Santiago Pagola (sanpa993)
Hao-Hsiang Liao (haoli436)

TDDC78 Lab 4

Particle simulation

Table of Contents

Introduction....2
Solution....2
  Handling the particles....2
  Finding out who my neighbors are....2
  Particle generation....3
  Particle interaction....3
    When two particles collide....3
    When a particle leaves my region....3
Results....5

Santiago Pagola (sanpa993)
Hao-Hsiang Liao (haoli436)

Introduction

The objective of this assignment is to do a particle simulation and to verify that the gas law holds: *pV = nRT.* There are two ways we are going to prove that this law holds: firstly by increasing the number of particles in the box, we expect the global pressure to increase, since the pressure *p* is directly proportional to the the number of particles *n* in the box. Secondly, a more parallel-computing related test will be used in order to prove that given a fixed number of particles in the box, *n,* the pressure *p* will remain constant but the execution time *t* will decrease hugely by using more computation cores. It is worth mentioning that this second test isn't exactly fair, since the inner loops within the main time-step loop run in $O(n^2)$, so by splitting the particles between the processors, more than a 50 % of speedup will be achieved by using double the number of resources. Let's analyze our solution.

Solution

This section aims to describe the developed solution in order to get an understanding of how the initially given sequential program has been parallelized.

Handling the particles

The particles are managed by doubly-linked lists, where every node holds a pointer to a particle structure (pcord_t), not raw data. This way it is faster to traverse the list and extract / append elements from / to it. The doubly-linked list is implemented in files dll.{c,h}. When a particle is to be extracted from the list, the function dll_extract() is used, and when a new particle coming from other cores is to be added to the current process' particle list, dll_append() is used. Forward / backward iteration through the particle list is a fast operation since both next and previous nodes on the list are known by the current node. Every process holds the following lists:

- my_list: the global list with the particles to process
- my_send_lists[4]: an array of lists, one for every neighbor. Particles in these lists will be abandoning the current process
- my_recv_lists[4]: an array of lists, one from every neighbor. Particles in these lists will be appended to my_list.

Finding out who my neighbors are

Since we want to create a two-dimensional MPI topology, processors will be arranged in a grid within their communicator. In order to do that, given the desired number of cores to use, the function compute_grid_dimensions() computes the dimensions that the topology will have, attempting to use the two closest factors that yield the number of cores, e.g., if 9 cores are available, this function shall return the dimensions 3x3, not 1x9; if 12 cores are available, this function shall return the dimensions 3x4, not 1x12, and so on. This is to optimize the dimension creation. The MPI built-in function `MPI_Cart_get()` will return the coordinates that every core has been assigned in the grid, and with the help of `MPI_Cart_shift()` we can obtain, for every core, who its neighbors will be. It is worth pointing out that since the 2-dimensional box is being split into squares, every process will have at most 4 neighbors: left, top, right and bottom.

Particle generation

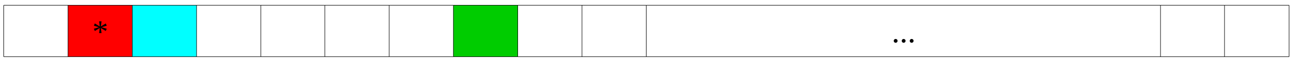
Before the simulation starts, every processor will generate a number of particles within its designated limits, again, depending on the MPI grid coordinates obtained with `MPI_Cart_get()`. The function that computes every process' limits is `get_my_grid_boundaries()`. Then, each particle will be appended to every process' main particle list, called `my_list`.

Particle interaction

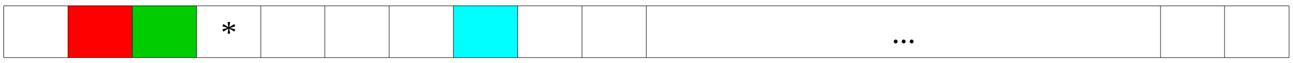
At this point, the behavior of the program is similar to that of the sequential program: particles within the same process are being checked for collisions. If a collision wasn't found between the particle at the current iteration and the rest of the particles, this particle is moved and the momentum is added should this particle collide with the box.

When two particles collide

In case there is a collision, then both collided particles are interacted and the collided particle is swapped with the next particle in the list in order not to go through it again in a later iteration. Graphically explained (the start (*) indicates the current node in the iteration):



Say that the red particle is being checked against the next particles on the list for collisions, and that the green one will actually collide with it. Say that we are at the second iteration on the list (the red particle is the second one in the list). Then, after interacting both particles, the list will end up like this:



And we will jump straight to the fourth element of the list. This way, we won't check the green particle for collisions at a later iteration, thus optimizing the performance. The particle swap operation is done via `dll_swap()` which actually swaps two nodes on the list, being a very fast operation with little overhead (pointer exchange solely).

When a particle leaves my region

After all particles are interacted and checked for collisions, the list is checked in search of particles whose position is outside every process' processing limits. If a particle turns out to be outside, the neighbor to which that particle will travel is determined, and the particle is extracted from the particle list (`my_list`) and appended to the corresponding outgoing list.

