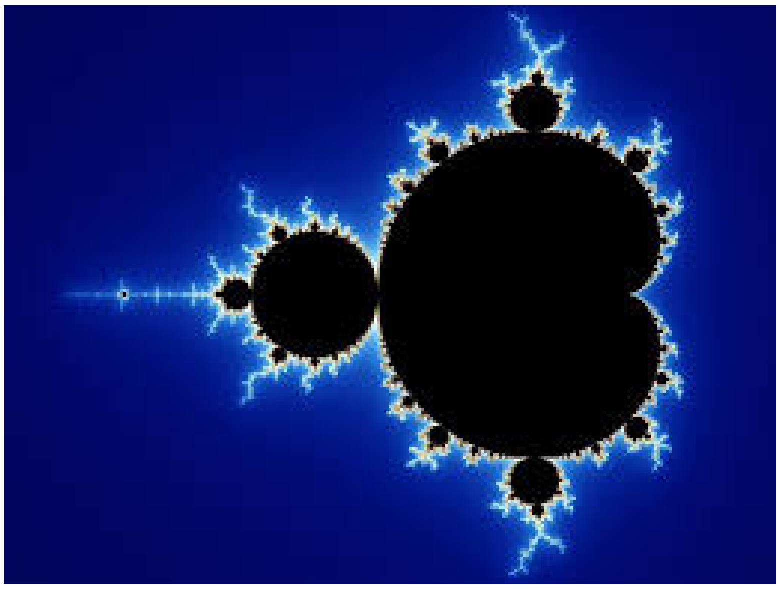
PARALLELISM LABORATORY 3

David Morais
david.morais

Laura van Dinteren
laura.van.dinteren



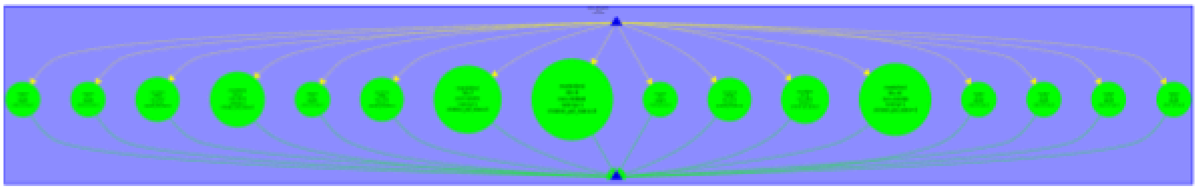
Comparison table:

Task Decomposition	Strategy	Run Arguments	T_1 (ns)	T(inf)	Parallelism	Load Unbalance (Yes/No) Why?
Iterative	Original		705.54	308.75	2.2851	Yes
	Original	-d	729.092	310.734	2.3463	Yes *
	Original	-h	713.924	309.804	2.3044	Yes *
	Finer grain		706.662	88.828	7.9554	Yes
	Column		705.542	496.780	1.5018	Yes
Recursive	Leaf		922.779	547.331	1.6859	Yes
	Tree		922.947	364.932	2.529	Yes

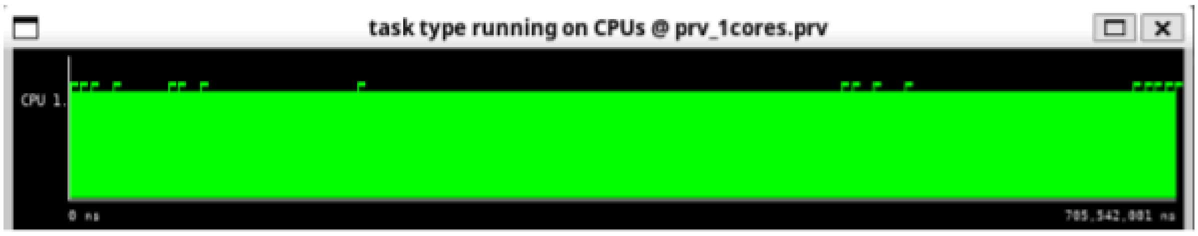
*We can see that core 9 does so much work, but on the other side core 8,12,15,16 ... They do barely any work (-d & -h)

