INF518 - Projet de 3A MAP-INFO Calcul haute performance
Patrick Carribault, Ludomir Oteski

Lukas Köstler
December 18, 2018

# Mid-Project Report

## Contents

## 1 Introduction

The objective of this project is to parallelize a particle simulation program using MPI[2] for inter processor communication and OpenMP [1] for shared memory parallelization. The resulting parallel program should be executed on a cluster and the scaling characteristics should be investigated. Different MPI communication strategies should be compared.

The underlying simulation is explained in subsection 2.1, the serial reference implementation and enhancements to it are given in subsection 2.2. Afterwards, the objectives for the parallel implementation are outlined in subsection 2.3 and technology choices are justified in subsection 2.4. The domain decomposition approach to parallelization is explained in subsection 2.5 and the resulting communication strategies are discussed in subsection 2.6. First results are shown in section 3 and discussed in section 4.

## 2 Main Part

### 2.1 Simulation

In the simulation, $N$ particles randomly traverse and interact with a 1-D domain. The physical domain is discretized into $M$ cells with constant physical properties, as given in Figure 1.

Each particle is represented by three physical properties:

$$\begin{array}{lll} \text{Position} & x & \in \mathbb{R} \\ \text{Horizontal velocity} & \mu & \in [-1, 1] \\ \text{Energy} & wmc & \in \mathbb{R}^+ \end{array} \tag{1}$$

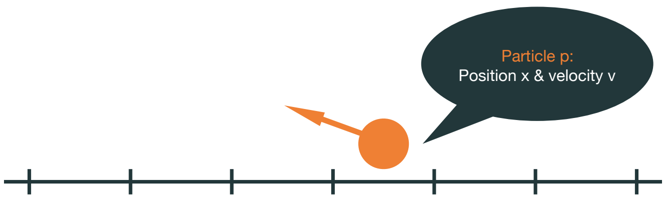


Figure 1: Schematic drawing of a particle, its velocity vector and the domain discretization into cells.

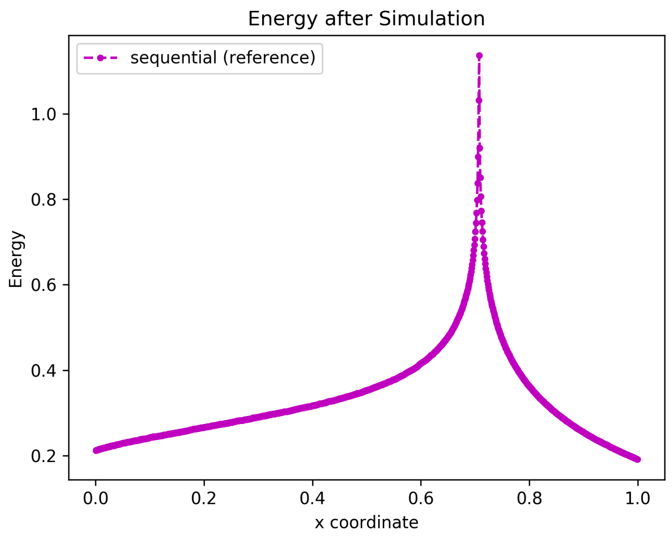


Figure 2: Energy per cell after a simulation with $M = 1000$ cells and $N = 1000000$ particles. The simulation was computed using the serial reference implementation.

and each cell is represented by two physical quantities:

$$\begin{array}{lll} \text{Effective absorption section} & sig_a & \in \mathbb{R}^+ \\ \text{Absorbed energy} & wa & \in \mathbb{R}_0^+ \end{array} \tag{2}$$

**Initialization** The initial values for the above properties are

$$\begin{array}{lll} x_i = & \frac{1}{\sqrt{2}} & \forall i \in \{1, \dots, N\} \\ \mu_i \sim & unif\,(-1, 1) & \forall i \in \{1, \dots, N\} \\ wmc_i = & \frac{1}{N} & \forall i \in \{1, \dots, N\} \\ sig_{a,j} = & \frac{1}{2} \exp\left(-x_{m,j}\right) & \forall j \in \{1, \dots, M\} \\ wa_{a,j} = & 0 & \forall j \in \{1, \dots, M\} \end{array} \tag{3}$$

where $x_{m,j}$ is the $x$ coordinate of the middle of cell $j$.

