Report

Lab 3: Analysis of parallel strategies: the computation of the Mandelbrot set

Josep Antoni Martínez García
Santiago Oliver Suriñach

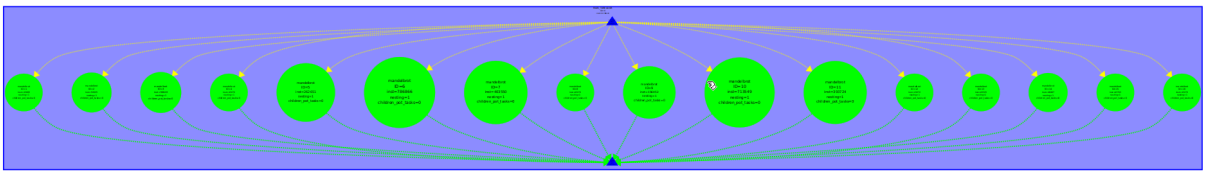
Index

1. Iterative task decomposition analysis

1.2 Analysis of strategies

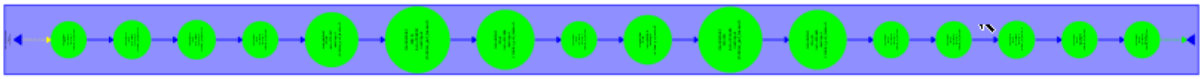
1.2.1 Original parallel strategy

As we can see in the following image, the TDG shows the potential parallelism to be applied to the original code.



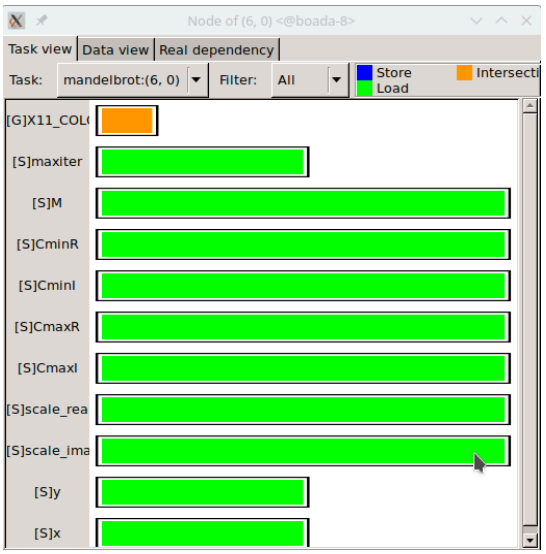
Display option execution

Now if we execute this code with ***-d***, in order to display the mandelbrot picture, we can appreciate by analyzing the TDG that now the code is not parallelizable anymore.



(Note that the image is rotated due to format reasons)

If we inspect a node (any of them is valid since all of them are the equivalent) we can see that the dependency is the variable X11_COLOR.



Analyzing deeply the code we find that the issue is generated by this two calls in the mandel_tiled function:

```C/C++
/* Scale color and display point  */
long color = (long) ((M[py][px]-1) * scale_color) + min_color;
if (setup_return == EXIT_SUCCESS)
 {
            XSetForeground (display, gc, color);
            XDrawPoint (display, win, gc, px, py);
 }
```

Working with OpenMP we could use the *critical* clause to solve the dependency issue:

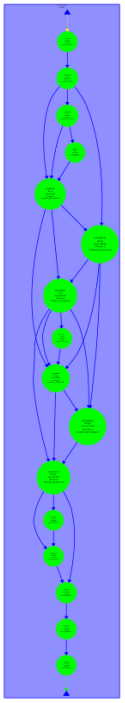
```C/C++
/* Scale color and display point  */
long color = (long) ((M[py][px]-1) * scale_color) + min_color;
if (setup_return == EXIT_SUCCESS)
 {
            #pragma omp critical
            XSetForeground (display, gc, color);

            XDrawPoint (display, win, gc, px, py);
 }
```

Using Tareador, in order to simulate the execution with the solved dependency, the resulting code should look like this:

```C/C++
tareador_start_task("XSetForeground");
tareador_disable_object("X11_COLOR");

tareador_enable_object("X11_COLOR");
tareador_end_task("XSetForeground");
```

About the proposed solution, use *critical* is not the best solution since the critical directive causes big overheads. An optimal solution could be parallelizing the color processment and sequentially setting them to pixels.

Histogram option execution



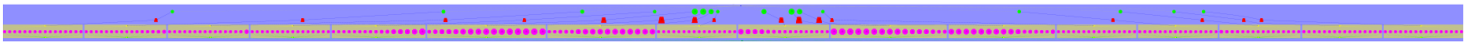
This time, when using the ***-h*** option to keep the histogram of colors, the TDG obtained is the following vertical TDG. As we can see, the result obtained is very similar to a sequential execution.

