PDC Summer School Project

Ernst Nordström Cederholm
Lukas Bähner
Marcial Sanchis Agudo

September 6, 2023

1 Introduction

This report describes our work on the project of the PDC Summer School 2023. It is part of the course ”FDD3260 High-performance Computing for Computational Scientists 5.0 credits” offered at KTH Royal Institute of Technology in Stockholm. The source code is accompanying this report and is referenced later. The first chapter describes how to run the code on the supercomputer Dardel at KTH and/or on a local machine. Afterwards, we describe our strategies when implementing parallelization. Next, we compare the run time of our different implementations. Then we discuss future optimization strategies and finally draw conclusions.

2 How to run the code

Both code files, energy_storms_mpi.c and energy_storms_omp.c can be compiled with the make file as on the git repository that was shared with us. We usually clean up the binary files with ”make clean” and then compile the code with ”make debug”, ”make all” or ”make [file name]” depending on if we want to print cells, compile every file or just one particular. We ran and tested the MPI and OpenMP codes on Dardel. To run the code on our local machine, where we will conduct the main analysis between languages, one compiles the code in the same manner as in Dardel.

```
    make clean
    make all
    ./energy_storms_omp n test_files/test_07_a1M_p5k_w1
```

Where $n = 10^3, 10^6$ and 10^9 for the later experiments.

2.1 OpenMP

For running the OpenMP code on Dardel, we allocate a certain number of CPUs per task, e.g. ”salloc –nodes=1 –ntasks-per-node=1 –cpus-per-task=32 -t 0:30:00 -A edu23.summer -p shared”. This gives us 32 threads on which OpenMP distributes the work in the parallel regions in the code. Afterwards, we can run the code with the srun command, e.g. ”srun -n 32 energy_storms_omp 32 test_files/test_01_a35_p5_w3”, which runs the code with all 32 threads.

2.2 MPI

For running the MPI code on Dardel, we allocate a certain number of CPUs, ”salloc -n 32 -t 0:30:00 -A edu23.summer -p shared”. 32 is the number of MPI ranks we can use. Then we can run the code with a certain amount of ranks, a certain layer size and test file, e.g. ”srun -n 5 energy_storms_mpi 35 test_files/test_01_a35_p5_w3”. Note that our implementation parallelizes the number of particles.

Therefore, the number of MPI ranks in the run command should be less or equal than the number of particles specified in the test file. If that is not the case, the program will throw an error.

2.3 CUDA

We can run any CUDA program on NVIDIA devices only. For this particular investigation we made usage of a local NVIDIA card [GeForce GTX 1650 Mobile]. It is important to remark too that the CUDA version is 12.0.

```
    nvcc CUDACode2.cu -o CUDACode2
    ./CUDACode2.cu " " test_files/test_07_a1M_p5k_w1
```

We have also made an jupyter note book version of the cuda code energy_storms_cuda_colab.ipynb that you find in our repository. In order to run it on Colab's T4 GPU you have to upload the storm files to Google Colab as well.

3 github-repository

Our github repository

3.1 OpenMP

As hinted in the project description, it is useful to start parallelizing with OpenMP bottom-up. The inner most loop is over cells in the layer, which we parallelize. Since we use OpenMP on the CPU, we start the parallel region with the command "#pragma omp parallel for" just before the for-loop starts. According to good practice, we declare data scoping clauses for each variable that is used in the loop and set "default(none)". Loop variables are private, so we can declare "private(k)". In our case, all other variables are shared, most prominently the variable "layer", which we declare as "shared(layer, ...)". As a consequence of sharing a variable, all threads access the same copy of the variable and we need to consider race conditions. They happen when the outcome of the program depends on the scheduling of the threads. In our case, the loop over cells calls the update function which adds some attenuated energy to each cell. This means that the writing is independent on other cells and cannot interfere with the writing of other cells from other threads. Therefore, we do not run into race conditions in our implementation.

