MASTER IN HIGH PERFORMANCE COMPUTING

Optimization of the Direct Simulation Monte Carlo code (DSMC3D) for rarefied gas flow simulations

Supervisor(s):
Dr. Ivan Girotto ICTP-MHPC
Reviewer:
Dr. Andrew Emerson CINECA, HPC DEPARTMENT

Candidate:
Dr. Orlenys Natali TROCONIS

7th EDITION
2020–2021

To all of the scientists in refugee situation.

Acknowledgement

Thanks to my supervisor *Ivan Girotto* for sharing with me his knowledge, for his patience, thanks for understanding me when I needed to perform a surgical procedure, for your understanding when I needed to meet my mother after so many years. Thank you even for the moments in which you lost your patience because it gave me a boost. Thank you for proposing this thesis to me as a door to the possibility to find a job.
Thanks to my Cineca colleague *Andrew Emerson* for his detailed review and valuable comments for this thesis. Thanks a lot to dedicate your time to read, comment and to explain me your observations. I really appreciate! Thanks to my Cineca colleague *Laura Bellantani* to guide me in the use of the parallel profiling tool intel APS.
Thanks to *Dr. Gianluca Di Staso* for his support in the initial phase of this project. Thanks for your patience and for sharing with me your code.
Thanks to the *International Center of Theoretical Physics (ICTP - Trieste, Italy)* for giving me the opportunity to study this master, for help me not only from the economical point of view but as well to recover my professional path after been in a difficult situation as a refugee in Spain.
Thanks to my dear friend Ralph Gebauer because thanks to you I realized the existence of the master in HPC. Thanks for supporting me everyday with you friendship. Thanks a lot my dear friend from the deep of my hart! You really know the meaning of the word: friendship! Thanks to Prof. Joe Niemela for his support when I asked him help when I was in a difficult situation. I will always be grateful to you!
Thanks to my MHPC colleagues *Saeid Aliei, Mattia Mencagli and Marco Celoria* for their help when I need it. Thanks to my ICTP friends: *Alessandro Porcella and Emily Sabina* for their friendship and support.
Thanks to the most important person in my life: *my mom Irma Troconis*. Thanks for all of your love. Te amo mami! Thanks to my syster *Aimara Troconis* for been with me from the distance everyday since I left my country. Te amo hermanita!
Thanks to two important people that help me during my hard time in Spain: *Nieves Alvarez (QEPD) and Luis Vidal*. Gracias por haberme ayudado siempre Luis!

Abstract

The Direct Simulation Monte Carlo DSMC method has become the method of choice for modeling rarefied gas dynamics in a variety of scenarios after its introduction 50 years ago. Several parallel DSMC codes have been developed over the last 30 years to leverage the increasing computational power of parallel machines. In this work we are going to focus on the DSMC3D code developed at Eindhoven University of Technology. The code is based on the message passing interface paradigm (MPI) using a cartesian domain decomposition. However, at large scales the code presents some load balancing issues that impact the performance. The main objective of the project is to introduce another level of parallelization using a shared memory approach with OpenMP that allows us to reduce the number of MPI's processes.

Keywords: direct simulation Monte Carlo, shared memory, distributed memory

List of Figures

List of Tables

Contents

Chapter 1

Introduction and Motivation

The Direct Simulation Monte Carlo method was invented more than 50 years ago by Graeme Bird to describe rarefied gas flows [1, 2]. Since that time, the power of the largest computers available for scientific simulation has been increased dramatically due to both continuous advances in processor hardware and the advent of massively parallel supercomputers based on cheap commodity components. Initially, parallel machines used single or few core CPUs; in the last decade, accelerator chips such as GPUs and many core CPUs (e.g., the Intel KNL processor) have been leveraged for scientific use, including in the largest parallel machines.

In 1988 the fastest supercomputer (as measured by the LINPACK benchmark) was an 8 processor Cray YMP, which could factor a dense matrix at the rate of 2.1 gigaflops (2.1^9 floating point operations per second). In 2019 the fastest current machine computes the same benchmark (for a much larger matrix) at 122 petaflops (122×10^{15} floating point operations per second). In 2022 the world arrived to the exascale (10^{18}) machines, the first one in the top 10 list (October 2023) being the Frontier machine installed at the Oak Ridge National Laboratory (ORNL) in Tennessee, USA, operated for the Department of Energy (DOE). It currently has achieved 1.194 PFlop/s using 8.699.904 cores. At the moment of writing this thesis (October 2023), the only new machine to grace the top 10 of the list was the No. 4 Leonardo system at EuroHPC/CINECA in Bologna, Italy. The machine achieved a linkpack performance R_{max} of 238.70 PFlop/s [11]. While these performance numbers are for dense-matrix operations, many scientific computational methods, including DSMC, have benefited from similar factors of speed increase due to Moore's Law and massive parallelism.

The fundamental DSMC algorithm is inherently parallel. In each time step, particles advect independently and may experience collisions with surface elements representing the surfaces of complex objects embedded in the gas flow. Particles are then grouped by grid cell, and pairwise collisions and chemical reactions are computed independently within each grid cell. Thus, the advection step can be parallelized over particles, and the collision-chemistry step can be parallelized over grid cells.

Several parallel DSMC codes have been developed over the last 30 years to leverage the increasing computational power of parallel machines. In this work we are going to focus on the DSMC3D code developed at Eindhoven University of Technology by Doctor Gianluca Di Stasso [10]. The code is based on the message passing interface paradigm (MPI) using a cartesian domain decomposition. The software was developed to study different kinds of flows, but we are going to focus in two specific cases: the Poiseuille flow and a flow with open boundary conditions.

In an MPI base code with a high number of processes load balancing means to distribute equal amounts of computation to each process while minimizing the volume of communication between them. The DSMC method is expensive in terms of time due to its particle-based and Monte Carlo nature. The design of the code may require a large amount of inter-processor communications, that implies load balancing problems and difficulties in reaching very good parallel scaling performances. In the case of the flow with open boundary conditions load balancing problems are naturally present since in this case there is a gradient in the number of particles that makes some processes do more computation than others.

Inspired by the load balancing problems, the main objectives of the thesis are to perform a deep benchmark of the DSMC3D code for the case of a flow with open boundary conditions, and introduce another level of paralellization with OpenMP in order to reduce the number of MPIs processes using a new data structure that allows threaded parallelism over the cells grid using a standard Poiseuille flow as a model.

The thesis is organized in three chapters: in the first one we present the performance study and bottlenecks for the DSMC3D code for a flow with open boundary conditions. In the second chapter we introduce a new data structure to be used for the DSMC code for a Poiseuille flow set-up and present a benchmark for the results obtained in MARCONI-skl Cineca cluster. Finally, some general conclusions will be presented.

Chapter 2

Performance study and bottlenecks

2.1 Architecture description and compilers used

We are using two HPC architectures: the ICTP's HPC system ARGO and the Cineca's HPC system MARCONI (figures 2.1 and 2.2). Overall, ARGO is a heterogeneous cluster, with nodes belonging to various generations of Intel CPU micro architectures. We are using the long queue, based on sandybridge and ivybridge architecture, where we are able to run up to 8 nodes with a time limit of one day. The characteristics of the long queue ARGO partition for ivybridge micro-architecture are [8]

- Nodes: 8.
- Processors: 2 x 10-cores Intel Ivybridge.
- Cores: 20 cores/node.
- RAM: 64 GB/node
- 40 Gbps Infiniband technology
- Peak Performance: not available.

On MARCONI we are using the third partition $A3$ based on Intel Xeon 8160 (SkyLake). The basic characteristics of Marconi A3 are [5]

- Nodes: 3188.

Figure 2.1: ICTP cluster: ARGO.

- Processors: 2 x 24-cores Intel Xeon 8160 (SkyLake) at 2.10 GHz.
- Cores: 48 cores/node.
- RAM: 192 GB/node
- 100 Gb/s Intel OmniPath technology [1]
- Peak Performance: $\sim$ 10 PFlop/s.

The compiler used for the next benchmarks on both architectures is the intel compiler and for each configuration we used the full node. On ARGO the benchmark is for 1, 2, 4 and 8 nodes, i.e. 20, 40, 80 and 160 cores, respectively. On MARCONI the benchmark is for 1, 2, 4, 8, 16, 32 and 64 nodes, i.e. the number of processes are 48, 96, 192, 384, 768, 1536 and 3072, respectively.

[1] At the time of setup, MARCONI was the largest Omnipath cluster in the world.

Figure 2.2: Cineca cluster: Marconi - A3 (Skylake).