Original without run arguments

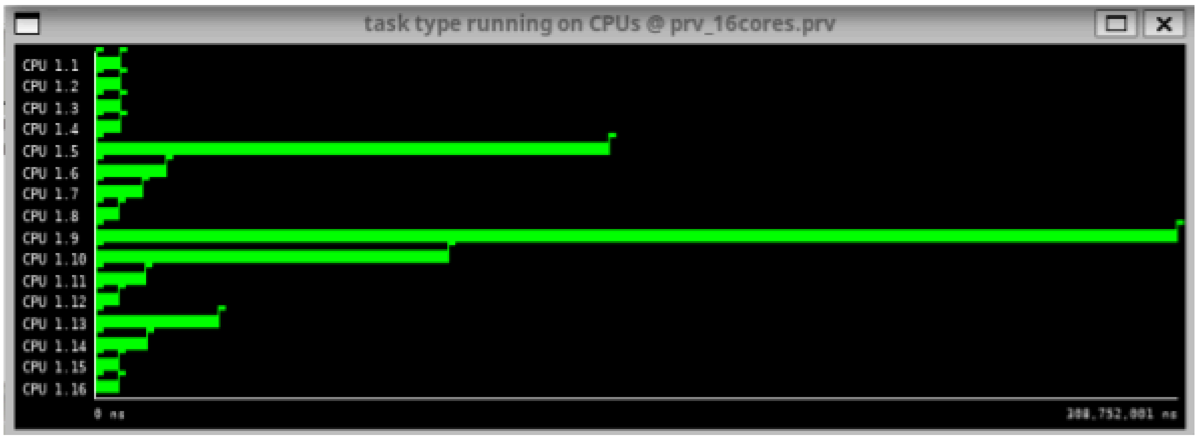
We changed nothing of the code as there weren't any data dependencies.



TDG from original without run arguments



Original without run arguments running in one thread



Original without run arguments running in 128 threads

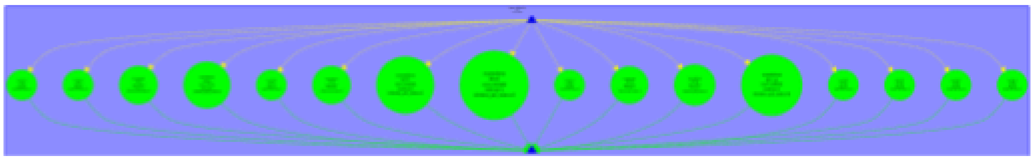
Dependence analysis:
It has no data dependencies, as we can see in the graph, so each task can be executed in parallel. Also, there is some load unbalance. We can see that thread 1.9 does most of the work, the threads 1.5 and 1.10 do a considerable amount of work while the others don't do much work.

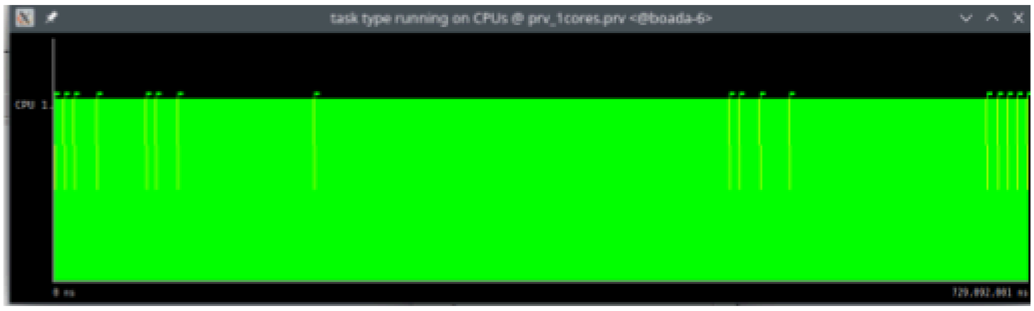
With “-d” arguments:

The changes that were made were:

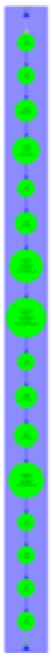
- We wrote “tareador_disable_object(&X11_COLOR_fake)”, just before creating the task
- We wrote “tareador_enable_object(&X11_COLOR_fake)”, just after ending the task.

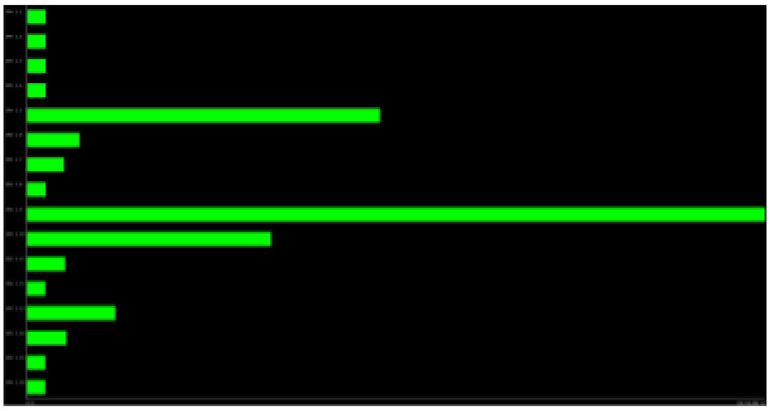
The variable “X11_COLOR_fake” was being used in each task, so we deleted that dependency so each task could be executed in parallel, as it is shown in the following images.


