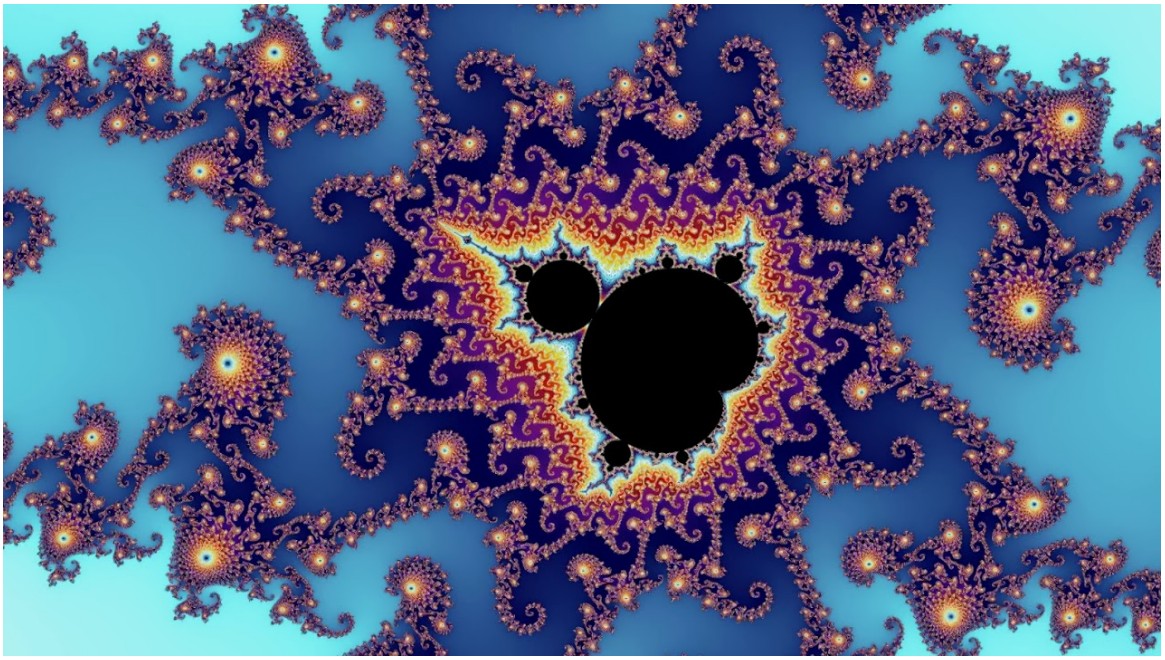
PAR Laboratory 3

Font i Cabarrocas, Marc — Jara Palos, Adrià

October-November 2024



Contents

1 Introduction

The purpose of the third laboratory assignment is to understand and explore the nuances of parallel computing strategies through both iterative and recursive task decompositions. To deepen our comprehension, we have been tasked with compiling, executing, and modifying a program that computes the well-known *Mandelbrot Set*. A particular set of points, in the complex domain, whose boundary generates a distinctive and easily recognizable two-dimensional fractal shape (see cover).

The assignment challenges us to implement these computations using both iterative and recursive approaches, each offering unique insights into parallel task management, load balancing, and performance analysis. With an emphasis in the use of Tareador to visualize task dependency graphs (TDGs) and measure key performance metrics like sequential execution time (T_1), parallel execution time with an infinite number of processors (T_∞), and overall parallelism achieved.

Through hands-on experimentation and modifications to the original code, we seek to uncover how various parameters and optimizations impact the execution. The iterative approach will provide insight into the Finer Grain and Column methods, while the recursive approach will allow us to understand the Tree and Leaf strategies.

2 Iterative Parallelism

2.1 Original Version

In the Original Version no parallel strategies or approaches shall be used. As the purpose of this part is purely to understand and disable dependencies due to data sharing to exploit parallelism among tasks which will later be useful for the rest of strategies and future parallelization.

2.1.1 No Arguments

To begin with the Original Version with no arguments of execution, we can clearly see (view Figure 1) that there are no task dependencies whatsoever and that every task in the graph can be executed in parallel.

