp single
/* Calculate points and save/display */
for (int row = 0; row < height; ++row) {
    for (int col = 0; col < width; ++col) {
        #pragma omp task firstprivate(row, col)
```

In this version, as we move the parallel region and the work-distribution outside the loop we don't invoke it at each iteration of row, so we only do it once. This slightly improves the performance as you can see in the strong scalability plots.

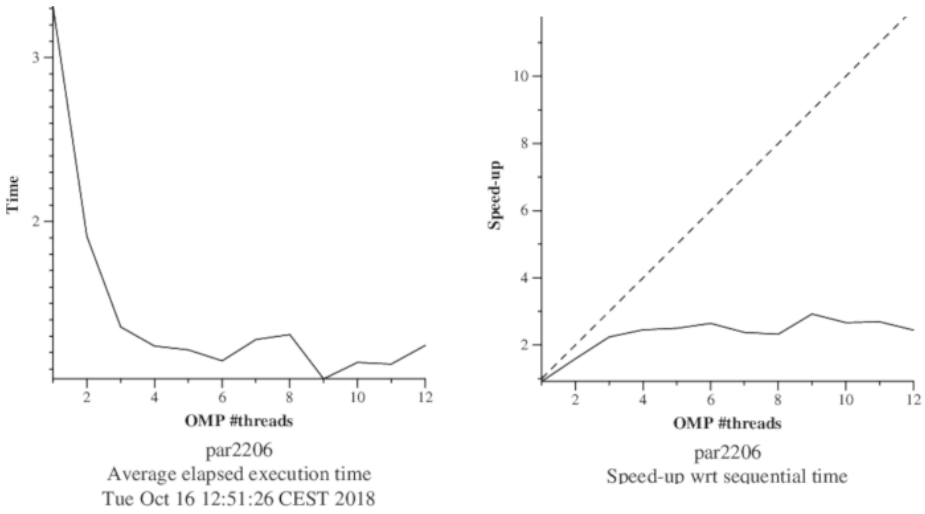

Figure 7: Strong scalability plots for the improved version

Taskloop version

In this version we introduce the taskloop pragma. This statement distributes more efficiently each task's work in a loop. Nonetheless, it creates and implicit taskgroup barrier. Since we observed that all iterations are independent, there is no need for synchronizations. To remove them we added a *nogroup* clause.

```
#pragma omp parallel
#pragma omp single
/* Calculate points and save/display */
for (int row = 0; row < height; ++row) {
    #pragma omp taskloop firstprivate(row) num_tasks(width) \
    nogroup
    for (int col = 0; col < width; ++col) {
```

If we look the strong scalability plots and we compare them to the previous ones we can see how big the improvement is. This is due to the optimization in the task creation and distribution and also in the elimination of the barrier.

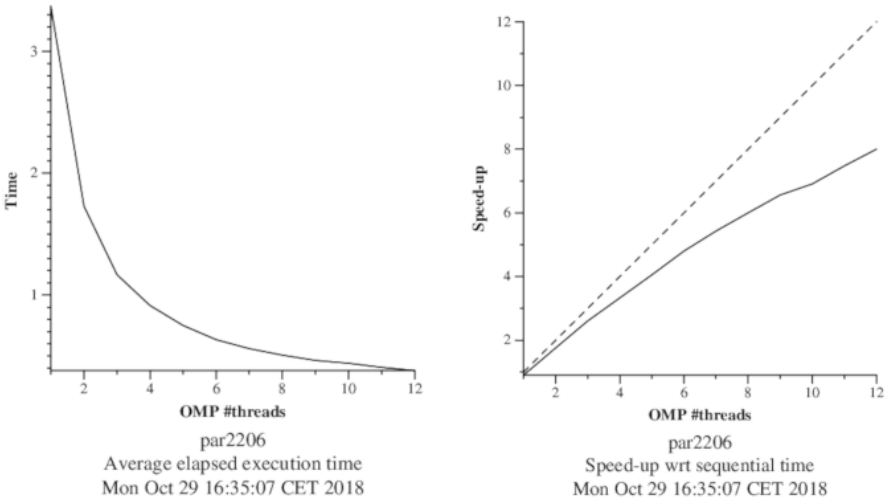

Figure 8: Strong scalability plots for the taskloop version

3.2 Row decomposition in *openMP*

In this section we implemented the row decomposition, which we analyzed in section: *Row decomposition*

As we observed in the previous sections, reducing overheads is the key to a better performance; so we expect to reduce the execution time when we reduce the granularity to one task for each row.
You can find the full code at *code/omp/mandel-omp-row.c*. Here you have the pragma setup:

```
#pragma omp parallel
#pragma omp single
/* Calculate points and save/display */

#pragma omp taskloop num_tasks(height) nogroup
 for (int row = 0; row < height; ++row) {
     for (int col = 0; col < width; ++col) {
```

To achieve row granularity, we placed the pragma outside both loops, and used the clause num_tasks(height) to set the granularity to one task per row. We also kept the nogroup clause to eliminate the implicit barrier.

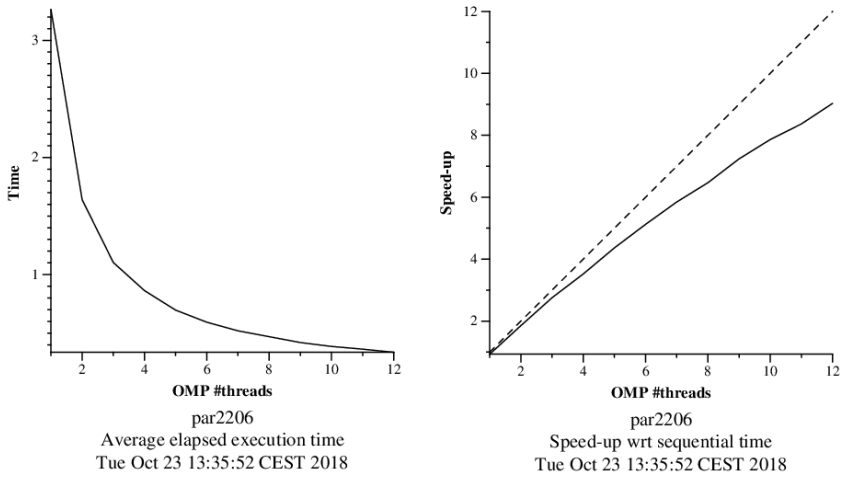

Figure 9: Strong scalability plots for the Row-Granularity version

As can be seen in the Figure 9 the strong scalability has slightly improved compared to the previous versions. This fulfills our expectations. In the following section we will be comparing the execution times to confirm this improvement.

We also tried to change the granularity (p.e: one task each to rows) modifying the num_tasks clause and as you can see in the following figure the reduction of overheads doesn't compensate the loss in parallelization. Notice that this is the opposite behaviour than in the point version. This is due to, the smaller the granularity in the point version, the more similar to the row one it is.

Point		Row	
Tasks/row	Time (s)	Tasks/col	Time (s)
800	0.513	800	0.47
400	0.474	400	0.485
200	0.481	200	0.489
100	0.474	100	0.509
50	0.488	50	0.524
25	0.47	25	0.673
5	0.451	5	2.515
1	0.46	1	3.481

4 Performance evaluation

When obtaining the execution times of both versions with default parameters we get *0.51 s* for the point granularity and *0.47 s* for the row granularity. To observe better the difference, we also executed both versions with *maxiter* variable equal to 300000. The results obtained with this parameter are *95.09 s* for the point version and *13.02 s* for the row version. We can see that as we increase the problem size the time difference is more noticeable.

First we are going to evaluate the percentage of time that each version wastes on task creation and synchronizations. We executed point and row version (both with no group) with paraver instrumentation and used omp_tasks configuration file. We can see in the following table that for each thread the time consumed by the overheads in the point version is greater than in the row one. In consequence, in the row version, threads spends more time calculating and the execution time is lower.

	Row strategy		Point strategy	
	Code exectuing	Taskloop construct	Code executing	Taskloop construct
THREAD 1.1.1	99.97	0.03	97.55	2.45
THREAD 1.1.2	99.9	0.1	97.45	2.55
THREAD 1.1.3	99.8	0.2	98.99	1.01
THREAD 1.1.4	99.81	0.19	97.58	2.42
THREAD 1.1.5	99.97	0.03	97.5	2.5
THREAD 1.1.6	99.98	0.02	97.63	2.37
THREAD 1.1.7	100	0	97.37	2.63
THREAD 1.1.8	99.99	0.01	97.33	2.67
Average	99.93	0.08	97.68	2.32

5 Conclusions

In this lab assignment we understand the importance of the work-distribution and task creation overheads in the parallelization process. We saw that the best result is obtained when there is a balance between parallelization and the reduction of overheads.It's important to set the grain size properly to optimize the execution time, because it affects directly to the parallelization and the overheads.

We also learned the different *OpenMP* pragmas that we can use to parallelize loop-based codes and to set different grainsize and control the barriers and synchronizations.

Another key factor that helped us to compare different versions and to confirm the results that we were expecting is the *Paraver* instrumented execution of the codes.