Original with “-d” after deleting data dependencies



Original with “-d” running in one thread


Original with “-d” before deleting data dependencies


Original with “-d” running in 16 threads

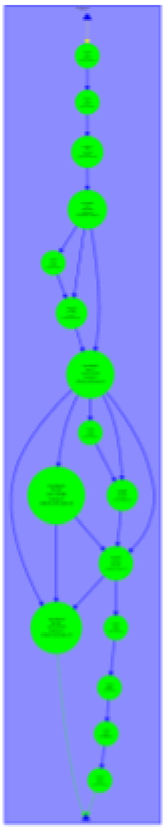
With “-h” arguments:

The changes that were made were:

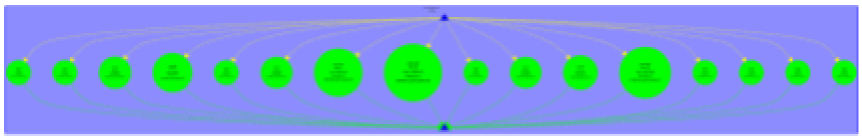
- We wrote “tareador_disable_object(histogram)”, just before creating the task
- We wrote “tareador_enable_object(histogram)”, just after finishing the task.

In Tareador we see on the TDG that all nodes have the same dependence, which is the histogram. So before we start the task with Tareador we disable the object histogram to eliminate the dependencies of the TDG. (tareador_disable_object(histogram)). We can now take advantage of the parallelism.

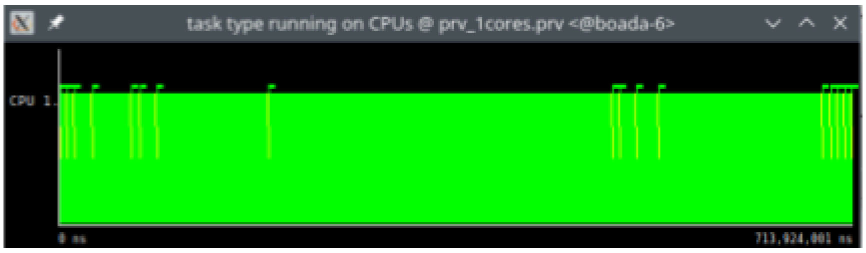
The following image is the TDG before deleting data dependencies.



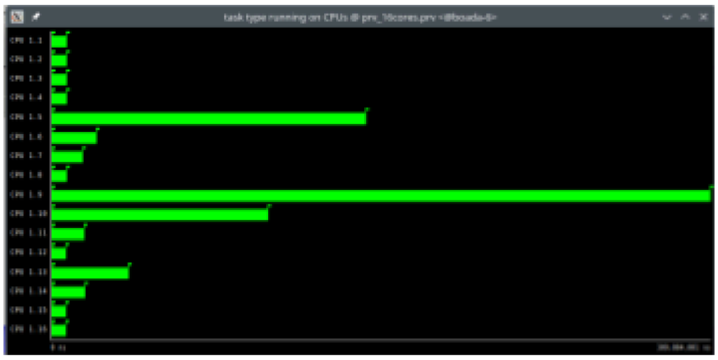
Original with “-h” before deleting data dependencies