1.2.2 Finer grain parallel strategy

As the guide proposes, we have defined the following tasks:

- **horizontal:** this task refers to the checking performed to the horizontal borders.
- **vertical:** similar to the previous one, but done in the vertical borders.
- **if-else:** this task wraps all the if-else block, responsible for assignment (and computation if needed) of the colors.
    - **py-fill:** each iteration of the loop that fills the pixel with the same value is a task.
    - **py-comp:** each iteration of the loop that computes the color is also a task.

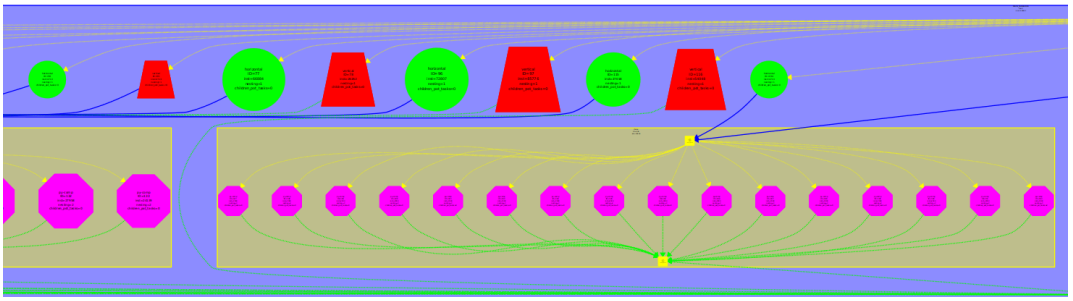
The resulting (and poorly visible) TDG shows this task decomposition (see Annex to have a little bit better look):



Now we can check the dependencies between vertical and horizontal tasks, which generate the parallelism issue:

As we can see, the variable causing the dependency is *equal*. A solution for this problem could be creating two *equal* variables, one for each task, vertical and horizontal. Then both tasks could be executed in parallel without any data sharing.

After performing this modification, the TDG looks like:



We have parallelized the horizontal and vertical tasks and conserved dependencies between this and the if-else block and our goal is achieved.
The modified code is:

We create a local variable for horizontal task (*equal0*):

```C/C++
//check horizontal borders
       tareador_start_task("horizontal");
       int equal0 = 1;
// ...
tareador_end_task("horizontal");
```

We create a local variable for vertical task (*equal1*):

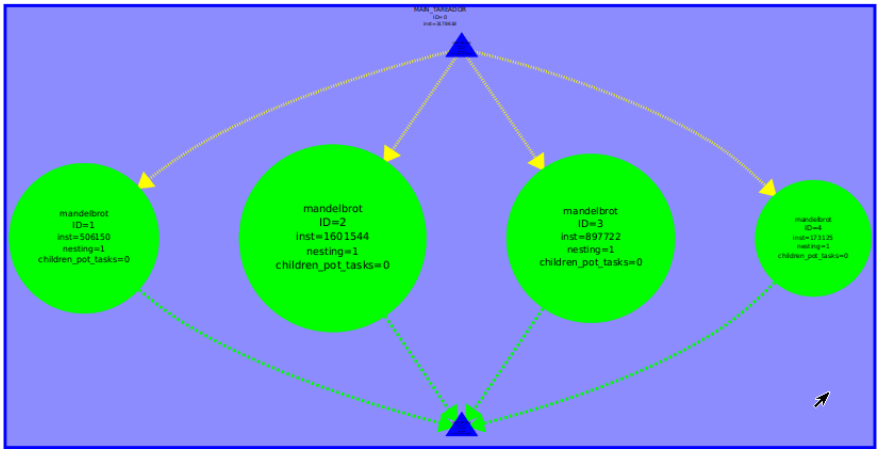
```C/C++
//check vertical borders
       tareador_start_task("vertical");
       int equal1 = 1;
// ...
tareador_end_task("vertical");
```

Then the if-else block uses equal so we need to create a equal variable that takes into account the previous ones:

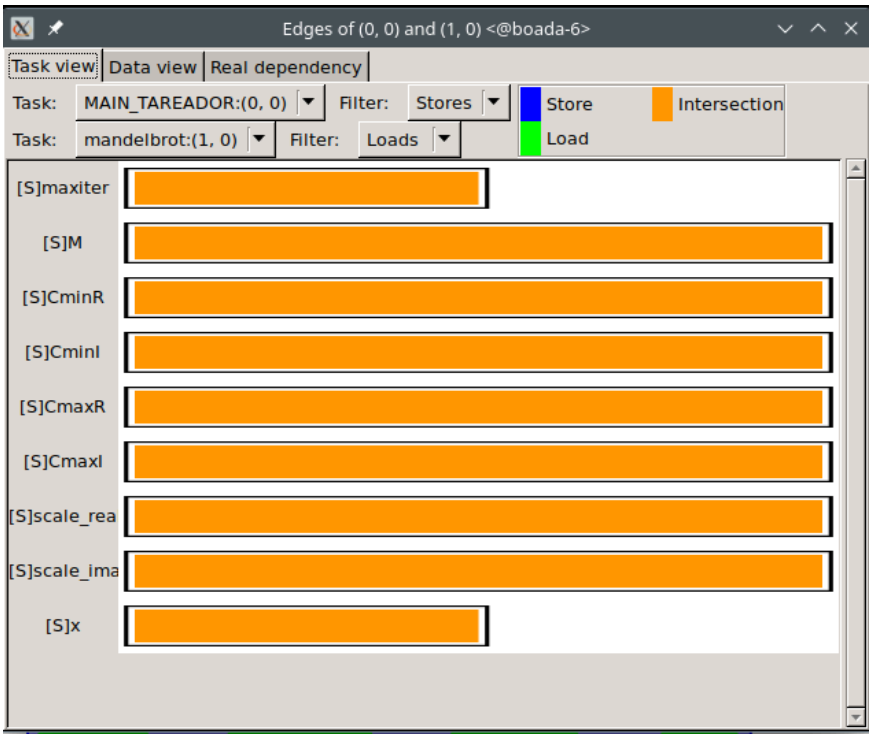
```C/C++
//check if we can avoid computation of tile
       tareador_start_task("if-else");
       int equal = equal0 && equal1;
       if (equal && M[y][x]==maxiter) { // ... }
```

1.2.3 Column of tiles parallel strategy

Now the task is a whole iteration of the for-x. We have switched both for loops in order to compute columns consecutively. Otherwise, the computation flow would be row by row. The resulting TDG looks like:



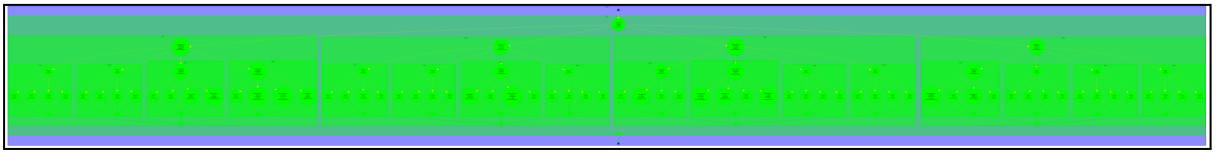
Dependencies found for this task decomposition are:



As we can see, this task decomposition strategy is already optimal and lets us execute the program correctly.

2. Recursive task decomposition analysis

After running the recursive program with default Tareador instrumentation, the TDG looks like:



As we can see in the original code, one task represents the whole recursive function. As a result, the task wraps all the recursive calls in it.

2.1 Leaf strategy

In order to apply a Leaf Strategy, we must define a task as the base case of the recursive function. In this occasion, we found two base cases:

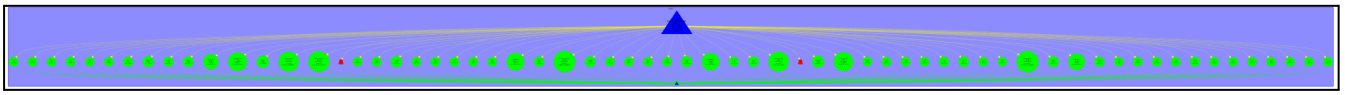
- ***base-case-edges***: when we can avoid color computation, as we have the same color in all edges of the tile.
- ***base-case-comp:*** the computation of the tile itself.

Here we can check the task decomposition:

```C/C++
// fill all with the same value: base case
     tareador_start_task("base-case-edges");
      for (int py=y; py<y+NRows; py++)
       for (int px=x; px<x+NCols; px++)
       {
          // ...
       }
      tareador_end_task("base-case-edges");
```

```C/C++
   // computation of tile: base case
         tareador_start_task("base-case-comp");
          for (int py=y; py<y+NRows; py++)
            for (int px=x; px<x+NCols; px++)
            {
                    // ...
            }
          tareador_end_task("base-case-comp");
```

The resulting TDG is:



As we can appreciate in the TDG, all created tasks corresponding to the base cases are executed in parallel. So no changes are needed to be made in the code, since this proposal is already optimal.