2.2 Performance metrics

There are different metrics for measuring the performance of the DSMC code. In this project we focus on three of them: the absolute time, the average time per call and the time per particle. The total time is the elapsed time for a specific kernel (in seconds) which is based on the use of the *gettimeofday* c function. The average time per call is defined as the total time divided by the number of calls of a specific kernel

$$AVG \times CALL = \frac{TotalTime}{NumCalls}(\text{seconds}). \tag{2.1}$$

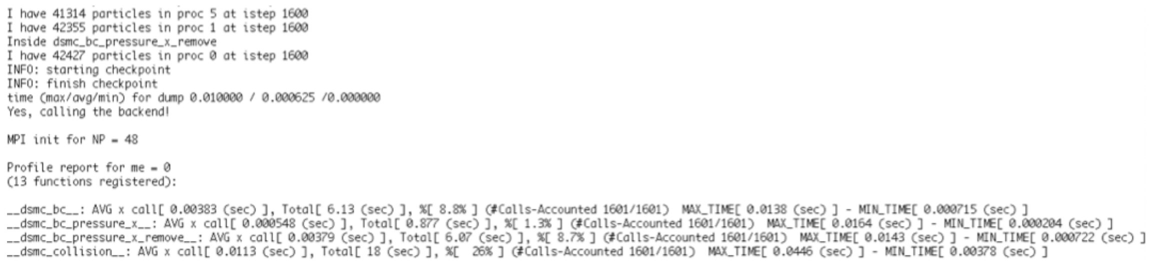
The time per particle is the total time of a specific kernel divided by the mean value of the number of particles for master process $< N >$,

$$Time\_per\_particle = \frac{TotalTime}{< N >}(\text{seconds}), \tag{2.2}$$

where $< N >$ is computed as the sum of the number of particles at each time step divided by the number of steps

$$< N >= \frac{\sum_{istep} N}{< number\_of\_steps >} \tag{2.3}$$

Figure 2.3 corresponds to an output example for a run on Marconi for 1600 steps. At each time step the number of particles for each process is printed, this allows us to compute the mean value of N for master process. The functions reported in the log file, are those which the time is higher than 0.5% of the total time of the simulation.



```
I have 41314 particles in proc 5 at istep 1600
I have 42355 particles in proc 1 at istep 1600
Inside dsmc_bc_pressure_x_remove
I have 42427 particles in proc 0 at istep 1600
INFO: starting checkpoint
INFO: finish checkpoint
time (max/avg/min) for dump 0.010000 / 0.000625 /0.000000
Yes, calling the backend!

MPI init for NP = 48

Profile report for me = 0
(13 functions registered):

__dsmc_bc__: AVG x call[ 0.00383 (sec) ], Total[ 6.13 (sec) ], %[ 8.8% ] (#Calls-Accounted 1601/1601)  MAX_TIME[ 0.0138 (sec) ] - MIN_TIME[ 0.000715 (sec) ]
__dsmc_bc_pressure_x__: AVG x call[ 0.000548 (sec) ], Total[ 0.877 (sec) ], %[ 1.3% ] (#Calls-Accounted 1601/1601)  MAX_TIME[ 0.0164 (sec) ] - MIN_TIME[ 0.000204 (sec) ]
__dsmc_bc_pressure_x_remove__: AVG x call[ 0.00379 (sec) ], Total[ 6.07 (sec) ], %[ 8.7% ] (#Calls-Accounted 1601/1601)  MAX_TIME[ 0.0143 (sec) ] - MIN_TIME[ 0.000722 (sec) ]
__dsmc_collision__: AVG x call[ 0.0113 (sec) ], Total[ 18 (sec) ], %[  26% ] (#Calls-Accounted 1601/1601)  MAX_TIME[ 0.0446 (sec) ] - MIN_TIME[ 0.00378 (sec) ]
```

Figure 2.3: Output example for the unbalance case ($P = 417.54$Pa).

Finally, the speed-up of the code for the different configurations is computed dividing the total time of the simulation for 1 node by the time for the different cases,

$$speedup = \frac{Time(1node)}{Time(nnodes)}, \tag{2.4}$$

for example the speed-up for 2 nodes is $Time(1node)/Time(2nodes)$.

2.3 Flow with open boundary conditions

A flow with open boundary conditions is simulated thanks to the imposition of a desired pressure at the inlet/outlet sections of the flow domain. The different pressures in the boundaries, inflow and outflow P_{in} and P_{out}, respectively, implies a gradient in the number of particles along the velocity direction. The problem is defined so that the inlet/outlet surfaces are normal to the x-component of the mean flow velocity vector.

Figure 2.4 explains why we expect load balance issues in a fluid with open boundary conditions when it is described with a full 3-D DSMC code using the message passing interface paradigm (MPI). The gradient in the number of particles along the velocity direction implies that some processes will receive more particles than others; in the figure the process identified as Proc a with Na particles, and near to the inlet boundary, will do more computation than a process near to the outlet boundary, Proc b with Nb particles, if some message needs to be passed from Proc a then Proc b has to wait until Proc a finishes the computation, impacting the performance of the entire simulation. Moreover, if for some reason the number of particles for the processes near to the inlet boundary increases a lot and they need to reallocate memory, then the communication part starts to be affected consequently impacting the performance.

In the next sections we describe the main algorithm for the open boundary conditions case and we will show the computation imbalance in place on both architectures described in subsection 2.1, from the qualitative and quantitative points of view.

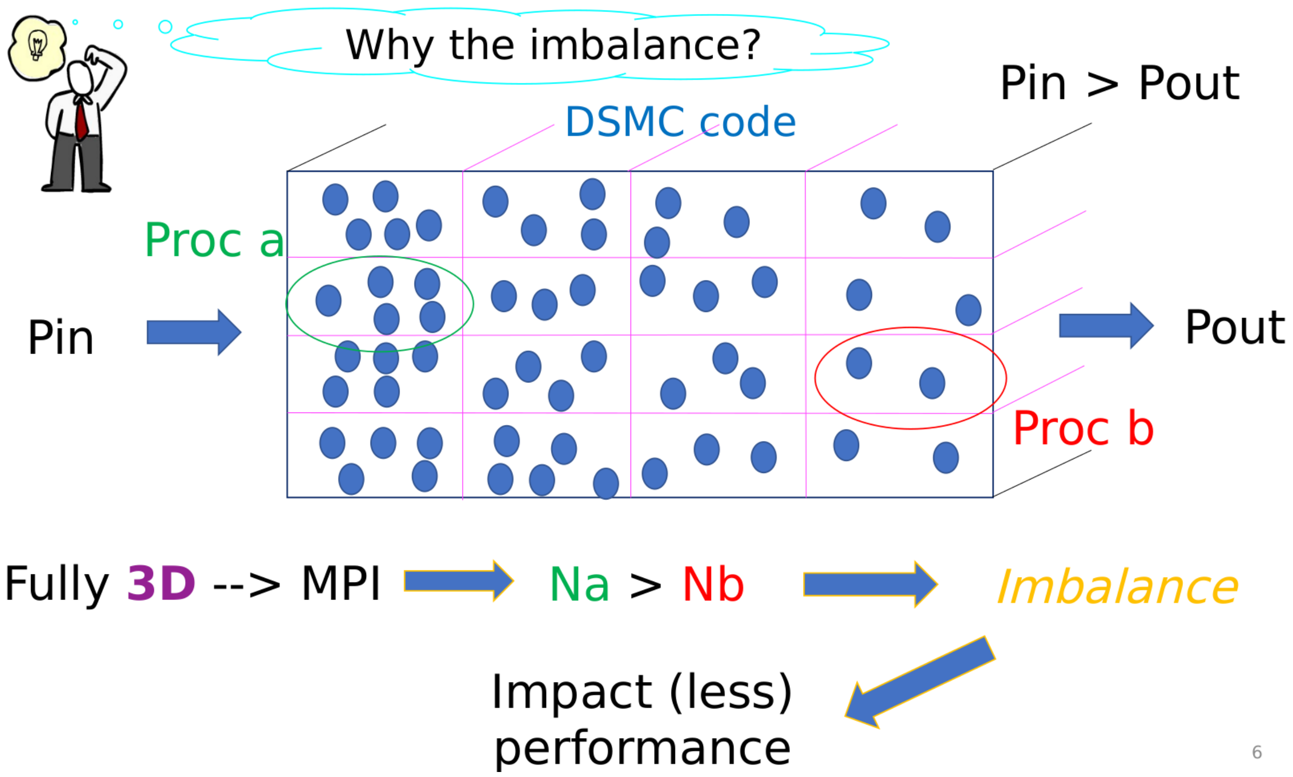


Figure 2.4: Flow with open boundary conditions.

2.3.1 Algorithm description for a flow with open boundary conditions

In order to understand the different plots for the benchmark on the two architectures (ARGO and MARCONI), in this section we are going to explain the main functions used in the algorithm that describes the DSMC3D code (see 1).

The starting point is the initialization of the grid, particles and sampling variables by the functions *init_dsmc_geometry*, *init_dsmc_particle*, *dsmc_init_sampling* (to initialize the different velocity components and kinetic energy) and *dsmc_hydrovar* (to initialize the different components of the momentum).

The time step loop consist of: *dsmc_bc_pressure_x* which determines the inlet boundary conditions according to the pressure and temperature of the incoming flow (pressure boundary conditions). The number of sampled particles for the inlet boundary is determined by a Poisson distribution which mean value is given by equation (3.18) in reference [10]. Particles are moved and computed their interaction with the boundaries, thanks to the functions *dsmc_step* and *dsmc_bc*. In *dsmc_bc_pressure_x_remove* kernel, particles are simply removed when they cross inlet or outlet boundaries. The communication part corresponds to the function *dsmc_mpi*; in this case we split it into the different parts namely NOMPI (the part in which there are no MPI communication calls), the Send-Receive part called MPI-Sendrecv (in which the different buffers and their size are sent and received), the MPI-Allreduce and MPI-Barrier part. It is important to make note that the kernels *dsmc_bc_pressure_x* and *dsmc_bc_pressure_x_remove* include collective operations (*MPI_Allreduce*). The *dsmc_indexing* kernel assigns an index to the particles according to its position in the cell and subcell (more details about the indexing function in the next chapter). In *dsmc_collision* stochastic binary collisions for particles that resides in the same subcell are performed in a random fashion. Finally, the variables are sampled and some statistics are calculated.