Original with “-h” after deleting data dependencies

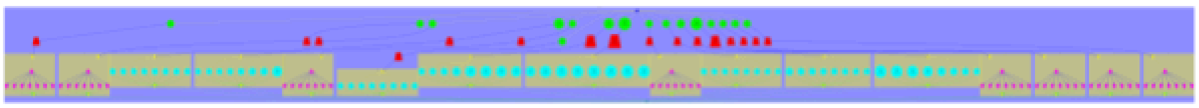


Original with “-h” running in one thread

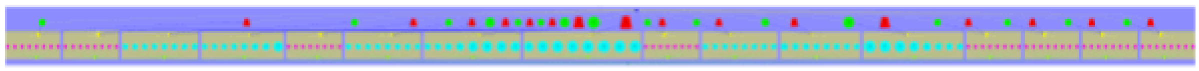


Original with “-h” running in 16 threads

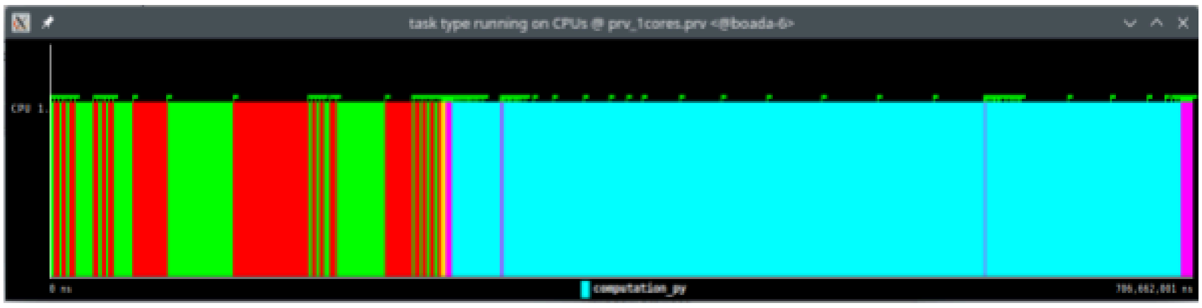
Finer grain



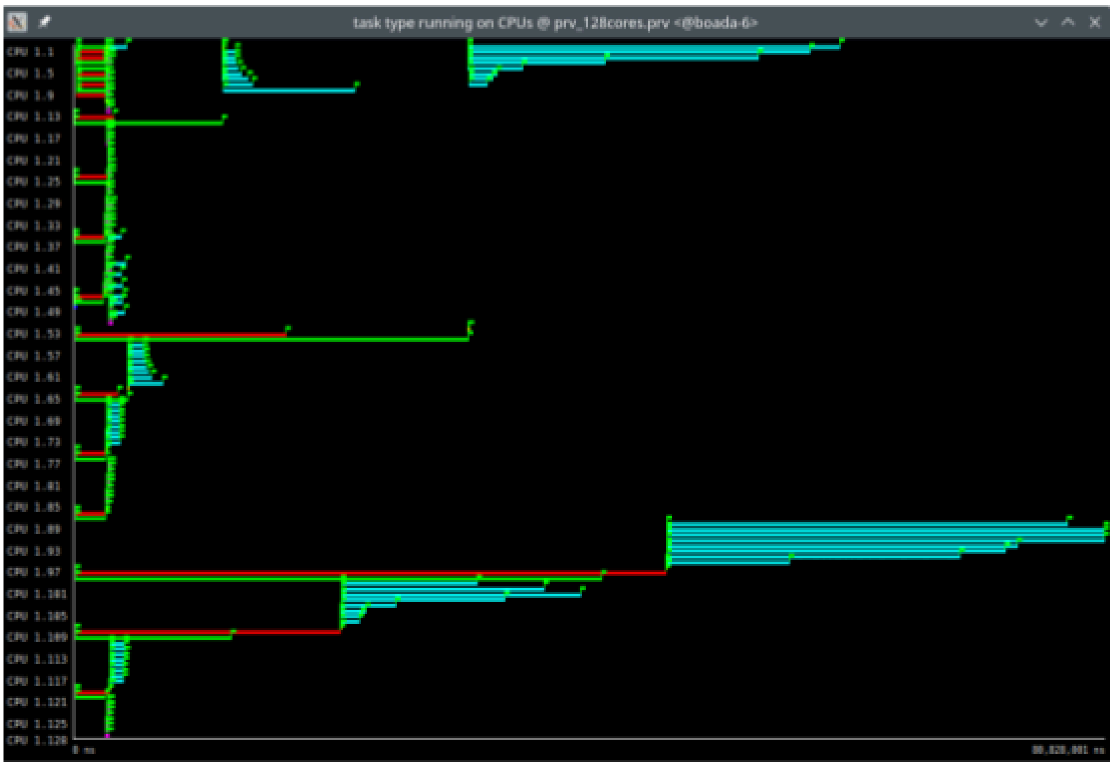
Finer grain before deleting data dependencies