2.2 Tree strategy

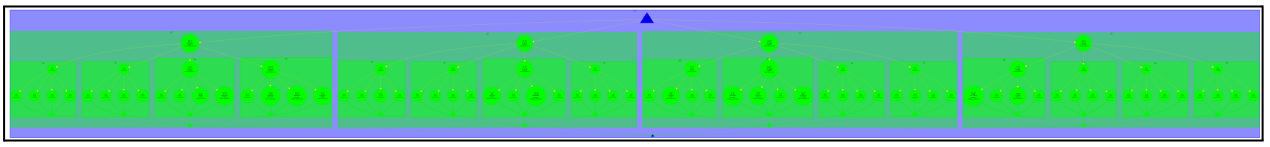
Now the strategy consists of defining tasks in every recursive call, in order to create a task decomposition based on a Tree Strategy.

Here we have the proposed code:

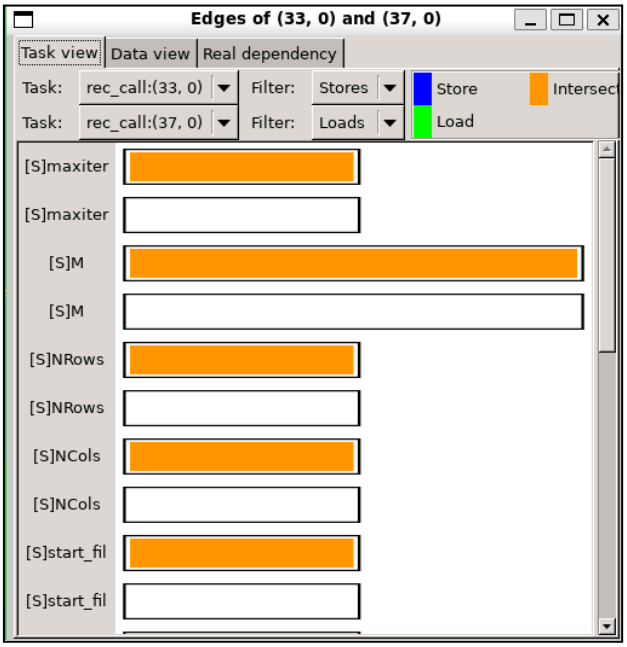
```
C/C++
    // computation of tile: recursive cases
      if (NRows > TILE)
      {
        tareador_start_task("rec_call");
         mandel_tiled_rec(M, NRows/2, NCols/2, start_fil, start_col, CminR,
CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
        tareador_end_task("rec_call");

        tareador_start_task("rec_call");
         mandel_tiled_rec(M, NRows/2, NCols/2, start_fil, start_col+NCols/2,
CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
        tareador_end_task("rec_call");

        tareador_start_task("rec_call");
        mandel_tiled_rec(M, NRows/2, NCols/2, start_fil+NRows/2, start_col,
CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
        tareador_end_task("rec_call");

        tareador_start_task("rec_call");
         mandel_tiled_rec(M, NRows/2, NCols/2, start_fil+NRows/2,
start_col+NCols/2, CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag,
maxiter);
        tareador_end_task("rec_call");
      }
      else
      {
        tareador_start_task("rec_call");
```

```
        mandel_tiled_rec(M, NRows, NCols/2, start_fil, start_col, CminR,
CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
         tareador_end_task("rec_call");

         tareador_start_task("rec_call");
         mandel_tiled_rec(M, NRows, NCols/2, start_fil, start_col+NCols/2,
CminR, CminI, CmaxR, CmaxI, scale_real, scale_imag, maxiter);
         tareador_end_task("rec_call");

         }
```

The TDG looks like:



If we check the dependencies, we conclude that we cannot improve the performance anymore. Because all variables implicated are arguments of the recursive call and it is not possible to disable them.



3. Comparison

Version	T_1	T_∞	Parallelism
Iterative: original strategy (no parameters) Iterative: original strategy (-d) Iterative: original strategy (-h)	3.178s 3.305s 3.244s	0.786s 3.305s 3.243s	4.043 1 1
Iterative: Finer grain	3.180s	0.141s	22.553
Iterative: Column of tiles	3.178s	1.601s	1.985
Recursive: Leaf	4.545s	2.702s	1.682
Recursive: Tree	4.545s	0.689s	6.597

The table shows the execution times for all strategies.

3.1 Iterative

As we can see, every time the program is executed with a single processor (T_1), the time is almost identical, since the code is the same for all executions and is runned sequentially.
For the parallel execution (T_∞), we can appreciate that the most significant improvement is achieved by the Finer Grain task decomposition.

3.2 Recursive

Finally, the reason why all T_1 are the identical again, is the same as the Iterative case. Referring to the improvement found in parallel executions (T_∞), the best performance is shown by the Tree strategy.