PDC SUMMER SCHOOL 2023 PROJECT
THE PARALLELIZATION OF STORMS OF HIGH-ENERGY PARTICLES SIMULATION

Abhijeet Vishwasrao[1], *Pol Suarez*[1]

[1] ***FLOW, Engineering Mechanics, KTH Royal Institute of Technology, Stockholm, Sweden***
avis@kth.se, polsm@kth.se

1 Introduction

In the harsh environment of outer space, space vessels and satellites face continuous bombardment by high-energy particles originating from various sources such as the sun, cosmic rays and interstellar medium, relentlessly bombard the surfaces of space vessels and satellites. These particles, consisting of protons, electrons, and other subatomic entities, carry immense kinetic energy. When they strike the exposed surfaces of spacecraft, they can cause a cascade of events, leading to the accumulation of energy and potential damage to the material, potentially compromising their functionality and mission objectives. To better understand and mitigate the risks associated with such particle bombardment, sophisticated simulations are essential.

This report explores the parallelization of a sequential code called as energy storms project developed by Arturo Gonzalez-Escribano and Eduardo Rodriguez-Gutiez in Group Trasgo, Universidad de Valladolid (Spain). The sequential code is designed to simulate the effect of high-energy particle storm bombardment on 1D geometry. Specifically, we focus on parallelizing this code using three distinct technologies: OpenMP, MPI, and CUDA. By leveraging the computational power of both CPUs and GPUs, we aim to enhance the efficiency and performance of the simulation, ultimately providing more timely insights into the vulnerabilities of spacecraft in outer space.

2 Problem Description

The code simulates interactions between storms of high-energy particles and materials outermost layer. The code represents the surface as a discrete set of control points, each storing the accumulated energy from particle impacts (see figure 1). The program calculates, for each storm, the point with the highest accumulated energy, which presents a higher risk of being damaged.

The program initially reads wave files and stores them as arrays of particles. For each wave, it follows a consistent process. First, during the **bombardment phase**, it converts particle energy to Joules and iterates through array positions to calculate energy accumulation at each point. Energy is transmitted from impacting particles to contact points and neighbouring ones, with the exact accumulation considering attenuation based on the distance from the impact (**Relaxation Phase**). Energy is not accumulated at points too distant from the impact to meet a minimum threshold, determined by the material. Points failing to reach this threshold due to distance do not accumulate energy from the impact. Figure 2 shows the output of the serial code in debug mode where we can see 35 control points after two waves impacted the outer layer of the material. In each line, the energy stored in each control point can be seen. The control point with the last character 'x' shows the point where maximum energy is stored after the impact of all waves.

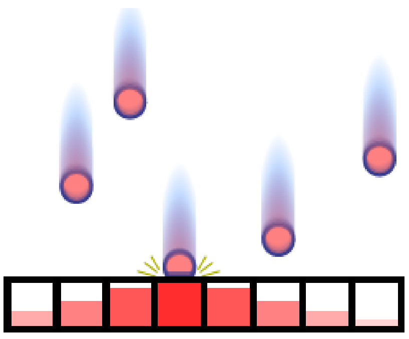


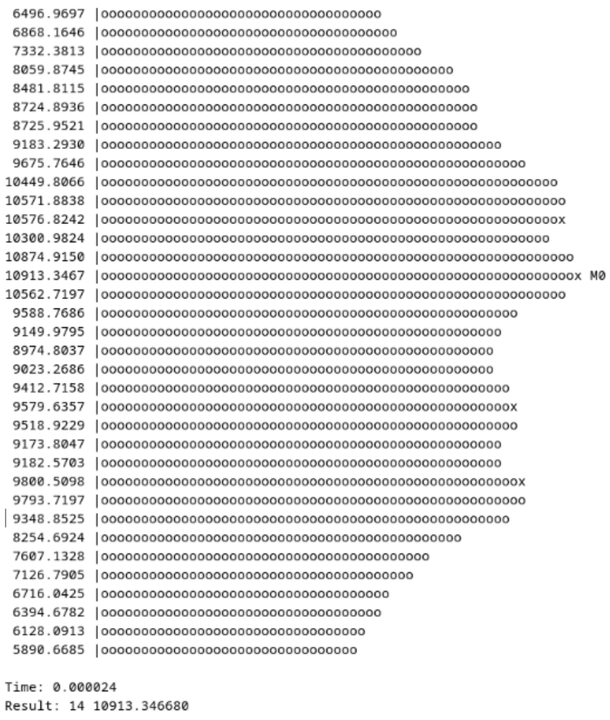
Figure 1: A collection of high-energy particles approaching the target surface. The impacting particle transfers energy to the impact point and to its neighbourhood

