PSE Molecular Dynamics: Sheet 5 Report

Group C: Flavius Schmidt, Luca-Dumitru Drindea, Mara Godeanu

January 31, 2025

Code

- **Pull Request**: `https://github.com/FlamingLeo/MolSim/pull/6`
- **Technical Details**: This is a short description of our main development environments.
  - **Operating Systems**: Ubuntu 24.04 (Home), SUSE Linux Enterprise Server 15 SP6 (CoolMUC-4)
  - **Compiler**: G++ 13.2.0
  - **Compiler Flags (Release)**: See Task 3.
  - **C++ Standard**: C++17
  - **External Libraries**: `libxerces-c-dev libgtest-dev libspdlog-dev libxsd-dev`
  - **Build Tools**: CMake 3.28.3, GNU Make 4.2.1

Report

Task 1: ”Simulation of a membrane”

- **Usage**: `build/src/MolSim input/input-lj-w5t1.xml`
- **Basic Idea**: We tried to implement the special membrane case with as few disruptions to the simulation environment as possible. We did not create an extra membrane class, but rather simply extended the `Particle` class with a few attributes: stiffness `k`, average bond length `r0`, the upwards force `fzup`, and the vectors `direct_neighbours` and `diagonal_neighbours`. The latter two store references to the direct and diagonal neighbours for each particle for the special membrane force calculations. In the beginning, if the input file contains the membrane component, the particles have their neighbour vectors initialized and their membrane attributes set. We also created a different force function `calculateF_Membrane_LC`, in order to preserve the efficiency of the main force function. The new function calculates the special membrane interactions.
- **Initialization**: The initialization procedure is similar to normal Cuboid initialization. First, `initialize()` gets called normally on the Cuboid. If `fzup` is not zero, it implies a membrane simulation and the particles are initialized with `type = 5`[1]. Unlike standard simulations, we do *not* add the Brownian motion[2]. Additionally, `specialCase()` is called to determine which particles should have the special `fzup` force applied and which don't. The coordinates of the "special" particles are given through the XML input. Immediately afterwards, `XMLReader` calls `initializeNeighbours()` for membrane input files.
- **Neighbours**: `initializeNeighbours()` works very simply. Because the function is called immediately after initialization, we know the start index of the Cuboid particles in the `particles` vector and can do simple addition and subtraction from an index to find out the neighbours. For example, in order to find the above neighbour's reference, we add `cuboid_dimensions[0]` to the particle's index and use this index to look into `particles`. After all neighbours of a particle are collected, they are stored into the `direct_neighbours` and `diagonal_neighbours` attributes of said particle. Naturally, we are careful not to add nonexistent neighbours for the particles at the edge of the membrane.

[1] We don't actually use this type, but it would be required in order to enable mixed membrane - normal object simulations. This was however not required in the worksheet. Also, the `type` attribute could be replaced with an `enum`, but we kept it as an `int` for backward compatibility and to avoid having to rewrite a *lot* of functions.

[2] The velocity given by the Maxwell-Boltzmann distribution was overpowering the `fzup` force and the membrane was falling to the ground, not building a "tent" as intended. Still, it was behaving like a membrane, "folding" unto itself, not disintegrating.

- **Forces**: Because we didn't want to include a lot of inefficient if statements in our main force calculation function `calculateF_LennardJones_LC()` *just* for this one case, we wrote a new function `calculateF_Membrane_LC`. This is automatically set as the simulation forces function if the XML input features a membrane. It works exactly like the standard function except for 3 points:
  - The special `fzup` force is added to particles.
  - For two particles, we first check if they are neighbours (through the neighbour vectors) in order to apply the special neighbour forces (as detailed in the worksheet).
  - If the particles are not neighbours, we check whether the distance is smaller than the `specialCutoff` defined for membranes. If yes, the repulsive Lennard-Jones force is applied.

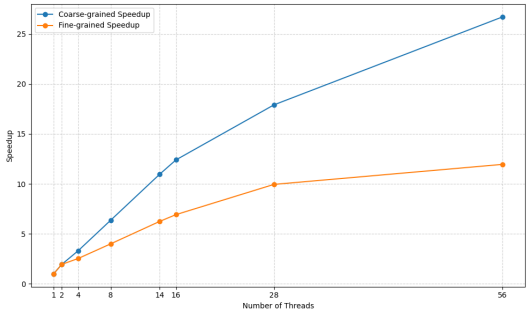
Task 2: "Parallelization"