Next, for every outgoing list, the particles on each list are sent to the corresponding neighbor but before that, each list is converted to an array via `dll_to_array()`. This way it becomes easier to send the particles through `MPI_Send()`. Note that there might be no particles at all for the current outgoing neighbor, in which case there will be 0 elements to be sent. The opposite neighbor will then probe for a message from its opposite neighbor via `MPI_Probe()`, and in case of being a message, it will get the number of elements that were sent to it with `MPI_Get_count()`. Finally, it will call `MPI_Recv()` with the item count set to the number of elements determined by `MPI_Get_count()`. All incoming particles shall be received into the corresponding list in `my_recv_lists[4]` each process holds.

Lastly, all received particles in `my_recv_lists[4]` are appended to `my_list` for further processing in the next time step.

Santiago Pagola (sanpa993)
Hao-Hsiang Liao (haoli436)

Results

This last section shows the obtained simulation results, both graphically and numerically. The first test that has been made is the following: by increasing the number of particles in the box, the pressure should increase, while the execution time should remain constant. For this purpose, we assume that every core is processing 10k particles, so the total number of particles in the box, *n*, increases when using more cores. Here are the numerical values obtained from the simulation:

# cores	Simulation time (s)	Global pressure	# particles / core	# particles (*n)*
1	11.804968	0.037142	10000	10000
2	12.334865	0.082535	10000	20000
4	12.349498	0.165876	10000	40000
8	13.947468	0.316338	10000	80000
16	15.357635	0.567281	10000	160000
32	13.572056	1.090733	10000	320000
64	11.247533	1.665412	10000	640000

And here is the graphical plot of the above numbers:

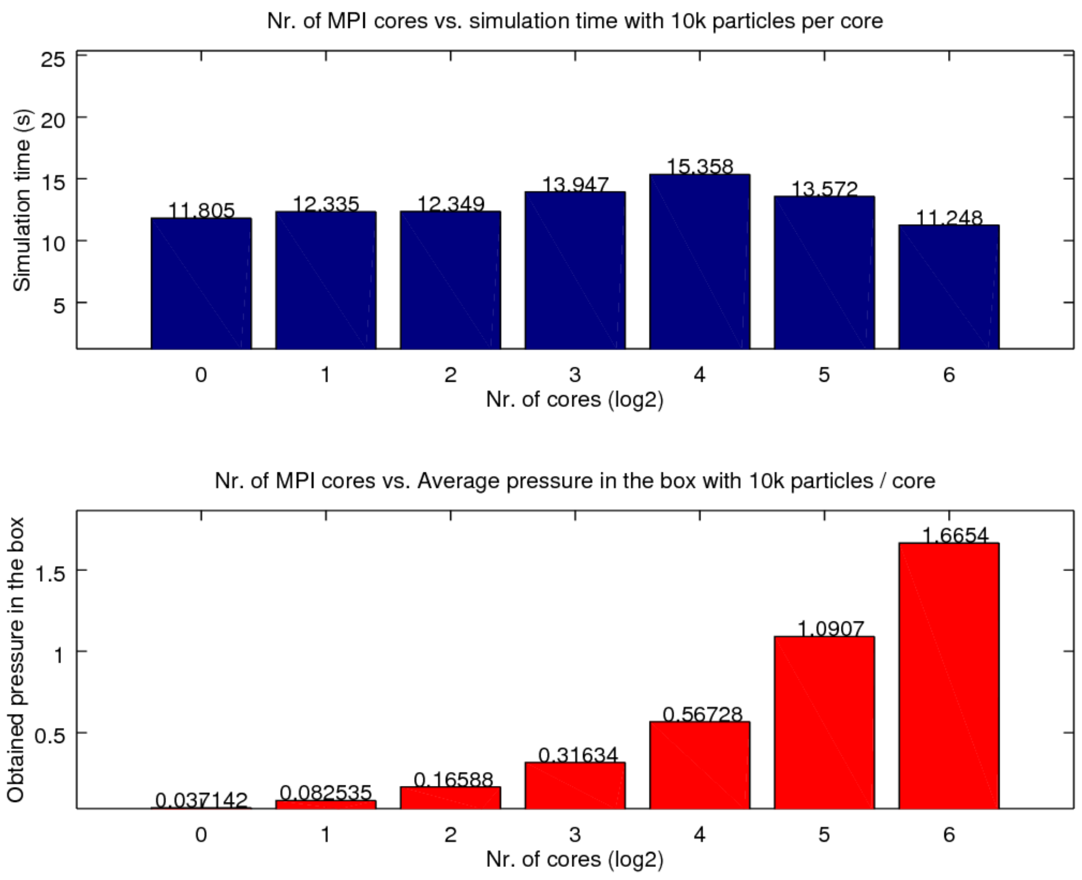


Figure 1: Execution times (top) and pressure increase (bottom) for different number of MPI cores. As expected, the pressure keeps increasing while adding up more particles to the same box volume.

