Performance Programming Coursework: Serial Optimization of a Molecular Dynamics Program

Abstract

This paper documents the serial optimization of a molecular dynamics program written in the Fortran programming language. The original version of the program took 1270 seconds to complete a simulation on the Cirrus supercomputer. The final version took 26 seconds to complete the same simulation. 98% of the runtime could be removed, simply by modernizing the source code and utilizing compiler optimizations. The main result of the optimization process is the observation that idiomatic, modern and easy-to-read Fortran code results not only in a far better performing program, but also in a more maintainable one. Utilizing the compiler properly was the main reason for the increased performance.

Keywords: Scientific programming, serial optimization, molecular dynamics, Fortran

1. Introduction

This paper documents the serial performance optimization conducted for a molecular dynamics program written in the Fortran programming language. The program reads data from n molecules or particles as its input and iterates a predefined amount of steps, which correspond to the progress of time in the simulation. Each step properties of the particles (position in a three dimensional Euclidean space and velocity) are updated. The program counts collisions between molecules and writes the data for each molecule of the simulation out after a certain interval of iterations. This interval is called a superstep. The original version of the program is not optimized for computational efficiency.

This paper describes, documents and discusses the process of optimizing the original version of the program serially. The program was optimized for the Cirrus supercomputer, a tier-2 UK national supercomputer of the engineering and physical sciences research council, which is hosted and maintained by the EPCC (EPCC, 2020a). The compiler used was Intel's Fortran compiler `ifort`, version 18.0.5 for the Linux operating system (Intel, 2018).

The conducted optimizations range from choosing the appropriate compiler flags over rewriting performance critical sections of the program to hardware specific optimizations, like leveraging vectorization and cache optimizations. Raw performance benefits are not all that is important for writing well performing and good programs. The maintainability of the program is also a significant factor. The performance optimizations are always looked at from a perspective of maintainability.

The original version of the program takes 1270 seconds to complete the simulation used for benchmarking. The final version takes only 26 seconds to complete the same benchmark. 98% of the runtime was removed by the optimization efforts described in this paper. The main result of this paper is the observation that idiomatic, modern and

easy-to-read Fortran code results in the best performing program, mostly by enabling the compiler to optimize effectively.

Not only is the performance of the program drastically improved, maintainability is also highly increased. Approximately 20% of the source code got removed, making it easier to follow what the program does. Only in one case was code explicitly added to increase performance (OpenMP's simd directive). On most other occasions performance was increased by replacing badly readable and inefficient code with modern Fortran code, killing two birds with one stone.

This paper continues in Section 2 with describing the molecular dynamics program in detail. Section 3 describes Cirrus and how the correctness of the program is tested with a regression test suite. Also the benchmark suite used for assessing the performance benefits of an optimization is outlined, as is the simulation setup used for benchmarking. Section 4 lists, describes and discusses all optimizations tested with their performance benefits. Afterwards, the results are discussed in Section 5. At least a conclusion is drawn in Section 6.

2. Molecular Dynamics Program

This section describes what the program does. The program is a simple molecular dynamics program. It reads data from n particles from a file. In the following, subscripts $1 \leq i \leq n$ refer to a particle of the simulation. The data for a particle i is its mass m_i, the viscosity of a fluid i is part of vis_i, the position of the particle's center in a three dimensional Euclidean space $\vec{p}_i$ and its velocity vector $\vec{v}_i$.

The program iterates a predefined amount of iterations. Each iteration represents a time step in the particle simulation. The time is updated by a constant value Δ_t. Every step the position and the velocity of each particle is updated, based on the gravitational forces operating on each particle. The gravitational forces are the forces between each particle and the gravitational force coming from a large central mass located at the origin of the Euclidean space. The particles are part of a fluid, which means the viscosity of the fluid must also be taken into account. The last external force operating on a particle is wind $\vec{w}$, which is a constant vector over the whole simulation.