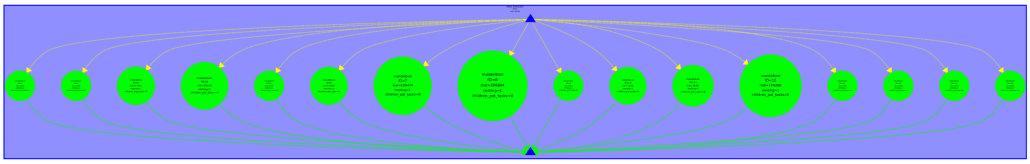


Figure 1: Task Decomposition without Arguments

After running the simulation with Tareador, a value for T_1 and T_∞ were calculated, and the value of the *Parallelism* too:

- $T_1 = 0.702$s
- $T_\infty = 0.308$s
- $Parallelism = 2.279$

The code however, presents a load unbalance, that is because the CPUs that have to execute the bigger tasks in Figure 1 will take more time to finish compared to CPUs dedicated to the smaller ones.

As the result of parallelizing the code with *-d* and *-h* arguments generates exactly the same TDG, we can also extend this reasoning to their respective cases as to why there is load unbalance.

2.1.2 -d Argument

This execution of the code is run with the $-d$ argument, which allows the code to generate a computed image of the Mandelbrot Set. Observe that there is a huge task dependency (view Figure 2). The image shows that no task T_i can start before the previous one has been completed, causing a huge traffic jam that could be avoided if the Task Dependencies were to be removed.

Figure 2: Task Decomposition with Computed Image

In order to get rid of the Task Dependencies, we used Tareador's tools to locate the dependency on a variable inside $X11$'s draw call. To solve this problem, we disabled the calls to the graphics library, so that we wouldn't have any dependencies related to the writing on $X11$'s buffer. After this modification, the code is executed more than two times faster. However, we would need to use data sharing synchronizations to ensure that $X11$'s functions are not called at the same time on two independent threads. We think *#pragma omp critical* should be used here to ensure only one thread at the time is executing the $X1$'s calls.

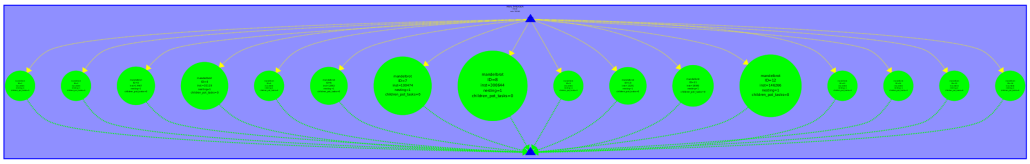


Figure 3: Task Decomposition with Computed Image after removing dependencies

Having removed all TD, the code was simulated in Tareador to find a value for T_1, T_∞ $Parallelism$:

- $T_1 = 0.729$s
- $T_\infty = 0.310$s
- $Parallelism = 2.351$

For the same reasons as the version with No Arguments, this one also has issues with the load unbalance, as there are a number of CPUs that will carry out more expensive tasks compared to others.

2.1.3 -h Argument

The final code to be analyzed before starting to parallelize anything is the Original Version with the $-h$ option, that produces the histogram of values in the computed image. The resulting TDG with this particular argument can be seen in Figure 4. Note that there are even more dependencies that in the $-d$ version.

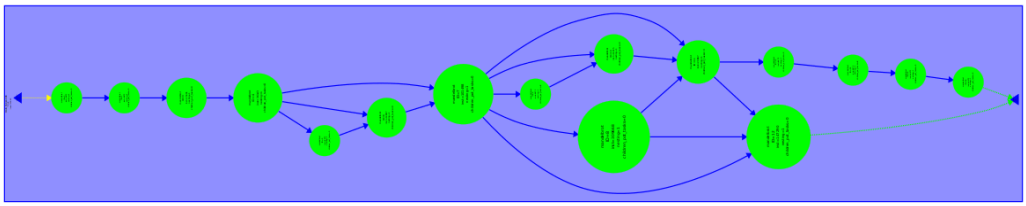


Figure 4: Task Decomposition with Histogram

In order to get rid of the Task Dependencies, Tareador was used to locate the dependency of -h version, which was found to be the histogram variable. We disabled it with *tareador_disable_object(&name_var)*. A data sharing synchronization is required, since we only need to ensure histogram is updated correctly, *#pragma omp atomic* is to be used.