The numbers in the table and in the above plot confirm what we expected that would happen: given the same volume *V* of the box, that is, 10000 x 10000, increasing *n* will only lead to a higher pressure in the box, while the execution times remain approximately the same since every core is processing the same amount of particles at the same time. It is worth commenting that the execution times may not be equal due to an additional reason: there might not be the same number of particles flowing from one core to another, i.e., perhaps there are more particles that leave a certain grid area in a given execution that particles incoming from other contiguous cores, hence ending with less particles to process at every iteration.

The second test, as described at the beginning of the document, attempts to test the speedup of the simulation by fixing the total amount of particles in the box to 50k and splitting the workload between cores. Our theory is that the global pressure *p* shall remain constant for every core simulation since the total amount of particles *n* in the box remains the same: the difference with respect to the previous test is that the number of particles per core will now decrease. In other words: each core will now have less particles to process than the first test. Here are the numerical results:

# cores	Simulation time (s)	Global pressure	# particles / core	# particles (*n*)
1	342.740080	0.202134	50000	50000
2	83.052188	0.202634	25000	50000
4	20.401397	0.205647	12500	50000
8	5.116517	0.207849	6250	50000
16	1.323736	0.200321	3125	50000
32	0.385024	0.219347	1562	50000
34	0.115883	0.194617	781	50000

And here is the graphical representation of the above results:

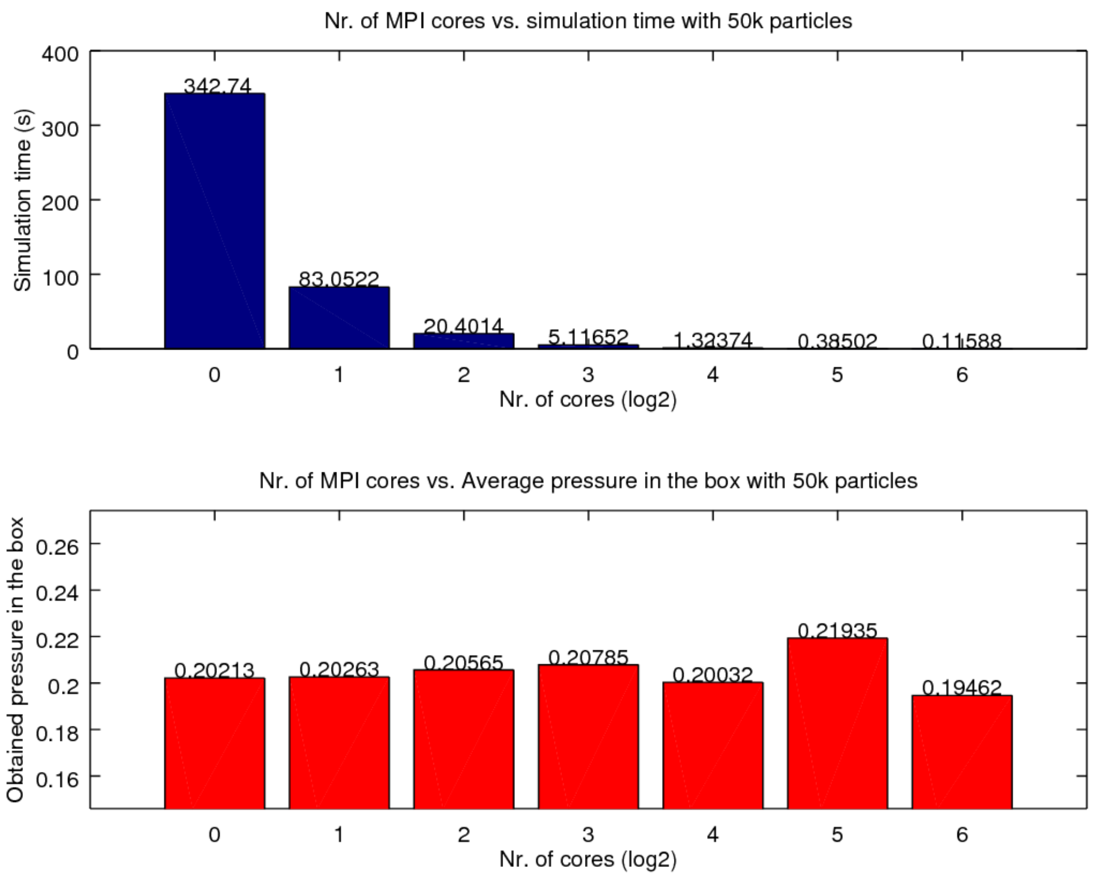


Figure 2: Simulation time (top) vs global pressure (bottom) with different cores. As expected, the global pressure remains constant at ~ 0.2 since the total amount of particles n in the box remains constant, while the execution times decrease drastically.

Once again, the obtained behavior is the expected one: by increasing the number of cores and by dividing the workload (50k particles) between cores, the execution times are decreased. Note how much time it takes for a single core to process all 50k particles (almost 6 minutes) compared to roughly one tenth of a second when using 64 cores. The pressure keeps constant since the total number of particles *n* in the box remains the same for all executions (keeping the same box volume *V*).