In this study, we will check the performance for GPU, OpenMP and MPI parallelization strategies for four different types of test cases, of different computational magnitude. This tests are `test_01_a35_p5_w3` for 35 control points, `test_02_a30k_p20k_w1` for 30000 control points, `test_07_a1M_p5k_w1` for 1M control points and `test_08_a100M_p1_w1` for 100M control points.

3 Parallelization

The basic Idea behind parallelization

The natural way of paralleling a sequential code is to take a bottom-up approach. Following this first,



```
 6496.9697 |ooooooooooooooooooooooooooooooooooo
 6868.1646 |ooooooooooooooooooooooooooooooooooooo
 7332.3813 |oooooooooooooooooooooooooooooooooooooooo
 8059.8745 |oooooooooooooooooooooooooooooooooooooooooooo
 8481.8115 |oooooooooooooooooooooooooooooooooooooooooooooo
 8724.8936 |ooooooooooooooooooooooooooooooooooooooooooooooo
 8725.9521 |ooooooooooooooooooooooooooooooooooooooooooooooo
 9183.2930 |oooooooooooooooooooooooooooooooooooooooooooooooooo
 9675.7646 |ooooooooooooooooooooooooooooooooooooooooooooooooooooo
10449.8066 |ooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
10571.8838 |oooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
10576.8242 |ooooooooooooooooooooooooooooooooooooooooooooooooooooooooox
10300.9824 |oooooooooooooooooooooooooooooooooooooooooooooooooooooooo
10874.9150 |ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
10913.3467 |ooooooooooooooooooooooooooooooooooooooooooooooooooooooooooox M0
10562.7197 |oooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
 9588.7686 |oooooooooooooooooooooooooooooooooooooooooooooooooooo
 9149.9795 |oooooooooooooooooooooooooooooooooooooooooooooooooo
 8974.8037 |ooooooooooooooooooooooooooooooooooooooooooooooooo
 9023.2686 |ooooooooooooooooooooooooooooooooooooooooooooooooo
 9412.7158 |ooooooooooooooooooooooooooooooooooooooooooooooooooo
 9579.6357 |ooooooooooooooooooooooooooooooooooooooooooooooooooox
 9518.9229 |oooooooooooooooooooooooooooooooooooooooooooooooooooo
 9173.8047 |oooooooooooooooooooooooooooooooooooooooooooooooooo
 9182.5703 |oooooooooooooooooooooooooooooooooooooooooooooooooo
 9800.5098 |oooooooooooooooooooooooooooooooooooooooooooooooooooox
 9793.7197 |ooooooooooooooooooooooooooooooooooooooooooooooooooooo
 9348.8525 |ooooooooooooooooooooooooooooooooooooooooooooooooooo
 8254.6924 |ooooooooooooooooooooooooooooooooooooooooooooo
 7607.1328 |ooooooooooooooooooooooooooooooooooooooooo
 7126.7905 |ooooooooooooooooooooooooooooooooooooooo
 6716.0425 |oooooooooooooooooooooooooooooooooooo
 6394.6782 |ooooooooooooooooooooooooooooooooooo
 6128.0913 |ooooooooooooooooooooooooooooooooo
 5890.6685 |oooooooooooooooooooooooooooooooo

Time: 0.000024
Result: 14 10913.346680
```

Figure 2: Example of output of the program in debug mode for an array of 35 positions and two waves of particles.

we identify the possibilities of parallelization in the inner loops and then gradually go to the outer ones. For the energy storms program, we have a total of four loops, consisting of bombardment, relaxation, finding the maximum energy point and finally calculating the impact of each storm. Out of these, the loops for the bombardment and relaxation phase can be categorised as inner loops. Hence we must focus on eliminating these loops to accelerate the code.