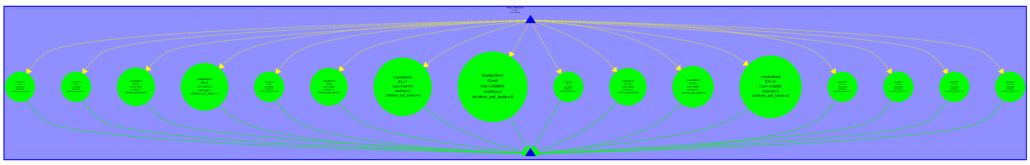


Figure 5: Task Decomposition with Histogram after removing dependencies

After removing all the task dependencies, the code was simulated in Tareador to find a value for T_1, T_∞ and the *Parallelism*:

- $T_1 = 0.713$s
- $T_\infty = 0.309$s
- $Parallelism = 2.307$

2.2 Finer Grain Approach

The finer grain approach consists of dividing a task or workload into smaller, more detailed sub-tasks to be executed in parallel. To do that, the Original Version was modified to accommodate that strategy.

Modifications:

- Task mandel_H that checks the horizontal lines.
- Task mandel_V that checks the vertical lines.
- Another task, mandel_IF that checks the conditional if-clause. This task will have two inner tasks, one that dies the filling for each py, and another that requires the computation of the same tile.
- Finally disable the variable *Hmatrix* (line 303) which trigered an intersection when examining Tareador's tasks carefully.

After implementing all of the aforementioned changes, the code was simulated in Tareador and it provided the following TDG (view Figures 6 & 7).

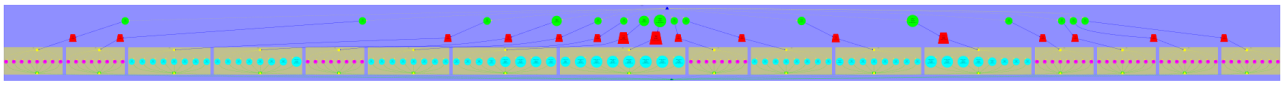


Figure 6: Task Decomposition with Finer Grain parallelization

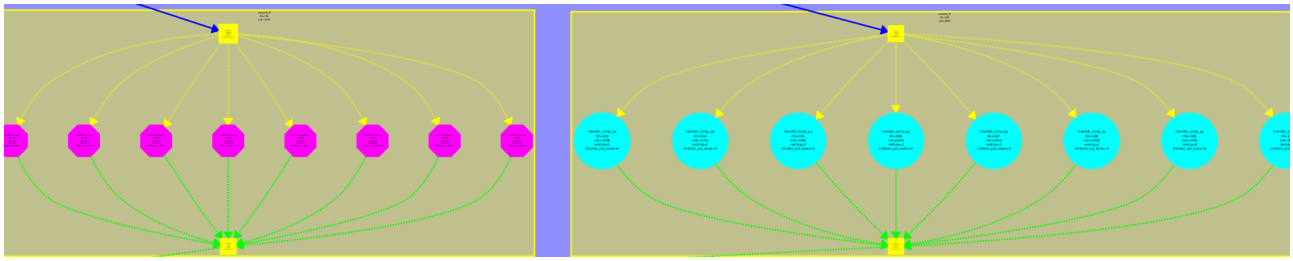


Figure 7: Zoomed version of Task Decomposition with Finer Grain parallelization

The values of T_1, T_∞ and the *Parallelism* were also calculated, and the results, in lack of a better word, were surprising to say the least.

