PSE Molecular Dynamics: Sheet 4 Report

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

December 20, 2024

Code

- **Pull Request**: `https://github.com/FlamingLeo/MolSim/pull/5`
- **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)
  - **Compilers**: Clang++ 18.1.3 (Home), Various (CoolMUC-4, see Task 4)
  - **Compiler Flags (Release)**: `-Wall -Wextra -Wpedantic -Werror (-O3 -DNDEBUG)`
  - **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: "Thermostats"

- The implementation of the thermostat was in of itself one of the easier tasks of this worksheet since its functionality is specifically given in the worksheet. Corresponding unit tests were also added.
  - **Kinetic energy calculation**: For calculating the kinetic energy E_{kin}, we just have to iterate over all active particles of the simulation and sum the products between the mass m of each particle and the squared euclidean norm of its velocity v, which is equivalent of calculating the dot product of a vector with itself.
  - **Current temperature calculation**: To calculate the temperature T of the system, we only need to divide the doubled kinetic energy we got in the previous step by the number of particles and dimensions of the simulation. Basically, we have derived the temperature from the aforementioned kinetic energy.
  - **Scaling Factor**: The scaling factor β is just the square root of the fraction between the desired new temperature of the system T_{target} and its current temperature T. Of course, if the absolute temperature change is larger than the difference between the current temperature and the target temperature, then the new temperature will simply be set as the target temperature.
  - **Update of the system temperature**: This is the function that is called on every iteration of the simulation. It is given the time step of the simulation and is only applied if this parameter is a multiple of the set number of steps n after which the thermostat is supposed to be applied on the simulation. The functions `calculateKineticEnergy()`, `calculateTemp()` and `calculateScalingFactor()` are called in order to be able to apply the scaling factor in the end on the velocities. This method also initializes the velocities on the first time step with Brownian Motion if the respective boolean parameter is set in the initialization of the thermostat object.
  - **XML**: See the updated `README.md` for how the thermostat was implemented in our XML input files.
  - **NOTE**: Currently, the thermostat can't explicitly be turned off. To remedy this, disable initializing particle velocities with Brownian Motion and set a really high number of iterations n, e.g. `INT_MAX`.

Task 2:"Simulation of the Rayleigh-Taylor instability"

- **Usage (small)**: `build/src/MolSim input/input-lj-w4t2-small.xml`
- **Usage (large)**: `build/src/MolSim input/input-lj-w4t2-large.xml`
- **Periodic Boundary Conditions**: Implementing periodic boundaries was the most challenging part of the task and only achieved in 2D. It consisted of two parts: moving particles that exit on one side of the domain to the other side and "mirroring" the particles in border cells in the opposite halo cells, to allow for the trans-boundary force calculations, necessary for periodic boundaries.
  - **Moving**: For this we just extended the existing framework for handling the particles, that after the new position calculations end up in the halo (the `handleHaloCell()` function).
    * With the `determineBoundaryCondition()` function from worksheet 3 we determine what boundary the particle first hit and implement that boundary's condition. If this condition is periodic, the new function `handlePeriodicCondition()` gets called. To find out what border cell we should be moving the particle to, we call `getOppositeOfHalo()`. Based on the cardinal direction of the halo cell (NORTH, SOUTH etc.) this subtracts/adds a certain number of cells so as to return the corresponding border cell on the other side of the domain. To find out the particle's new position, we calculate its relative position within the halo cell it is currently in and add it onto the position of the border cell it should be moved to. Once we set the new position within the particle, we also move its reference from the halo to the border cell. In most cases, this concludes the process.
    * However, if the halo cell was a corner cell, then `getOppositeOfHalo()` actually returns another halo cell, not a border cell. In this case, it means that the particle actually interacted with two boundaries. Thus, after we move the particle "periodically" to the second halo cell, we call `handleHaloCell()` again on the particle, but with the new cell and new location. It is possible, that for the second call, another boundary condition is used (see Figure 1 for more intuition). This edge case was a lot of trouble and made our whole simulation blow up until we found it (it also impacted the reflective boundary condition where it met with periodic).