- **Basic Idea**: The position, force and velocity calculation routines have been parallelized using OpenMP. The force calculation was especially time-consuming before parallelization, so that's where we chose to apply two different strategies.
- **Strategies**: The user can select two parallelization strategies for the linked-cell force calculation function, `coarse` and `fine`.
  - **Coarse-Grained (Standard)**: The first strategy simply uses OpenMP's standard for-loop parallelization directives (`#pragma omp parallel for`). Here, OpenMP divides the iterations of the loop into chunks and assigns each chunk to a different thread in the thread pool. As an aside, we tried using different scheduling strategies and found that `dynamic` with a chunk size of 16 yielded the best results, albeit only very marginally compared to the default scheduling strategy.
    * Parallelization occurs at the *cell* level, with each thread processing large chunks of work independently.
    * **Pros**: Simplicity, lower overhead, more uniform workload in a homogeneously distributed particle system.
    * **Cons**: Higher potential for load imbalance (especially in inhomogeneous systems).
  - **Fine-Grained (Task-Based)**: The second strategy makes use of OpenMP's task system. The algorithm starts creates a pool of threads ready to execute tasks. Inside the parallel region, the `#pragma omp single` directive ensures that only one thread creates the tasks for processing cells. For each computational cell, a task is created. Once a task is created, any available thread in the thread pool can execute it. Threads dynamically pick tasks as they complete their current work, which balances the load across threads.
    * Parallelization occurs at the *task* level, with dynamic distribution of smaller work units for better load balancing.
    * **Pros**: Better dynamic load balancing among threads which may terminate quicker.
    * **Cons**: Worse performance in homogeneous systems, more overhead.
- **Choosing the Strategy**: The parallelization strategy can either be chosen using the `<parallelization>` tag in the XML's `<args>` object or using the `-p` option. The argument must be `coarse` or `fine`.
- **No OpenMP? No Problem!** We wrapped our OpenMP integration inside `utils/OMPWrapper.h`, which either includes `<omp.h>` if it is found (and linked correctly) or defines standard / placeholder functions instead of OpenMP's preexisting ones. This minimizes the amount of `#ifdef`s needed and allows the program to still run, even if OpenMP is not available. Additionally, CMake checks for OpenMP support during compilation using `find_package(OpenMP)` and applies the correct flag based on the compiler used.
  - You may also disable OpenMP intentionally using `-DENABLE_OPENMP=OFF` in CMake (or by passing `-o` to the build script).
- **Pitfalls...?**: Thankfully, because we wrote our functions in an already easily-parallelizable way, we barely ran into any pitfalls. Only certain parts had to be manually synchronised using locks to prevent data races and / or corruptions (moving particles from one cell to another, updating the force using N3L). We didn't make the functions thread-safe inside the `Particle` and `CellContainer` classes directly and instead chose to only make the position, force and velocity calculations thread-safe, because in almost all cases, the threads would not access the same data concurrently. This avoids needless locking and unlocking.
- **Speedup**: Figures 1a and 1b show the speedup and execution times in milliseconds of our program using both parallelization strategies for various thread counts.

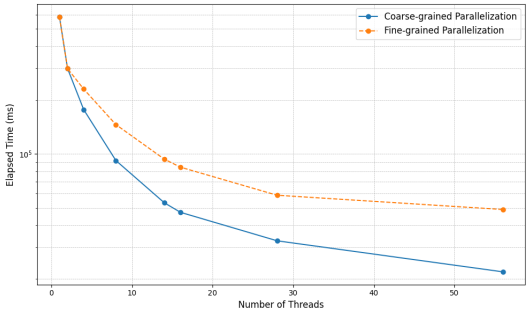
- **Environment**: CoolMUC-4, `cm4tiny`, compiled with G++ 13.2.0, `Release` build.
- **Method**: The `OMP_DYNAMIC` variable was set to `false` to ensure consistent thread counts.
- **Input**: We used the contest environment for Task 3 as our input (3D, 100k particles, 1000 iterations).
- **Speedup**: The Speedup is calculated using the formula

$$S(n) = \frac{T(1)}{T(n)},$$

  where $T(n)$ is the execution time for n threads. Table 1 shows the obtained speedup and times.
- **Results**: The coarse-grained parallelization strategy performs better for the initial 1000 iterations of the simulation. We notice a (mostly) steady increase in performance, but after about 16 threads, the overhead starts to become noticeable.



(a) The speedup based on the number of threads, using 1 thread as a base.



(b) The raw execution time in milliseconds plotted against the number of threads.

Figure 1: Graphs showing speedup and absolute times for a varying number of threads using a 3-dimensional input of 100.000 particles for coarse-grained and fine-grained parallelization.