Algorithm 1 Main loop of the dsmc code for a flow with open bc

Initialization: *cells, particles and sampling variables*

```
 1: for Every time step do
 2:     dsmc_bc_pressure_x (&dsmc_particle,istep_DSMC);
 3:     dsmc_step ();
 4:     dsmc_bc ();
 5:     dsmc_bc_pressure_x_remove(&dsmc_particle,istep_DSMC);
 6:     dsmc_mpi ();
 7:     dsmc_indexing ();
 8:     dsmc_collision ();
 9:     dsmc_data_sampling ();
10:     dsmc_hydrovar ();
11:     statistics_dsmc ();
12: end for
```

2.4 The observed imbalance

The imbalance is due to the unequal distribution of workload between the processes and this impacts parallel performance.

At the end of the simulation we were able to get the log (2.3) file for each process and as well to identify the process with the maximum and minimum number of particles N_{max} and N_{min}, respectively. Using such information we can plot the breakdown of the main loop for each of them as well for the master process. From figure 2.5 we can conclude that the process with the highest number of particles is spending most of the time on the kernels related with computation (*collision*, *indexing* and *data_sampling*), on the other hand the process with fewer number of particles is spending most of the time on kernels with MPI calls inside (*dsmc_bc_pressure_x*, *dsmc_bc_pressure_x_remove* and the different parts of *dsmc_mpi*).

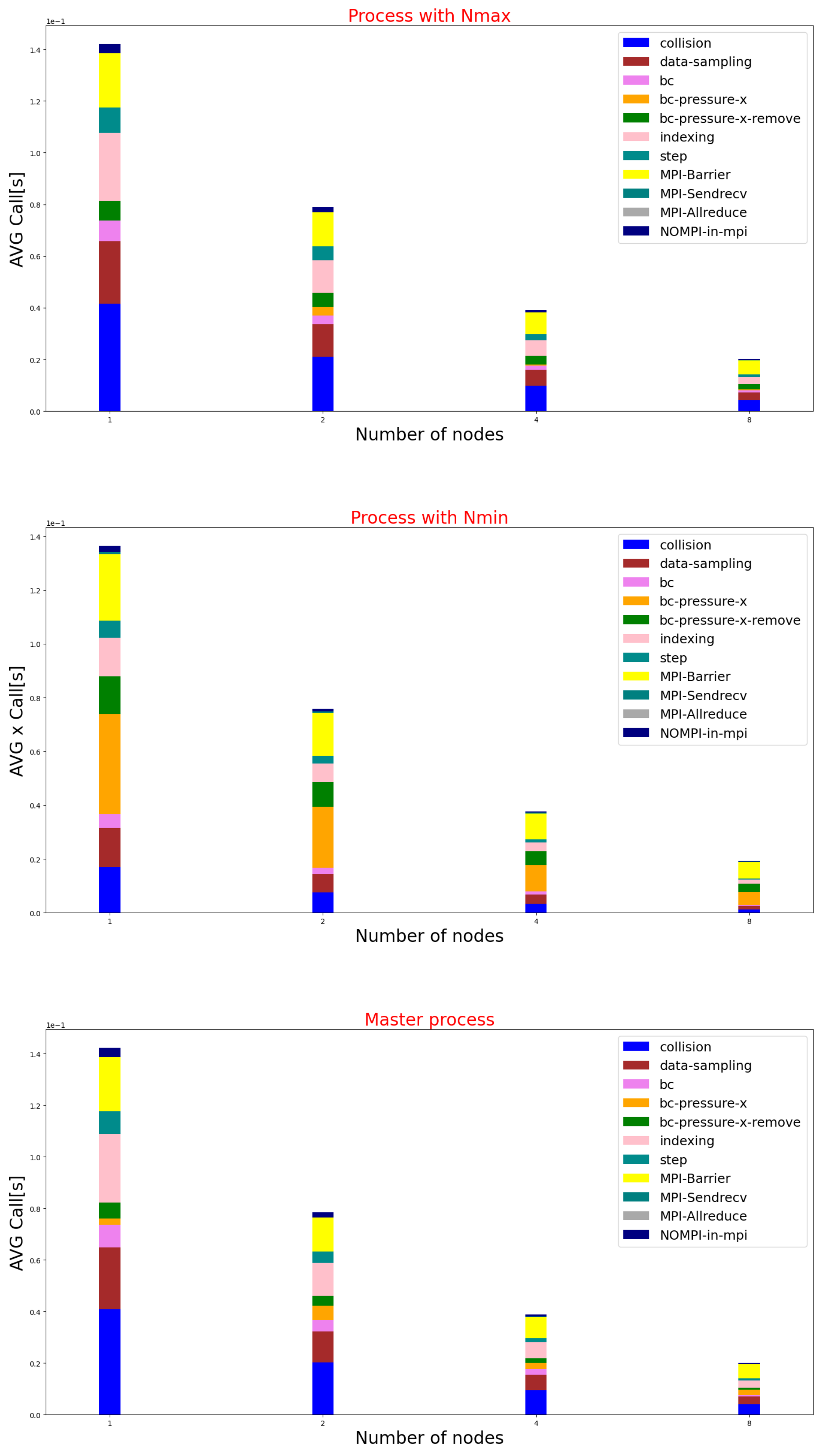


Figure 2.5: Breakdown of the average x call time for $P = 835.08$Pa, density $= 1.008 \times 10^{23}$, number of molecules=400000, $v_x = 10$m/sec, $\gamma = 1.67$, mass $= 6.63 \times 10^{-26}$kg. Benchmark for processes with maximum (N_{max}) and minimum (N_{min}) number of particles and master process. Run on ARGO long queue using the intel compiler. Grid 200 x 40 x 10 cells.

Chapter 3

Code Optimization

3.1 Original vs the new data structure. The Poiseuille flow

A serial or parallel DSMC code involves two important kinds of data: particles and grid cells. They can be implemented using two kinds of data layout approaches: Array of Structures (AoS) or Structure of Arrays (SoA). In the DSMC code the AoS is used as the data layout approach. Particles are defined using a type definition structure identified as *dsmc_type* with around 27 features for a 3-D simulation. Grid cells are defined with a type definition structure called *cell_type* with around 34 features.

The *dsmc_type* stores the information related with the particle in a 3D fashion and its connection with the *cell_type*. Part of the features of the data structure *dsmc_type* are: the molecule's position components and its variation, velocity components (all of them declared as double), molecule position in cell and sub cells [1] (integers), one integer called cross reference list to be used in the collision phase, and some double declared physical characteristics: mass, diameter, omega (viscosity temperature power law exponent), scattering (reciprocal of scattering parameter) and the collision cross section.

[1]In our case the number of sub cells are equal to the number of cells.

The *cell_type* data store the information for grid cells. Part of the features are: minimum and maximum cell coordinate, the cells center coordinates, a counter for the molecules in each cell, the molecules sum per cell, the cell address, sum of molecules kinetic energy and momentum flux tensor terms, the fluid velocity in the cell and the selected couples (used in the collision phase).

The main loop routine for the dsmc code is described in the algorithm sketch 2. Particles are allocated as an array of size equal to the number of molecules in the simulation, adopting the AoS data layout approach each of the main kernels are loops over the particles and when it is needed each of the elements of the *dsmc_type* structure is accessed through the dot operation.

Algorithm 2 Main loop of the dsmc code

```
Initialization: cells, particles and sampling variables
1: for Every time step do
2:     Move particles: dsmc_step ();
3:     Set boundary conditions: dsmc_bc ();
4:     Communicate (migrate particles): dsmc_mpi ();
5:     Index particles into cells: dsmc_indexing ();
6:     Select collision pairs and perform collisions: dsmc_collision ();
7:     Data sampling: dsmc_data_sampling ();
8: end for
```

The move step *dsmc_step* updates each particle that advects (independent of all others) for a straight-line distance determined by its velocity and the time step (see Fig. 3.1). In the DSMC code this is implemented as one large loop over all particles a process owns. Once the particles experiment the move step, then periodic boundary conditions are imposed thanks to the *dsmc_bc*.

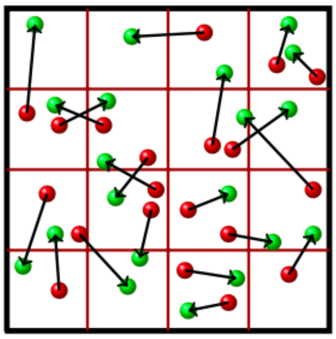


Figure 3.1: Particles configuration before (red dots) and after the streaming step (green dots) [10]

In the communication part, particles which ended their advection in a ghost grid cell owned by another process are sent to that process. In the first stage each process copies the particles into a send buffer and count how many should be sent in each direction. The process then compresses its own particle list and remove those that it is losing. In the second stage, each process sends a message to each neighbour with the count of particles it is going to receive. Each process allocates memory (if it is needed) for the incoming particles. In the original version of the code the reallocation is done by one particle at a time. The particles buffer is sent thanks to the MPI_Sendrecv call using a data type defined as MPI_DSMC_TYPE, which is created using the option MPI_Type_contiguous (sizeof (dsmc_type), MPI_BYTE, &MPI_DSMC_TYPE) in order to preserve the exact same bit sequence.

