Parallelism
1202

# Lab 1: Embracing parallelism with OpenMP: Mandelbrot set

Héctor Ramón Jiménez

Alvaro Espuña Buxo

April 25, 2014
Facultat d'Informàtica de Barcelona

# 1 Reading activity

In this first part of the report we first briefly describe the basic formulation for the **Mandelbrot set** (see 1.3) and then its implementation in the sequential code available (`mandel-serial.c`).

## 1.1 Description

The **Mandelbrot set** is a particular set of points in the complex domain named after the mathematician **Benoit Mandelbrot**, who studied it and popularized it. The **Mandelbrot set**'s boundary is an easily recognizable two-dimensional fractal shape (see figure 1).

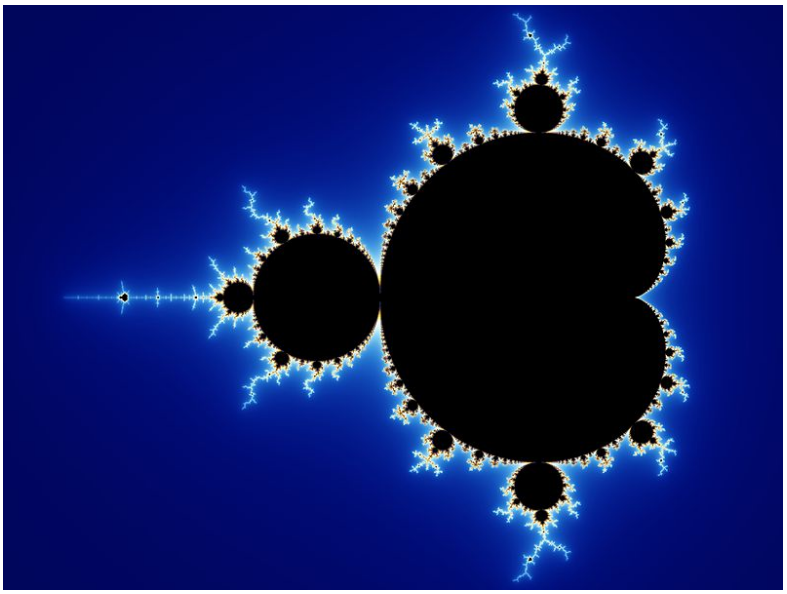

**Figure 1:** The Mandelbrot set boundary

Mandelbrot set images are made by sampling complex numbers and determining for each whether the result **tends towards infinity** when a particular mathematical operation is **iterated** on it. The real and imaginary parts of each number are treated as image coordinates. The pixels are colored according to how rapidly the sequence diverges.
More precisely, the **Mandelbrot set** is the set of values of $c$ in the complex plane for which the orbit of 0 under iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ remains bounded.

## 1.2 Algorithm

The simplest algorithm for drawing a picture of the Mandelbrot set is the following. We **discretize** the complex space in a set of points and each point corresponds to a **pixel** in a two dimensional plot. To color any such pixel, let $c$ (represented by the

complex variable $c$ at the `mandel-serial.c` code) be the midpoint of that pixel. Then we calculate $z_0$, $z_1$, ... (stored in the complex variable $z$) and beyond until **divergence** occurs or the **maximum number of iterations** is reached. We assume that divergence happens when the resulting complex $z_j$ is **not contained in the problem space**. More precisely, let the problem space be $\{(r, i) \,|\, -N < r < N,\ -N < i < N\}$, then divergence occurs when:

$$length(z_j) \geq N$$

The **intensity of the color** of the point $c$ is directly proportional to the **number of iterations performed** $k_c$ without divergence. Thus the points that belong to the **Mandelbrot set** are going to be the most intense ones (usually black color). That can be easily done calculating the `scale_color` variable, which is the factor that needs to be applied to the `min_color` for every performed iteration:

$$\texttt{scale\_color} = \frac{\texttt{max\_color} - \texttt{min\_color}}{\texttt{maxiter} - 1}$$

$$\texttt{color} = (k_c - 1) \cdot \texttt{scale\_color} + \texttt{min\_color}$$

### 1.3 References


1. Mandelbrot set - Wikipedia, the free encyclopedia. [Internet]. url: `http://en.wikipedia.org/wiki/Mandelbrot_set` (visited on Apr 23, 2014).

## 2 Task granularity analysis

1. **Which are the two most important common characteristics of the task graphs generated for the two task granularities (*Row* and *Point*) for mandel-tareador? Include the task graphs that are generated in both cases for `-w 8`.**

   The two most important characteristics we can see from both graphs are:

   (1) There are no data dependencies between tasks. They are fully parallelizable.

   (2) There are tasks that take much more time (instructions) than others. Tasks are not homogeneous timewise.

