Storms of high-energy particles

Project for the PDC summer school 2023

Victor Baconnet - Yu-Cheng Lu - Clément Marsone

August 2023

KTH Royal Institute of Technology

1 Introduction

This project performs the simulation of high-energy particle storms on a layer of cells. The simulation is divided into two phases:

- A bombardment phase, and
- A relaxation phase, which use a three-point stencil to average the energy of each cell.

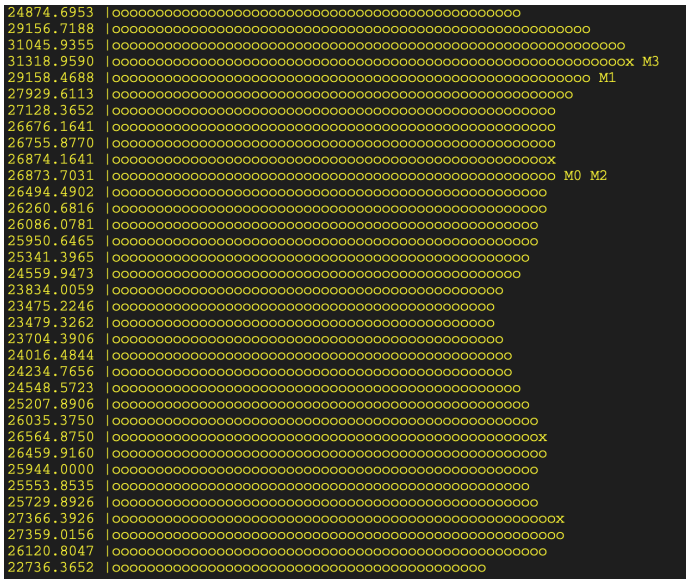

Figure 1: Energy distribution of particles bombardment for 35 cells, where the labels M0,M1,... identify the cell with maximum energy after the bombardment of particles from storm 0, 1, ...

Each storm contains a certain number of particles, whose data is stored in different test files, used as an input to the code. For each storm, and after the bombardment and relaxation phase, the program determines the cell with maximum energy and its position in the layer. An example of this process is shown on figure 1. To parallelize this code, we compared 3 different strategies:

- CPU parallelisation with `OpenMP` on shared memory,
- CPU parallelisation with `MPI` on distributed memory, and
- GPU parallelisation with `CUDA`.

. A detailed comparison with regards to performance and computational time is given in section 6.

2 Parallelisation Strategy

To successfully parallelize the code, the focus will primarily be on the bombardment and relaxation phases. These phases involve computationally intensive tasks that can be divided

among multiple threads or processes. Parallelizing these segments, along with potential optimizations in data structures, should lead to improved performance, making the simulation process more efficient and scalable.

It is essential to emphasize that while the primary objective is to enhance performance, strict adherence to the prescribed parallel programming model and compliance with the assignment's limitations must be upheld during code modifications. Additionally, thorough testing and validation are imperative to guarantee the correctness and efficiency of the parallelized code.

3 OpenMP implementation on Shared Memory

Parallelizing the provided C code using OpenMP involves introducing parallelism in the main script to simulate the energy particle storms with the nested for-loop. This parallelization aims to reduce execution time and make efficient use of multiple CPU cores. After making these modifications, we can compile the code with OpenMP support and evaluate its performance. To compile the code with OpenMP support, we typically add the `-fopenmp` flag to our compiler command.

3.1 Bombardment phase

In the main storm simulation loop for adding the energy impacts, the command `#pragma omp parallel for default(none) private(k) shared(layer,layer_size,position, energy)` is used to parallelize the loop. Each thread works on a separate storm simulation, and data from different storms are independent, making parallelization feasible without synchronization issues. We assigned all variables to be shared except for `k` to be private. The reason for this declaration is because those variables are used in the whole calculation but `k` is only used in this for loop. Therefore, `private(k)` makes the local copy to each thread, which saves memory and reduces part of the data transportation.