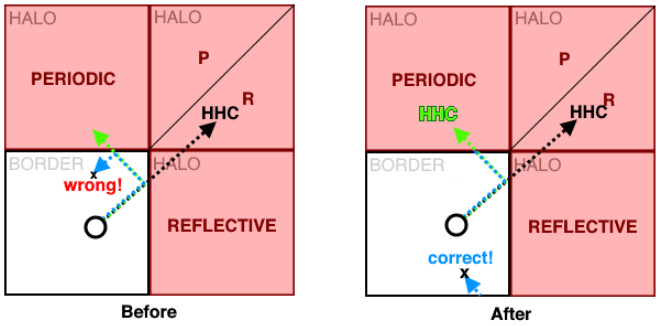

Figure 1: The updated halo cell handling routine. Before, applying a reflective boundary condition on a corner halo cell assumed that *both* the vertical *and* horizontal boundaries are reflective. As such, in the above example, the particle would bounce off the top boundary, even though it is a periodic boundary. This has been fixed by only reflecting the particle on one axis, then calling `handleHaloCell()` (HHC) to handle the new corresponding boundary condition. The same goes analogously for periodic boundaries.

  - **Mirroring**: Because copying/destroying actual objects is a no-go, we chose to work with references once again. Our approach has three main parts: putting the references of "ghost" particles in the appropriate halo cells, calculating the inter-particle forces with a slighlty modified function and then deleting all the ghost-particle references.
    * The function `mirrorGhostParticles()` handles the first part. For every cardinal direction of the border cell (NORTH, SOUTH, EAST, WEST), it checks whether in that direction a periodic boundary applies and if so, uses `getOppositeOfBorder()` to get the corresponding halo cell on the other side of the domain. In this cell we place a reference to the particle, without modyifying the position in the particle object itself. A special case is represented by corner border cells. For these, we mirror across the two directions of the corner (North and West, North and East etc.) and additionally place a ghost-reference in the opposite corner (e.g. for a particle in the SW corner place a reference in the NE halo corner cell).
    * With the groundwork laid, `calculateF_LennardJones_LC()` calculates the forces between all particles. The modified forces function skips over the halo cells (so we don't apply forces to the ghost-references). Otherwise, normal third-law forces are computed with the mention that the function

`getTruePos()` is called. This simply returns the position of a particle if it's in a normal cell. If the particle is in a halo cell (i.e. is a ghost-particle), it calculates its relative position in the cell of the true particle and adds it on the position of the halo cell, returning the position of the ghost-particle, required for the force calculations.

- **Gravitational Force**: This was trivial to implement since we only needed to add an easy to calculate force to all active particles. Naturally, the XML was also extended to take this extra parameter; see the `README.md` for more details.
- **Extra Particle Parameters**: To accomodate multiple substances, it was required to extend the Particle class to feature the Lennard-Jones parameters ϵ and σ. This was done, along with their inclusion into the XML file format. The formulas in the worksheet were implemented in the force function. Otherwise, small revisions were undertaken to make `calculateF_LennardJones_LC()` more readable, without changes in functionality.
- **Experiments**: The experiments mentioned in the task were undertaken and are included in the submission. **NOTE**: Because of the way our linked-cell algorithm is implemented, the positions of the cuboids were slightly shifted up and to the right. This is because the bottom-left corner of the domain $(0, 0)$ is a halo cell. The same goes for all future simulations.

Task 3: "Simulation of a falling drop - Liquid"