Once the communication step is ended, particles are sorted and indexed into the cell network. This step is done through a big loop over the particles that each process owns, in that way a list of particles for all its owned grid cells is created. Because particles continuously move to new cells or are added and deleted, the list rapidly becomes unordered with respect to grid cells. The sort creates a linked list for each grid cell of the particles it contains. This requires one integer vector of length number of grid cells, which stores the index of the first particle in the cell, and a second integer vector of length number of particles, where each particle stores the index of the next particle in the same cell. These two vectors can be created by a single loop over all the particles [9]. At the end of indexing kernel the cross_reference_list (to be used in the collision phase) is created.

Figure 3.2 shows the data structure used in the DSMC original code

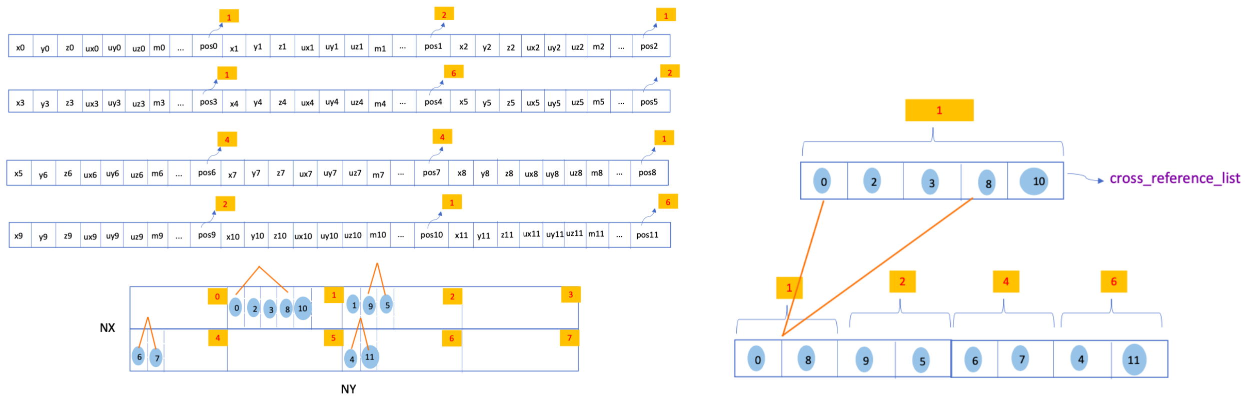


Figure 3.2: Sketch of the data structure used in the DSMC original code. Explanation for the *dsmc_indexing* (left) and the particles selection for the collision phase *dsmc_collision* (right). The highlighted number in the yellow square represents the cell id number. In the example we are using a 2D grid $NX \times NY = 2 \times 4$ cells.

The collision phase is performed in the kernel *dsmc_collision* where stochastic binary collisions between particles that reside in the same cell are done using the information obtained in the indexing function, specifically the *cross_reference_list*. The aim of this step is to determine the post collision velocity satisfying the balance of momentum and energy, given by the following equations

$$m_1\xi_1 + m_1\xi_2 = m_1\xi_1^* + m_1\xi_2^* = (m_1 + m_2)\xi_{cm}, \tag{3.1}$$

$$m_1\xi_1^2 + m_1\xi_2^2 = m_1\xi_1^{*2} + m_1\xi_2^{*2}. \tag{3.2}$$

where ξ_1, ξ_2 are the pre-collisional velocities, ξ_1^*, ξ_2^* the post-collisional velocities and ξ_{cm} is the mass center velocity.

Moreover, the magnitude of the relative velocity should remain unchanged:

$$|\xi_r| = |\xi_1 - \xi_2| = |\xi_r^*| = |\xi_1^* - \xi_2^*|. \tag{3.3}$$

Figure 3.3 is a graphical representation of the pre and post collision particles velocities. The momentum and energy conservation were used as validation test for the new data structure implementation.

More details about how to fully define the post collision relative velocity ξ_r^* in terms of the solid angle for the hard sphere model (HS) used in the code and how to determine the number of attempted collisions in terms of the collision cross section and particle's diameter can be found in reference [10].

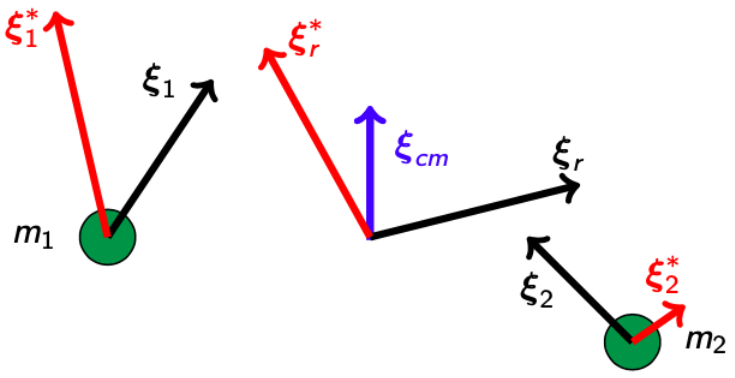


Figure 3.3: Sketch representing the pre-collision, ξ_1, ξ_2, and post-collision, ξ_1^*, ξ_2^*, particles velocities. The velocity of the center of mass, ξ_{cm}, is given by Eq. 3.1. The pre and post relative velocities are ξ_r and ξ_r^*, respectively [10].

3.1.1 The new data structure implementation

In the previous section we see that in the DSMC code most of the loops are over the particles, except in the collision kernel. The parallelization is achieved by distributing cells and particles among MPI processes, in a pure distributed memory approach [6]. Hence if we introduce another level of parallelization, but with a shared memory approach, using OpenMP threads [3], the parallelism will be over particles. In the indexing kernel we will face the case in which two different particles may reside in the same cell and in order to consistently update the number of particles for each cell, an atomic or critical operation is needed.

Figure 3.4 shows the data structure used in this thesis allowing threaded parallelism over cells. For each cell we allocate a fixed size of particles named as MAX_PART_CELL. In the initialization step a loop over particles is made in order to assign the cell position of each particle. Once particles are assigned to the corresponding cell, we can see in the figure 3.4 the cell address is defined according to the cell location and the selected MAX_PART_CELL, meanwhile the particle location is determined by the cell location, the selected MAX_PART_CELL and the counter of the number of molecules that the cell is receiving.

The main functions are loops over the cells and the particles that each cell owns (see the algorithmic description in 4). Since each cell owns its particles, there is no need to create a cross_reference_list for the collision phase mentioned in the previous section.

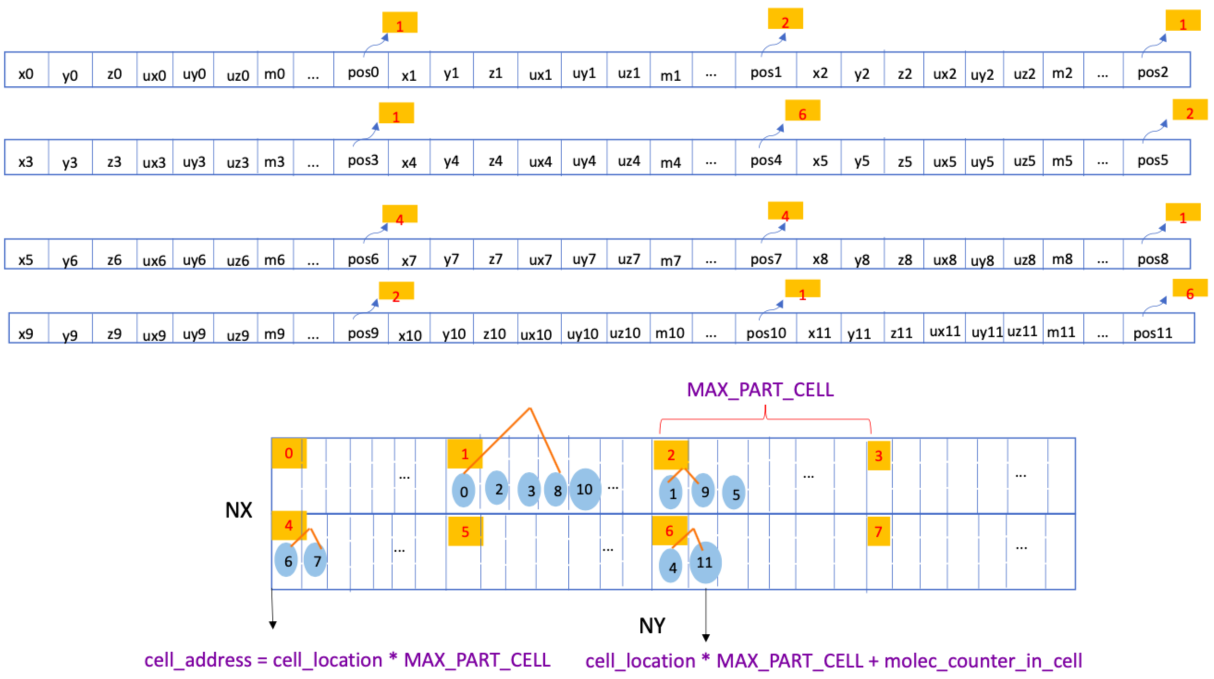
As in the original code, each particle experiments the moving step accordingly to their velocity and time step, thanks to the *dsmc_step* function. The boundary conditions are defined in *dsmc_bc* function, in a similar way as in the original code but with a parallel for loop over cells.

