INF560 - Report

Lukas Köstler, Ezequiel Morcillo Rosso

March 13, 2019

Abstract

This work aims to provide a comprehensive report on the different approaches to parallelize a Monte Carlo particle simulation using MPI, OpenMP & CUDA. A resource & domain decomposition algorithm is implemented that aims to evenly distribute work among all compute nodes according to their available resources. We demonstrate the effectiveness of the implemented parallelization through a heterogeneous example.

Contents

1 Introduction

The goal of this project is the hybrid parallelization of a Monte Carlo particle simulation program using MPI [2], OpenMP [1] and CUDA [3]. We therefore target execution on a cluster with nodes that might be equipped with NVIDIA GPUs. We develop a domain decomposition approach that estimates the compute power during runtime and thus adapts to different execution environments. In a comprehensive demo we show the successful adaptation to a very heterogeneous environment without prior knowledge.

The sequential reference implementation and accompanying documentation was distributed as part of the INF 560 course at École Polytechnique and will therefore not be described within this report.

2 MPI

MPI was used in order to parallelize the simulation by distributing the domain to several MPI processes. This method of parallelizing, called domain decomposition, lends itself to a distributed memory method (MPI in this case) since the different sections of the domain will not share particles being simulated. Each MPI process is assigned a different layer, thus being as many layers as there are processes in the distributed memory model. Each process will then be able to simulate the particles inside its layer until the particles either get small enough or exit the layer.

The communication approach taken in this work is purely a synchronous one, given that an asynchronous one escapes the scope of the objective. When a process simulates all particles in its layer, it has a collection of particles that has either exited through a side or gotten disabled in the layer. A process needs to communicate to adjacent layers the particles now entering their respective domain. In order to do so, two strategies were implemented, both of them using point to point communication in between adjacent layers, the difference laying in the use of different MPI functions.

The first and more simple strategy implements communication by the functions MPI receive and MPI send. These functions limit the process in its ability to send and receive at the same time, and a process can't send nor receive to or from both adjacent ones at the same time. The buffer provided to receive the particles is big enough to hold the worst case scenario, and the amount that a layer receives is obtained through the MPI status. Figure 1 shows an illustration for this communication. The arrows represent the particles exiting one layer and entering the adjacent one. It should be noted

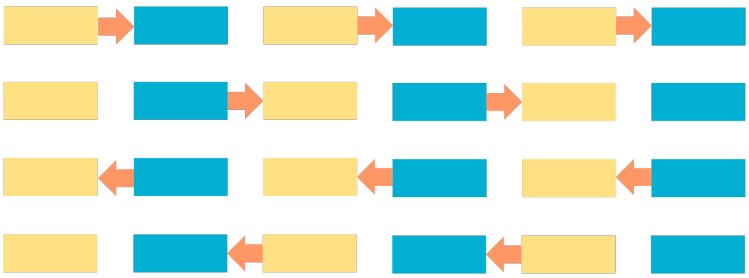


Figure 1: Communication using MPI receive and send. Each row represents a different stage, starting top most. In the first row, even numbered layers (yellow) send to their right neighbour, which are odd numbered layers (blue), who in turn receive from the left. In the second row, odds send to the right and evens receive from the left. The last two rows are similar to the previous ones, but all sends are to the left and all receives are from the right.

that the ordering of the rows is not important, but doing these four steps guarantees all layers have sent and received to and from their neighbours.

The second strategy reduces the number of point to point communications through the use of the MPI send-receive function. Once again the buffer provided to receive was large enough to hold all particles in the worst case scenario. Figure 2 shows an illustration for the second strategy. It has only two stages and is similar to figure 1 if the first row was merged with the third one and the second one with the fourth.

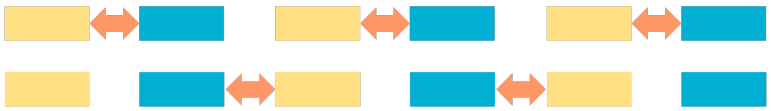


Figure 2: Communication process using MPI send-receive. Each row represents a different stage, starting top most. In the first row, even numbered layers (yellow) send and receive to and from their right neighbour, while odd numbered layers (blue) send and receive to and from the left. The second row is similar to the first one, but now evens send and receive to and from the left and odds send and receive to and from the right.

In the end, the second strategy is chosen for being superior to the first one in the time it takes for the layers to communicate with all their neighbours.

3 OpenMP

OpenMP was used to parallelize the simulation step within each MPI process. Because the particles do not interact, the simulation of each particle is

independent of the others. In contrast, the absorbed weights of the domain couple all simulation steps together. We therefore zero-initialize a local array for the absorbed weights within each OpenMP thread. After the simulation the local arrays are added by using an OpenMP critical section.