3.2 MPI

Our idea for MPI parallelization was to distribute the array of particles between ranks on CPUs. The other options would have been storms or cells in the layer. There are usually more processors available than storms specified in the input, which makes storms not suitable for parallelization. If each rank would process a number of cells in the layer, things would become complicated because cells affect their surrounding cells due to e.g. energy relaxation, which would require ghost cells between ranks. This is because MPI uses distributed memory, that means that the ranks have their own, distinct memory. It turns out that particles are very suitable to split between ranks, because they are independent and we can sum up the effect from all particles with a single MPI reduce operation. Things we had to consider when implementing were the following: each rank gets the correct particles of the ordered list; the reduce operation is correctly implemented; we need a new variable that stores the sum; the energy relaxation can be done on each rank before the reduce; the maximum value must be found after the reduce. Note that we declare the integers n, remainder, lb and ub in line 154 with all the other integer declarations.

3.3 CUDA

We choose to run the code on a NVIDIA GPU using CUDA (mainly because Dardel was under maintenance and we only had NVIDIA graphic cards to work with). In our solution we made the GPU handle the update function used in the most inner loop. So instead of iterating through the particle in a wave we make the GPU handle the particles in parallel using different amounts of TPB. The implementation work well since the update function use simple enough operations that add energy to one position in the layer array.

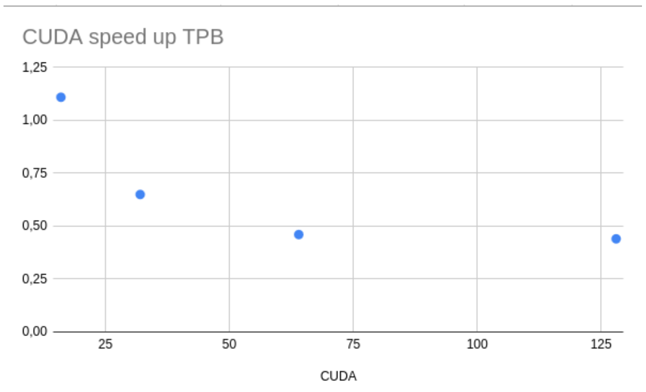

Figure 1: CUDA speed up TPB

4 Speed Comparsion

In order to conduct this analysis, as mentioned earlier, we will run all 3 codes uploaded on the github folder on a GeForce GTX 1650 Mobile. We compared not only the performance of each code for different storm sizes but also modify the hyper-parameters to gain a better understanding of the language itself.

An one can see below 1, the improvement on performance appears as the size of storms increases. For smaller size input the MPI is drastically better than the rest of languages, however as the size increases the CUDA code is considerably better, as one is able to exploit the full potential of the NVIDIA GPU card. Finally, it is important to remark that the performance on Table 1 regarding MPI, was computed with 5 cores, as we see increasing the number of cores after the latter does not affect the timing.

	10^3	10^6	10^8
CUDA	0.03	0.009	0.44
OMP	0.01	0.04	3.88
MPI	0.22	0.002	0.81

Table 1: Speed analysis

5 Futher optimization strategies

It would be interesting to try and use both CUDA and OpenMP on different parts of the code. Another strategy might be to use both OpenMP and MPI as was done in one of the quiz questions. For example the number of particles could be divided on several processors using MPI and then OpenMP could be used to update the layer on the different cores.

6 Conclusions

In conclusion, for lower input size the full performance of the GPU is not exploited. One can clearly see above, Table 1, how for smaller size, 10^3, it is not clear which language could scale better as the input size increases too. Due to coding complexity, for smaller inputs one would prioritise simplicity, as good performance is achieved within the three possible codes. However, as the size input increases it becomes more obvious the superior performance of the GPU (Cuda) against the CPU (OMP, MPI). Even more, if one simplifies the computational algorithm, the GPU will be outperforming the CPU.

Ideally, one wants to implement both MPI and GPU languages (ACC,OMP,CUDA or HIP). Doing so, we could parallelize multiple tasks into multiple cores while copying the required kernels into the GPU with the objective of optimizing the computation fully.