- **Usage (setup)**: `build/src/MolSim input-lj-w4t3-base.xml`
- **Usage (main)**: `build/src/MolSim input-lj-w4t3-disc.xml`
- **Checkpointing**: Checkpointing was made possible by passing the `--generate-serialization` argument to `xsd` for the previous worksheet, allowing serialization to and from XML files.
  - **XMLWriter**: To generate an output XML file which can once again be used as input for a future simulation, we created a new class, `XMLWriter`, which uses `xsd` to create a valid input file. As opposed to the other output writer classes, this one does NOT inherit from `FileWriter`, since those subclasses are only used for periodically writing the particles to an output file, not the whole state of the simulation after a complete run. This might be changed in the future, however, to unify all writer classes.
  - **Output**: The result is stored in the same directory from which the program was ran. It uses the basename as it is specified in the input XML file, with `_results` appended.
  - **Details**: A few conscious decisions were made about which data should be stored.
    - Each *active* particle is individually stored in a `<particle>` tag. Unfortunately, this blows up the size of the output file, but there isn't really any other feasible way of storing the data of the simulation particles. As a small optimization, the total number of active particles is also saved to allow reserving enough space in the particle vector beforehand.
    - The start and end times of the simulation are not changed. This is because we can't know beforehand how the user wants to continue. Thus, the new start and end times must be set manually.
    - The starting temperature for the output file T_{init} is the current temperature at the end of the simulation. If the target temperature T_{target} is not specified, the initial temperature T_{init} from the previous simulation is used. This is done to prevent gradually moving the target temperature away from it's true value (e.g. if the temperature at the end of the simulation is $40.5K$ and the unspecified target is 40, we don't want to progressively move the target temperature away from 40).
    - Initialization of particle velocities with Brownian Motion is *disabled*, since this is a continuation of a simulation, not a new one. Also, the particles should all have some velocity by now.
- **Experiments**: The simulation results are once again included in the submission.
  - **Setup**: The setup is performed using the parameters given in the worksheet (albeit under the linked-cell implementation constraint mentioned in Task 1).
  - **Main**: The simulation using the disc is done using a slightly modified version of the XML output. The disc was added manually and the start and end times of the simulation had to be set by hand. Assuming the drop is of the same type of liquid, the m, ϵ, σ and h particles are the same as the liquid's.

Task 4: "Performance Measurement and Profiling"

- **Runtime Measurement**: Time benchmarking was performed on CoolMUC-4 using varying iterations counts and compilers, see below for the results. All dependencies were dynamically fetched and compiled.
  - For benchmarking, we created a new header-only class, `utils/Timer.h`, which times a certain block of code and allows calculating the molecule-updates per second (MUPS/s).
  - Only the time integration loop itself is measured. This is because almost all other elements rely on I/O, which is too inconsistent to reliably measure. Moreover, for typical simulations with a large enough iteration count, the simulation loop takes up the most amount of runtime anyway.
  - The MUPS/s metric is calculated by accumulating the number of active particles after each timestep, then dividing it by the total runtime in seconds. We interpreted a "molecule update" as a complete change of a particle's data after one iteration, including position, velocity and force updates.
  - Much like logging, benchmarking is configured at compile-time and works using macros. In order to enable benchmarking, `ENABLE_BENCHMARKING` must be turned on when using CMake. Alternatively, one can pass `-c` to the build script.
  - Benchmarking and profiling simultaneously is disabled, because profiling tools and profiling flags, such as `-pg` for `gprof`, add extra instrumentation code / steps, making timing inaccurate.
- **Profiling**: Profiling was done primarily using `gprof`, `perf` and Valgrind's `callgrind` tool. Due to issues with profiling on the Linux Cluster[1], profiling was done on a personal machine.
  - **Environment**: 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, Clang++ 18.1.3 with -O3 (Release build) and automatic CPU frequency scaling disabled.
  - **Methodology**: Multiple benchmarks were performed using varying amounts of iterations (10000, 20000, 50000) to scope out any potentially large variations. All I/O was disabled, except for the required functionality to load XML input files. This is minimal, however, and restructuring the code to remove this one small bit of I/O left would have been needlessly tedious. The overall results were mostly the same across all iterations, so we will only analyze the profiling data for 50000 iterations going forward, as this gives us the most complete overlook.
  - **Quirks**: While the overall call graph was mostly the same across all profiling tools, `perf` and Valgrind report memory managements calls (`malloc, free`) in `CellContainer::getNeighbors()`, while `gprof` doesn't. This is *not* done explicitly - we're just creating and populating a new vector, which we then return to the caller function. These memory allocations happen under the hood.
- **ICPX and Xerces-C**: Getting `icpx` to work was a mess. Attempting to compile the code using `icp` and `icpx` as-is nets you a linker error with Xerces-C, so we tried loading the corresponding module on CoolMUC-4. That didn't work, because now, a necessary header file is missing, perhaps due to a Xerces-C version mismatch. Then, we tried compiling all other parts of the code beside the main `mdsrc` library using G++ and only use `icpx` for our own code. That resulted in a linker error, this time with `spdlog`. Finally, we gave up and simply removed all references to Xerces-C and hard-coded the input data into the program itself.
- **Input Data**: Benchmarking and profiling was done using the large experiment from Task 2. It is suitable due to the large number of particles (10000) and frequent (mixed) boundary condition handling. Thus, we get to run through most of the program's function paths and see which parts consume the most runtime.
- **Results**: Table 1 shows the average raw results from running the simulation multiple times for varying iteration counts on CoolMUC-4, compiled in `RELEASE` mode using the default CMake compiler option (`-O3`) and the default compiler when no modules are loaded (G++ 7.5.0). Table 2 shows the average raw results from running the simulation multiple times for 50000 iterations on CoolMUC-4, this time using different compilers. Figure 2 shows a hierarchical view of the call graph generated by `gprof`. Figure 3 shows some call graphs generated by Valgrind's `callgrind`, highlighting hidden memory allocations across our code.
  - **Table 1**: The runtime of the simulation scales linearly with respect to the number of iterations, with the runtime-to-iteraton ratio being around 0.2. As (mostly) expected, the MUPS/s mostly stay the same across all iteration counts, with some slight divergence as the iteration count grows. This is likely because, further into the simulation, particles become increasingly unstable, causing boundary conditions and cell updates to be evaluated more frequently, alongside varying cell density. It could also be due to inconsistent load of the `serial` cluster - either way, we're not re-testing a 35-minute simulation.