- $T_1 = 0.705$s
- $T_\infty = 0.112$s
- $Parallelism = 6.294$

In this case there was no vestige of load unbalance, as all of the CPU assigned to the code in Tareador seamed to balance the work quite evenly.

2.3 Column Approach

The Column Approach consists in dividing the workload into vertical columns, and each column is assigned to a separate processing unit or thread for parallel execution. The changes in the Original Version were as follow.

After the changes had been implemented, the code was simulated in Tareador and it returned the following TDG (view Figure 8).

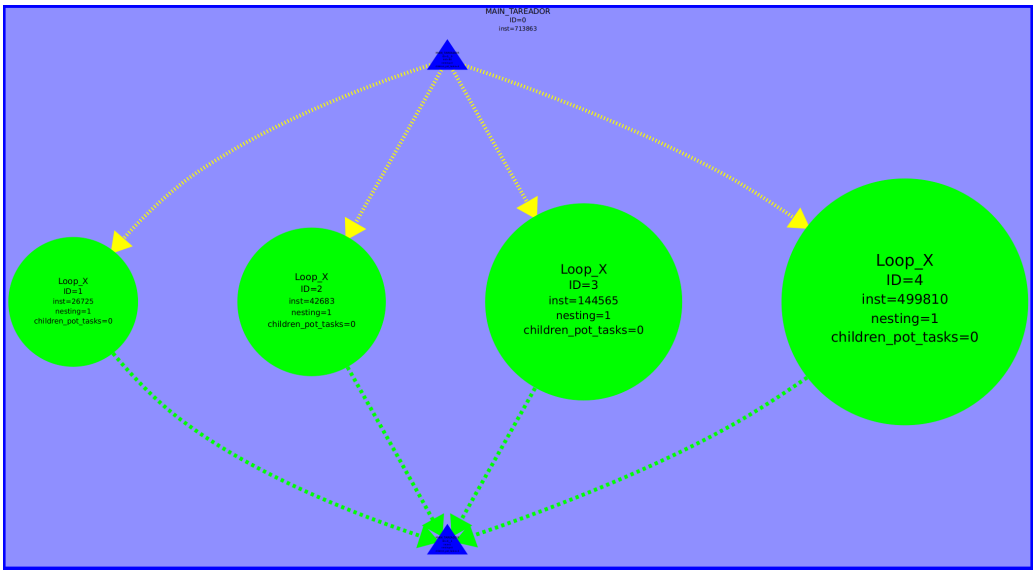


Figure 8: Task Decomposition with Column parallelization

After analizing the TDG, values forT_1, T_∞ and the *Parallelism* were obtained.

- $T_1 = 0.714$s
- $T_\infty = 0.499$s
- $Parallelism = 1.428$

In this approach there is a case for load unbalance, as the right-most task (the biggest) in Figure 8 takes a lot more of time than the left-most task (the smallest). And, as we can see in the histogram in Figure 9 , that's just what happens.

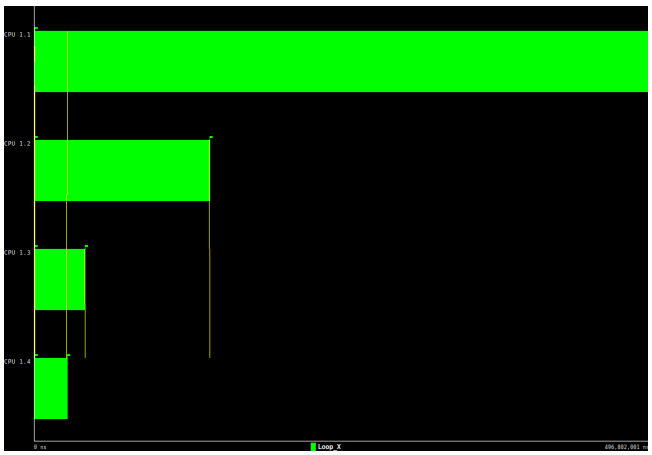


Figure 9: Histogram of Column Approach

3 Recursive Parallelism

3.1 Leaf Strategy

In this strategy, tasks are assigned to the leaf nodes of a tree-like structure. Leaf nodes are the end-points or ”leaves” of the tree, meaning they have no child nodes. Each leaf node is responsible for a small chunk of work, which can be computed in parallel.