The results for `schedule(static)` vs. `schedule(dynamic)` are shown in fig. 3. Based on the result we choose a static schedule. We achieve nearly perfect speedup for two and three threads. Even for repeated runs the speedup with four threads is smaller than expected. Overall, i.e. with eight threads, we achieve a speedup of roughly five times on a machine with four cores.

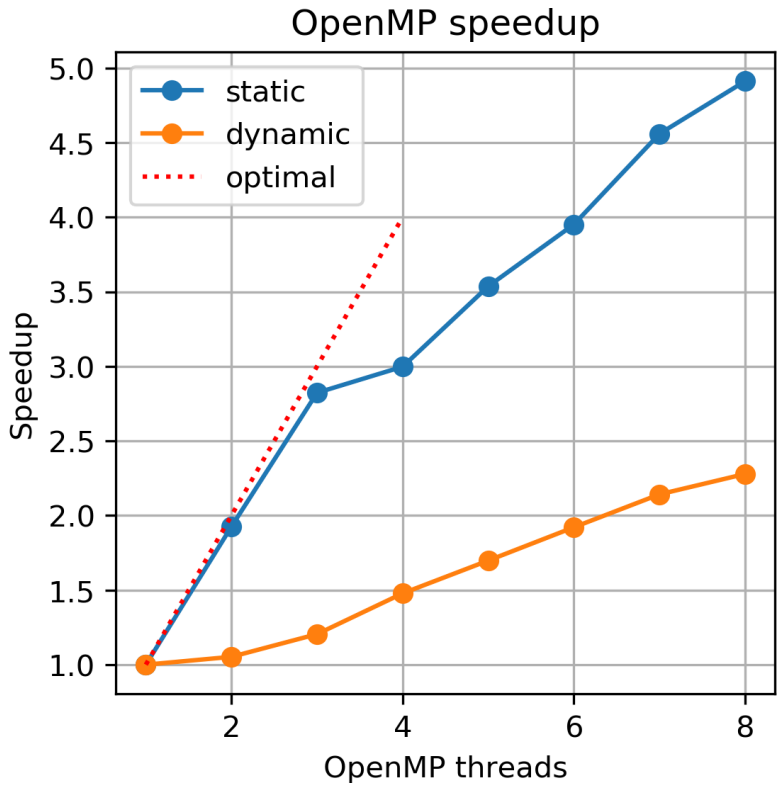


Figure 3: OpenMP speedup for `schedule(static)` vs. `schedule(dynamic)`. For one to four threads the optimal speedup would be one to four times because the machine has four cores. Afterwards the speedup depends on the efficiency of hyper-threading. Static scheduling is advantageous as each iteration of the parallelized loop has similar runtime and thus no explicit load-balancing is needed.

4 CUDA

Similar to OpenMP, CUDA is also used to parallelize the simulation step within each MPI process. We assign one particle per CUDA thread and map all domain properties to shared memory which is on-chip and thus faster than global GPU memory.

The computation kernel performs a fixed number of steps on all particles. Before actual simulation the shared memory is initialized per thread-block. During simulation the thread-block-local copy of the absorbed weights array is updated using `atomicAdd`. After simulation each thread-block adds its copy of the aforementioned array to the global array using `atomicAdd`.

Before the kernel is invoked for the first time the cell properties are copied into GPU memory. The kernel is then invoked in a loop and after each invocation the particles are copied back to CPU main memory. The CPU then sorts the particles into active and inactive and then invokes the kernel again with only the active particles. When all particles have been deactivated all remaining data is copied back from the GPU.

Through `nvprof` we can see that above 97% of the GPUs computation time is spent inside the kernel and that the data shuffling from CPU to GPU and back is not a performance bottleneck.

We achieve pipelining within one streaming multiprocessor through choosing thread-blocks of size 256. This still leaves the number of thread-blocks to be larger than 30 for most invocations of the kernel and thus guarantees good load balancing.

We test the performance of the pure CUDA implementation on the benchmark problem and compare it to the sequential performance and to the performance achieved through using OpenMP. Sequentially the execution takes 25.1 seconds, with eight OpenMP threads 4.84 seconds (compare fig. 3) and using the GPU 0.45 seconds. This results in a speedup of 55 over the sequential version and 10 over the OpenMP version.

5 Hybrid Models

Once MPI, OpenMP and CUDA are implemented, it is possible to combine these in order to further parallelize the simulation, since not all of these methods use the same type of parallelization.

Since MPI does a domain parallelization, whereas OpenMP and CUDA do not, it is possible to combine MPI with either one of the other two. As previously discussed, OpenMP can be used to faster simulate the particles

in an MPI process, MPI still being able to parallelize the program through domain decomposition.

Since CUDA is implemented much in the same way OpenMP is, it is also a possible hybrid to do CUDA and MPI, the former parallelizing the simulation inside a layer and the latter continuing to parallelize by domain decomposition.