3.2 Relaxation phase and maximum search

The first for-loop copies the energy value from the local cell `layer` to the local cells `layer_copy`. The second for-loop then calculates the average energy value and updates the cell `layer` from those local cells. The OpenMP commands here parallelise the for-loop with `#pragma omp parallel for`. Since the information of `layer` and `layer_size` kept updating at every stage, they were assigned to be shared. The for-loop with nested conditional statement finds the cell with the maximum energy accumulation. Similar to the other for-loops, `#pragma omp parallel for` is implemented to parallelise the iteration.

Computing on 32 threads on the Dardel cluster, the OpenMP parallelisation reduces massive computation time in the cases test02 and test07, which are composed of a large number of cells as well as particles in a single wave of storms. However, for the small amount of iteration, especially small numbers of particles, OpenMP has no advantages on the performance. The detailed comparison is presented in Section 6.

4 MPI implementation on Distributed Memory

Our MPI implementation is mainly based on subdividing the cell layer among all the processes and letting each rank perform their local `update()`, `relaxation()`, and maximum search.

4.1 Bombardment phase

The first step to achieve this is allocating a so-called "local layer size" to each rank. This is done by the function `redistribute_layer_size()`. Additionally, we use a `local_sizes` array, which contains the `local_layer_size` of each rank. This will come in handy later when converting from local to global indices, which will be explained below.

The reading of the test files is done by `rank 0`, which then broadcasts storm information to all other ranks.

With all ranks having their own chunk of cells and all relevant storm/particle information, the `update()` is called by each rank, using the `position` and `energy` of each particle `j` in a given storm `i`. Our modified `update` function uses additional parameters, like the `local_sizes` array. It is used to make sure that the `distance` between the local cell index `k` and the impact point of a particle `j` is calculated based on the global position of the cell `k`. The mapping from local to global position is done by the `get_global_index()` function, which uses the `local_sizes` array to return a global index given a `local_index` and a `rank`.

4.2 Relaxation phase and maximum search

After copying their local cell layer to a local copy `layer_copy`, each rank enters the `relaxation` function. The relaxation at each cell `k` involves a three-point average using the neighbouring `k-1` and `k+1` cells. This is quite straightforward for all inner cells but requires some communication at the edges. Each `rank` needs to send the first (resp. last) cell of the local cell layer to their `rank - 1` (resp. `rank + 1`) neighbour. This means that each `rank` receives the last (resp. first) cell of the layer belonging to their `rank - 1` (resp. `rank + 1`) neighbour. The order of the communication is done differently depending on if `rank` is odd or even. This allows for faster communication between the ranks by avoiding unnecessary waiting by each rank on the `Mpi_Recv` instructions.

The final step is the search for the global maximum among all the cell layers, and its (global) position. Firstly, the `find_maximum()` function searches for the local maximum of each cell layer as well as its local position. Secondly, we find the global maximum with an `MPI_Allreduce` using the `MPI_MAXLOC` operation. This operation allows us to identify the rank that owns the maximum, which in turn can help us in computing the global position of the maximum. To achieve this, we use a simple `struct` made of a `float` and an `int`. Passing this `struct` to the `MPI_Allreduce` with an `MPI_MAXLOC` operation will store the global maximum in the `max` attribute, and place the owning rank in `rank_owner`. Once the owning rank is identified, it will broadcast the local position of its maximum to all other ranks, allowing us to compute the global index of the maximum with `get_global_position`.

5 CUDA implementation on Nvidia® GPU

For the CUDA implementation, and similarly to the MPI implementation, the natural choice was to parallelize the cell layer. Our program makes use of two kernels, one for the `update()` and one for the `relaxation()` phases.

5.1 Bombardment phase

For the bombardment phase, we added the `bombardment()` kernel, which calls the modified `update()` function. Essentially, each thread block `k` will loop through all the particles and call the `update()` function, which will only modify `layer[k]`. For the cases where there are more cells than available threads, we use a grid stride technique which makes sure that the entirety of the cell array is parsed.

5.2 Relaxation phase and maximum search

To account for the three-point averaging, we use ghost cells in shared memory, where we allocate a `__shared__` array with two extra elements at the edges to receive the edge elements from neighbouring blocks.