[1] The `gprof` output was completely wrong, with only select few functions being displayed, all reporting times of 0 seconds; `perf` does not have the required permissions to work, and Valgrind doesn't even exist.

- **Table 2**: GCC and Clang are close to eachother in terms of runtime and MUPS/s. The Intel C++ compiler is noticeably faster, finishing more than 2 and a half minutes quicker than the rest with 1.2x the molecule updates per second.
- **Figure 2**: As we thought, the force calculation function takes up the vast majority of the runtime, at roughly 91%. For some reason, simple getters seem to take up a significant chunk of the total runtime. This is probably not because of the getters themselves; rather, it's because they are used in conjunction with arithmetic to update the particle metadata. Thus, the graph might seem a bit misleading.
- **Figure 3**: As mentioned before, `callgrind` observes hidden memory allocations in certain parts of the code.

Iterations	Runtime (s)	MUPS/s
20000	397.869	502675.490
50000	1008.553	495768.699
100000	2156.857	463631.365

Table 1: The runtime in seconds and the molecule-updates per second for varying iteration counts of the large experiment from Task 2, built and tested on CoolMUC-4 using G++-7.5.0.

Compiler	Runtime (s)	MUPS/s
`g++ 13.2.0`	899.085	556131.449
`clang++ 16.0.2`	879.336	568622.233
`icpx 2023.2.1`	723.601	691001.407

Table 2: The runtime in seconds and the molecule-updates per second for 50000 iterations of the large experiment from Task 2, built and tested on CoolMUC-4 using different compilers.

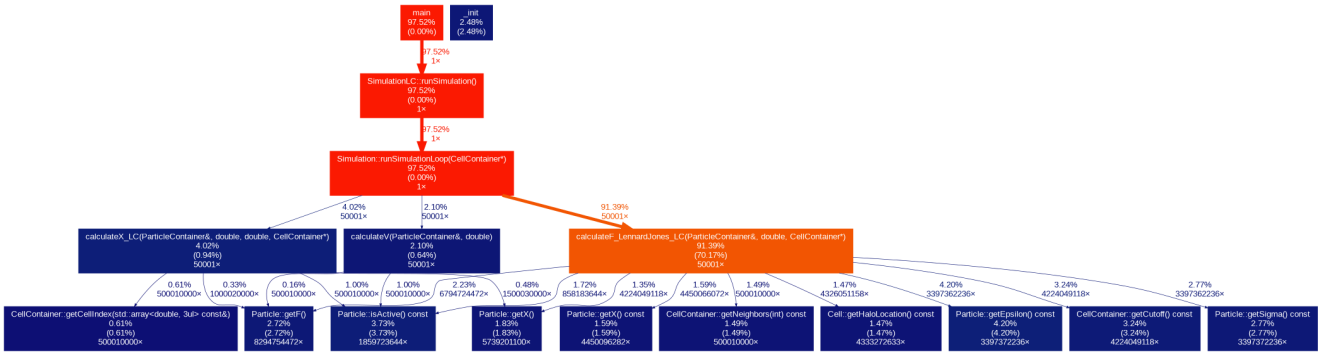

Figure 2: The call graph of the program for 50000 iterations of the large experiment from Task 2, generated by `gprof` on a local machine and visualized using `gprof2dot`. The red and orange nodes show the hot path of the program. Each edge contains the percentage and absolute count of each function call.

