Part 1 :

At the first place, we used a simple parallelization where thread all increment value in the original array. This parallelization caused a lot of true sharing since the same array is cached and modified by the different cores. This leads to a lot of time lost in sharing updated value between cores caches, this cause to nullify the benefits of parallelization, it makes the program twice slower than without parallelization.

Number of threads	computation time (5 runs average)	speedUp
1	0.97032s	1
2	1.8428s	0.53
4	1.9044s	0.51
8	1.9196s	0.51

Table 1: Simple parallelzation run on cluster

We implement another version with smarter parallelization. Each thread works on different arrays that are then merged, this removes the sharing operations between threads during the computation. With this version we saw that in average, the paralellization has the expected effect. With 2 threads, it has no impact, this is probably due to the merging process that takes some time to execute. With 4 and 8 threads, we see a speedup of respectively 1.8 and 3.2 which is quite good. Note that different computations lead to relatively large difference in the execution time (on cluster), that's why we used a average of 5 differents computation.

Number of threads	computation time (cluster) (5 runs average)	speedUp
1	0.97032s	1
2	1.04572s	0.93
4	0.5428s	1.79
8	0.30724s	3.16

Table 2: smarter parallelzation run on SCITAS

Additionnal note :
Running the program on my machine (6 cores) leads to more constant result through different running. Futhermore, paralellization reach better result that in cluster (even with less cores for 8 threads computation)). I suppose that this is because merging is done faster due to smaller numbers of cores that makes sharing faster. First version also show bad impact of true sharing however, it's not as bad as in the cluster. It doesn't makes computation slower,

it just doesn't make it faster (not sure of how it works, it seems my laptop manage to use all 6 cores in 4 threads computation, and there's multithreading too).

I tried to push comparaison further for curiosity, for 100x more sample and 28 threads, my computer: 12.07s, cluster: 45.11s.

Number of threads	bad version	good version	speedUp (good version)
1	1.152s	1.143s	1
2	1.483s	0.58s	1.97
4	1.75s	0.3159s	3.62
8	-	0.2142s	5.34

Table 3: run on local machine (Intel core i9 (6cores)

Part 2 :

1. 1. Algorithm explanations

   (a) Version 1: Simple process parallelization The first optimization level is to consider the 2-dimensional array as an index array, and this is already done with the provided clause. The next optimization is straight forward parallelizing the main work in the nested for structure. To achieve this, we used the collapse(n) directive which allow to collapse the two-level nested for into individual operations distributed on the threads.

   (b) Version 2: Parallelize with sharing optimization The next step is to consider the cache access and to make parallelize row by row so that we avoid accessing to many times to data that is not in the cache anymore. To achieve this, we separate the process in two helper functions updateRow and copyRow. The first one takes care of updating the values on the grid and the second one copies the values from the input to the output. We create first the threads with pragma opm parallel and then we parallelize the two loops with pragma omp for since we have two loops its more performant to separate the omp for declaration instead of using the pragma omp parallel for directive on the outer loop.

   (c) Version 3 (final): Use of an if statement to avoid parallelizing on to small array where the overhead brought by the merging of the threads and distribution of the work would be too important compared to the advantage of parallelization. From itterations we can see that the parallelisation is irrellevant for array less than 26. This come from the fact that merging data has a cost and at small size the parallel speedup is not paying back for the merging cost.

2. Optimization result analysis We decide to run on a demo of 1000 by 1000 array on 500 itteration from 1 to 8 cores. First the non parrallel version runs in 1.6 seconds on 1 core.

The following results are showing worse results on one core for the parrallel version. This can come from two elements. 1 the parallezation overhead 2 the inconsitency over runs on SCITAS (we got to rerun some of our stress test to obtain consistent results)

The simple parrallelization of the main code (update of the rows in the grid) with the collapse statement is already bringing the time on 16 threads down to 0.42s and the final optimisation with the parallelization of the writes is bringing down the time to 0.26s. which lend to a speedup of 1..6x.

The version 3 of our optimisation does not change the final results since it only concerns an optimisation for small arrays.

Number of threads	Running time (in seconds)
1	2.813
2	1.416
4	0.7172
8	0.4224

Table 4: Algorithm version 2 on SCITAS cluster

Number of threads	Running time (in seconds)
1	1.953
2	0.9772
4	0.513
8	0.2649

Table 5: Algorithm version 3 on SCITAS cluster

3. Presentation of the results

Number of threads	Running time (in seconds)
1	47,75
2	24,82
4	13
8	8.511
16	7.049

Table 6: Run of the optimized heatmap algorithm on SCITAS cluster