**Time step** Each active particle gets updated according to the following scheme:

- Compute the traveled distance $h \sim \exp\left(\frac{sigs}{1 - absorption\ rate}\right)$
- Decide if the particle interacts or leaves
  - If $|h \cdot \mu| < dist$, where dist is the appropriate distance to the next cell, an interaction happens
  - If $|h \cdot \mu| > dist$, the particle leaves the cell
- Reduce the particles energy and add to the cells energy
- Then:
  - Interaction: Displace the particle in the cell and draw a new velocity $\mu \sim unif\,(-1, 1)$
  - Exit: Displace the particle to the border of the new cell and leave $\mu$ constant
- If the particles energy crosses a lower threshold, or the particle leaves the computational domain, deactivate it

**Evaluation** After all particles have been deactivated, the energy for each cell can be computed and displayed as seen in Figure 2.

## 2.2 Serial Program

The serial program aims at completing the simulation as described in subsection 2.1 serially. In the following the reference implementation will be explained. The current code version (v0.2.2[1]) includes some enhancements which will be given afterwards.

**Reference** For the reference implementation the particles attributes (position $x$, velocity $\mu$, energy $wmc$, cell index $nc$, random number seed $sd$, particle event $ev$, i.e. interaction or exit, distance to the event $di$ and disable flag $disable$) are stored in arrays of length $N$. All cell attributes ($sig$ and energy $wa$) are stored in arrays of size $M$.
The reference implementation simulates one time step for each particle in a round-robin fashion. While this leads to stride-one access for the particle attributes, it causes nearly random access to the cell attributes. Additionally, data from many arrays has to be loaded into cache simultaneously.

**v0.2.2** For the v0.2.2 serial implementation I tried to enhance the cache friendliness. Therefore, the essential particle attributes (position $x$, velocity $\mu$, energy $wmc$, cell index $nc$, random number seed $sd$) are stored in a struct. All particles are stored in an array of structs of size $N$. The cell attributes are stored as in the reference implementation.
The v0.2.2 implementation simulates each particle until it is disabled. This leads to the particle-data in cache being constant and to stride plus or minus one access for cell data.

**Comparison** Although the v0.2.2 implementation uses C++ features such as classes, `std::vector` and other abstractions, the runtimes for $M = 1000$ and $N = 1000000$ are ca. 23 seconds (v0.2.2) and ca. 60 seconds (reference)[2].

## 2.3 Objectives

The objective of this project is to parallelize the simulation given through the reference implementation as described in subsection 2.2 using MPI for inter processor communication and OpenMP for shared memory parallelism. Additionally to finding an optimal parallel communication strategy, it should be investigated how synchronous vs. asynchronous communication influences the runtime behavior. The parallel program should be executed on a many node cluster and its scaling should be analyzed. Lastly, a simulation with a number of particles $N$ that does not fit into RAM should be run.

## 2.4 Technology Choices

The project will use MPI and OpenMP for parallelism. The parallel program will be implemented in C++ using either the C++11, C++14 or C++17 standard depending on the utilized features of the component. C++ enables easier abstractions which often come at zero overhead [4]. CMake [3] is used as the build system, because it enables relatively easy compilation into libraries which simplifies unit tests and modularization.
Automatic report generation, data handling, etc. is done through a mix of python, bash and LaTex.

## 2.5 Domain Decomposition

Domain decomposition is a standard approach in parallel computing. It is chosen for this project to reflect standard numerical practice. Nevertheless it should be noted that decomposing the problem into sets of particles would most likely lead to easier and faster parallelism.
The computational domain $D = [0, 1]$ is split into $K$ layers and each layer is assigned to exactly one MPI process and usually each MPI process is mapped to one processor.

## 2.6 Communication

The asynchronous communication is realized through the `AsyncComm<T>` class which has the following public interface

```
template <typename T> class AsyncComm {
public:
  void init(int world_rank, MPI_Datatype const mpi_t, size_t max_buffer_size);

  void send(T const &instance, int dest, int tag);
  void send(std::vector<T> &data, int dest, int tag);
```

[1]Code can be obtained by E-Mail at `lkskstlr@gmail.com`.
[2]CPU: `Intel(R) Core(TM) i5-3210M CPU @ 2.50GHz`, RAM: `16 GB 1600 MHz DDR3`, OS: `macOS`.

```
  bool recv(std::vector<T> &data, int source, int tag);

  void free();
}
```

The template parameter `T` denotes the type of the objects to be sent, e.g. `struct Particle`. The two `send` methods take ownership of the data, i.e. the `std::vector<T> &data` will be cleared. The `AsyncComm<T>` object has an internal buffer (essentially a `std::vector` of C arrays) that temporarily holds the data.