When added the tasks in the base case of the recursion, as the Leaf Strategy dictates, the following TDG was generated in Tareador (view Figure 10).

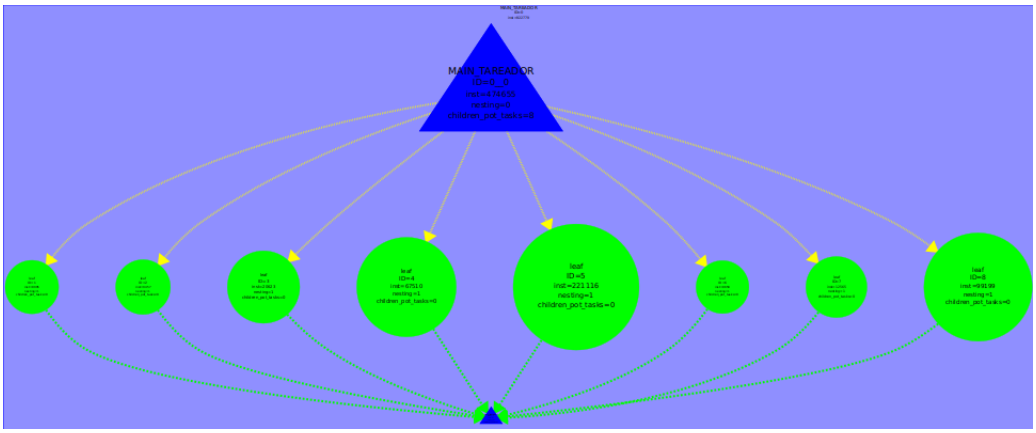


Figure 10: Task Decomposition with Leaf Strategy

And later on, the values for T_1, T_∞ and the *Parallelism* were calculated through Tareador's simulation.

- $T_1 = 0.922$s
- $T_\infty = 0.559$s
- $Parallelism = 1.649$

In this case we were also able to observe a Load Unbalance. When executed, a clear lack of equitative distribution was evidenced as observable in the histogram in Figure 11

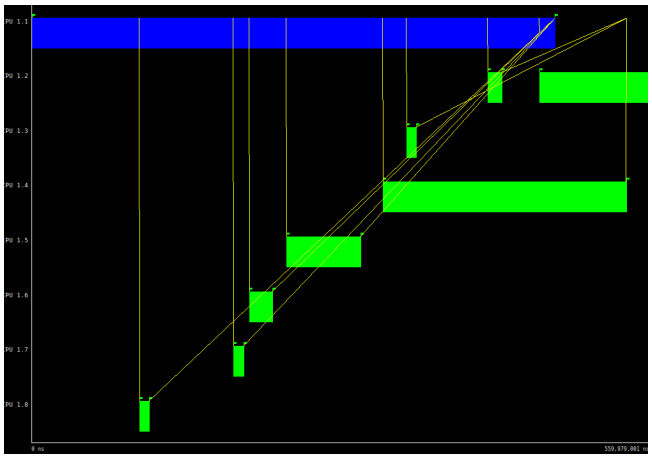


Figure 11: Leaf Strategy Histogram

3.2 Tree Strategy

The Tree strategy involves computation at all levels of the tree, including both leaf and non-leaf nodes. The work is distributed across the tree structure in such a way that each node (including intermediate nodes) performs some part of the computation, often involving a divide-and-conquer approach.

Tasks for horizontal and vertical checks were added, alongside a task for each recursive call. We also deactivated the variable *equals* since there was a collision on the data nodes. This modifications in the code allowed Tareador to generate the TDG visible in Figure 12.

