**Development Methodology**

In terms of development methodology, we defined the following strategy: 1st) identifying all *loop dependencies*; 2nd) checking the possibility of merges between *loops*; 3rd) in the *loops* that have removable dependencies, remove them in order to parallelize the loops; 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 has all the three dependencies (flow, output and anti).
* The loop 4.1 has an output-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 ended up concluding that this merge is impossible due to a 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 dependency in k+1, flow dependency in k-1  
(line 256)

As such, we returned the code to its 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 the Callgrind tool as well as the CLion IDE profiling tool) and maintained the same conclusions. In the following IMAGE X, obtained by using Kcachegrind visualizer for the Valgrind-Callgrind profiler using the sequential version of the program, we obtained an understanding of which methods have the most impact on the performance of our program. With this tool we can check the entire call tree which let us know which methods consume longer processing times. The majority of the calls were to the *update* method and the remaining were calls to procedures concerning the reading of the files which wasn’t accounted for the times obtained.   
Another point we noticed from the other profiler tool (CLion IDE profiling tool) was that one of the methods that had more samples (meaning, the one that was most “called”), together with the *update* method, were the processes concerning the calls to *omp parallel*. From this information we can determine that some operations are not worth parallelizing because the gains from achieved through parallelizing such operations are surpassed by the costs of the procedures concerning the parallelization itself (like opening and closing threads).

(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)