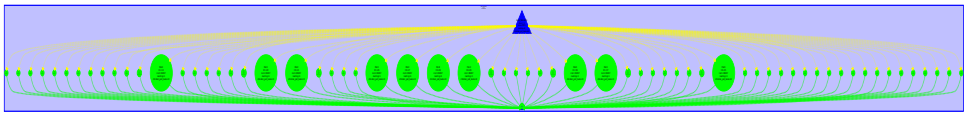

**Figure 2:** Task decomposition with *Point* granularity.

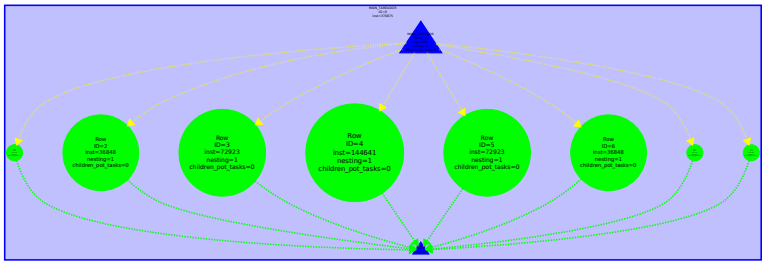

**Figure 3:** Task decomposition with *Row* granularity.

As we'll see later the different sizes of the tasks is very relevant in terms of load balancing.

2. **Which section of the code is causing the serialization of all tasks in mandeld-tareador? Include the task graph generated for the graphical version of Point after isolating the section of the code**

   The section that causes the serialization is the painting of the points. All threads must wait after computing a point to paint it, since the display system doesn't allow concurrent modifications of the window. The function `XDrawPoint` blocks the execution, so tasks must be serialized.

   If we do `tareador_disable_object(display)` and `tareador_disable_object(&win)` we can parallelize it more:

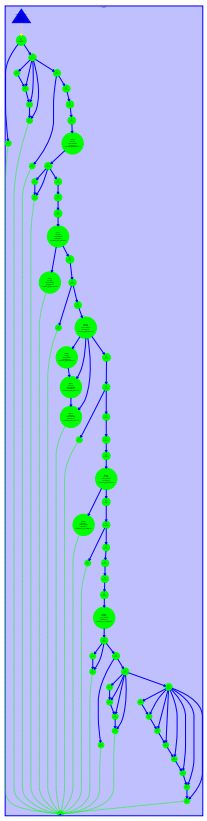

**Figure 4:** mandeld after disabling the blocking objects.

In Figure 4 we can see that it doesn't look as parallel as `mandel`. It is more parallel than `mandeld` without disabling nothing though. If we inspect data dependencies in *Tareador* we can see that there are still dependencies, even when we disabled all the objects that are relevant to the canvas painting. The problem is in the function calls to the X library. As we can not modify the library internals, we leave it this way.

3. **Using the results obtained from the simulated parallel execution for mandel-tareador and for a size of -w 16, complete the following table with the execution time and speed-up (with respect to the execution with 1 processor) obtained for the non-graphical version, for both individual task granularities. Comment the results highlighting the reason for the poor scalability, if detected?**

   The results obtained of simulating the execution of `mandel-tareador` with *Paraver* are:

| | Row | | Point | |
|---|---|---|---|---|
| **Num. processors** | **Time (ms)** | **Speed-up** | **Time (ms)** | **Speed-up** |
| **1** | 1080.584 | 1.000 | 1120.216 | 1.000 |
| **2** | 554.698 | 1.948 | 563.475 | 1.988 |
| **4** | 334.376 | 3.231 | 306.504 | 3.654 |
| **8** | 330.156 | 3.272 | 163.536 | 6.849 |
| **16** | 329.724 | 3.277 | 93.059 | 12.037 |

   We can see how *Point* granularity seems to scale better than *Row* granularity. At first, with two and four processors, we can see that both execution times are reduced and scale reasonably well, but as we add more processors, *Row* stalls.

   The problem is that, with *Row* granularity we have 16 tasks, and as we saw earlier, there are tasks that take a lot of time. We have a load balance problem, since some processor will have to execute the longest task while other processors are done and waiting. The longest row becomes the bottleneck, and adding more processors, won't produce speed-up.

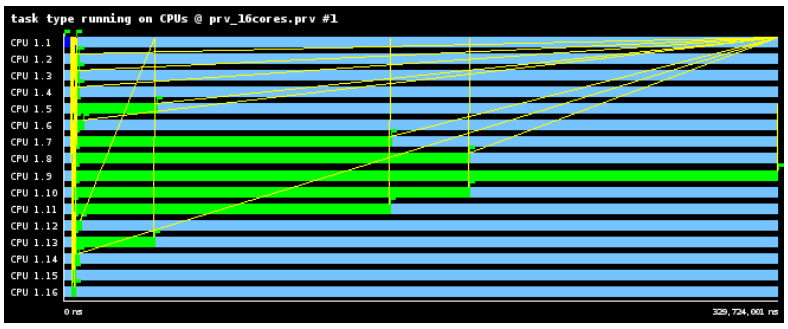


**Figure 5:** Row granularity: Most of the threads are in idle waiting for the longest task to finish.

With *Point* granularity, since we have 256 tasks ($16 \cdot 16$) while the longest points are being computed, the other processors can do other tasks, hence longest points are not the bottleneck now.

