PAR Laboratory Deliverable

Course 2019/20 (Fall semester)

Lab 3: Embarrassingly parallelism with OpenMP: Mandelbrot set

Rafel-Albert Bros Esqueu, Joan Vinyals Ylla-Català

User: Par2201

October, 2019

Contents

1 Introduction **2**

2 Task decomposition and granularity analysis **2**

3 *Point* decomposition in *OpenMP* **4**

3.1 *Point* strategy implementation using *OpenMP* `task` 4

3.2 Granularity control with `taskloop` 5

4 *Row* decomposition in *OpenMP* **6**

4.1 *Row* strategy implementation using *OpenMP* `task` 6

4.2 Granularity control with `taskloop` 6

5 `for`–based parallelization **7**

6 Conclusions **9**

1 Introduction

During the second half of 20th century mathematicians took profit from the rapidly advancing computing power to draw computer images of several functions they were analysing. What they discovered deeply surprised them: incredibly complicated fractal structures similar to those found in nature. The most common one being the **Mandelbrot set**, which is widely defined as the set of all the complex numbers c for which the complex numbers of the sequence $z_n = c + n$ remain bounded in absolute value. In other words, the complex values of c under the sequence $z_n = c + n$ does not escape infinity.

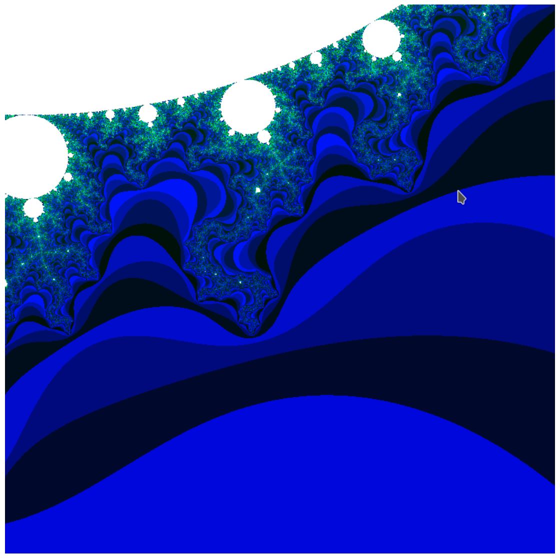


Figure 1: Mandelbrot set.

As the value of each point of a specific structure following the **Mandelbrot set** only depends on its own position, the calculation of every point is embarrassingly parallel. That is what interests us for this specific deliverable.

2 Task decomposition and granularity analysis

In order to analyze the potential parallelism for two given possible task granularities that can be exploited in the program created from the code `mandel-seq.c` we will use *Tareador* as in previous deliverables. The two options to be analyzed are: *Point:* every task is the computation of a single point `(row,col)`; and *Row:* every task is the computation of a whole row; in both cases working with the **Mandelbrot set**.

When using **row strategy** for each row of the matrix we are creating an individual task, therefore tasks are bigger as seen in Figure 3. On the other hand, with **point strategy** we are creating a single task per every element of the matrix as seen in Figure 2. Of course this means tasks are smaller than with row strategy. Having more tasks will mean that in order to schedule all those tasks the chances for large amounts of overhead will increase.

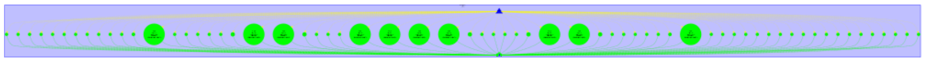


Figure 2: Point task graph.