The communication step is very similar to the original version. The particles that a specific process owns are checked to determine if the position corresponds to the domain of that process or if the particle ended its advection in a ghost grid cell owned by another process. The main differences between the original and the hybrid version of the *dsmc_mpi* function are: the loops are over cells and particles that each cell owns, each process allocates enough memory for the incoming particles at the beginning of the *dsmc_mpi_omp* function and not one particle at a time (as in the original version) and finally after the *MPI_Sendrecev* communication, we assign each incoming particle (from right or left) to the corresponding cell (see algorithm 5).

In the *dsmc_indexing_omp* version we classify particles as the ones which are in the correct cell position (consolidated particles) or those which are not. If a particle finishes their advection in an incorrect cell number we need to assign it to the correct one and then copy the particle in *dsmc_sparta_new* buffer [2]; since the parallelism is over cells, a critical operation is needed in order to correctly update the number of particles. After the first loop in which we check the particle position in cell, we copy the consolidated particles to the *dsmc_sparta_new* buffer and at the end we swap the pointers *dsmc_sparta_new* and *dsmc_sparta* (see algorithm 6).

The *dsmc_collision_omp* version is similar to the original one: loops are over the cells. In this case since each cell has its own particles, we randomly select the particles in the cell and perform the binary collisions without using the *cross_reference_list*.

[2]We identify as sparta the new data structure for particles inspired in the open source code SPARTA DSMC [9]



x0 y0 z0 ux0 uy0 uz0 m0 ... pos0 x1 y1 z1 ux1 uy1 uz1 m1 ... pos1 x2 y2 z2 ux2 uy2 uz2 m2 ... pos2
x3 y3 z3 ux3 uy3 uz3 m3 ... pos3 x4 y4 z4 ux4 uy4 uz4 m4 ... pos4 x5 y5 z5 ux5 uy5 uz5 m5 ... pos5
x5 y6 z6 ux6 uy6 uz6 m6 ... pos6 x7 y7 z7 ux7 uy7 uz7 m7 ... pos7 x8 y8 z8 ux8 uy8 uz8 m8 ... pos8
x9 y9 z9 ux9 uy9 uz9 m9 ... pos9 x10 y10 z10 ux10 uy10 uz10 m10 ... pos10 x11 y11 z11 ux11 uy11 uz11 m11 ... pos11
MAX_PART_CELL
NX
NY
cell_address = cell_location * MAX_PART_CELL
cell_location * MAX_PART_CELL + molec_counter_in_cell

Figure 3.4: Sketch of the new data structure. The highlighted number in the yellow square, represents the cell id number. In the example we are using a 2D grid $NX \times NY = 2 \times 4$ cells.

Algorithm 3 Original dsmc code

```
Init: cells, particles, sampling variables
1:  for Every time step do
2:      for each particle do
3:          step;
4:      end for
5:      for each particle do
6:          bc;
7:      end for
8:      for each particle do
9:          check position x;
10:     end for
11:     Communicate mpi in x;
12:     if it is needed then
13:         Allocate more memory
14:     end if
15:     dsmc_indexing();
16:     for each cell do
17:         Select pairs from cross_reference_list;
18:         binary collisions:
19:     end for
20:     dsmc_sampling();
21: end for
```

Algorithm 4 New data structure dsmc code

```
Init: cells,particles,particles-new-structure,sampling variables
for Every time step do
    #pragma omp parallel for ... collapse(3)
    for each cell do
        for each particle in the cell do
            step;
        end for
    end for
    #pragma omp parallel for ... collapse(3)
    for each cell do
        for each particle in the cell do
            bc;
        end for
    end for
    dsmc_mpi_omp();
    dsmc_indexing_omp();
    #pragma omp parallel for ... collapse(3)
    for each cell do
        for particles in the cell do
            binary collisions;
        end for
    end for
    dsmc_sampling_omp();
end for
```

Algorithm 5 *dsmc_mpi_omp*

```
1: Allocate memory for the incoming particles.
2: #pragma omp parallel for ... collapse(3)
3: for each cell do
4:     Set cell location;
5:     for each particle in the cell do
6:         Check if the particle is in the process domain;
7:         if particle is in a ghost grid cell owned by other process then
8:             fill $toright$ or $toleft$ buffer
9:             #pragma omp critical
10:            counter how many particles $toleft$ or $toright$;
11:        else
12:            particle remains in the process;
13:        end if
14:    end for
15: end for
16: $MPI\_Sendrecv$ communication how many particles will receive other process;
17: $MPI\_Sendrecv$ communication for the particles buffers;
18: for each incoming particle: fromleft or fromright do
19:     Set the correct cell position for the particles;
20:     Update the number of particles of those cells;
21: end for
```

Algorithm 6 *dsmc_indexing_omp*

```
1: #pragma omp parallel for ... collapse(3)
2: for each cell do
3:     Set cell location;
4:     for each particle in the cell do
5:         Check the cell position;
6:         if cell position is not the correct then
7:             fill $dsmc\_sparta\_new$
8:             #pragma omp critical
9:             update the number of particles in the cell
10:        else
11:            particle is consolidated in $dsmc\_sparta$
12:        end if
13:    end for
14: end for
15: #pragma omp parallel for ... collapse(3)
16: for each cell do
17:     copy consolidated particles to $dsmc\_sparta\_new$
18:     Update the number of particles
19: end for
20: swap $dsmc\_sparta\_new$ and $dsmc\_sparta$
```

3.2 Performance results

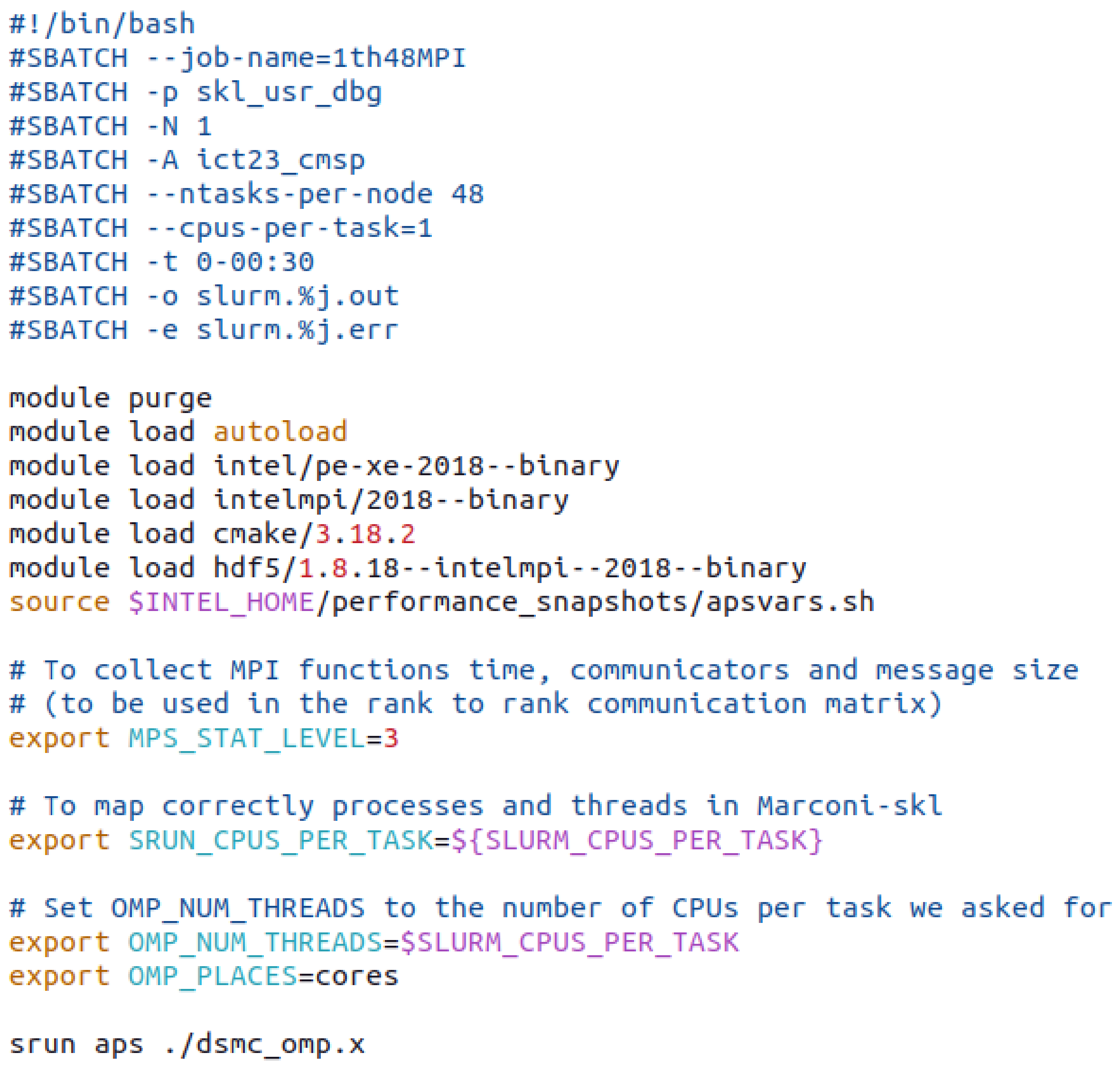
In this section we show the performance results corresponding to a Poiseuille flow described by the parameters of table 3.1 in which we report the values corresponding to Argon gas for the Hard Sphere model (HS) [2].

