**NOTAS:**

* No 2º Capitulo, o colocar de 1st, 2nd, 3rd, etc., é para eliminar. Não é para se colocar no relatório. É so para ser mais fácil nós percebermos enquanto elaboramos o capítulo.

**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.
* We didn’t detect any anti-dependency.
* 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) CHANGED 20210530 DR

In the last step of our strategy, we used a profiler: we experimented with multiple profiling tools (like Valgrind and CLion IDE profiling tool) where we retain the same conclusions. In the following IMAGE X, obtained by using Valgrind with a visualizer (qual é nome do visualizer Hugo?), 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. We then tried to optimize the *update* method but unsuccessfully.  
Also one thing that we notice from the other profiler tool (CLion IDE profiling tool) was that one of the most 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 the method are surpassed by the procedures concerning the parallelization itself (like opening threads and closing them).

IMAGEM DO PROFILING do VALGRIND (no commit do Hugo)