Threads	Runtime (s) (c)	Speedup (c)	Runtime (s) (f)	Speedup (f)
1	584,959	1,0000	584,390	1,0000
2	300,650	1,9456	300,187	1,9468
4	177,118	3,3027	230,640	2,5338
8	91,854	6,3684	145,944	4,0042
14	53,343	10,9659	93,524	6,2486
16	47,164	12,4027	84,361	6,9273
28	32,672	17,9039	58,741	9,9486
56	21,908	26,7007	48,899	11,9509

Table 1: The runtime in seconds and the speedup for different thread counts using coarse-grained and fine-grained parallelization.

- **Profiling**: Finding a good, working profiling tool for OpenMP applications was surprisingly tedious. Intel's VTune didn't work, since we used a computer with an AMD processor. MAQAO didn't quite do everything we wanted, and by the time we figured out how to get PAPI up and running, we were already looking for alternatives. In the end, we used *AMD uProf*, which analyzes different program aspects (such as function calls, threading, cache-misses, etc.), including OpenMP parallelization. Figures 2 through 7 show the collected data.
  - **Overall**: Almost 80% of the total time taken is parallel execution time, with the remaining 20% presumably the unavoidable XML input and simulation initialization, among other things. There is also little to no thread overhead, with almost 98% of CPU time being used up for actual work.
  - **CPI Spikes**: Occasionally, the parallel program will be anywhere from 50% to 150% slower than usual. This corresponds to spikes in the CPI profiling metric when analyzing with uProf. According to the uProf manual, "Cycles Per Instruction (CPI) is the multiplicative inverse of Instructions Per Cycle (IPC) metric. This is one of the basic performance metrics indicating how cache misses, branch mispredictions, memory latencies, and other bottlenecks are affecting the execution of an application. A lower CPI value is better." Looking deeper into the code, the most frequent function called in such a situation is an unknown function in `libgomp`, the GNU Offloading and Multi Processing Runtime Library. It appears

to be some sort of busy-waiting function, which either jumps back to the top of the loop or outside of it, if the comparisons match. Since we couldn't consistently reproduce these slowdowns, we assume this bottleneck is not caused by our program.

Profile Time	3.6090s
Time Inside Parallel Region	2.8274s
Time Outside Parallel Region	0.7816s
Parallel Time %	78.3429
Total Threads Created	12

▾ CPU Time	32.324s
Work Time	31.234s
▾ Overhead Time	1.090s
Other Overhead Time	1.090s
Idle Time	1.569s

Figure 2: Summary of the collected OpenMP statistics for 12 threads in an ideal scenario, with roughly 78% parallel time and 97% effective work time.

Region	Avg Idle Time (secs)	Avg Sync Time (secs)	Avg Overhead Time (secs)	Avg Work Time (secs)	Total Elapsed Time (secs)	Thread Count	Loop Chunk Size	Schedule Type
calculateVompparallel_for:12@VelocityCalculation.cpp:9	0.0465	0.0000	0.0178	0.4384	0.5027	12	1	Dynamic
calculateF_LennardJones_LCompparallel_for:12@ForceCalculati	0.0375	0.0000	0.0565	1.7168	1.8108	12	1	Dynamic
calculateX_LCompparallel_for:12@PositionCalculation.cpp:29	0.0468	0.0000	0.0165	0.4476	0.5109	12	1	Dynamic

Thread No.	Thread Id	Total Idle Time (secs) ▼	Total Sync Time (secs)	Total Overhead Time (secs)	Total Work Time (secs)
0	137630	0.0473	0.0000	0.0171	0.4384
1	137634	0.0467	0.0000	0.0175	0.4385
2	137635	0.0468	0.0000	0.0181	0.4378
3	137636	0.0467	0.0000	0.0174	0.4386
4	137637	0.0461	0.0000	0.0176	0.4390
5	137638	0.0454	0.0000	0.0181	0.4392
6	137639	0.0470	0.0000	0.0173	0.4383
7	137640	0.0458	0.0000	0.0190	0.4378
8	137641	0.0473	0.0000	0.0175	0.4379
9	137642	0.0460	0.0000	0.0183	0.4385
10	137643	0.0461	0.0000	0.0183	0.4384
11	137644	0.0463	0.0000	0.0178	0.4386

Figure 3: OpenMP statistics for the parallelized regions and each unique thread in an ideal scenario.