Parameter	Nomenclature	Value
Number of cells	sxxsyxsz	73728
Number of subcells	subellsx x subcellsy x subcellsz	1
Boundaries	$x_m = y_m = z_m$, $x_p = y_p = z_p$	0, 1.0000
Number of particles	dsmc_molecules_number_total	1769472
Number density	dsmc_number_density	2×10^{20}
Temperature	dsmc_temperature	300 K
Molecular mass	dsmc_mass	6.63×10^{-26} kg
Viscosity index (HS model)	ω	0.81
Molecular diameter	dsmc_diameter	4.09×10^{-10}m

Table 3.1: Parameters of the simulation

In order to measure the benefits of the new data structure and code optimization we use the Application Performance Snapshot (APS) tool developed by Intel VTune [7]. The application allows us to perform analysis for share memory codes and identify critical areas impacting the code performance, such as MPI and OpenMP usage, CPU utilization, memory access efficiency, etc. On Marconi it is possible to use the APS report tool generating a directory that contains the necessary files to be analysed in a post process step. In figure 3.5 we show a script example on Marconi for submitting a job script for the hybrid code.

Once the job is finished we will have a directory with the nomenclature: aps_result_date from where we can extract the data in a post process step, asking for an interactive session and running the command: aps –report=aps_result_date. The result will be a .html file that we can open in a browser (see figure 3.6).

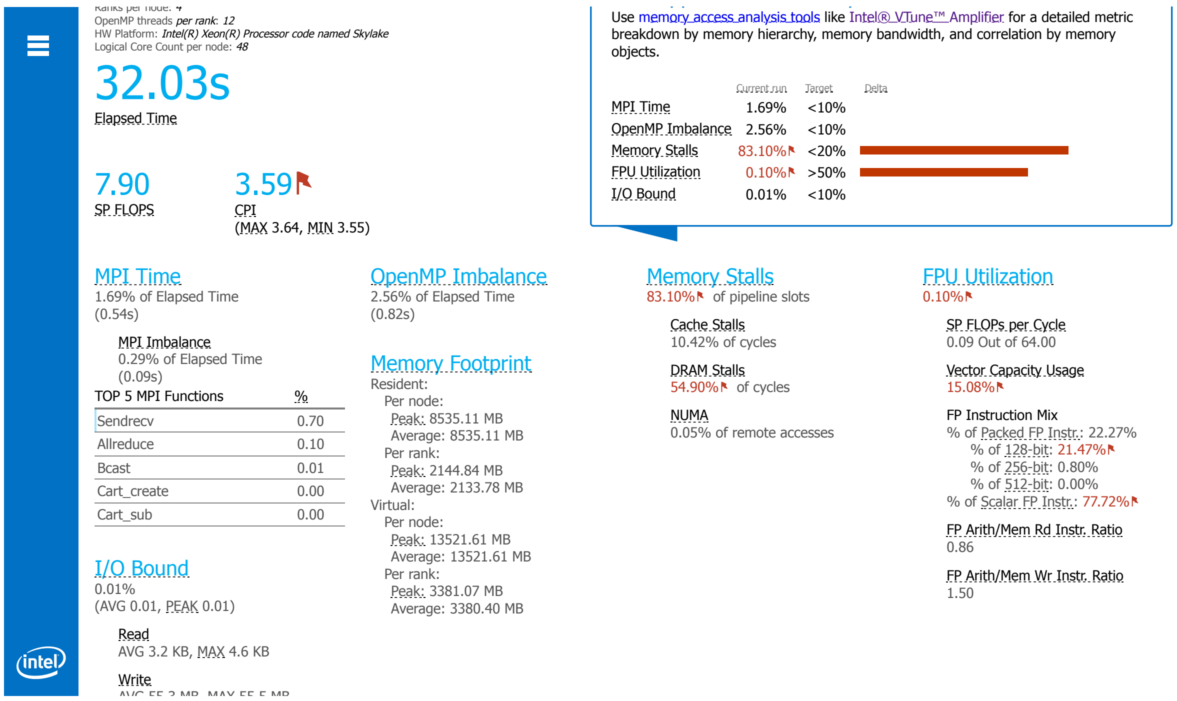


```
#!/bin/bash
#SBATCH --job-name=1th48MPI
#SBATCH -p skl_usr_dbg
#SBATCH -N 1
#SBATCH -A ict23_cmsp
#SBATCH --ntasks-per-node 48
#SBATCH --cpus-per-task=1
#SBATCH -t 0-00:30
#SBATCH -o slurm.%j.out
#SBATCH -e slurm.%j.err

module purge
module load autoload
module load intel/pe-xe-2018--binary
module load intelmpi/2018--binary
module load cmake/3.18.2
module load hdf5/1.8.18--intelmpi--2018--binary
source $INTEL_HOME/performance_snapshots/apsvars.sh

# To collect MPI functions time, communicators and message size
# (to be used in the rank to rank communication matrix)
export MPS_STAT_LEVEL=3

# To map correctly processes and threads in Marconi-skl
export SRUN_CPUS_PER_TASK=${SLURM_CPUS_PER_TASK}

# Set OMP_NUM_THREADS to the number of CPUs per task we asked for
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export OMP_PLACES=cores

srun aps ./dsmc_omp.x
```

Figure 3.5: Script example to generate the aps report directory in Marconi-skl.

We focus our attention on the elapsed and top MPI functions time of both versions: original vs hybrid code. In figure 3.7 we see the performance results in terms of speedup for the original vs hybrid code, we see that the best configurations, i.e. those that offer a speedup higher or similar the orginal version, in terms of number of MPIs tasks and OMP threads corresponds to those with 6 or 12 threads. In the particular case of 4 nodes (i.e. 192 cores in total) we see that the configuration with 4 nodes is the one that offers a better performance.

Figure 3.8 shows that besides we do not have a big benefit in total time of the hybrid version compared to the original one, the fact that we reduce the number of tasks replacing



Ranks per node: 4
OpenMP threads *per rank*: *12*
HW Platform: *Intel(R) Xeon(R) Processor code named Skylake*
Logical Core Count per node: *48*

32.03s
Elapsed Time

7.90
SP FLOPS

3.59
CPI
(MAX 3.64, MIN 3.55)

Use memory access analysis tools like Intel® VTune™ Amplifier for a detailed metric breakdown by memory hierarchy, memory bandwidth, and correlation by memory objects.

	Current run	Target	Delta
MPI Time	1.69%	<10%	
OpenMP Imbalance	2.56%	<10%	
Memory Stalls	83.10%	<20%	🟥
FPU Utilization	0.10%	>50%	🟥
I/O Bound	0.01%	<10%	

MPI Time
1.69% of Elapsed Time
(0.54s)

MPI Imbalance
0.29% of Elapsed Time
(0.09s)

TOP 5 MPI Functions	%
Sendrecv	0.70
Allreduce	0.10
Bcast	0.01
Cart_create	0.00
Cart_sub	0.00

I/O Bound
0.01%
(AVG 0.01, PEAK 0.01)

Read
AVG 3.2 KB, MAX 4.6 KB

Write
AVG 55.3 MB, MAX 55.5 MB

OpenMP Imbalance
2.56% of Elapsed Time
(0.82s)

Memory Footprint
Resident:
Per node:
Peak: 8535.11 MB
Average: 8535.11 MB
Per rank:
Peak: 2144.84 MB
Average: 2133.78 MB
Virtual:
Per node:
Peak: 13521.61 MB
Average: 13521.61 MB
Per rank:
Peak: 3381.07 MB
Average: 3380.40 MB

Memory Stalls
83.10% of pipeline slots

Cache Stalls
10.42% of cycles

DRAM Stalls
54.90% of cycles

NUMA
0.05% of remote accesses

FPU Utilization
0.10%

SP FLOPs per Cycle
0.09 Out of 64.00

Vector Capacity Usage
15.08%

FP Instruction Mix
% of Packed FP Instr.: 22.27%
% of 128-bit: 21.47%
% of 256-bit: 0.80%
% of 512-bit: 0.00%
% of Scalar FP Instr.: 77.72%

FP Arith/Mem Rd Instr. Ratio
0.86

FP Arith/Mem Wr Instr. Ratio
1.50

Figure 3.6: APS report graphic for the optimized code 1 node (48 cores) - 4 MPI tasks and 12 OMP threads.

them by OMP threads impacts in a positive way the reduction in communication time. The main reason that explains the lack of a big benefit in total time of the hybrid version is the fact that the *dsmc_indexing_omp* function contains a critical call that is needed to correctly update the number of particles in each cell, resulting in a slow down when increasing the number of threads.

