Zhikai Yang | Anastasiia Andriievska | Siyuan Chen
zhikai@kth.se | aa11@kth.se | siyuanch@kth.se

Project "Simplified simulation of high-energy particle storms"

PDC Summer School FDD3260

PlasmaS: Zhikai Yang, Anastasiia Andriievska, Siyuan Chen
KTH Royal Institute of Technology

December 15, 2023

Contents

Introduction

In this project, the task is to work in groups of 1-3 students and parallelize the "energy storms" code by using OpenMP, MPI, and CUDA/HIP. In addition, the group must write a short report (maximum five pages in the standard latex template) where they describe the techniques they used and reason around possible future opportunities for parallelization. The energy storms project was developed by Arturo Gonzalez-Escribano and Eduardo Rodriguez-Gutiez in Group Trasgo, Universidad de Valladolid (Spain).

Materials and Methods

Code available: https://github.com/fomalhautn/plasmas

1 Version parallelized with MPI

1.1 How to compile and run

First, we need to request some computational resources in Dardel

```
salloc -A edu23.summer -N 1 -n 8 -t 1:0:0 -p shared
```

Second, for compile under the debugging (take `energy_storms.c` file as an example)

```
make debug energy_storms.c
```

if just want to compile without the debugging, then just directly make it

```
make energy_storms.c
```

Then, if we want to check the output. For seq version, we don't need the multi-processors, so we can directly (where 10 means the number of layers)

```
./energy_storms_seq 10 test_files/test_01_a35_p5_w3
```

While for mpi version, we need to specify the number of processers. (where 8 means the number of processers, and 10 is the number of input layers)

```
srun -n 4 energy_storms_mpi 10 test_files/test_01_a35_p5_w3
```

1.2 How to parallelize with MPI

There are several strategies to parallelize this problem with MPI. We can assign different processers to handle different input files if the input files were so much lot. Also, we can assign different processers to handle different particles' computation for one input file. Of course, we can combine them together as well. Here, we adapt the second strategy and the rank 0 will be responsible to summing up all the other processers' layer results by using MPI_Reduce.

We create a new variable named particle_per_proc to assume the amount of particles handled by every processer (rank with 1,2,3...).

```
particle\_per\_proc = storms[i].size/(size-1);
```

But it is noted that probably for the number of particles handled by the last processor could be different from others. To differentiate them, we have

```
if(rank==0) {
    MPI_Reduce(MPI_IN_PLACE,layer,layer_size,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);
} else if(rank == size -1){
    for(int j = (rank-1)*particle_per_proc; j<storms[i].size;j++) {
        // ...
        MPI_Reduce(layer,layer,layer_size,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);
        for(k=0; k<layer_size; k++) layer[k] = 0.0f;
    }
} else {
    for(int j =(rank-1)*particle_per_proc;j<rank*particle_per_proc;j++) {
        // ...
        MPI_Reduce(layer,layer,layer_size,MPI_FLOAT,MPI_SUM,0,MPI_COMM_WORLD);
        for(k=0; k<layer_size; k++) layer[k] = 0.0f;
    }
}
```

2 Version parallelized with HIP

2.1 How to compile and run

First, we need to request some computational resources in Dardel

```
salloc -A edu23.summer -p gpu --nodes=2 -t 00:30:00
```

Second, we need to have the ROCm platform installed on our system if we want to run your code on AMD GPUs, or the CUDA Toolkit if we want to run it on NVIDIA GPUs.

```
module load rocm/5.3.3
```

Make sure that your energy_storms_hip.hip file is in a directory that is accessible from the system where you loaded the ROCm module. Remove the old binaries and compile the new binary.

```
make clean && make all
```

Use the hipcc compiler, which is part of the ROCm platform, to compile the HIP code. For example, we can use the following command to compile the energy_storms_hip.hip file:

```
hipcc energy_storms_hip.hip -o energy_storms_hip
```

Then, if we want to check the output. After compiling the code, we can run the resulting executable file to start our simulation. For example, we can use the following command to run our hip executable:

```
srun -n 2 --nodes 2 ./energy_storms_hip test_files/test_01_a35_p5_w3
test_files/test_01_a35_p7_w2
```

For HIP version, we need to specify the number of nodes allocated. Output:

```
Time: 0.000001
Result: 0 0.000000
Time: 0.000001
Result: 0 0.000000
```