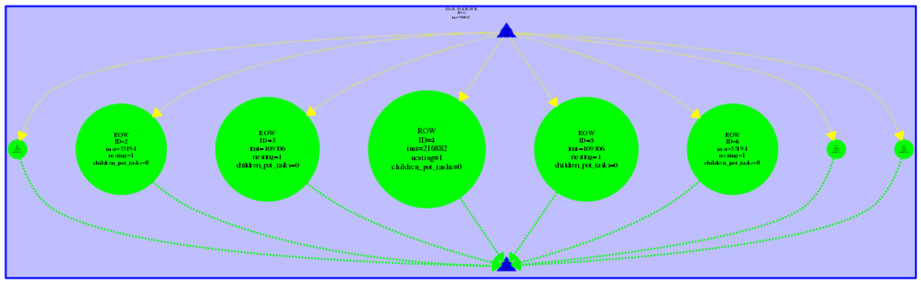


Figure 3: Row task graph.

Furthermore, notice how there are no dependences between the tasks generated by the main loop in neither option, and also in both cases not all tasks have the same size. Apparently nodes closer to the middle have greater granularities than those in the outskirts, we deduce that this is due to the fractal shape we are dealing with.

```
/* 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 4: Zone of code that needs to be protected.

In program `mandeld-tar` the serialization is caused by the piece of code seen in Figure 4. Looking deeper into the code we observe that the library-based variable `X11_COLOR_fake`, used in both functions `XSetForeground` and `XDrawPoint`, is the reason why this function causes the serialization of the execution.

In order to get the program executed as expected and to avoid data races we will protect the functions with `#pragma omp critical`, shown in Figure 5.

After this process we finally got the desired result, which is the same we got from the previous program. That is to say, a potential parallel execution of all tasks with no dependences inside the main loop.

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

Figure 5: Parallel version of the code, protected with a critical clause.

3 *Point* decomposition in *OpenMP*

In this section we will explore different options in the *OpenMP* tasking model to express the *Point decomposition* for the Mandelbrot computation program.

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

One of the simplest ways to parallelize the code that creates a **Mandelbrot set** by using a *Point* task-based strategy is the structure seen in Figure 6 (refer to the code `mandel-omp-point-task-1.c` to see it all).

```c
    #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)
```

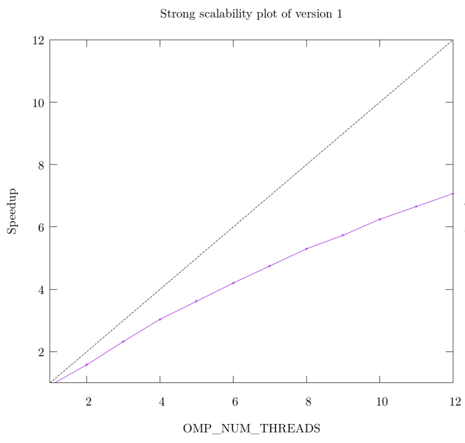
Figure 6: First parallel version based on *point* strategy.

Take into consideration that the `firstprivate` clause is important to avoid data races. Refer to Table 1 for specific results when executing the serial version, the parallel version with a single thread, and the parallel version with eight threads. Notice that the difference in execution time between the serial version and the parallel one with a single thread is a *slowdown* caused by OpenMP overhead.

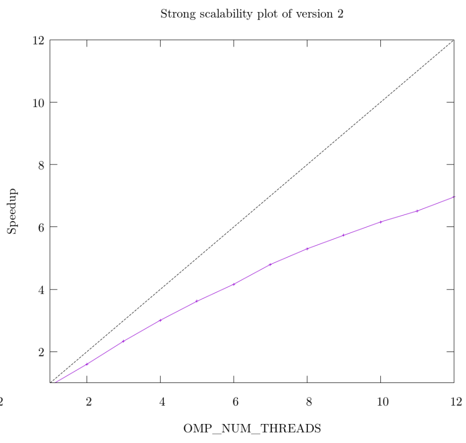
Version	Time (s)
Serial	3.15
point-1	3.41
point-8	1.41

Table 1: The timing for different executions of *Point* decomposition using tasks.

Next we are lead towards a very similar structure based on the same *Point* strategy, but slightly more complicated than the previous version. The only difference is that now the synchronization construct `#pragma omp taskwait` has been added. This addition implies that right at that point the program will have to wait until all tasks for each one of the rows finish in order to keep execution, generating a new bunch of tasks right after advancing one single iteration of the `row` loop.



(a) Strong scalability with `firstprivate`.



(b) Strong scalability with `taskwait`.

Figure 7

The plots of both *Point* implementations when executing an amount of threads raging from 1 to 12 are to be seen in Figure 7. Apparently, in both cases the results are quite similar. Pay attention to the fact that for more threads the speed-ups have bigger decreases. We consider that the addition of the `taskwait` clause is useless at least for this specific amount of threads as there is no change between executions.

3.2 Granularity control with `taskloop`

When trying to control the granularity with the clause `taskloop` we will use the code found in program `mandel-omp-point-taskloop.c`, and the piece of code of interest is shown in Figure 8. The word `NUM_TASKS` has to be changed into the specific amount of tasks we want.