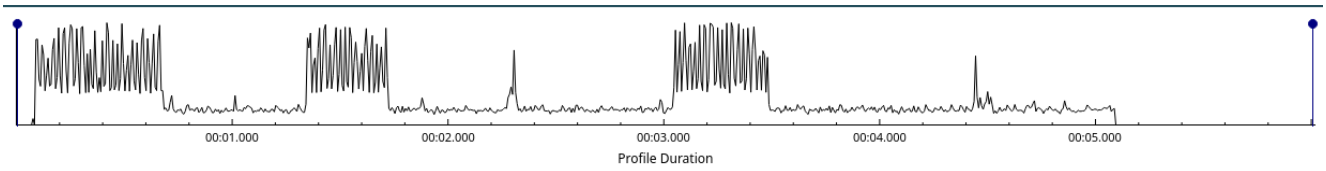


Figure 4: Spikes in the CPI metric which correspond to noticeably slower execution.

Address	Line	Assembly	CPU_TIME(s)
0x258a6		cmpq %rdx, %rax	35.6410%
0x258a9		je 0x258e0	
0x258ab		movl (%r12), %ecx	
0x258af		cmpl %ecx, %ebx	
0x258b1		je 0x258a0	

Figure 5: Disassembly of the most commonly called function in a slow scenario, an unknown function in `libgomp`.

Figure 6: Thead timeline in a slow scenario. The salmon parts represent our own program code, the green parts represent `libgomp` code.

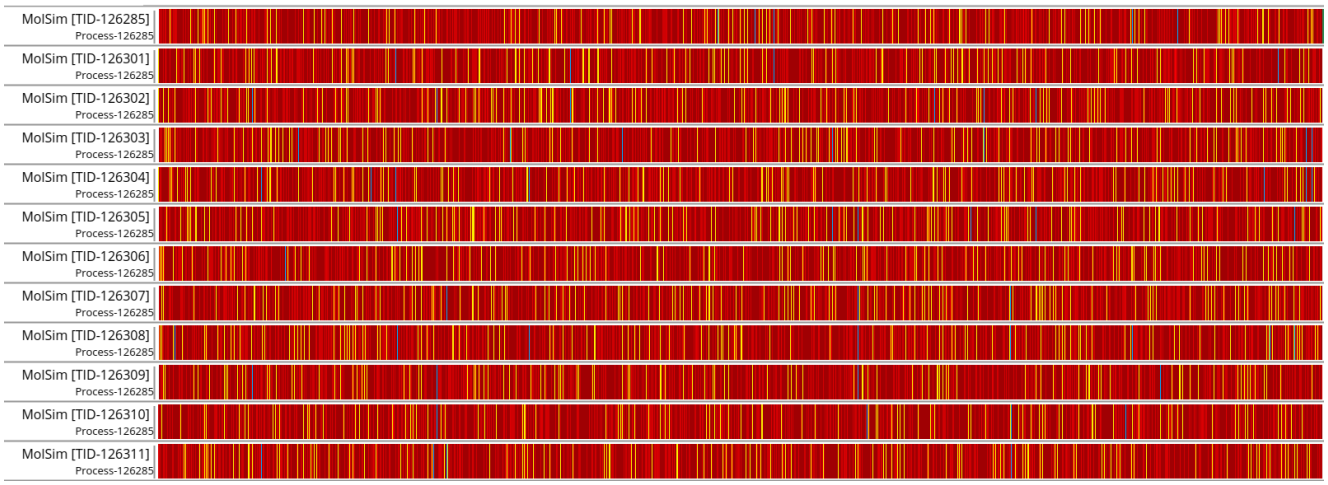


Figure 7: Thead timeline in an ideal scenario. The red parts represent force calculation (with each shade being a different subroutine, e.g. `getTruePos()`), the yellow parts represent position calculation, the blue parts represent velocity calculation. Everything else is not our own code.