2.2 How to parallelize with HIP

There are several strategies to parallelize this problem with HIP. This is how we integrated the HIP code into the existing program:

1. Include the HIP header file at the top of your program:

```
#include<hip/hip_runtime.h>
```

2. Define the HIP error checking macros after the includes:

```
#define CHECK_HIP_CALL( a )    { \
    hipError_t ok = a; \
    if ( ok != hipSuccess ) \
        fprintf(stderr, "--- Error HIP call in line %d: %s\n", __LINE__,
        hipGetErrorString( ok ) ); \
    }
#define CHECK_HIP_LAST()    { \
    hipError_t ok = hipGetLastError(); \
    if ( ok != hipSuccess ) \
        fprintf(stderr, "--- Error HIP last in line %d: %s\n", __LINE__,
        hipGetErrorString( ok ) ); \
    }
```

3. Allocate memory for the layer array on the GPU device after you have initialized layer and layer_copy in your main function:

```
float *d_layer;
CHECK_HIP_CALL(hipMalloc((void **)&d_layer, sizeof(float) * layer_size));
```

4. Copy the data from the host to the device right after the previous step:

```
CHECK_HIP_CALL(hipMemcpy(d_layer, layer, sizeof(float) * layer_size,
hipMemcpyHostToDevice));
```

5. Replace the update function call inside the nested loop with a call to launch the update_kernel. This should be done inside the loop over storms and particles:

```
/* Launch kernel to update all cells in parallel */
int blockSize = 256;
int gridSize = (layer_size + blockSize - 1) / blockSize;
hipLaunchKernelGGL(update_kernel, dim3(gridSize), dim3(blockSize),
0, 0, d_layer, layer_size, position, energy);
```

6. Copy the updated layer data back from the device to the host after all updates have been performed and before you start finding maximum values:

```
CHECK_HIP_CALL(hipMemcpy(layer, d_layer, sizeof(float) * layer_size,
hipMemcpyDeviceToHost));
```

7. Free the memory allocated on the device at the end of your program where you free other resources:

```
CHECK_HIP_CALL(hipFree(d_layer));
```

8. Define your update_kernel function somewhere in your code outside of any other function. It could be at the top of your file or at the bottom, as long as it's not inside another function:

```
__global__ void update_kernel(float *layer, int layer_size, int position,
float energy) {
    int k = hipBlockIdx_x * hipBlockDim_x + hipThreadIdx_x;
    if (k < layer_size) {
        /* Compute the absolute value of the distance between the impact
        position and the k-th position of the layer */
        int distance = position - k;
        if (distance < 0) distance = -distance;
        /* Impact cell has a distance value of 1 */
        distance = distance + 1;
        /* Square root of the distance */
        float atenuacion = sqrtf((float)distance);
        /* Compute attenuated energy */
        float energy_k = energy / layer_size / atenuacion;
        /* Do not add if its absolute value is lower than the threshold */
        if (energy_k >= THRESHOLD / layer_size ||
        energy_k <= -THRESHOLD / layer_size)
            layer[k] = layer[k] + energy_k;
    }
}
```

3 Version parallelized with OPENMP

3.1 How to compile and run

We use linxu to compile the openmp program.

```
sudo apt install libomp-dev
```

Use the gcc compiler, run below command line

```
gcc -fopenmp energy_storms_omp.c -o energy_storms_omp
```

Then, if we want to check the output. After compiling the code, we can run the resulting executable file to start our simulation. For example, we can use the following command to run our energy_storms_hip executable:

```
./energy_storms_omp test_files/test_01_a35_p5_w3
```

3.2 How to parallelize with OPENMP

Inspired by feedback, we optimized the code and corrected the data racing problem. Below is the procedure.

1. Parallelize the initialization of the layer array: This is a straightforward operation where each initialization can be done independently.

```
#pragma omp parallel for
for( k=0; k<layer_size; k++ ) layer[k] = 0.0f;
for( k=0; k<layer_size; k++ ) layer_copy[k] = 0.0f;
```

2. Parallelize the particle energy addition to layer cells: This can be done because each particle's energy impact on the layer is independent of the others. By adding the reduction(+:layer[:layer size]) clause to the pragma omp parallel for directive, each thread will have its private copy of the layer array, and at the end of the parallel region, the reduction operation + will be performed to aggregate the results from each thread into the shared layer array.

```
#pragma omp parallel for private(j, k) reduction(+:layer[:layer_size])
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 each cell in the layer */
    for( k=0; k<layer_size; k++ ) {
        /* Update the energy value for the cell */
        update( layer, layer_size, k, position, energy );
    }
}
```