In figure 3.9 we see in more detail the most relevant MPI functions and the reduction in communication time for the hybrid version. In most of the cases we see that the best configuration in terms of number of MPI tasks and OMP threads corresponds to the case in which the number of threads is set to 6 (as we concluded before from Fig. 3.7). The most expensive MPI calls correspond to MPI_Allreduce and MPI_Sendrecv, but we see the effect when increasing the number of tasks of the MPI_Bcast, such effect is reduced in the hybrid version of the code.

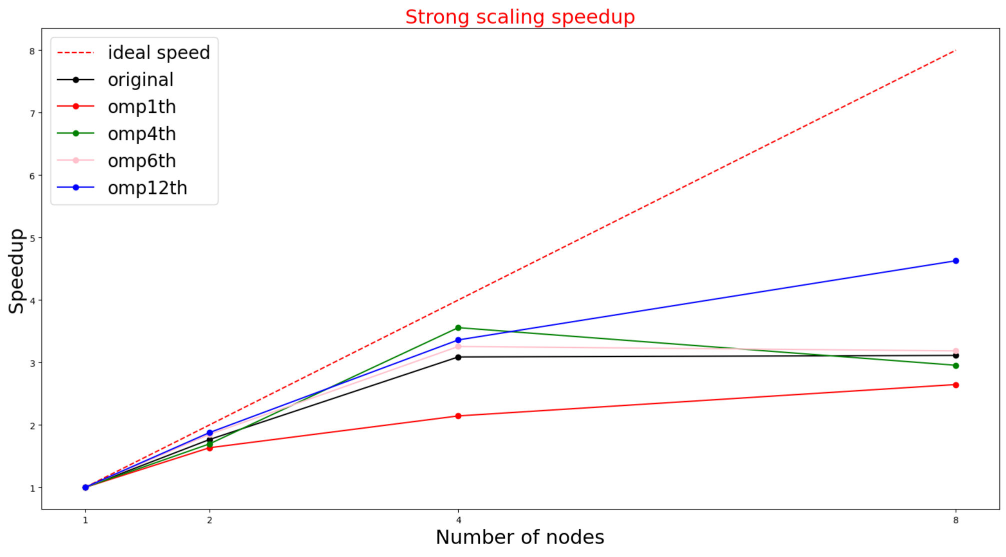


Figure 3.7: Speedup for original vs hybrid (denoted by omp1th, omp4th, omp6th and omp12th) code in Marconi for 1, 2, 4 and 8 nodes. The hybrid version corresponds to the use of 1, 4, 6 and 12 threads. Parameters defined on table 3.1, for a total number of steps=1000.

We analyze in more details the communication time between ranks with the communication matrix, which can be generated to report either the communication time between ranks (in seconds) or the amount of data transferred in the communication (in megabytes). Since the Intel APS version in Marconi-skl does not support the graphical representation .html file of the rank to rank communication matrix (the intel aps version is too old), for the post processing I used the Galileo 100 Cineca cluster [4]. To generate such a plot in a post process step, once you have the directory aps_result_date, you can use the command: aps-report -D -x –format=html aps_result_date that will result in .html file like the ones shown in figure 3.10

In figure 3.11 we collect the average and maximum times reported in the rank-to-rank communication matrix for each of the configurations and again we see the positive effects in both times of the reduction in number of tasks for the hybrid code.

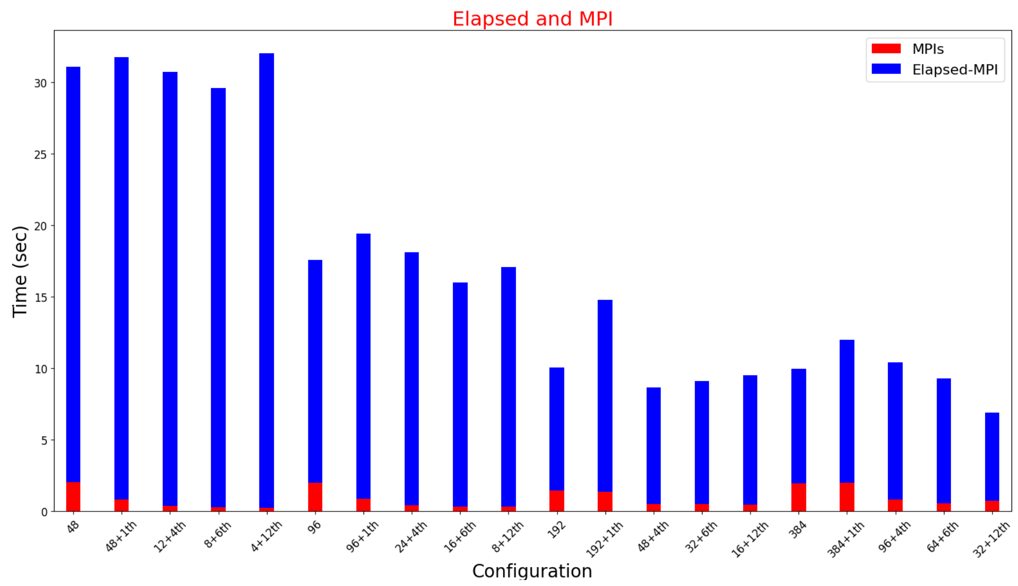


Figure 3.8: Elapsed and time spent inside the MPI library of the total simulation using aps-report profiling intel tool for original vs hybrid code in Marconi for 1, 2, 4 and 8 nodes. The stacked plots for the original code are those with 48, 96, 192 and 384 MPIs. The stacked plots for the hybrid version are those combining number of MPIs and OMP threads (for example 48 + 1th, 8 + 6th, etc). The parameters are defined on table 3.1, for a total number of steps=1000.

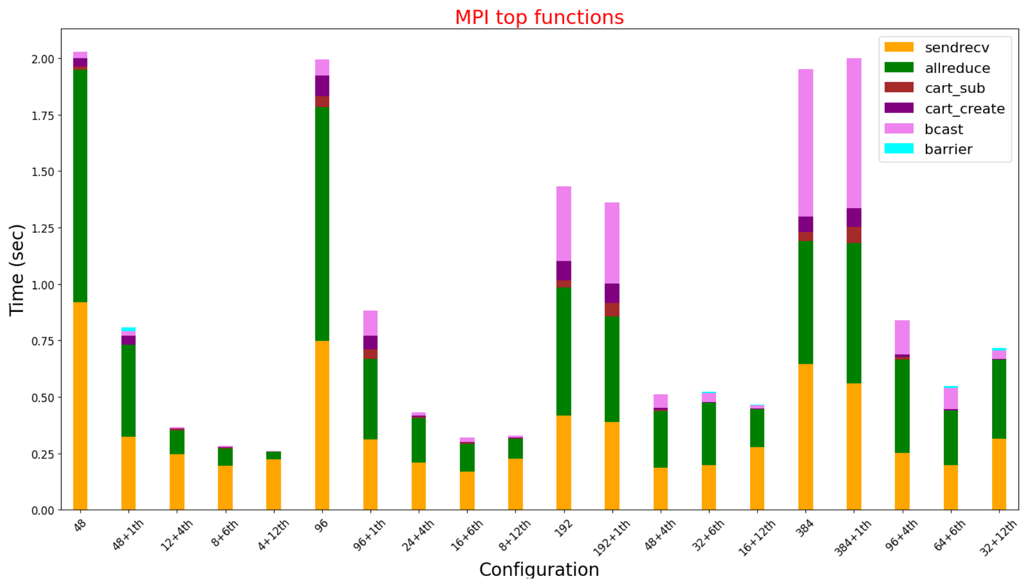
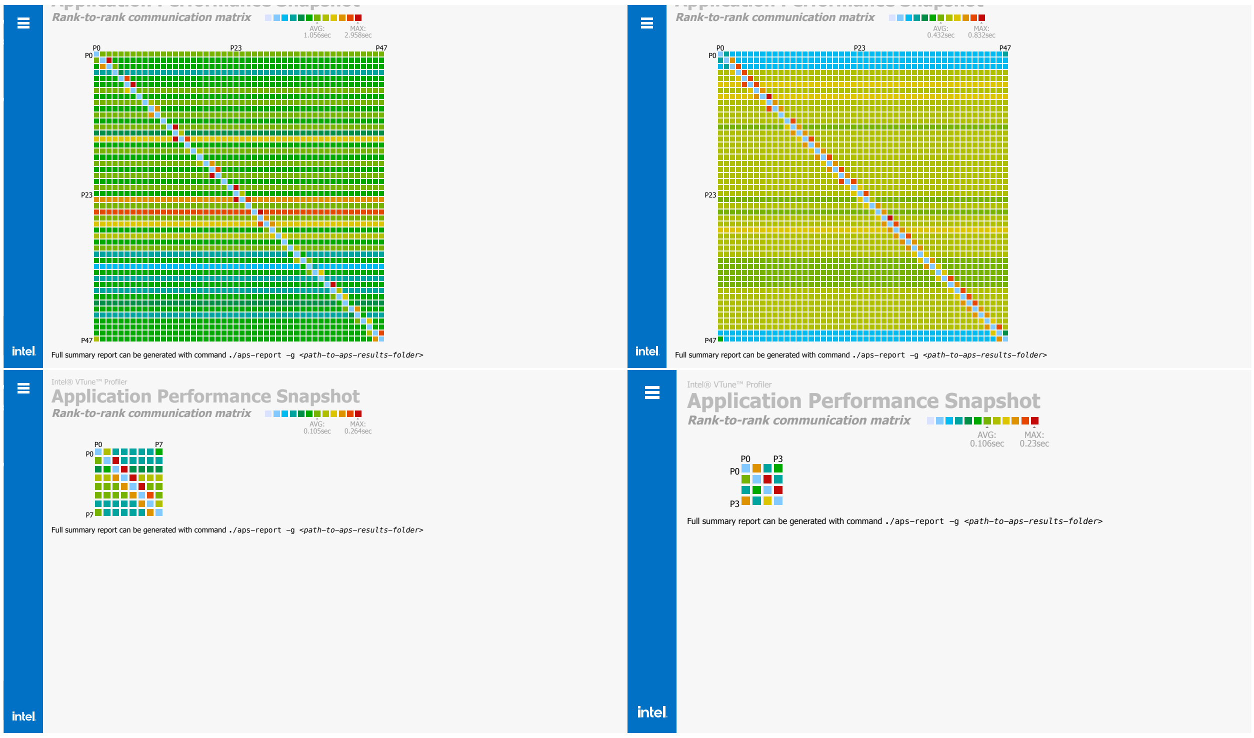


