**Development Methodology**

(talvez denominar Development Methodology and First Analysis)

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.

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.
* We didn’t detect any anti-dependency.
* QUAL O TIPO DE DEPDENCY NO 4.1 e 4
* Acho que o 4 tem todas as dependências
* A única dependência que consigo ver no 4.1 seria de output

2nd)

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 anti-dependency detected after merging the loops, we concluded that this merger 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)

In the last step of our strategy, we used a profiler: we experimented with multiple profiling tools (like Valgrind Profiler using massif visualizer for example) and ended up choosing the CLion IDE profiling tool, as we found it to be especially helpful in the 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 stree and method list that let us know which methods have more samples registered, in the sense of, the methods that are called more often. In the last version of our program, the majority of the calls were to the *update* method and the rest were calls to procedures concerning the *omp parallel*. We then tried to optimize the *update* method but unsuccessfully.

IMAGEM DO PROFILING