Task 3: "Rayleigh-Taylor instability in 3D", Contest 2

- **Usage (coarse)**: `build/src/MolSim input/input-lj-w5t3-coarse.xml`
- **Usage (fine)**: `build/src/MolSim input/input-lj-w5t3-fine.xml`
- **Usage (contest)**: `build/src/MolSim -e0.5 input/input-lj-w5t3-coarse.xml`
- **Extending to 3D**: See "Miscellaneous" below, as this part is specific to the entire worksheet (and simulation framework as a whole).
- **Further Optimizations**: As mentioned in the previous worksheet, there were more optimizations that we could've attempted, but didn't, either due to a lack of time or lack of experience. We falsely assumed that the compiler would do most optimizations, such as inlining or small algorithmic optimizations, leaving little room for improvements, but as it turns out, that's not always the case. Don't trust everything you read on the internet.
  - **Base**: We mainly tested the improvements locally on the usual machine, not on the Linux cluster. This is why the improvements will be given in a relative way, using percentages. For reference, our base implementation using the 2D experiment of Contest 1, running on 16 threads, had a runtime of about 1.8 seconds, before the optimizations mentioned below.
  - **Environment**[3]: Asus ZenBook 13, Ubuntu 24.04 LTS, Linux kernel version 6.8, 16 GB RAM, AMD Ryzen 7 5800U (8 cores, 16 threads) @ 4.51 GHz, G++ 13.3.0 with -O3 (Release build) and automatic CPU frequency scaling disabled.
  - **Function Inlining**: The getters inside the `Particle` class took up a significant portion of time due to the underlying `call` instructions. Inlining them already gave us a 15% boost.
  - **Periodic Boundary Checks**: Previously, in the force calculation function, in order to check if we needed to mirror ghost particles, we would repeatedly check the boundary condition vector of the `CellContainer` for any instances of `PERIODIC`. We replaced this with a boolean which is initialized once during the creation of the `CellContainer`. The improvement is minimal, but noticeable in the long run.
  - **Link Time Optimization**: We enabled link-time optimization to allow optimization of calls among compilation units, which gave us another 13,3% performance increase.
  - **Compiler Optimizations**: We used more compiler flags alongside `-O3` to further optimize our program.
    - `-march=native, -mtune=native`: Optimizes for the current machine's architecture.
    - `-fdata-sections, -ffunction-sections`: Allows the linker to "perform optimizations to improve locality of reference in the instruction space".
    - `-fno-math-errno`: Does not set `errno` after calling math functions that are executed with a single instruction. This assumes that the functions don't return any errors.
    - `-funroll-loops`: Unrolls loops whose number of iterations can be determined at compile time or upon entry to the loop.

    In total, these increased our performance by about 8%.
  - **Fast Math**: This is where optimizations could potentially become dangerous. Fast (floating-point) math is turned off by default, as it may lead to some numerical inaccuracies. It also doesn't work well with non-finite values, as these are undefined. This means that tests which rely on infinite (or NaN) values *will* fail. However, assuming the input is valid, such caveats are negligible in favor of the gained time, which in this case is about 16%.
  - **Skip Activity Checks**: Another dangerous optimization is the omission of particle activity checks (i.e. `if(!p.isActive())...`). Last week, we tried getting rid of these by deleting inactive particles after the position calculation. However, this *still* didn't work because of... well, C++, causing simulations with outflow boundary conditions to break. So, we reverted this change. However, in order to still be able to get rid of these, if the user *knows* that there are no outflow boundaries, they may now compile with a new option, `-DNO_OUTFLOW=ON` (or `-O` using the build script), which removes these checks at compile time. Attempting to run a simulation with at least one outflow boundary condition will abort the program. This is actually an improvement over our attempt last week, because we also skip having to check if there are any inactive particles after each position update, eliminating *all* activity checks (and giving us our 5% performance increase back).

[3] This is how we tested our improvements - the final contest results were still achieved on CoolMUC-4.

  - **Profile Guided Optimization (PGO)**: Lastly, we integrated PGO into our CMake environment. First, the user must compile the program with PGO instrumentation code using `-DPGO_GENERATE=ON` (or `-p` using the build script). Afterwards, the program must be run once using the desired input to generate profiling data which the compiler may use to better optimize for the hot paths. The profiling data will be in the project's root directory, in the `profiledata` directory. Finally, the program can be compiled again with `-DPGO_USE=ON` (or `-P` using the build script), this time using the generated data. This gave us our final improvement of about 15%. Locally, our best time was about 750ms for the 2D simulation using 16 threads.

- **Final Results (Contest 2)**[4]: Table 2 shows the best results using commit hash `bfcea4e`, tested on CoolMUC-4 (`cm4_tiny` with `cpus-per-task=224`) using the latest available versions of G++ and the Intel C++ compiler. The molecule updates per second are calculated the same way they were for the previous worksheet.
  - **2D Input**: 1000 iterations, 10.000 particles, Worksheet 4 Task 2 Large, coarse parallelization
    - **Usage**: `build/src/MolSim input/input-w4t5-large.xml`
  - **3D Input**: 1000 iterations, 100.000 particles, Worksheet 5 Task 3, coarse parallelization
    - **Usage**: `build/src/MolSim -e0.5 input-lj-w5t3-coarse.xml`
  - **General Build Script Flags**: Fast math, no outflow, benchmarking mode.
    - **Usage**: `./build.sh -c -f -O`
    - **CMake Flags**: `-DENABLE_BENCHMARKING=ON -DNO_OUTFLOW=ON -DENABLE_FAST_MATH=ON`
  - **Profile Guided Optimization**: G++ results were produced with PGO enabled, using the steps mentioned in the previous bullet point.
    1. `./build.sh -c -f -O -p` (CMake: `-DPGO_GENERATE=ON`)
    2. `build/src/MolSim ...` (*generate* profile data for corresponding simulation)
    3. `./build.sh -c -f -O -P` (CMake: `-DPGO_USE=ON`)
    4. `build/src/MolSim ...` (*use* profile data for corresponding simulation)

