**Development Methodology**

In terms of development methodology, we defined the following strategy: 1st) identifying all *loops dependencies*; 2nd) checking the possibility of merges between the *for loops*; 3rd) in the *loops* that have removable dependencies, removing them so the loops can be parallelized; 4th) using a profiler to check which methods have the most calls and how that affects the performance of the program. During the development and analysis stage, this process had to be repeated to check if everything remains correctly parallelized after all the changes.

**Dependency Analysis and Code Improvements**

1st

As previously stated, we started by identifying all the *for loops* dependencies. This way we could check whether each loop could be directly parallelized (using *omg parallel for*). In this first step, we found that:

* the loops in 4.2 (4.2.1 and 4.2.2) and 4.1.1 (the one with the update function) did not have dependencies. So they could be directly parallelized.
* the loop 4.3 had an output-dependency and flow-dependency.
* the loop 4 have all the three dependencies (flow, output and anti).
* The loop 4.1 have an output-dependency.
* We didn’t detect any anti-dependency.

2nd) CHANGED DR 20210601 after HUGO revision

We then tried to look for loops with the same parameters to check the possibility of a merger. In this step, we chose the 4.3 and 4.2.2 *for loops*: we tried several approaches but due to an flow-dependency and anti-dependency, detected after merging the loops:

layer[k] = … and if (layer[k] > layer[k - 1] && layer[k] > layer[k + 1]) → anti in k+1, flow in k-1  
(line 256)

Due to this, we concluded that this merge is impossible and returned to the previous form.

3rd)

As previously acknowledged, we detected an output-dependency on the 4.3 *loop for*. This dependency was resolved by splitting the loop in two: a parallelizable one, that uses two auxiliar arrays and the number of the thread to calculate local thread maximums, and the second one that compares the thread maximums to find the storm maximum and its position. With this change, we were able to improve the overall performance.

4th) CHANGED 20210601 DR after HUGO REVISION

In the last step of our strategy, we used a profiler: we experimented with multiple profiling tools (like Valgrind with Callgrind tool as well as CLion IDE profiling tool) where we retain the same conclusions. In the following IMAGE X, obtained by using Kcachegrind vizualizer for the Valgrind-Callgrind profile using the sequential version of the program, we obtain an understanding of which methods have the most impact on the performance of our program after every possible improvement we made. With this tool we can check all the call tree that let us know which methods consume longer processing time. In the last version of our program, the majority of the calls were to the *update* method and the remaining were calls to procedures concerning the read of the files.   
Also one thing that we notice from the other profiler tool (CLion IDE profiling tool) was that one of the method that has more samples (meaning, one that is most “called”), together with the *update* method, were the procedures concerning the calls to *omg parallel*. From this we can retain that some operations are not worth it to parallelize because the gains from parallelize them are surpassed by the procedures concerning the parallelization itself (like opening threads and closing them).

(O HUGO FALOU QUE ESTA ÚLTIMA CONCLUSAO É MANHOSA, MAS PENSO QUE ERA IMPORTANTE MENCIONAR ALGO DO GENERO. FOI O MOTIVO DE NÃO TERMOS POSTO POR EX INICIALIZAÇÕES E ASSIM PARALELIZABLE)

IMAGEM DO PROFILING do VALGRIND (no commit do Hugo)