Finer grain after deleting data dependencies



Finer grain running in one thread



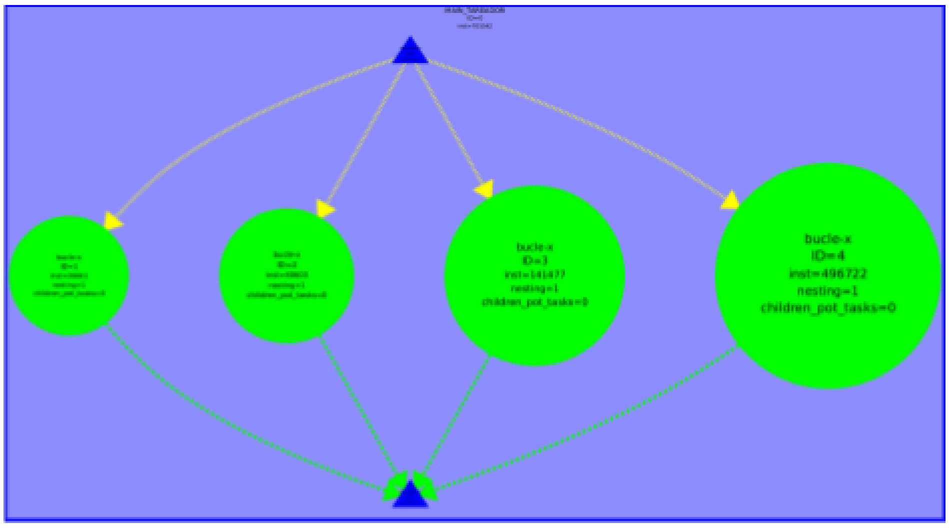
Finer grain running in 128 threads

The first dependency we see in the TDG is the equal variable between the vertical and horizontal tasks. What we can do to solve this dependency is to make a manual reduction.

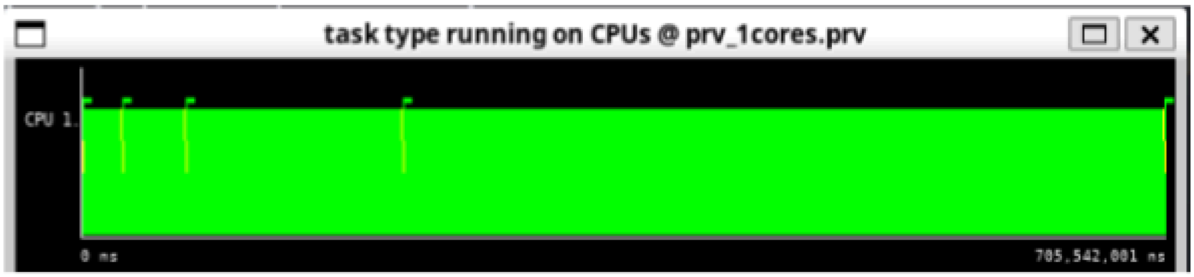
For this, we create two new variables that will be equalH and equalV, then for the execution, we use their respective equals instead of the global. Then, just before executing the if-else task, we make equal = equalH & equalV (it is important that it is the binary AND, this way we do not have lazy evaluation).
The second dependency we see is in the equal-py task. This is because it needs to first access M[y][x] in order to loop. As this value is the same for the whole loop px, we can disable the variable M[y][x]. This way, we can have all the executions in parallel instead of needing an iteration first.

Column of tiles

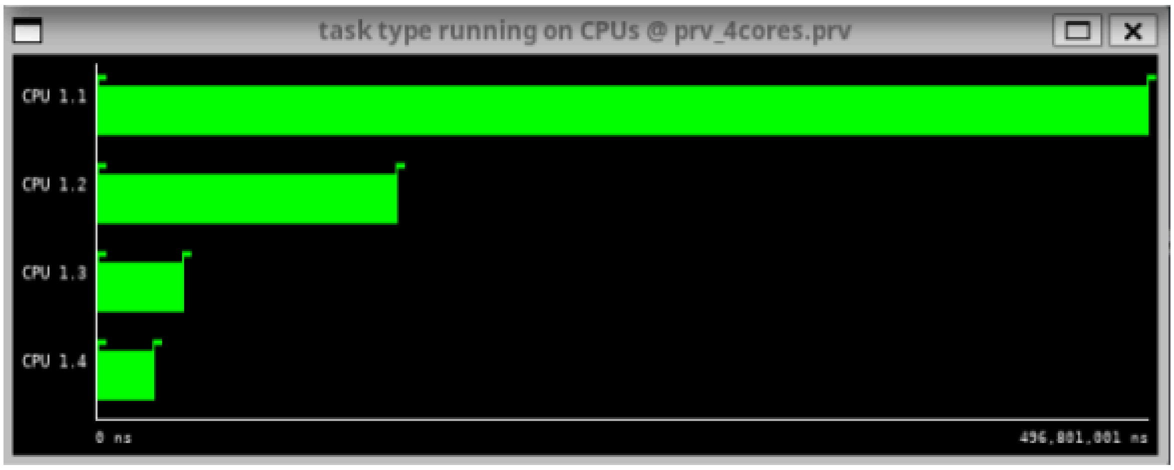
TDG:
The loops are swapped, and as we can see in the TDG, there are no data dependencies. There is a considerable amount of load unbalance, we can see that thread 1.1, does most of the work, while threads 1.2, 1.3 and 1.4 don't have that much work to do.