Input	Compiler	Runtime (s)	MUPS/s	Notes
2D	**g++ 13.2.0**	**0,479**	**20.876.826,72**	**112 Threads, PGO enabled**
2D	icpx 2023.2.1	0,570	17.543.859,65	112 Threads
3D	**g++ 13.2.0**	**11,533**	**8.670.770,83**	**224 Threads, PGO enabled**
3D	icpx 2023.2.1	12,138	8.238.589,55	150 Threads

Table 2: The fastest times and MUPS/s for both simulations, tested on CoolMUC-4. The **bold** results show the best times for the 2D and 3D inputs respectively.

[4] See "Miscellaneous" section below for how we got ICPX up and running.

Task 4: ”Nano-scale flow simulation (Option A)”

- **Usage (normal)**:`build/src/MolSim input/input-lj-w5t4-normal.xml`
- **Usage (increased gravity)**: `build/src/MolSim input/input-lj-w5t4-increased_gravity.xml`
- **Usage (inner wall)**: `build/src/MolSim input/input-lj-w5t4-inner_wall.xml`
- **Usage (no walls)**: `build/src/MolSim input/input-lj-w5t4-no_walls.xml`
- **Thermostat**: For this task, it was necessary to extend the thermostat from the previous worksheet to support further functionality to accommodate the needs of this task.
  - First, we had to make use of the `type` attribute of the `Particle` object with which we differentiated between wall and fluid particles (1 for wall and 0 for fluid).
  - After that, we needed to also update the forces and velocity methods implementations in order to not update these values for the particles that are part of the walls.
  - For the thermostat itself, we added another attribute called `nanoFlow` which determines whether the thermostat is used for the ”nano-scale flow” simulation or for another simulation. In the case that this attribute is set to true, the thermostat doesn't only take into account the velocity itself of the particles but instead computes its thermal motion by subtracting the average velocity from this velocity. The scaling factor is then applied on this thermal motion after which it is added onto the average velocity. This in the end is the new velocity of the particle. Of course only the particles contained in the fluid are used for the calculation of velocities.
- **Analyzer**: We implemented this task's analyzer in the class `FlowSimulationAnalyzer.cpp` which simply takes as parameter the `ParticleContainer` of the simulation, the number of bins we want to assign the analyzed particles to (`binNumber`), the number of iterations after which the the analyzer should periodically be applied (`n_analyzer`), as well as the x coordinates of the left and right wall (`leftWallPosX` and `rightWallPosX`).
  - The idea of the analyzer is to simply use two vectors for the densities and average velocities of each bin. These vectors are resized with the length of the `binNumber` passed as a parameter in the constructor of the class. The size of the bin, `binSize`, is also calculated here. In the function `calculateDensitiesAndVelocities()` is where the logic of assigning particles to the bins by their specific x coordinates happens. We go through each bin which has a `leftRange` and a `rightRange` and if the position of the particle on the x coordinate is between those ranges, then we increase the counter of the element in the densities vector on the bin position. We do the same thing by just adding the velocities on y coordinate in the velocities vector. At the end we simply divide the sum of velocities by their specific sum of particles. All this information is written in the end in `.csv` files every `n_analyzer` iterations.
- **Results**[5]: By running the simulation with various input, we got the following results:
  - **Normal Simulation**: When running the simulation with the default input we didn't notice anything extraordinary or unusal, only that the densities are somewhat constant and that the velocities gradually decrease on the Y-Axis, especially in the bins around the middle. All in all we got exactly what we paid for, a bland flow of particles which goes down and comes back from above.
  - **Increased Gravity Simulation**: By changing the gravity value from -0.8 to -10, the particle velocities started to gradually increase. From an average base velocity which nears 0, it skyrocketed (groundrocketed?) to a value of around -250 on the y scale.
  - **Inner Cuboid**: There is a noticeable, somewhat constant, spiky pattern in the middle of the density graph, where the inner cuboid is. The velocities seem to remain constant as opposed to the normal simulations, neither increasing nor decreasing on the Y-Axis because of the block that repels particles.
  - **No Walls**: We also ran the simulation with no walls, once with reflective boundaries and once with outflow boundaries.
    - **Reflective**: We notice in the simulation with reflective boundaries that the densities remains constant throughout the bins and that the velocities gradually decrease.
    - **Outflow**: In this simulation, as expected, the particles begin to disappear. As such, the values from the densities bins decrease.