The gravitational forces are computed based on Newton's law of universal gravitation. It states, that any two physical objects attract each other with a force, which is proportional to the mass of both objects and inversely proportional to the squared distance between both (Feynman, 1963). The scalar gravitational force F between two objects i and j can be mathematically described as:

$$F_{i,j} = G\frac{m_i m_j}{||\vec{p}_i - \vec{p}_j||_2^2}, \tag{1}$$

where G is the gravitational constant, m_i the mass of object i and $||\vec{p}_i - \vec{p}_j||_2$ the Euclidean or L_2 distance between the centers of both objects. The vector form of the gravitational

force object j operates on object i, which also accounts for the direction of the force, is given by:

$$\vec{F}_{i\leftarrow j} = -F_{i,j}\frac{\vec{p}_i - \vec{p}_j}{||\vec{p}_i - \vec{p}_j||_2} = -G\frac{m_i m_j(\vec{p}_i - \vec{p}_j)}{||\vec{p}_i - \vec{p}_j||_2^3}. \tag{2}$$

The gravitational force, which object i operates on object j is the additive inverse of $\vec{F}_{i\leftarrow j}$: $\vec{F}_{j\leftarrow i} = -\vec{F}_{i\leftarrow j}$. If particles collide, the gravitational forces between the two object are negated (the forces are flipped). The program finds collisions by checking if the distance of two particles is smaller than a threshold $\tau_{i,j}$, which is the sum of the radii of the two particles. In the simulation used for benchmarking all particles have a radius of 1/2, so the threshold for a collision is 1. We can define a pairwise gravitational forces function that takes the force flipping into account for the particles of the simulation as:

$$f_{i\leftarrow j} := \begin{cases} \vec{F}_{i\leftarrow j} & \text{if } ||\vec{p}_i - \vec{p}_j||_2 \geq \tau_{i,j} \\ -\vec{F}_{i\leftarrow j} & \text{otherwise} \end{cases}. \tag{3}$$

Other than the pairwise gravitational forces between the particles, there is the gravitational force of the central mass $\vec{F}_{i\leftarrow central}$. The central mass lies at the origin of the three dimensional space, which means its position vector $\vec{p}_{central} = 0$.

The viscosity of the fluid is another force operating on a particle. It is simply the negative of the viscosity of the fluid multiplied by the velocity vector of particle $i : -vis_i\vec{v}_i$. Shear velocity of the fluid is not taken into account. Lastly there is the wind force, which is simply the negated viscosity of the fluid particle i is part of multiplied by the wind vector: $-vis_i\vec{w}$. The overall force per iteration operating on particle i can now be described as:

$$\vec{F}_i = -vis_i(\vec{v}_i + \vec{w}) + \vec{F}_{i\leftarrow central} + \sum_{j\neq i}^{n} f_{i\leftarrow j}. \tag{4}$$

Based on $\vec{F}_i$ we can now update $\vec{p}_i$ and $\vec{v}_i$:

$$\vec{p}_i = \vec{p}_i + \Delta_t\vec{v}_i \tag{5}$$

$$\vec{v}_i = \vec{v}_i + \Delta_t\frac{\vec{F}_i}{m_i}. \tag{6}$$

The iterations of the program are broken down into supersteps. On completion of a superstep, the updated particles are exported to a file with the same format as the input file.

3. Setup

This section outlines the settings, under which the program was optimized for performance. Information about the used hardware is given. The way correctness of the program was

tested is described. Lastly the settings and the criterion for benchmarking the computational performance are presented.

Like stated in Section 1, the program was optimized for the Cirrus supercomputer (EPCC, 2020a). Since we are running the program serially, we are not concerned with the amount of nodes or the interconnect, but will focus on a single compute node. A single compute node of Cirrus contains two 2.1 GHz, 18-core Intel Xeon E5-2695 processors (code name: Broadwell). The processor supports the AVX2 vector instruction set (Buxton, 2011). Each processor is connected to 128 Gigabyte of memory. Both processors are within a NUMA region, so 256 Gigabyte of memory are actually at ones disposal (EPCC, 2020b). The compute node offers three levels of cache:

1. 32 Kilobyte instruction and 32 Kilobyte of data cache (per core)
2. 256 Kilobyte (per core)
3. 45 Megabyte (shared)

Testing the correctness of the program is not as straight-forward as it seems at first glance. Like stated in Section 2, after each superstep, the updated particles are written out to file. Comparing the output of the optimized version of the program with the original one would be a sufficient test for correctness if it were not for floating point rounding errors. These accumulate and after a certain amount of time the numbers generated by the optimized version will be too different from the original ones.

The program was augmented by a special test setting in order to avoid getting different results just because of floating point errors. This test setting differs from the normal program, because it reads the data it has written out after a superstep back in. That way, the next superstep will work with the floating point numbers that are crippled by writing them to file. The floating point numbers are written to file text based in exponential form with 16 digits, eight digits on the right side of the decimal point (see e.g. Shene, 2020, for formatting IO in Fortran). Changing the output format to a more precise representation is not possible, because this would mean the files generated as output would not have the same format as the input file with the initial states of the particles. The test setting allows for effectively comparing the output files of both versions, because floating point rounding errors now only accumulate over a single superstep instead of the whole simulation.