3. Parallelize the energy relaxation between storms: This can be done because the new layer value for each cell is calculated independently of the others.

```
#pragma omp parallel for
for( k=1; k<layer_size-1; k++ )
    layer[k] = ( layer_copy[k-1] + layer_copy[k] + layer_copy[k+1] ) / 3;
```

4. Parallelize the finding of the maximum value in the layer: This is a reduction operation, where the maximum value and its position have to be found. OpenMP has built-in support for such operations.

```
#pragma omp parallel for reduction(max:maximum[i]) private(k)
for( k=1; k<layer_size-1; k++ ) {
    /* Check it only if it is a local maximum */
    if ( layer[k] > layer[k-1] && layer[k] > layer[k+1] ) {
        if ( layer[k] > maximum[i] ) {
            #pragma omp critical
            {
                maximum[i] = layer[k];
                positions[i] = k;
            }
        }
    }
}
```

Results

In the results part, we compare the three files at different numbers of layers running time in OPENMP and MPI. We find the larger number of layers means more time-consuming. The experiment on the MPI was conducted on the PDC server and OPENMP is conducted on the PC (because of the expired of PDC account). So the two method time consumption hard to directly comparison since it using different level of computation recsources. OpenMP can be more efficient for shared-memory systems with a moderate number of processors because it allows for easier thread coordination and data sharing. It is often used for applications with fine-grained parallelism, such as loop-level parallelism.

MPI, on the other hand, is more suitable for distributed-memory systems with a large number of processors. It excels in scaling to larger clusters and is commonly used for applications with coarse-grained parallelism, such as simulations or data-intensive computations that require frequent communication between processes.

Table 1: Performance of the time on different methods at different number of layers $Testfiletest\_01\_a35\_p5\_w3$

Methods	100	1e3	1e4	1e5	1e6
MPI	0.000443	0.000206	0.001394	0.005474	0.053069
OPENMP	0.000034	0.000032	0.000200	0.001937	0.013937

Table 2: Performance of the time on different methods at different number of layers $Test\_file\_test\_02\_a30k\_p20k\_w1$

Methods	100	1e3	1e4	1e5	1e6
MPI	0.005645	0.089812	0.558420	5.794647	101.799823
OPENMP	0.027084	0.245781	2.042242	21.414772	216.2597

Table 3: Performance of the time on different methods at different number of layers $Test\_file\_test\_07\_a1M\_p5k\_w1$

Methods	100	1e3	1e4	1e5	1e6
MPI	0.002333	0.023249	0.136987	1.351210	15.092521
OPENMP	0.011542	0.070375	0.5321	5.10	52.18796

Discussions (and conclusions)

OpenMP, HIP, and MPI are different parallel programming models that can be used to speed up the execution of applications on multiple processors or cores. They have different advantages and disadvantages depending on the type of problem, the hardware architecture, and the programming style. Brief comparison of OpenMP, HIP, and MPI with regard to run time and parallelization strategy:

- OpenMP is a directive-based model that allows you to add parallelism to your existing sequential code by using compiler directives, such as 'pragma omp'. OpenMP is mainly designed for shared-memory systems, where all processors can access the same memory space. OpenMP is easy to use and can achieve good performance for loop-based computations. However, OpenMP has some limitations, such as limited scalability, lack of explicit communication control, and difficulty in handling irregular or dynamic workloads. In the context of the "energy storms" project, OpenMP could be used to parallelize the computations within a single node. However, a race condition was identified in the OpenMP code, which needs to be fixed to ensure correct results.
- HIP is a C++ extension that allows you to write portable code for heterogeneous compute platforms, such as AMD and NVIDIA GPUs[1]. HIP is similar to CUDA, but it can run on different devices without changing the source code. HIP is suitable for data-parallel problems, where the same operation is applied to many data elements independently. HIP can achieve high performance and efficiency by exploiting the massive parallelism and memory hierarchy of GPUs. However, HIP requires more programming effort and knowledge

than OpenMP, and it may not work well for problems that have complex control flow or inter-thread communication. In the "energy storms" project, HIP could be used to offload the computations to the GPU, which can significantly speed up the execution time. However, the performance of HIP depends on the size of the data and the complexity of the computations. Therefore, it's important to optimize the data transfer between the CPU and the GPU, and to choose the right block size and grid size for the GPU kernels.