[5] All videos are provided in the submission.

Miscellaneous

- **No Xerces-C? ~~No~~ Big Problem**: Because of how finicky and inconsistent it is to try and fetch Xerces-C via CMake, Xerces-C is now mandatory again for compilation.
  - **Local Machines**: On local machines, you must ensure that Xerces-C is installed and discoverable via CMake. This should be the case when installing using the system's package manager (e.g. `sudo apt-get install libxerces-c-dev` on Debian).
  - **Linux Cluster**: On CoolMUC-4, the `xerces-c` module must be loaded beforehand.
    * The build script will attempt to do this automatically (as in, running `./build.sh` will suffice).
    * Should compilation not work, try loading the module manually using `module load xerces-c`. According to CMake, it should be located here: `/dss/lrzsys/sys/spack/release/23.1.0/opt/x86_64/xerces-c/3.2.1-gcc-fuaugd5/lib/libxerces-c.so`
    * Should compilation still not work, try first loading the module, then running `module load user_spack` and `spack install xerces-c`, then compiling (preferably using the build script).
    * Should compilation *still* not work... we're sorry.
- **ICPX Support**: Because the issues regarding the Intel C++ compiler had to do with locally compiling and linking Xerces-C, ICPX now works, since we require that Xerces-C be installed and available beforehand.
  - **Side Note**: You can now check which compiler was used in the help string of the program (`-h`).
- **Pre-5.0 OpenMP Support**: Before OpenMP 5.0, the `#pragma omp parallel for` did not support range-based for-loops. As such, in order to ensure compatibility with the default CoolMUC-4 compiler (g++ 7.5), which uses OpenMP 4.5, we created a new macro in `OMPWrapper.h` for iterating over containers which conditionally compiles either a standard range-based for loop or a "traditional" for loop. Thus, you can compile the program with the default compiler (but seriously, I think it might be time to update it...).
- **Multithreaded Death Tests**: GTest will warn the user that death tests (tests used to check if the program terminates) are unsafe in a multithreaded context. Using `GTEST_FLAG_SET(death_test_style, "threadsafe")` disables the warning; however, this lead to compilation errors on different machines, so we decided not to use these macros. The tests should still work, given how simple they are.
- **Extending to 3D**: All tasks of this worksheet required 3D simulations, so we first had to extend our simulation environment to support 3D simulations. This was not extremely difficult, as the system was designed to accommodate this change we knew was coming. Importantly, the number of dimensions (2D or 3D) is now a simulation parameter that should be given by the user. It determines how the linked cells are initialized and boundary conditions implemented. Many functions suffered slight modifications not worth mentioning, so the following is a list of the more important changes:
  - `CellContainer`: In the `CellContainer` class the attribute `dim` was added to always store the number of dimensions, required in different functions. Otherwise, most "spatial" functions were modified to support 3D: `calculateNeighbours()`, `getOppositeOfHalo()` etc.. The function `getOppositeOfBorderCorner()`, tasked with finding the cells in which corner cell particles should be duplicated as ghost particles, suffered the most modifications. Any border cell with ≥ 2 border locations is considered a corner cell (e.g. North, East). To us it intuitively seemed that a corner particle should be mirrored across every individual periodic border location it find itself in *and* across every combination of these locations (mathematically, you would build the power set of border locations).
    * **2D**: For 2D this meant at most 3 mirrorings in a fully periodic corner (e.g. for the SE corner we would place ghosts in the North side, West side and NW corner).
    * **3D**: For 3D this can mean as much as 7 mirrorings in a fully periodic corner (e.g for the Below (B) SE corner, ghost are placed in the Above (A), North and West sides and in the AN, AW, NW and ANW corners).

    The combinations are found through the `getBorderCombinations()` function.
  - `BoundaryConditions`: In this class, the reflective boundary conditions were adapted. Namely, if the halo location is Above or Below and the boundary reflective, `handleReflectiveCondition()` now flips the third velocity component and mirrors the particle normally as before.
  - `Cell`: This class suffered the biggest update as it contains the `getCornerRegion()` function, which decides, based on the particle's relative position inside the corner cell, which boundary condition should be applied to it. Namely, it calculates which boundary (North, Above etc.) the particle first hit before ending up in the corner cell. This is done through some basic geometric calculations. Since the basic concepts had already been implemented for 2D (`handle2DCorner()`), they were only extended to the 3D scenario (for more details on 2D see Report 3, 4). This meant a painful exponential increase in handled cases: from 4 to 20. The following subdivision was used:

* **Double Corners**: These are the corners with only 2 halo locations (e.g. NW, BE). For these corners (12 distinct types, grouped into 6 categories sharing a demarcation plane), the 2D concept was simply moved into 3 dimensions. As an example, a Below-East corner cell would have the demarcation plane of fig. 8.