TDG (no data dependencies, so no before and after)



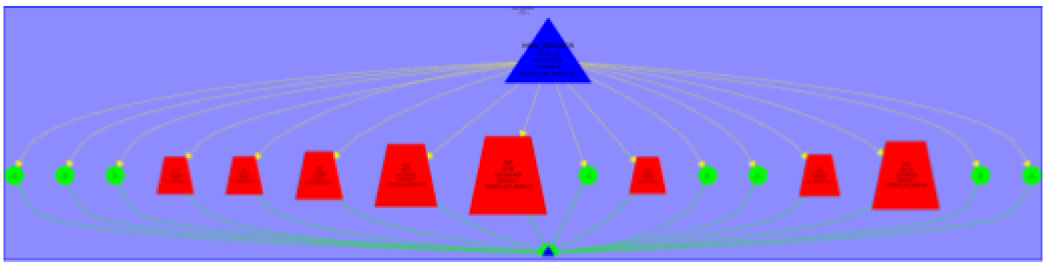
Column of tiles running in one thread



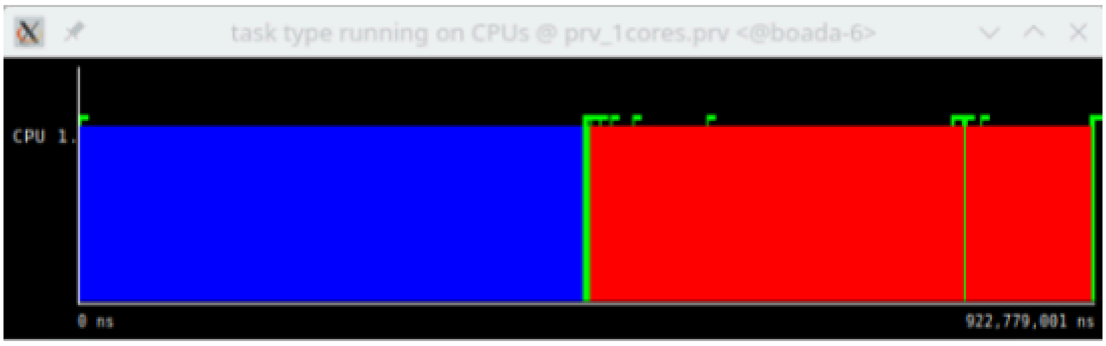
Column of tiles running in 128 threads

Leaf recursive strategy

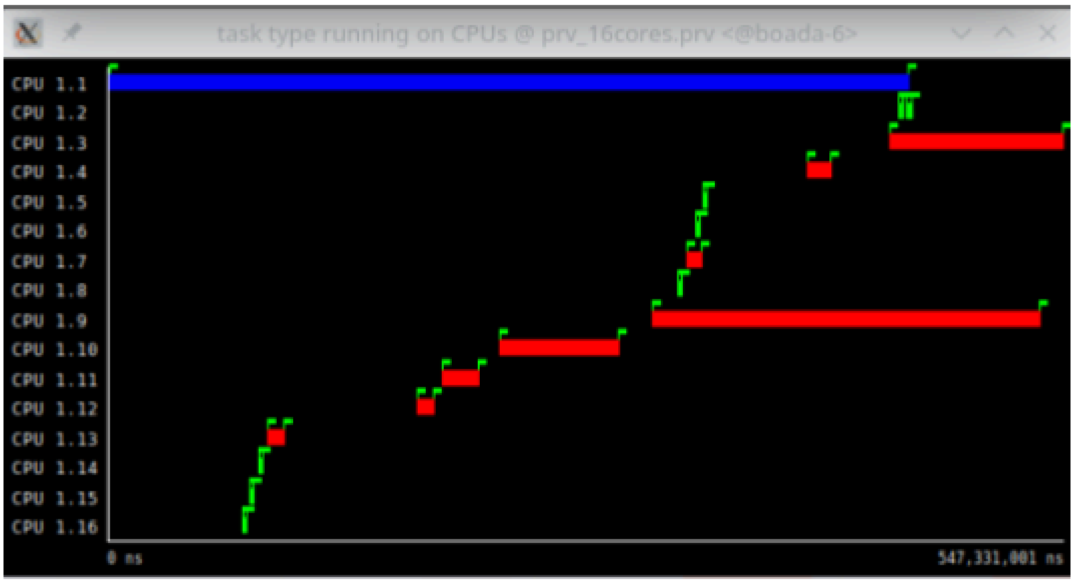
We created a task for each base case. There are no data dependencies.
In order to carry out this leaf strategy, we have created two new tasks, the first in the first base case which is if the equal is true, which we have called base-case and the second is in case the equal is false but the TILE is greater than or equal to NCols, this task we have called leaf.



