# PDC Summer School – Project report

Gabin Schieffer
KTH

August 2023

## Introduction

In this project we investigate ways to parallelize the application "Storms of high-energy particles". We propose three version of a sequential *baseline* code: OpenMP, MPI, and CUDA.

The code is available at https://github.com/gabin-s/pdc-summer-23-project.

For each version of the code, we evaluated the correctness for all test cases, against the reference sequential implementation. All experiments are repeated 10 times, and all results are shown as average values.

## OpenMP

For this implementation, we parallelize the code in the main function, by adding pragma directives in the code. We also swap two loops to further improve the effectiveness of the parallelization. For this version, we run our tests on Dardel's *main* partition. Each node has two AMD EPYC 7742 CPUs, resulting in a total of 128 cores per node (256 threads).

The parallelization strategy chosen for the OpenMP version is to divide the iterations over the particles, across all threads – each thread handles the update of a specific portion of the `layer` array. For our evaluation, we use the static schedule, as all computations in the threads are predictable, and using the guided or dynamic schedules is not worth it due to the incurred overhead – albeit very limited, this impact was confirmed by our testing.

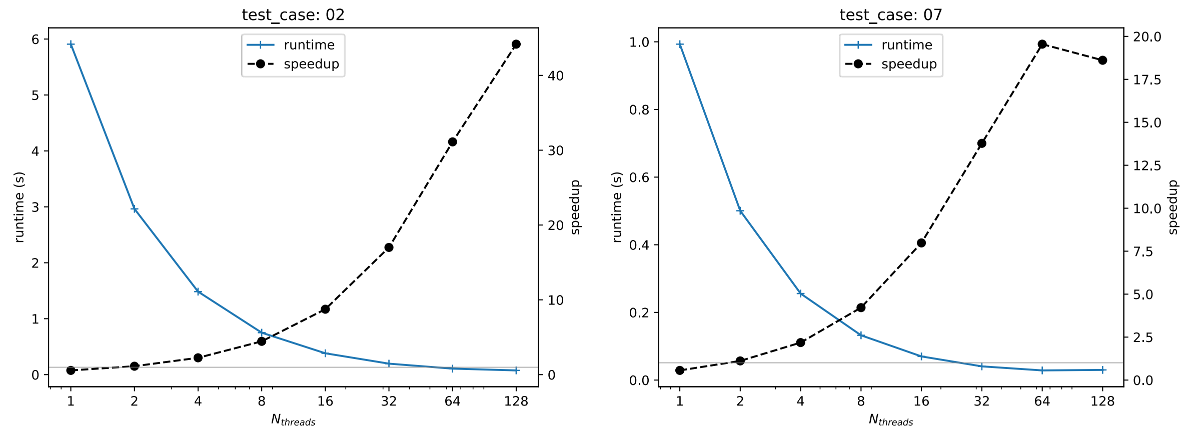

*Figure 1: runtime and speedup for two test cases (02 and 07), using 10000 cells.*

Figure 1 shows the runtime and speedup obtained when running our Open-MP-parallelized code, for two test cases provided in the original repository. Those test cases were chosen as their problem allowed for testing on higher number of threads. Indeed, when using other test cases, poor results were observed as the number of threads increased. This is because the problem sizes for those test cases (here, the number of particles per wave) were too small, and parallelizing the algorithm was not worth it.

Future improvement can be done to detect those limited-size problems and treat them in single-threaded fashion.

We observe that the speedup only exceeds 1 when using 4 threads or more. We achieve a maximum speedup of ×44, for the test case `02` using 128 threads, and ×20 for the test case `07` using 64 threads. Interestingly, while the maximum speedup for the 02 test case is reached for the maximum number of threads, the speedup for the 07 test case decrease when increasing the number of threads above 64, namely 128 threads.

## MPI

We use a similar approach to parallelize the code with MPI as with OpenMP: we divide the full domain in same-size chunks, where each MPI process handle the workload for one chunk. Note here that the last chunk may manage a larger domain, as the number of cells may not be divisible by the number of MPI ranks.

**Communication.** In this problem, the update for one cell needs information about the two-neighboring cell. As each process perform computation on different parts of the whole domain, information about neighboring cells for the first and last cell of the local domain is not trivial to obtain. In order to solve this problem, the local domain is created so as two "ghost" cells are added to the domain – one at before index 0, and one after the last index. Those ghost cells are filled by using two `MPI_Sendrecv` calls. This approach limits code modifications, while preserving correctness and maintaining readability.