It might be possible for a computer to be running more than one MPI process, and since it is assumed each computer to have at most one GPU, it might be possible that a single computer has two layers, one's simulation being parallelized with CUDA and the other ones through OpenMP; so in this case one is dealing with a hybrid of all three methods.

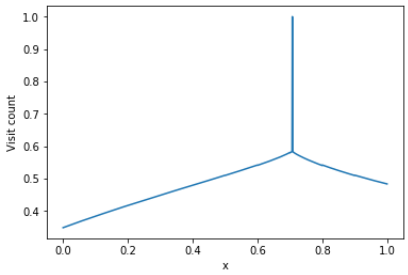
6 Resources & Domain Decomposition

In order to make the most out of the parallelized simulation, two factors must be taken into account. The first one is the amount of resources an MPI processes has, and which one of those are shared with other MPI processes that might be in the same node. The second one is the domain decomposition, and how to evenly distribute the size of the layers among the processes in order for them to take the same amount of time simulating.

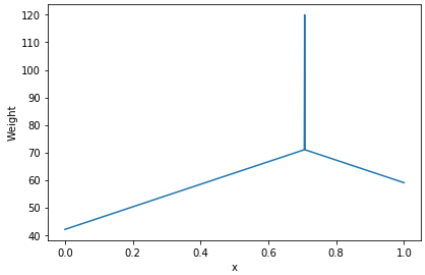
To tackle the first factor, a scheduler when first initializing the program determines which of the ranks share the same node and whether said node has a GPU available for use. If only one process exists in the node, then it gets all resources. However, if more than one is in the node, then the one with the lowest local rank gets use of the GPU and only one thread, since it does not parallelize through OpenMP, and each of the processes remaining in the same node get the remaining threads as evenly distributed as possible. This configuration for each MPI process is saved in order to be able to make a better domain decomposition.

To tackle the second factor, a domain decomposition, meaning the amount of cells given to each process of the simulation, algorithm is implemented. In order to best explain it, a first basic approach to domain decomposition is explained, later on improving this method with the final solution.

The first basic approach is a uniform distribution of the domain among the layers, attributing the same number of work load to each part of the domain. Therefore, all layers are the same size, independently of computing power for the process administrating that layer, or the position of the layer in the domain. This would mean an even distribution of workload based on computing power for all processes under the assumption that all cells



(a) Normalized visit count for each cell. All particles are initialized at $x = 1/\sqrt{2}$ and move either left or right. A higher visit count might indicate a heavier work load for a layer containing said cell.



(b) Final cell weights, based on the visit count.

Figure 4: Comparison between visit count of a cell and the cell weight assigned to it

imply the same amount of work load and that all processes have similar computation power. However, neither of these assumptions holds true, since it could be that some processes are running on the same node, sharing resources and decreasing each of those processes' computing power, while other processes are the only ones in a node, thus being able to handle a bigger layer than those sharing a node in the same amount of time. This decreases run time performance because due to the synchronous nature of the communication in between the different layers, the process is as fast as its slowest process. By having each process complete its simulation in the same amount of time, the optimal run time is achieved. And thus the final algorithm is implemented.

This approach consists of assigning a numerical value to each process indicating computing power, and a weight to each cell in the domain indicating the work load that cell implies. With both of these indicators, an even distribution of work load can be obtained, according to each process' computing power and thus obtaining the same simulation time for each process. In order to get the computing power of the process, a small scale simulation is run and timed at the beginning of the program, with the hardware resources obtained in the scheduler, effectively benchmarking the process' computing power by inverting the time it took to complete said simulation. As for the cell weight, figure 4a shows the result of a simulation ran where the visit count of each cell was tracked, under the assumption that the work load a cell implies is related to the visit count is has throughout the simulation. Figure 4b shows the final value given to each cell in the simulation.

The shape of the cell weight distribution is similar for both figures, but the latter one parametrizes the function as two straight lines and a delta in the cell that creates the particles.

7 Demonstration

Node	Num. GPUs	OMP threads	MPI processes
0	1	8	2
1	0	8	2
2	1	8	1
3	0	8	1
4	0	2	1

Table 1: Demo node specification.

The setup for this demonstration was chosen to showcase a wide variety of hardware resources on different nodes. We skew the environment s.t. the domain decomposition algorithm assumes the hardware availability given in table 1. It will then react with the distribution of resources and domain decomposition given in table 2. Figure 5 illustrates how despite the various hardware resources of each MPI process, each rank spends a similar amount of time computing throughout the whole simulation. This, as mentioned before, is advantageous because idle time (part of `MPI_Sendrecv` in the figure) is dependent on the process that takes the most time to do its computation. However, it can be observed that the domain decomposition done in this demonstration successfully manages to distribute work load among the processes, as evidenced by the similar amount of computation time for each process.