TDG (no data dependencies, so no before and after)



Leaf recursive running in one thread

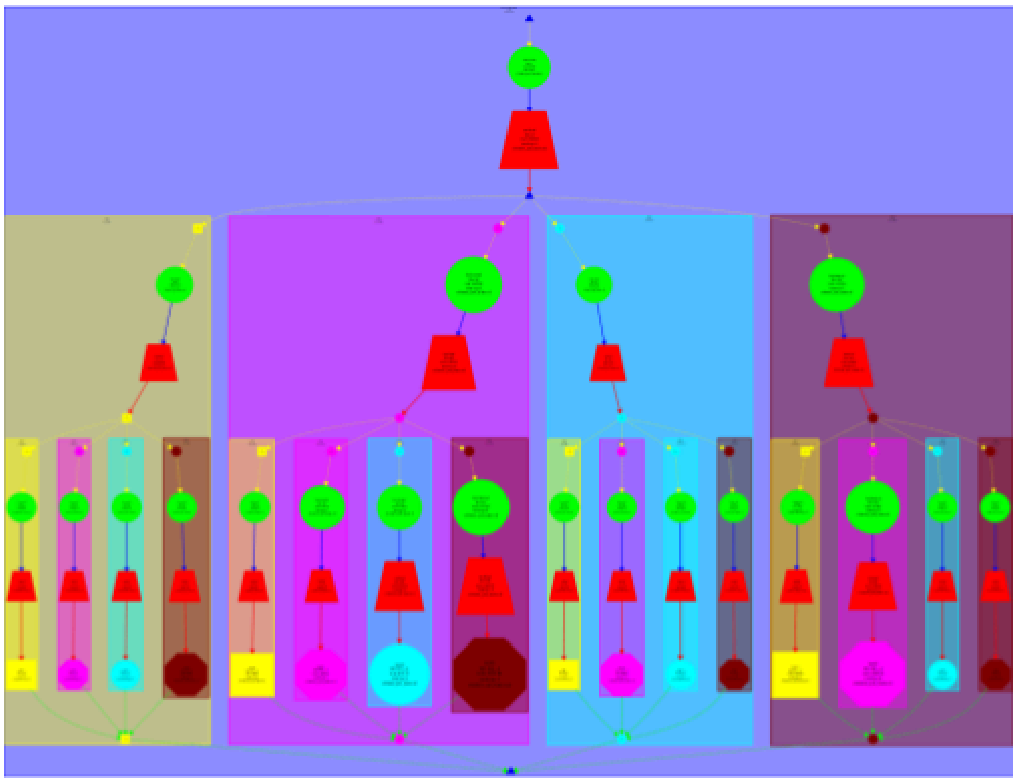


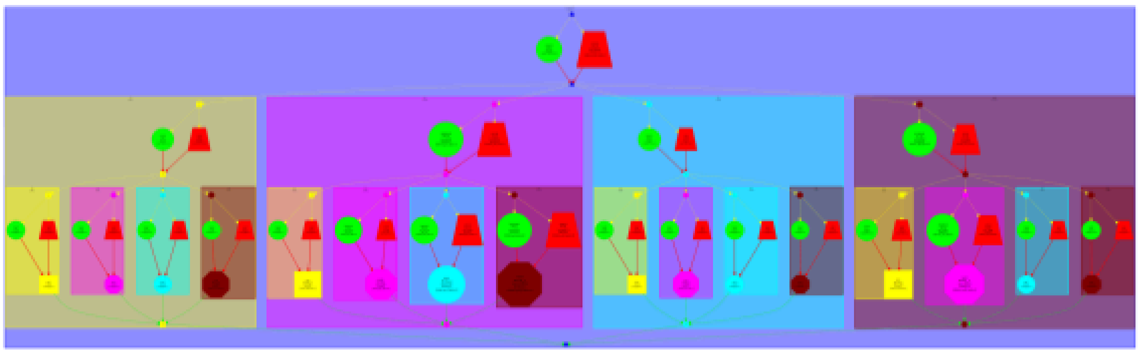
Leaf recursive running in 16 threads

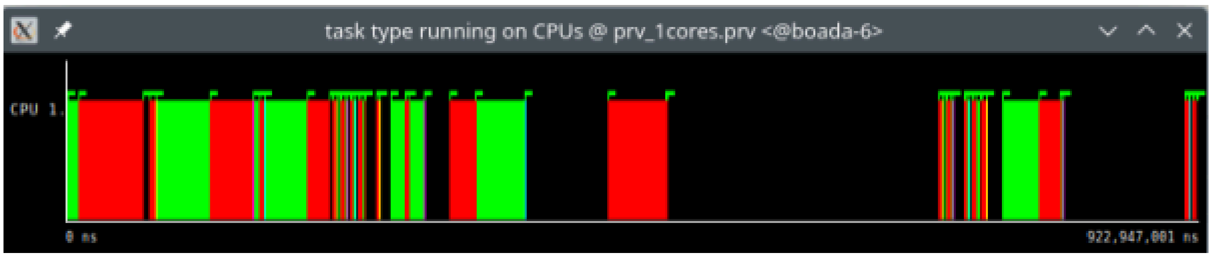
We can also observe load unbalance for the threads, we can see how the first one has a big cost to create all the tasks while the core 9 has the biggest workload. While threads 12-16 have almost no workload.

Tree recursive strategy

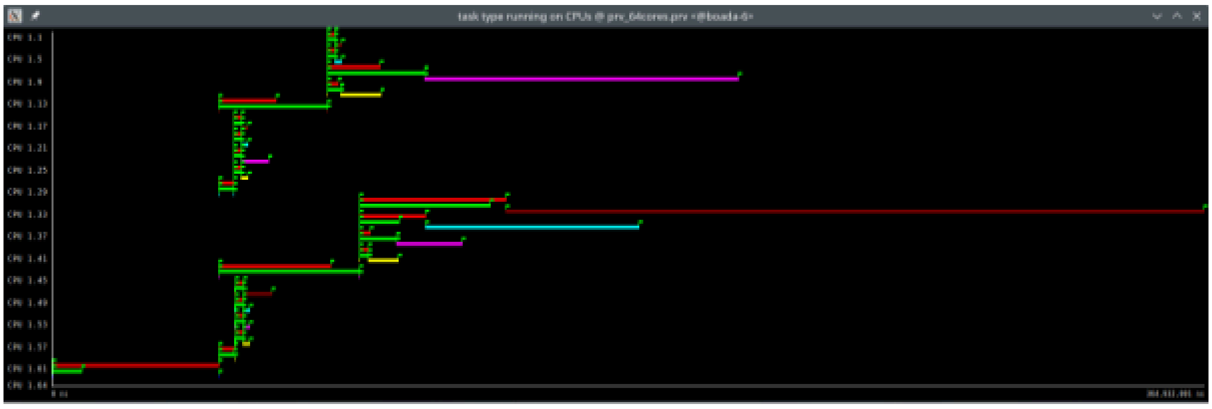
First we have used the same strategy as in the finer grain, for the decomposition of the two tasks that check the tile edges. Then we also add a task for each recursive call we make to the function, thus implementing the tree strategy.
The first dependency we see in the TDG is the equal variable between the vertical and horizontal tasks. What we can do to solve this dependency is to make a manual reduction. For this, we create two new variables that will be equalH and equalV, then for the execution, we use their respective equals instead of the global. Then, just before executing the if-else task, we make equal = equalH & equalV (it is important that it is the binary AND, this way we do not have lazy evaluation). Then, for the tree recursion, we create a task for each recursive call.


TDG before deleting data dependencies


TDG after deleting data dependencies



Recursive tree strategy running in one thread



Recursive tree strategy running in 128 threads

As can be seen in the second graph, there is a large load unbalance especially with the middle thread, while the ones at the beginning (top) and at the end (bottom) have a very short execution time.