Each call to `send` triggers and MPI_Isend call. Each call to `recv` triggers potentially multiple MPI_Iprobe and MPI_Recv calls until no more MPI_Isends are pending. A call to `free` essentially removes all buffers from the buffer vector for which the MPI_Isend call has completed. The user code should call `free` repeatedly, e.g. every 10 sends.

The design of the communicator is specific to the simulation and relies on the fact that the particles do not interact with each other. Therefore no restrictions on when data has to arrive are given and no synchronization is needed.

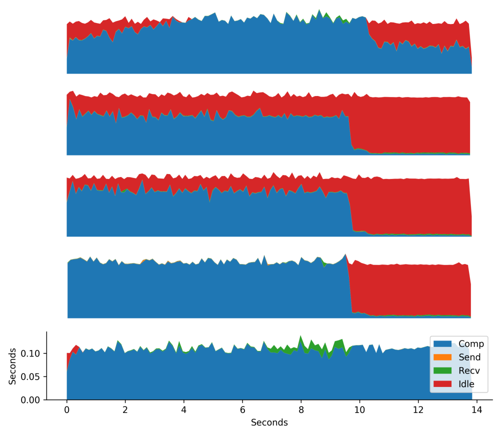
Each MPI process runs a loop in which, first, a fixed number of particle steps are computed. Afterwards, all particles that crossed the boundary are sent to neighboring processes, and the number of disabled particles is sent to the root. After a fixed number of loops, statistical data is accumulated and saved. In each loop the process receives all particles from its neighbors that are buffered within the neighbors `AsyncComm<T>` object. If the process has no active particles left, it waits for the termination message from the root process.

## 3 Results

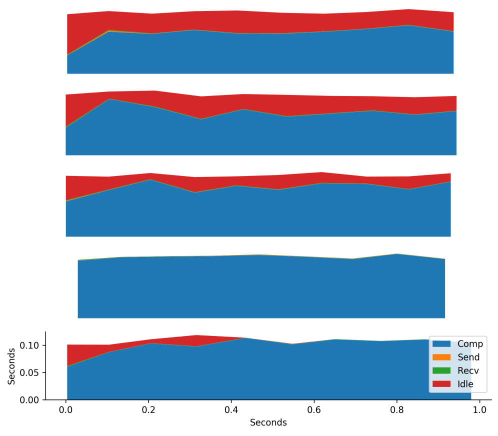
The following results are an excerpt from an automatically generated `single_run` report.

The run started at December 4, 2018 10:32:11 and the root machine was Lukass-MacBook-Pro.local[2]. The simulation used 5 processes to simulate 1000000 particles in a total of 1000 cells, i.e. 200 cells per process. The runtime for the simulation was: 13.86 s. Figure 3 displays the loading.

The simulation used a total of 10009 MPI_Send & MPI_Recv calls which excludes calls after the simulation end to aggregate statistics data.



(a) Loading for whole simulation.



(b) Loading for the first second.

Figure 3: Load for all processes. Process 0 is shown at the top. The sum of all times (stack height) should equal the statistics_cycle_time, i.e. 0.1 sec.

## 4 Discussion

Figure 3 indicates that the current asynchronous communication strategy leads to rapid startup, i.e. time until every node can compute. Also the communication overhead is small. It must be noted that all simulations were performed on a single machine[2] and scaling may be different on a cluster.

The following should be addressed in the second half of the project:

- Implement suboptimal (synchronized) communication
- Compare different communication strategies, and investigate scaling on the cluster
- Add OpenMP parallelization
- Add remote memory access (MPI)
- Larger than RAM simulation
- (Bonus) add GPU parallelization utilizing CUDA

## 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] Kitware Inc. *CMake*. 2018. URL: `https://cmake.org`.

[4] Bjarne Stroustrup. "Foundations of C++". In: *European Symposium on Programming*. Springer. 2012, pp. 1–25. URL: `http://www.stroustrup.com/ETAPS12-corrected.pdf`.