- MPI is a message-passing model that allows you to write parallel programs that communicate by sending and receiving messages between processes. MPI can run on both shared-memory and distributed-memory systems, and it can support a wide range of parallel patterns, such as point-to-point, collective, or one-sided communication. MPI can scale well to large numbers of processors or nodes, and it can give you full control over the communication and synchronization. However, MPI can be complex and verbose to use, and it may incur high overhead or latency for communication-intensive or fine-grained problems. In the "energy storms" project, MPI could be used to parallelize the computations across multiple nodes. However, the performance of MPI depends on the network latency and the communication overhead. Therefore, it's important to balance the load between the nodes and to minimize the communication cost.

Possible future optimization strategies:

- One can also use a hybrid approach that combines different models, such as MPI+OpenMP or MPI+HIP, to leverage their strengths and overcome their weaknesses. This could help to overcome the scalability limitations of OpenMP and to make better use of the hardware resources.
- Analyze the performance bottlenecks and hotspots of your parallel code using profiling tools or benchmarks. Use tools such as gprof, nvprof, or TAU to measure the execution time, memory usage, communication overhead, or GPU utilization of your code. This could provide valuable insights into the performance characteristics of the code and help to identify areas for optimization.
- Optimize the code by applying techniques such as load balancing, vectorization, loop unrolling, cache blocking, memory coalescing, or kernel fusion. One can also use libraries or frameworks that provide optimized implementations of common parallel algorithms or operations, such as Thrus, OpenACC, or RAJA. These techniques and tools could help to improve the performance of the parallel code and to reduce the execution time.

References:

1. Comparison between pure MPI and hybrid MPI-OpenMP - Springer.
https://link.springer.com/article/10.1007/s40571-018-0213-8
2. Toward Heterogeneous MPI+MPI Programming: Comparison of OpenMP and MPI.
https://link.springer.com/chapter/10.1007/978-3-030-48340-1_21

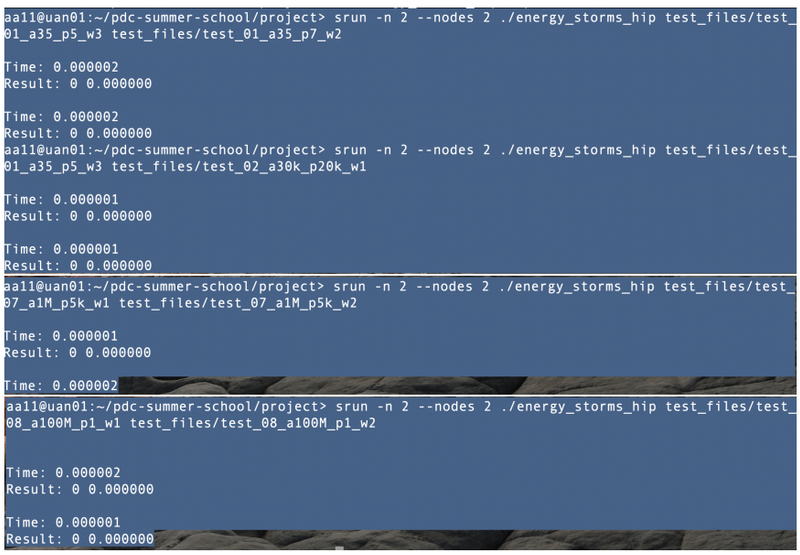
2 Supplentary materials

Screenshot MPI

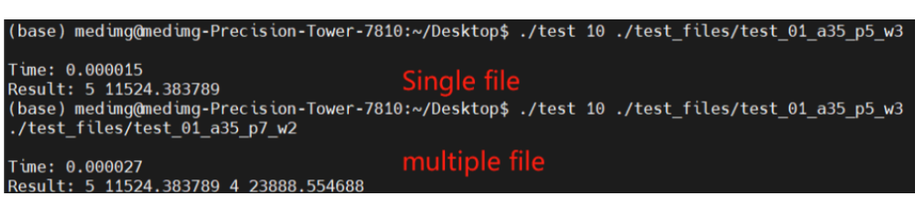
For multiple test files:

Screenshot MPI

We need to set the number of threads that OpenMP can use with omp_set_num_threads(num_threads). By default, OpenMP uses as many threads as there are cores on CPU.



Screenshot HIP



Screenshot OPENMP