The `bombardment()` and `relaxation()` kernels are called separately with the same number of threads per block and blocks.

The array of cells is allocated on the GPU before the entire storm simulation, whereas the `posval` array is allocated at every storm iteration `i`. This is because each storm may have a different size, therefore we need to make sure the GPU receives the proper arrays and sizes.

6 Results and performance comparison

Different computation strategies have their corresponding performance. Among the test cases, they can be categorised into three groups according to the relative number of cells and particles: (1) few cells with few particles, (2) massive cells with massive particles, and (3) massive cells with few particles. These three categories enable us to understand how the parallelisation works. The CPU computation was performed on the Dardel cluster (PDC, KTH) with AMD EPYC^{TM} processors, and the CPU parallelisation ran with 32 threads/`MPI` ranks. The GPU computation was performed on the Alvis GPU cluster (C3SE, Chalmers) with NVIDIA® A100 GPUs.

In test01, consisting small number of cells and particles per wave, the serial computation has the best performance, followed by the GPU parallelization. Regarding CPU parallelization, the `MPI` outperforms the `OpenMP`. Because of the small amount of iteration in the for-loops, the serial code directly computes the result without communication between threads/cores. It saves time for data transfer. In contrast, `OpenMP` with shared memory parallelization strategy requires information transfer between computing cores and memory, which is less efficient in this case. On the other hand, `MPI` has better performance on CPU parallelisation since each thread iterates the loop with its distributed memory. Regarding

large amounts of cells and particles, such as test02 and test07, GPU parallelization has the best performance.

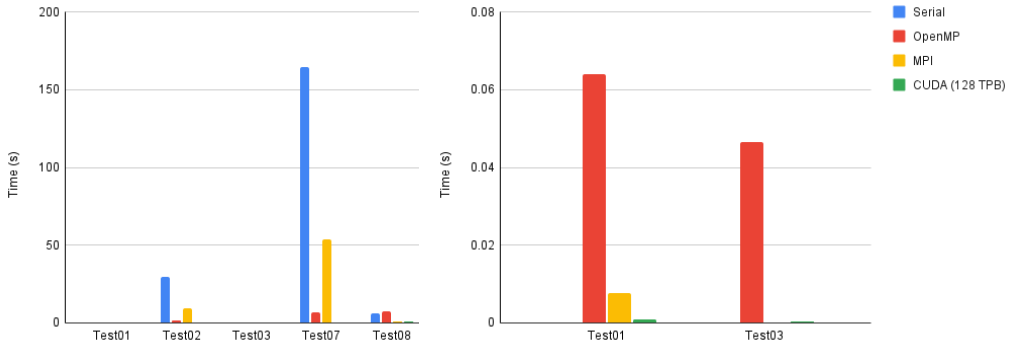

Figure 2: Performance comparison. Left: All cases, Right: Zoom-in for Test01 & Test03

7 Conclusion & Future work

We have shown three different techniques to parallelize the storm simulation code. Some possible improvements include:

- The parallelization of the maximum search in the CUDA code. Having a maximum search on the device would keep the cell layer array in the GPU memory, and we would only need to return the maximum value and its position in the array. This would avoid to move the cell layer array at every storm iteration.
- Parallelizing the `update()` function in the CUDA code to have each thread perform the bombardment for 1 cell and 1 particle. This would involve using 2-dimensinonal blocks and grids. However the performance gain would only be noticeable for a large number of particles (e.g. test 02).
- The communication in the MPI code for the relaxation phase could probably be handled a bit more efficiently and combined a bit better, but we were not able to find any better way to do it so far.

Table 1: Results of Computation Time, unit: [ms]

Case	Serial	`OpenMP`	`MPI`	`CUDA`
test01	0.017	63.879	7.68	0.925
test02	29728.637	1702.592	9450.0	38.355
test03	0.011	46.486	0.124	0.377
test07	164588.572	6897.874	53839.04	113.747
test08	6238.676	7296.922	941.243	909.151