Task 5: "Tuning the sequential Performance"

- Some optimizations were performed and tested, some with greater success than others. The commit hash for each incorporated version is included in the parentheses.
  - **Base (`917ef6f`)**: Below are the results from the base version, tested on CoolMUC-4.

Compiler	Runtime (s)	MUPS/s
`g++ 13.2.0`	16.87	586701.434
`icpx 2023.2.1`	13.81	717079.53

  - **Success - Precomputing Cell Neighbors (`724df3f`)**: Precomputing the cell neighbors on creation and storing them in a vector instead of dynamically calculating them each time they were accessed in the force calculation function drastically reduced the overhead caused by `getCellNeighbors()`.

Compiler	Runtime (s)	MUPS/s
`g++ 13.2.0`	15.07	665483.585
`icpx 2023.2.1`	12.06	813368.826

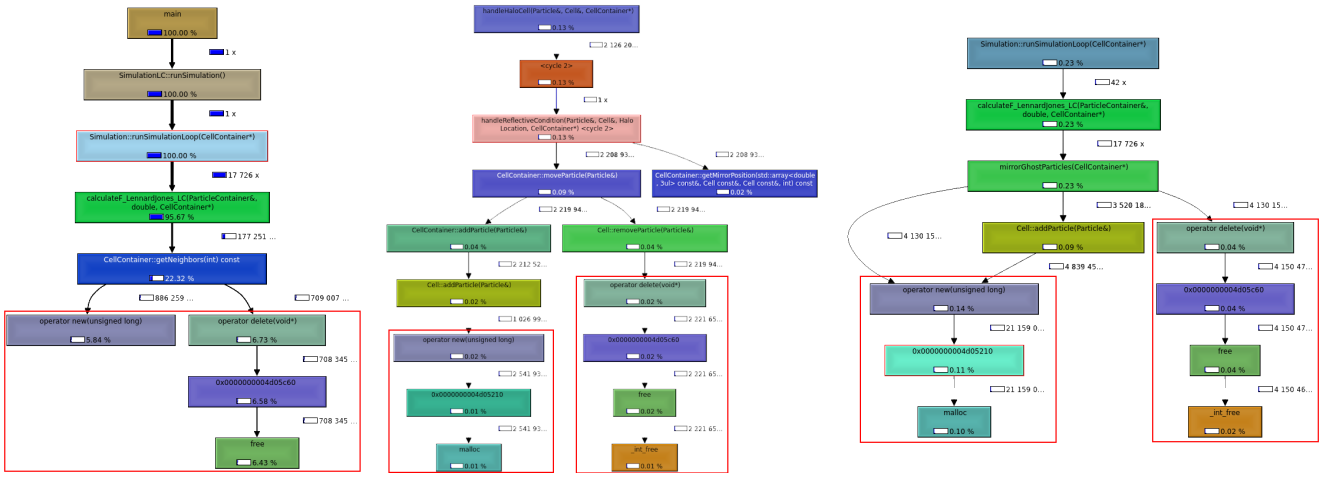

Figure 3: Call graphs generated by `callgrind` for 50000 iterations of the large experiment from Task 2. Note the dynamic memory allocations marked in red, even though we never explicitly call `new` or `delete` in our code.

– **Success - Active Particle Iteration Redesign (`0dd617e`)**: To avoid checking *each* particle's activity for *each* iteration, we redesigned the way active particles are handled. Since a particle may only become inactive after updating its position (due to boundary conditions or, worse, an error), inactive particles are completely removed from the `ParticleContainer` after each `calculateX()`-call in which at least one particle became inactive. For all other container iterations, we don't need to check for a particle's activity anymore, since it can't be changed anywhere else and since they're completely gone now.