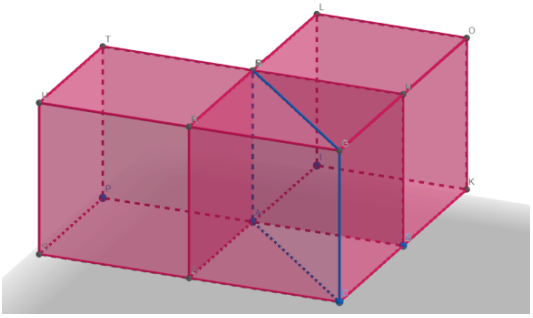


Figure 8: The demarcation plane of a Below-East halo corner cell. If a particle is left of the plane, it must've hit the left cell first (Below) and if it is right of the plane, it must've hit the right cell first (East). *Image generated with GeoGebra*.

* **Triple Corners**: For triple corners we decided to use 2 demarcation planes per corner: one demarcation plane for the vertical direction (did the particle hit north/south or the "sides" first?) and then one for the horizontal direction (which "side" was hit first?). This results in a two-phase decision system: if the first demarcation plane says that the North/South boundary was hit first, then this is the result, otherwise one of the "sides" (e.g. West or Above) must've been hit first and the result is decided as in the double corner scenario. For all of the 8 triple corners in a 3D domain, a different case had to be defined. For example, fig. 9 shows the demarcation planes for the Below-South-East corner.

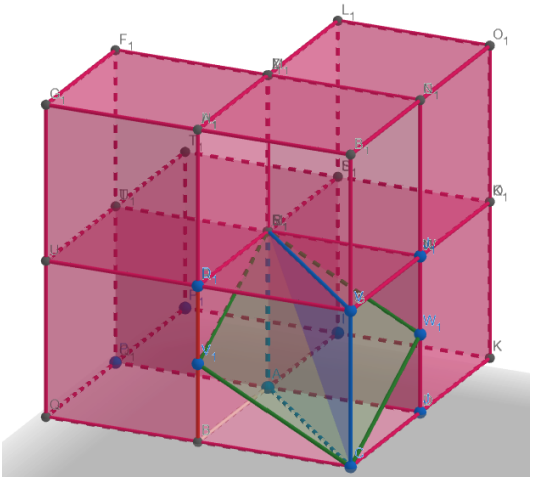


Figure 9: The two demarcation planes of a Below-South-East halo corner cell. In the first phase, if the particle is below the green plane, it must've first hit the bottom cell (South). If it is above the green plane, we calculate whether it's left or right of the blue plane (left - Below, right - East). *Image generated with GeoGebra*.

Structurally, in the case of a 3D simulation, based on the number of halo locations, either `handle3DDiagonal()` or `handle3DTripleCorner()` gets called. Both work similarly, first determining the concerned corner and based on its demarcation planes, the boundary to be enforced. Because all corner cases are treated the same except for the *exact* demarcation plane(s), we defined macros like `RETURN_DIAG_CONDITION` to remove some redundancy and make the code easier to read.

- **Extra Geometry**: For the interested reader, the "vertical" planes for the triple corners were determined the following way:
  - A line was drawn between the "innermost" corner A and the outermost corner B (line AB).
  - Then, from the corner C, that lies above the line (not on it), the perpendicular to AB was drawn. This perpendicular is the the normal vector of the plane.

– The particular dimensions of the cell were used to define the points, so that the plane would be correctly defined for any simulation.

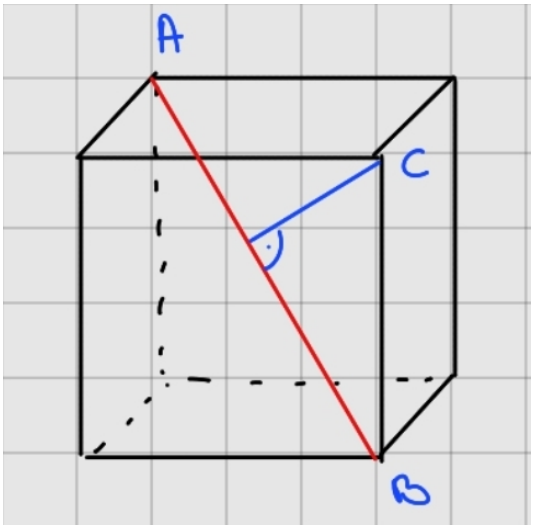


Figure 10: Geometry exemplified on the Bottom-South-East Corner.