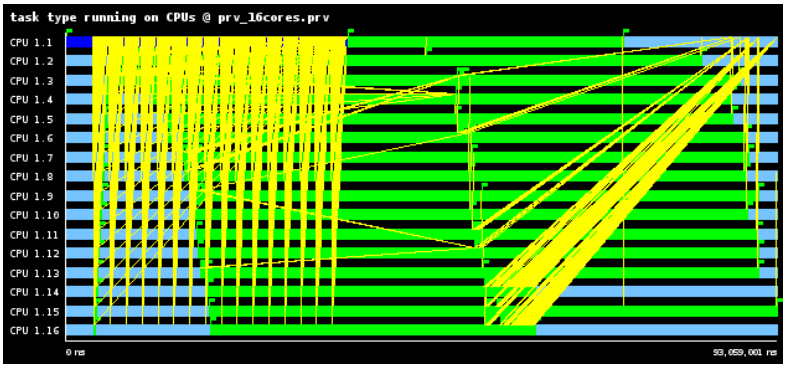


**Figure 6:** Point granularity: Since there are many more tasks than threads, the load can be distributed better.

From this exercise we can conclude that when $n_{processors} << n_{tasks}$ is more difficult to face load balance problems.

# 3 OpenMP execution analysis

4. **For each parallelization strategy of the non-graphical version, complete the following table with the execution time for different loop schedules and number of threads, reasoning about the results that are obtained.**

We did these exercise along with our colleagues of group 1201 (Lorenzo Arribas and David Solà). They performed timing for the not `collapsed` version. These are the results:

| Num. threads | static | static, 1 | dynamic, 1 | guided, 1 |
|---|---|---|---|---|
| 1 | 30.207 | 30.207 | 30.207 | 30.207 |
| 2 | 15.223 | 15.233 | 15.206 | 15.224 |
| 4 | 15.228 | 7.715 | 7.612 | 16.654 |
| 6 | 14.716 | 5.391 | 5.334 | 11.438 |
| 8 | 12.757 | 4.025 | 4.025 | 8.147 |
| 10 | 10.924 | 3.364 | 3.376 | 7.127 |
| 12 | 9.580 | 2.802 | 2.817 | 6.006 |

**Table 1:** Results for the uncollapsed version. Times are in seconds.

- `static` gives the worst performance. This can be explained considering that longest rows are contiguous, so a thread can have a chunk spanning multiple long rows, increasing the length of the critical path.
- `static,1` gives better results than `static` because rows are assigned one by one. This way, long rows end up being executed by different threads and can be more parallelized.
- `dynamic,1` gives similar results as `static,1`. It's not important if threads are assigned dynamically or statically, since the bottleneck will be the same. The important thing is that chunks have size 1.
- `guided,1` size of the chunks decreases as they are assigned to different threads. Because of this, we can see that with a low number of threads it behaves like `static` (since at the beginning chunks have size $n_{tasks}/n_{threads}$). At the end, though, as the chunk size decreases, it behaves better than `static`.

When using the `collapse` pragma we can only notice differences when the chunks are big enough, otherwise they behave very similar to the uncollapsed version. Having big chunks reduces the difference on the task sizes.

5. **For each parallelization strategy, complete the following table with the information extracted from the Extrae instrumented executions with 8 threads and analysis with Paraver, reasoning about the results that are obtained.**

| | **static** | **static, 1** | **dynamic, 1** | **guided, 1** |
|---|---|---|---|---|
| Running avg. time per thread (s) | 3.800 | 3.986 | 3.984 | 3.872 |
| Execution balance | 0.298 | 0.986 | 0.998 | 0.459 |
| SchedForkJoin ($ms$) | 12667.786 | 70.510 | 0.504 | 7805.279 |

With this exercise we can confirm what we said in the previous one. Even all 4 strategies perform similarly in terms of average time per thread, they are very different. `static` performs very badly because the maximum time is very high. This is a consequence of the grouping of long tasks. `static,1` and `dynamic,1` have a good balance (pretty close to 1) because the number of threads is very low compared to the number of tasks and allows to cluster shorter tasks in parallel with a longer task. Also, the lowest time spent for scheduling belongs to the dynamic strategy, because threads are assigned as they become available.