Compiler	Runtime (s)	MUPS/s
`g++ 13.2.0`	13.75	729335.494
`icpx 2023.2.1`	11.11	891410.049

– **Unknown - Compiler Flags**: We tried different compiler flags, such as G++'s `-Ofast` (typically unsafe, but mostly qualitatively OK here) or `icpx`'s `-ip, -ipo`, but the improvements were marginal at best and inconsistent at worst, with regular `-O3` occasionally beating out the other optimizations. We didn't try `-O2`.

– **Failure - Avoid Checking Corner Neighbors**: We thought that we could avoid checking particles in neighboring corner cells based on where the particle is in the current cell, but as it turns out, because we have to check the condition for each particle and for each of the 4 corners, this actually adds more overhead than simply checking the pairwise distances between the particles, slowing the simulation down. Figure 4 explains the experiment.

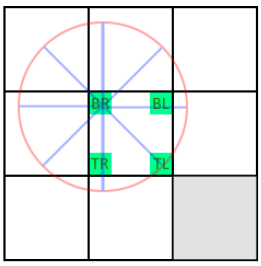

Figure 4: If a particle is in one of the green corner areas, assuming the cell size is equal to the cutoff radius r_c it is guaranteed to be far enough away from the opposing corner neighbor. As such, it isn't needed to check that cell.

– **Failure - Gonnet's Method**: To reduce the amount of spurious distance checks, we attempted to use the *sorted interaction algorithm*, as presented by Gonnet.[2] However, our implementation was broken (and not *much* faster than the already existing implementation), and we couldn't fix it in time, but out of all other attempts, this one seems the most promising, and might be revisited in the future.

– **Skipped - AoS vs. SoA**: We skipped testing the performance of AoS (Arrays of Structs, *our current method*) vs. SoA (Structs of Arrays) due to time constraints.

[2] P. Gonnet, A simple algorithm to accelerate the computation of non-bonded interactions in cell-based molecular dynamics simulations, J. Comput. Chem. 28 (2007) 570–573.

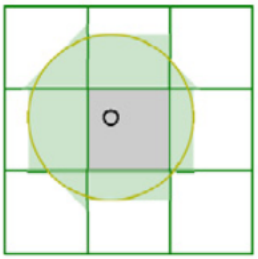

Figure 5: A visual representation of the search area, using Gonnet's algorithm.

- **Skipped - Reordering Particles**: We skipped reordering the particles inside the particle container for better cache locality, since the overhead for constantly moving particles around in the overarching container would dramatically outweigh the (slight) gains of better memory accesses, especially for very large simulations. Also, profiling using `perf` showed that, both for 10000 and 25000 iterations, we had roughly the same number of page faults, meaning that thrashing wasn't an issue.
- **Skipped - Fixed Point Arithmetic**: We skipped using fixed point libraries, because most modern CPUs have powerful enough FPUs that eliminate the benefit of using fixed point numbers, which would inherently be less accurate anyway.
- **Skipped - Various Minor Opimizations**: Other minor optimizations (using C++'s "fast" datatypes, using naive algorithms instead of `std::accumulate`, skipping redundant divisions by 2 then multiplications by 2) were ignored, because the compiler would optimize them out anyway.

Miscellaneous

- **Removals**: Various components were removed due to lack of support.
  - Removed text file input. All input files going forward must be XML files.
  - Removed `CuboidGenerator`. There was no need for this anymore, since we removed text file input.
  - Removed Google Benchmark integration, since we now have our own timing library.
  - Removed all other force calculation functions besides the main one to streamline optimization.
  - Removed `filesystem` to enable support with CoolMUC-4's default compiler, G++ 7.5.0.
- **Common Cluster Interface**: The `Cuboid` and `Disc` classes now inherit common functionality from a `Cluster` class.
- **Reference Wrappers**: Certain containers now use reference wrappers instead of raw pointers for safety. Personally, I still stand by the fact that using raw pointers is fine as long as you know what you're doing, and since we don't allocate any dynamic memory here or move particles around inside their respective containers, I don't see the issue, but I suppose if C++ offers you the possibility of RAII, why not use it.
- **Compiler Selection**: You can now specify the compiler used in the build script using `-C` for the C compiler and `-X` for the C++ compiler.