**Reduction**. One maximum-finding task is required for each wave in the simulation. To solve achieve this operation, we use MPI_Reduce, with the `MPI_MAXLOC` operation, which combines both a *max*, and *argmax* operations in a single `MPI_Reduce` call. The result is received by the first thread (rank `0`) and printed at the end of the program.

For this version, we run our tests on Dardel's *main* partition. Each node has two AMD EPYC 7742 CPUs, resulting in a total of 128 cores per node (256 threads).

### Single-node evaluation

We first evaluate our implementation in a single-node configuration. We limited our evaluation to 128 MPI processes, as using more processes yielded worst performance, due to hardware limitations. We use a problem size of 10000 cells for this set of experiments.

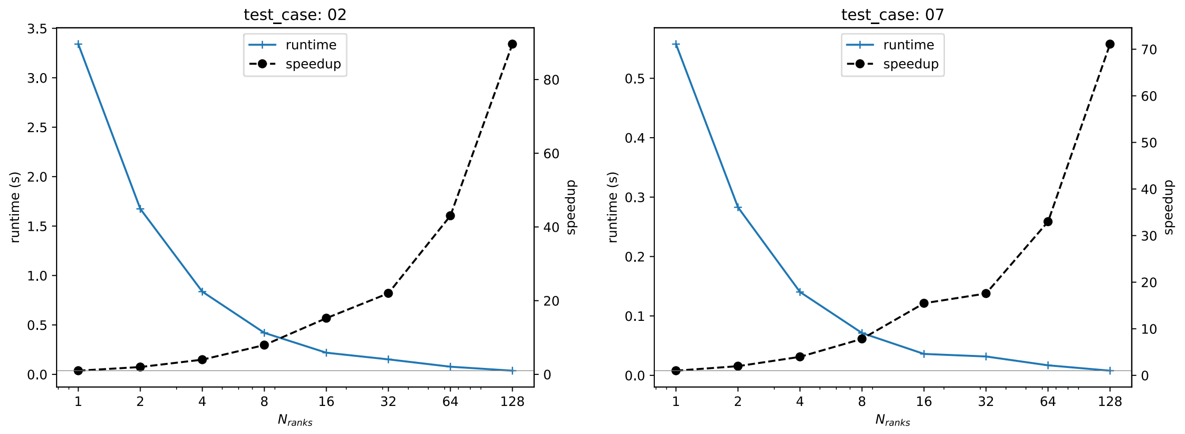

*Figure 2: single-node run of the MPI-parallelized version, with a fixed 10000-cell problem size*

Figure 2 shows the runtime for the single-node evaluation. We obtain a consistent speedup as the number of ranks is increased: ×89 for the test case `02`, and ×71 for the test case `07`; both obtained for the maximum number of MPI processes (128).

### Multi-node evaluation

Our multi-node evaluation did not yield any higher speedup compared to the single-node evaluation. This highlights the limits of our implementation, which is unable to scale on multiple nodes. Future work may focus on improving the implementation to be able to scale efficiently on multiple nodes.

## MPI & OpenMP comparison

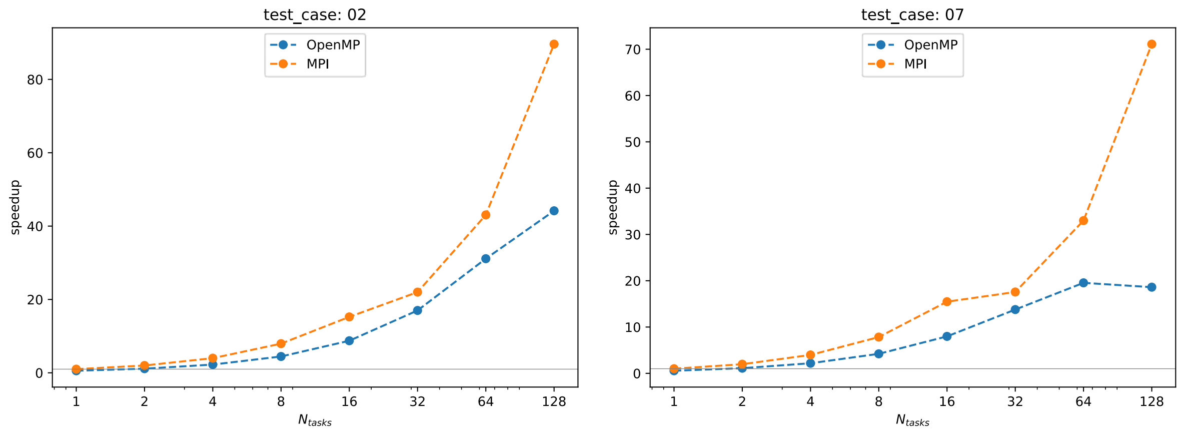

*Figure 3: Speedup for two test cases, with a problem size of 10000 cells, compared between OpenMP and MPI.*