```c
#ifndef NUM_TASKS
#define NUM_TASKS 64
#endif
```

```c
    #pragma omp parallel
    #pragma omp single
    for (int row = 0; row < height; ++row) {
        #pragma omp taskloop firstprivate(row) num_tasks(NUM_TASKS)
        for (int col = 0; col < width; ++col) {
```

Figure 8: Parallel version based on *point* strategy using `taskloop` clause.

4 *Row* decomposition in *OpenMP*

After analyzing the *Point decomposition* for the Mandelbrot computation program in the previous section we will do the same process but now exploring *Row decomposition*.

4.1 *Row* strategy implementation using *OpenMP* `task`

Observe the structure in Figure 9 to see how we have parallelized by rows the given code. The whole program is found in the file `mandel-omp-row-task.c`.

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

Figure 9: First parallel version based on *Row* strategy.

Through this process, we are generating tasks that will execute the inner loop several times, depending on the grainsize. We use the `private` clause to avoid data races, and the `firstprivate` clause to keep the initial value of the variable `row` every time we create a new task.

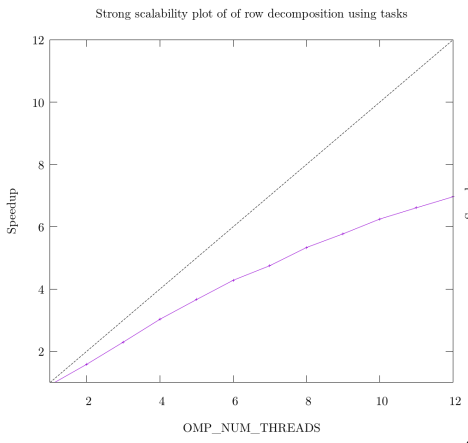
4.2 Granularity control with `taskloop`

When trying to control the granularity with the clause `taskloop` we will use the code found in program `mandel-omp-row-taskloop.c`, and the piece of code of interest is shown in Figure 10. The word `NUM_TASKS` has to be changed into the specific amount of tasks we want.

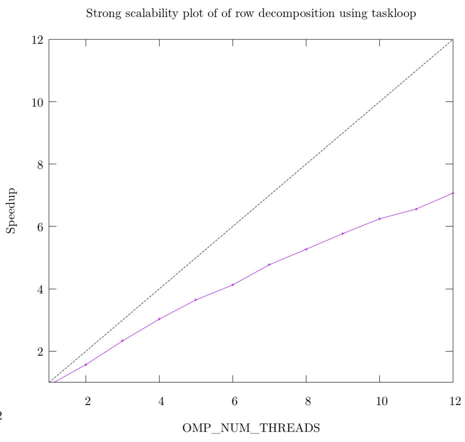
```
    /* Calculate points and save/display */
    #pragma omp parallel
    #pragma omp single