4 Bombardment loop

The impact of high-energy particles at any control point on the top layer of the material also affects the nearby control points(see 1).

To simulate this, a variable is computed to calculate the distance between all the control points and the impact position. This variable is designed to give more weightage to the control points closer to the impact location. Using this distance and the attenuation, the energy stored in every control cell due to each impact is updated through a loop for all the cells.

5 Relaxation loop

After each storm, the program deploys a relaxation function, where the energy stored in all the cells is allowed to diffuse to nearby cells. For this operation, we replace the energy stored in the cell with the average energy stored in three consequent cells.

6 GPU parallelization using CUDA

Coding

The sequential code takes two inputs while running the code, viz. number of control points and the list of wave files (or storms). The wave file contains information such as the number of impact particles in the particular wave, location of impact and energy of impact. As the energy stored in a particular cell after the impact is a function of the distance between the impact point and the concerned cell, the location of impact for each particle is given in the wave file even for sequential code. However, from the perspective of parallelization, we can use this information to sort out all the particles in a storm based on their impact location. In the kernel `Bombardment`, we can task a GPU thread to perform all the impact energy calculations for the particular control point. After each loop for a control point location (in every storm file) we update the energy for that control point in all layers(for all storms) in `layer_d`. To do this practically, we copy the particle impact position data to the GPU's global memory and by identifying the GPU thread index using global indexing and assigning each cell to it, such as,

$$cell = blockIdx.x * blockDim.x + threadId.x \quad (1)$$

As explained above, in the relaxation loop we need to perform an averaging operation for three consequent cells. Hence to parallelize the relaxation loop we must use GPU's shared memory. The shared memory can be accessed by all the threads in a thread block. For this purpose, we need to break down the global array containing stored energy (after each storm) into tiles and allocate these tiles accordingly. The halo cells at the end of the shared tile are filled up such that leftmost neighbour is stored in the rightmost hallo cell and vice versa (see fig 3).

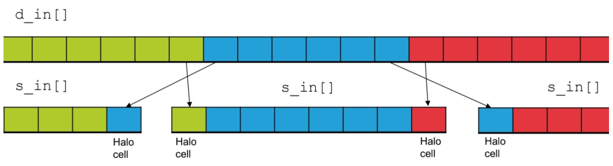


Figure 3: Allocating the hallo/ghost cells in GPU shared memory

Results

We check for the parallelization performance of CUDA GPU V100 on supercomputer Alvis from Chalmers with different threads per block in the x-direction for three different cases. In these cases, we increased the number of control points from 35 to 1M to check the scaling. We saw that for 35 control points, the GPU performed very poorly, worse than sequential code. The wall clock time remained almost constant as we increased the number of threads per block (TPB). For the case with 30000 control points, we got exceptionally good speedups of the

TPB	total time (sec) test 35	test 30k	test 1M
1	0,000005	1,6408	13,68
4	0,000331	0,022887	0,179695
8	0,000452	0,012511	0,086367
16	0,000501	0,005419	0,042639
32	0,000294	0,004635	0,023515
64	0,000252	0,004643	0,023669
128	0,000323	0,004656	0,023351
256	0,000417	0,005648	0,027021
512	0,000356	0,005725	0,027541

Table 1: Computational cost for CUDA in wall-clock time in 3 different cases with increasing threads per block(TPB)