8 Conclusion

To conclude, this work effectively parallelized the given particle simulation using MPI, OpenMP & CUDA. MPI is advantageous for doing domain decomposition, more so, when spreading work load evenly and fairly among all MPI processes.

Also, as shown in fig. 3 the OpenMP implementation scales well i.e. achieves a speedup of nearly 5 times on a four core (eight thread) CPU. The proposed CUDA implementation attains a speedup of 55 times over the sequential implementation.

MPI process	Num. GPUs	OMP threads	Num cells
0	1	1	420
1	0	7	96
2	0	4	58
3	0	4	55
4	1	8	208
5	0	8	117
6	0	2	46

Table 2: Demo resource allocation. Rank 0 & 1 are on the same node and thus rank 0 gets exclusive access to the GPU while rank 1 gets all but one OpenMP threads. Still, rank 0 computes a larger part of the domain, because the GPU is faster than seven OpenMP threads. Rank 2 & 3 are on the same node and share the eight OpenMP threads equally, thus the domain sizes are nearly identical. Rank 4 has access to a GPU and eight OpenMP threads, but because its domain includes the particle initialization point, it is responsible for less cells than rank 0 which has a similar compute power. Rank 5 & 6 both get all resources of their respective nodes and the domain sizes are appropriate.

It has also been shown how to mix more than one parallelization model to obtain better results and make full use of of the available resources. After having combined the models, there were still ways to improve the program. Taking into account available resources helps to decrease runtime by doing the most even and fair work distribution, assigning more work load to those processes with more available resources, and making the program adapt to a great array of situation, up to the most heterogeneous resources case (as seen in the demo). Through this method, it can be seen that GPUs are faster than a machine running with all OpenMP threads, at least for this simulation.

References

[1] OpenMP Architecture Review Board. *OpenMP*. 2018. URL: `https://www.openmp.org`.

[2] MPI Forum. *MPI Forum*. 2018. URL: `https://www.mpi-forum.org`.

[3] John Nickolls, Ian Buck, and Michael Garland. "Scalable parallel programming". In: *2008 IEEE Hot Chips 20 Symposium (HCS)*. IEEE. 2008, pp. 40–53.

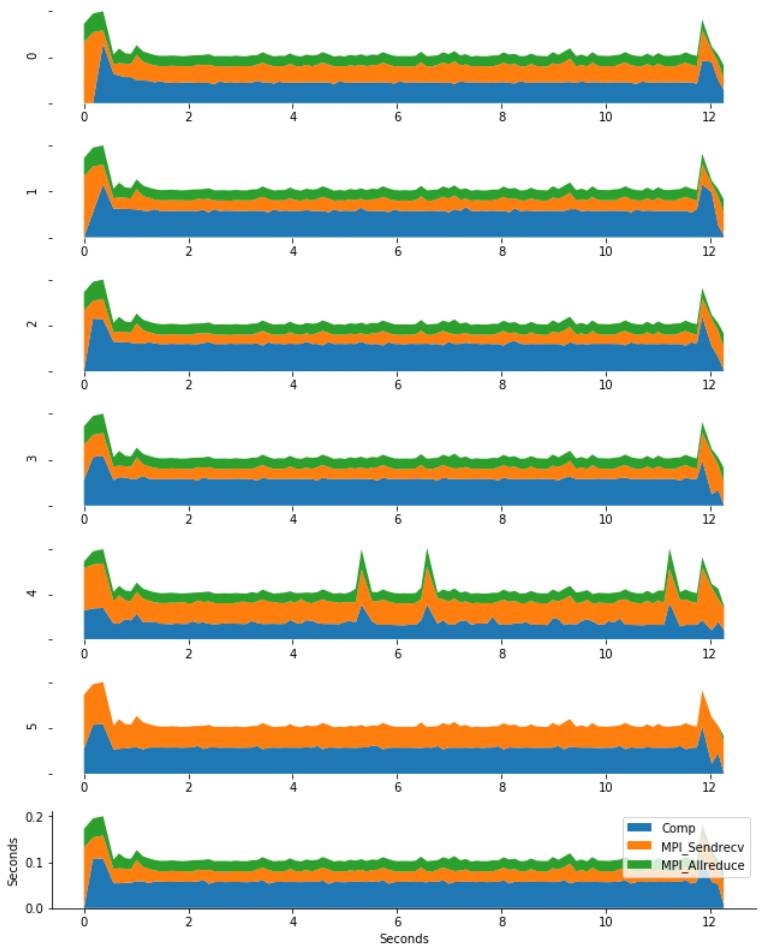


Figure 5: Demo simulation run. Each stacked plot represents a different MPI process, where the X axis represents simulation time and the Y axis represents time spent in each task, as shown in the labels.