    #pragma omp taskloop num_tasks(NUM_TASKS)
    for (int row = 0; row < height; ++row) {
        for (int col = 0; col < width; ++col) {
            complex z, c;
```

Figure 10: Parallel version based on *Row* strategy using `taskloop` clause.



(a) Strong scalability of *Row* strategy.



(b) Strong scalability of *Row* strategy with the usage of `taskloop`.

Figure 11

In Figure 11 the plots of both the initial *Row* strategy analysed in previos section and the now planned *Row* strategy using the `taskloop` clause can be seen. Both are very similar and they share the same characteristics with the plots seen in Figure 7 back when we were working with *Point* strategy.

5 `for`–based parallelization

Finally, we will analyze the possibilities of the `for` directive in *OpenMP* with the **Mandelbrot set**. We have three different types of scheduling options when working with the `for` directive:

- **static:** All threads have a fixed scheduling and the same amount of tasks among all.
- **dynamic:** Every task generated is assigned to the first free thread.

- **guided:** The amount of tasks executed per every thread decreases along time in order to achieve a load balance.

The `for`-based command to parallelize the program we have been using is quite simple, as seen in Figure 12, which is a piece of code from file `mandel-omp-for.c`. The word `SCHEDULE` is to be changed by any of the three possible scheduling options: `static, dynamic` or `guided`.

```
for (int row = 0; row < height; ++row) {
  #pragma omp parallel for schedule(SCHEDULE)
  for (int col = 0; col < width; ++col) {
```

Figure 12: Basic code structure using the `parallel for` construct.

After explaining all three scheduling possibilities we next move into analysing them. We executed each schedule policy with a number of threads raging from 1 to 12, obtaining the plots seen in Figure 13.

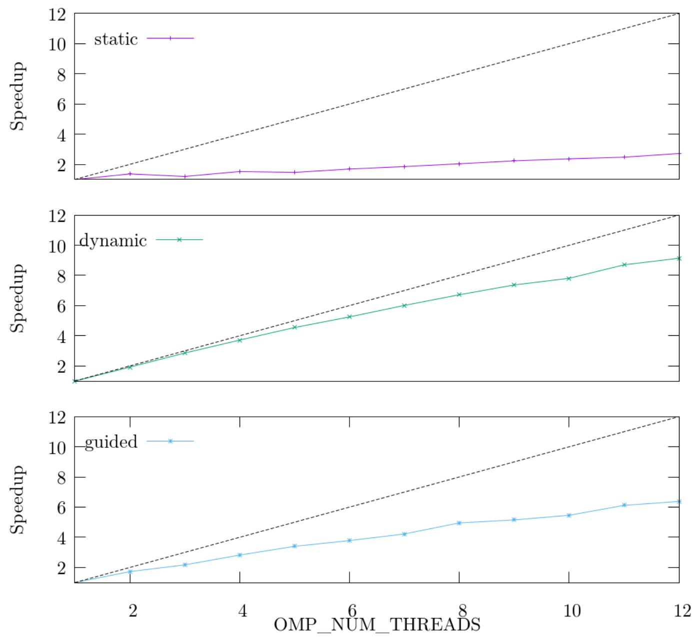


Figure 13: Strong scalability of the `mandel-omp-for.c` with the three different schedule policies explained and studied.

The `static` gets the worst results among all three policies, as there is large load unbalance between threads because all threads get the exact same number of tasks as explained. Next we have the `dynamic` schedule that gets the best results as tasks are assigned to threads as they become free. Finally, the `guided` schedule gets some results in the middle grounds, but still closer to the `dynamic` ones than the `static`.

6 Conclusions

Throughout this project we have get to know different strategies to parallelize a sequential code that works with a **Mandelbrot set**.

We have seen that different task generation strategies as well as grainsize can greatly alter results, and it is important to deeply analyze which are our expectations in order to decide which parallelization strategy is best. Still, a more comprehensive analysis should be done to identify possible differences among both *Point* and *Row* strategies that have not been detected due to the limitations of this project.