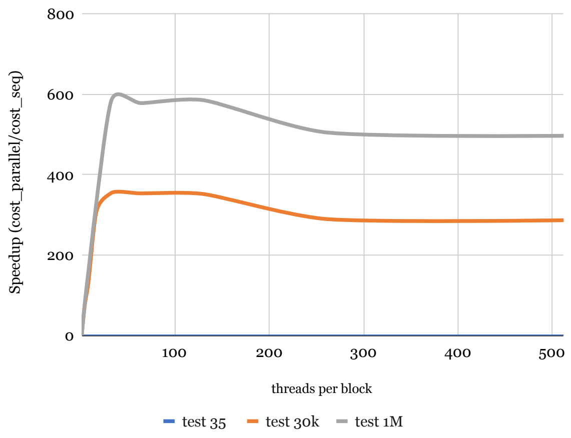


Figure 4: Speedup performance with CUDA parallelization

order of 100. The speed-up first increased linearly with TPB, to decrease to a constant value. Which showed that multiple GPU threads remained ideal for higher TPBs. For the largest case with 1000000 control points, we saw a similar trend as in the 30000 control points case with a better speed-up performance from all the threads. The overall results for all three cases show that GPU acceleration are much better, surprisingly more than we expected, as the computations increase. The GPU acceleration for the last test of 100M particles could not be performed due to the error `CUDA illegal memory access`.

7 MPI

Coding

As commented before, the main inner loops we have to focus on speedup through MPI parallelization are the bombardment and relaxation processes. In particular the function `update()` consumes almost all computational time. So, first of all we have to profile this specific loop to check before going to the outer loops. We did not take into account multiple storms from different test files. The study is applied to the case of single storms.

In order to prepare MPI directives, we should prepare our data depending on the rank, so a local and global index differentiation is needed. The loop to update the energy values per cell, is parallelized and now takes the values in a subdomain within the total `layer_size`, the `local_size`. So if this loop runs in `n` threads, will compute `n` times faster ideally. The modified loop looks like:

```
for( j=0; j<storms[i].size; j++){
    /* Get impact energy (expressed in thousandths) */
    float energy = (float)storms[i].posval[j*2+1] * 1000;
    /* Get impact position */
    int position = storms[i].posval[j*2];

    for( k=0; k<local_size; k++ ) {
        /* Update the energy value for the cell */
        update( layer, layer_size, k, position, ...
        energy, local_size, rank);
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

Note that `MPI_BARRIER()` is needed to synchronize the computation for the whole layer before goes to the next iteration. Just with this simple modification the code is being accelerated succesfully. However, the next relaxation phase is also improved by MPI using those subdomains.

The other loops are not so heavy, so the speedup is not noteceable. But to find the maximum energy values in the full layer, we use `MPI_REDUCE()` using `MPI_MAX()` operator to find global maximum:

```
MPI_Barrier(MPI_COMM_WORLD);
MPI_Reduce(&local_maximum[i], &global_maximum[i], ...
    1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
```

Results

The scalability is not promising, see figure 5, but there is no drop in performance. The speedup slopes are continuously increasing up to 128 threads. Then, the smaller case, 30k, reaches a plateau meaning that it does not make sense to go for more threads. On the other hand, with the bigger cases, like 1M and 100M, there is a big benefit to use the full node. Special mention to the 1M case, that can achieve over 100 times speedup and maybe can continue beyond if we exceed another node. For the case 35, there is no need to parallelize anything.

threads	total time (sec) test 35	test 30k	test 100M
1	0,000002	3,025233	0,707545
4	0,000618	1,520648	0,394254
8	0,000815	0,811806	0,224229
16	0,000074	0,474373	0,167316
32	0,000105	0,26838	0,156609
64	0,00019	0,159509	0,081761
128	0,000454	0,121374	0,043875
256	0,000901	0,117231	0,021501

Table 2: Computational cost in wall-clock time in 3 different cases.

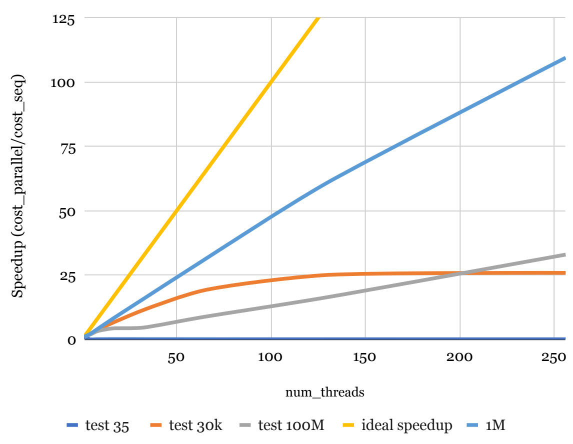


Figure 5: Speedup performance with MPI parallelization

8 OpenMP

Coding

In this case OpenMP directives were much easier and quick to implement into the code. The inner loops are parallelized using `#pragma omp parallel for` directive and depending if the expected memory usage is very well distributed, we use `schedule(static)`. Again, the major gain in performance is due `update()` function:

```
#pragma omp parallel for schedule(static)
for( k=0; k<layer_size; k++ ) {
    /* Update the energy value for the cell */
    update( layer, layer_size, k, position, energy );
}
```

The outer loop is not parallelized. The other inner loop are also parallelized, but we do not see any noticeable performance gain.

Results

The results show how scale very well in the beginning, see figure 6. Around 64 threads reaches the peak and then decrease rapidly. Those problems are maybe due because of too many communications, that start to take much more time than the actual computation. Maybe it means that OpenMP is worse managing the communication and we can optimize the schedules. This results shows how the use of OpenMP for more than 64 threads will waste more computational resources, even worse performance. The 1M case is an exception, scaling better and having its peak performance at 128 threads. Anyway the speedup drop still there.

As in the other cases, for the 35 cell test is no need of parallelization running in Dardel cluster. However, in weaker machines may suppose a difference.

9 Conclusions

The parallelization has been successfull in all tests cases. Nevertheless, the maximum speedup for the best case we ran, are quite different: for CUDA around 600 times faster than sequential in single CPU; MPI 100 times and OpenMP 45 times approximately. Furthermore, if we study how scale this 3 different implementations, we se how OpenMP is struggling to go for more thread experiencing a drop in performance. In the MPI case we have a continuous slope that is smaller gradient but achieve better performance using the full node. The use of GPU is a game-changer, there is no comparison with the maximum speedup we can get thanks to CUDA acceleration. There is a plateau when reaches 64 threads per block, so it does not make sense using the full hardware either.

	total time (sec)		
threads	test 35	test 30k	test 100M
1	0,000002	3,025233	0,707545
4	0,005513	1,898468	0,359289
8	0,005663	0,975602	0,182528
16	0,006383	0,529785	0,099294
32	0,008335	0,32335	0,058662
64	0,011842	0,200106	0,041888
128	0,019203	0,18114	0,042758
256	0,074881	13,085344	0,078745

Table 3: Computational cost in wall-clock time in 3 different cases.

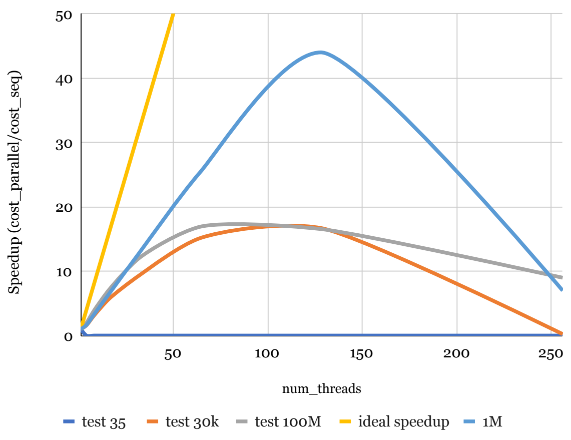


Figure 6: Speedup performance with OpenMP parallelization