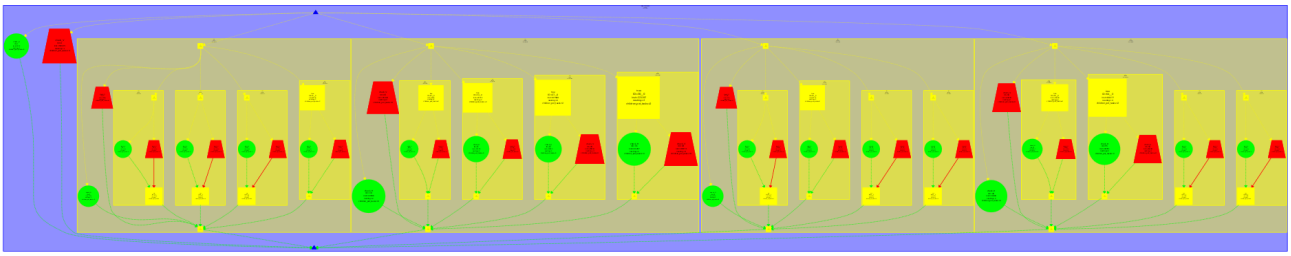


Figure 12: Task Decomposition with Tree Strategy

The values for T_1, T_∞ and the *Parallelism* were calculated through Tareador's simulation.

- $T_1 = 0.922$s
- $T_\infty = 0.221$s
- $Parallelism = 4.520$

In this case we were also able to observe a Load Unbalance since one processor took most of the work load, but, that issue aside, almost half the program is balanced.

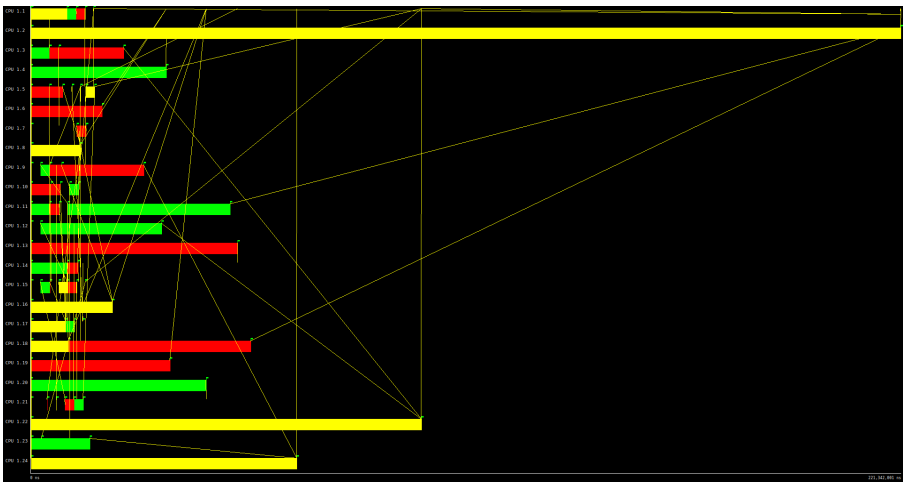


Figure 13: Tree Strategy Histogram

4 Conclusion

In order to summarize all the details of the parallelism used on each version, be that recursive or iterative parallelism view Table 1.

Task Decomposition	**Strategy**	**Run Arguments**	T_1	T_∞	**Parallelism**	**Load Unbalance (Yes/No) Why?**
Iterative	Original		0.702	0.308	2.279	Yes
	Original	-d	0.729	0.310	2.351	Yes
	Original	-h	0.713	0.309	2.307	Yes
	Finer grain		0.705	0.122	6.294	No
	Column		0.714	0.499	1.428	Yes
Recursive	Leaf		0.922	0.559	1.649	Yes
	Tree		0.922	0.221	4.520	Yes

Table 1: Summary of the parallelism performance of each of the versions

We conclude that the Mandelbrot Set is most efficiently parallelized with an Iterative Task Decomposition. More specifically, the Finer Grain approach was the most efficient one. With a parallelism value of 6.294 and an execution time of 0.122s, it has proven to be the best strategy, yet, to compute the Mandelbrot Set.

As for the Recursive Task Decomposition, the Tree Strategy takes an edge over Leaf, with a parallelism value of 4.520 and an execution time of 0.221s.