Figure 3.9: Top MPI functions using aps-report profiling intel tool for original vs hybrid code in Marconi for 1, 2, 4 and 8 nodes. The stacked plots for the original code are those with 48, 96, 192 and 384 MPIs. The stacked plots for the hybrid version are those combining number of MPIs and OMP threads (for example 48 + 1th, 8 + 6th, etc). Parameters defined on table 3.1, for a total number of steps=1000.



Rank-to-rank communication matrix

AVG: 1.056sec MAX: 2.958sec

Full summary report can be generated with command `./aps-report -g <path-to-aps-results-folder>`

Rank-to-rank communication matrix

AVG: 0.432sec MAX: 0.832sec

Full summary report can be generated with command `./aps-report -g <path-to-aps-results-folder>`

Intel® VTune™ Profiler

Application Performance Snapshot

Rank-to-rank communication matrix

AVG: 0.105sec MAX: 0.264sec

Full summary report can be generated with command `./aps-report -g <path-to-aps-results-folder>`

Intel® VTune™ Profiler

Application Performance Snapshot

Rank-to-rank communication matrix

AVG: 0.106sec MAX: 0.23sec

Full summary report can be generated with command `./aps-report -g <path-to-aps-results-folder>`

Figure 3.10: Rank-to-rank communication matrix for 1node Marconi-skl original code. Top left original code 48MPI vs hybrid code (top right: 48 MPI + 1 thread, bottom left: 8 MPI + 6 threads and bottom right: 4 MPI + 12threads)

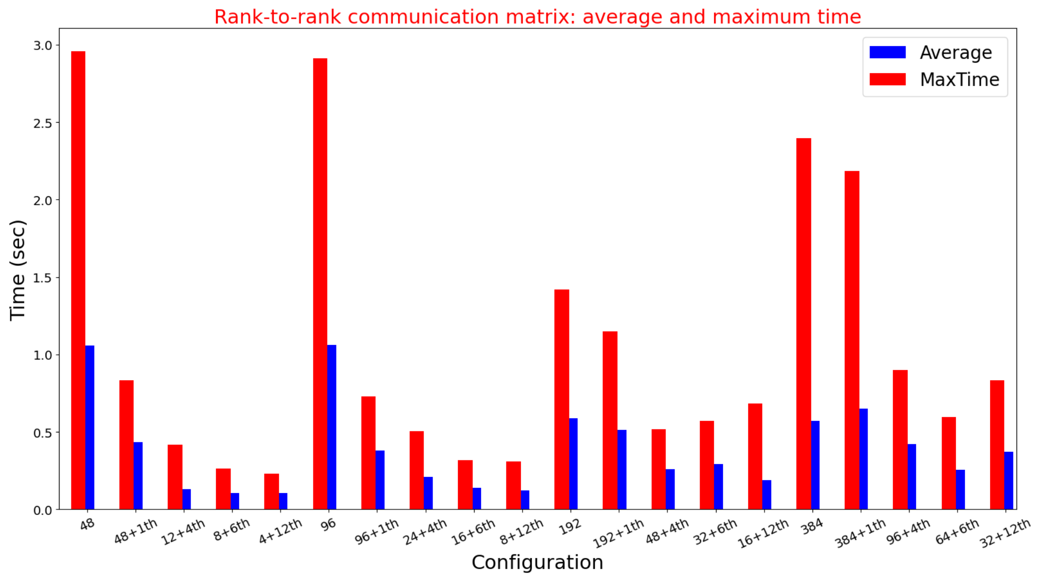


Figure 3.11: Average and maximum time in the communication between two ranks (time collected from the rank-to-rank communication matrix). The stacked plots for the original code are those with 48, 96, 192 and 384 MPIs. The stacked plots for the hybrid version are those combining number of MPIs and OMP threads (for example 48 + 1th, 8 + 6th, etc).

Chapter 4

Conclusions

In this project we investigated a DSMC application and it's performance for two kind of approaches: pure MPI and hybrid MPI + OpenMP. We have found that the introduction of another level of parallelization with OpenMP threads improves the load balancing problems impacting the communication part of the code, but with a no big benefit in total time of the simulation due to threads overhead in one of the most relevant kernels: the one that indexes particles through the cells.

We start from the benchmark of the MPI code describing a flow with open boundary conditions, that has an implicit load imbalance nature due to the gradient in the number of particles in the flow direction. The analysis of the log file allows us to compute the process with the highest and lowest number of particles. We found that the process with the highest number of particles expends most of the time in computation, while process with the lowest number of particles expends more time in communication kernels. We show in numbers the imbalance issues of the MPI code.

After identifying the main bottlenecks of the MPI code, we proceed to introduce another level of parallelization with OpenMP threads using as a model case the Poiseuille flow and changing the data structure of the original code in such a way to parallelize over cells and not over particles. The strong scaling analysis on Marconi-skl CINECA's cluster with an intel tool called Application Performance Snapshot (APS) for a flow with 1769472 particles and 73728 cells allows us to study the best configuration in terms of number of MPI processes and OpenMP threads that scales similar or better than the pure MPI version. We found that for the case of 8 nodes (384 cores) the configuration with 4 tasks per node and 12 threads per task shows a speedup of around 1.5 times better than the pure MPI code.

For future possible studies I suggest applying the hybrid version of the code for a flow with open boundary conditions where the benefit for the load balancing problems should be more evident. An improvement in the kernel that indexes the particles would be necessary to overcome the slow down due to the critical section that is present in order to correctly update the number of particles in the cell.

Bibliography

[1] G. A. Bird. Approach to Translational Equilibrium in a Rigid Sphere Gas. *The Physics of Fluids*, 6(10):1518–1519, 10 1963.

[2] G.A. Bird. *Molecular Gas Dynamics and the Direct Simulation of Gas Flows*. Number v. 1 in Molecular Gas Dynamics and the Direct Simulation of Gas Flows. Clarendon Press, 1994.

[3] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, and Jeff McDonald. *Parallel programming in OpenMP*. Morgan and Kaufmann, San Diego, CA, 2001.

[4] Cineca. Galileo 100 user guide: https://wiki.u-gov.it/confluence/display/scaius/ug3.3 [G100].

[5] Cineca. Marconi user guide: https://wiki.u-gov.it/confluence/display/scaius/ug3.1 [Marconi-skl partition].

[6] G. Di Staso, H.J.H. Clercx, S. Succi, and F. Toschi. Dsmc–lbm mapping scheme for rarefied and non-rarefied gas flows. *Journal of Computational Science*, 17:357–369, 2016. Discrete Simulation of Fluid Dynamics 2015.

[7] Intel Application Performance Snapshot (APS) linux guide. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide-application-snapshot-linux/2023-0/introducing-application-performance-snapshot.html. Introducing application performance snapshot — Intel Corporation,, 2023. [Version: 2023.0].

[8] International Center for Theoretical Physics (ICTP). Argo user guide: https://argo-doc.ictp.it ICTP, 2020.

[9] S. J. Plimpton, S. G. Moore, A. Borner, A. K. Stagg, T. P. Koehler, J. R. Torczynski, and M. A. Gallis. Direct simulation Monte Carlo on petaflop supercomputers and beyond. *Physics of Fluids*, 31(8):086101, August 2019.

[10] Gianluca Di Staso. *Hybrid discretizations of the Boltzmann equation for the dilute gas flow regime*. PhD thesis, Eindhoven University of Technology, 2018. PhD thesis, Applied Physics.

[11] Top 500 list website. http://www.top500.org. Ornl's exaflop machine frontier keeps top spot, new competitor leonardo breaks the top10 — Copyright 1993-2022 TOP500.org (c), top 500, 2023. [Online; accessed 12-January-2023].