We compare the speedup between the OpenMP and the MPI implementations. We observe that the MPI systematically provides a higher speedup than the OpenMP implementation, for all amount of threads/process. We make the hypothesis that the poor performance of the OpenMP parallelization is due to an attempt to parallelize as much as possible in the OpenMP version (including some lower-cost loops) – which causes a significant amount of thread synchronization, which could lead to lower scalability of the code.

## CUDA (GPU-accelerated)

We implemented a GPU-accelerated version of the simulation using the CUDA platform. We use a single kernel for this simulation, which is launched after the storm-relative data is transferred to the device from the host. This kernel uses one thread block, where each threads handle computing for part of the cells in the domain.

Our testbed to evaluate this CUDA implementation, our testbed is equipped with a single AMD EPYC 7302P CPU, 256 GB DRAM, and two NVIDIA A100 PCIe 40GB – of which only one is used in our tests.

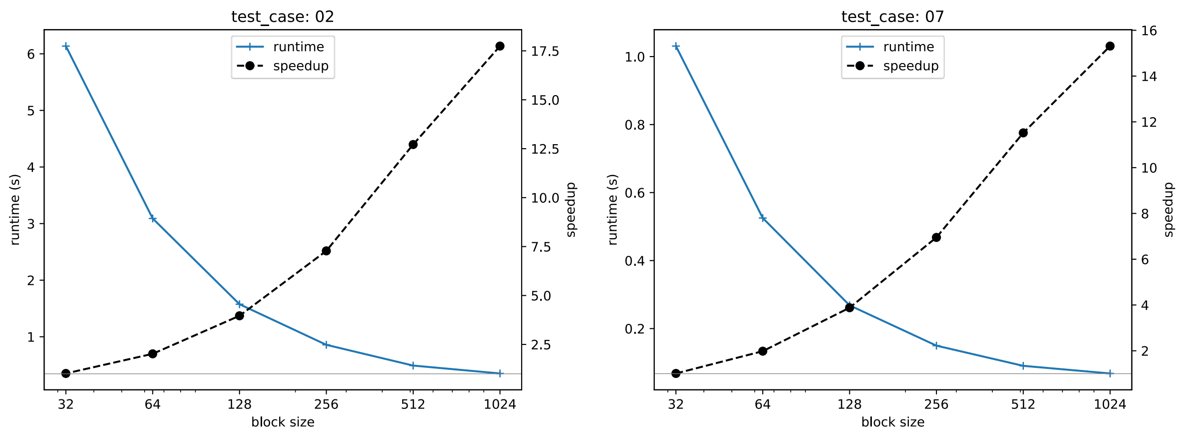

*Figure 4: runtime and speedup of our CUDA implementation over the sequential version, on A100, for various block sizes and a fixed problem size of 10000 cells, using two of the provided test cases.*

Figure 4 shows the results of our evaluation for a 10000-cells problem size, using increasing block sizes. We use a minimum of 32 threads per block as the hardware executes code in parallel 32 threads at a time (referred to as *warps*), using a lower number of threads per block would thus under-utilize the hardware and yield worst performance. The higher bound of 1024 threads per block is a constraint of the CUDA programming model. We observe that our implementation provides a speedup above one for all test cases, and all block sizes. Increasing the number of threads per block reduces the runtime. This is clearly expected as the hardware is largely under-utilized in our test case, as our implementation can never exceed 1024 concurrently executing threads, while the hardware can execute a far higher number of threads in parallel.

While our approach yields satisfactory results with regards to the sequential version, it does not leverage the ability of GPUs to launch kernels with several independent thread blocks, to further increase parallelization, and achieve higher performance; future work could focus on achieving this further level of parallelization. Possible improvements also include using warp shuffle operations to improve the argmax reduction operation.

## Conclusions

In this project, we implemented three parallelized versions of the energy storm simulation: OpenMP, MPI, and CUDA. All those versions provided consistent speedup over the sequential baseline. However, further improvement could be done to further increase performance. For example, a combined OpenMP-MPI approach could be interesting to investigate. In addition, our CUDA implementation does not leverage the two-level parallelization allowed by the hardware (block/threads) – transforming our implementation to follow the best practices of GPU programming would significantly improve performance and could even surpass the CPU implementations.