Once the discrepancy between the output values of the optimized version and the original one surpasses a predefined error level, the optimization is deemed to result in an incorrect version of the program. The predefined error level was set to be 0.05. If any output file contains a `NaN` value, the program is also deemed incorrect. The regression test suite was implemented with a Python script.

The program runs five supersteps. Each superstep encompasses 100 iterations. Computation is done using double precision floats. The input file which was used for optimizing contained 4096 particles. The goal of the optimization process was to reduce the wall-clock

time of the program to a minimum, while bearing in mind portability and more importantly maintainability. The program measures the time it needs for each superstep and its overall time, including the file output. In order to build the benchmark suite around the program, the timings are exported to another file when the program has finished the simulation. The program was benchmarked by running it ten times on a compute node of Cirrus and taking the average from those ten runs as the performance measurement. Running it ten times is sufficient to get a stable average, because running the program as a job on a compute node of Cirrus means exclusive hardware access to that node. Only IO performance can be influenced by other users, because Cirrus uses Lustre for its file system which is shared (EPCC, 2020b). As will be shown below, IO performance is actually negligible when it comes to performance optimization when compared to the computational effort of the simulation.

4. Optimizations

This section documents the process of the serial performance optimization of the program. Focus lies more on the process, not the results. All the successively performed optimizations are described. Code quality in form of readability, portability and maintainability is taken into account during the whole process and the optimizations are all looked at from this perspective. The optimization process can basically broken down into five phases: (i) rewriting the source code to modern, free-formatted Fortran and restructuring the source code without changing the critical section, (ii) enabling basic compiler optimizations, (iii) rewriting the program for better performance and maintainability, (iv) taking hardware and environment into account (mainly vectorization and memory access patterns) and (v) trying out more advanced compiler optimizations on the rewritten version of the program again.

4.1 First Phase

The first phase of optimization only concerns itself with increasing the maintainability of the program. The original version of the program is written in fixed format Fortran (see e.g. Sandu, 2001, for free vs. fixed format Fortran). Readability for screen based devices was deemed more of an issue than formatting source code for punched cards, which are unfortunately not supported by Cirrus. So the first step was to reformat the source code to free format Fortran to increase maintainability.

The original version of the program is spread across four files. `control.f` contains the main program. It performs initialization of the program. This includes defining constants and reading the particles from the initial file. It contains the superstep loop and performs the output of the intermediate states of the particles to file. It also collects the timings for every superstep and the combined time for all supersteps together. The `MD.f` file contains the `evolve` subroutine. This subroutine performs the main computations for

the simulation. It is called each superstep and iterates 100 times over the simulation, updating the state of the particles. The particle data is shared between the main program and the `evolve` subroutine with a `COMMON` block (see e.g. Harper and Stockman, 2020). The `COMMON` block is defined in the `coord.inc` file, which also contains the global constants G and $m_{central}$. Lastly, there is the `util.f` file containing utility subroutines and functions, e.g. `visc_force` or `wind_force`, which compute $-vis_i\vec{v}_i$ and $-vis_i\vec{w}$ respectively (see Section 2).

Modern Fortran compilers like `ifort` version 18.0.5 support all Fortran 2008 features (Intel, 2018). Fortran introduced modules in Fortran 90, which make it much easier to share data between subroutines and coupling can be much improved by using them (Moin, 2002). Because using modules increases maintainability a lot, all routines of the program are put into the main program in `control.f`, inside its `contains` block. That way the particle data can be shared with the `evolve` subroutine without a `COMMON` block. These changes greatly increased maintainability, because of the enhanced readability of the source code. Also the build process is simplified, because only a single Fortran 90 file needs to be compiled, rather than having to link `control.f` and `MD.f` with `coord.inc`.

Both, the original version and the new Fortran 90 version were modified to incorporate the test setting, where they read the intermediate outputs back in, in order to compute the next superstep. This setting should not interfere with the original setting used for benchmarking. Intel's Fortran preprocessor was used to implement the test setting (Intel, 2019d). The additional reading back of the intermediate file is put into an `#ifdef` directive. This way, the original version of the program used for benchmarking is not damaged by additional checks at runtime.

Another aspect to consider in favor of the new version is the fact, that the `COMMON` block is not aligned. Figure 1 shows the compiler output, when compiling the original version. The `COMMON` block has alignment issues for the wind vector $\vec{w}$, which could have an impact on the programs performance. Removing the block removes the alignment issue. So not only is the new version better maintainable, it also removes the first performance issue with the original version.

Both versions were compiled using `ifort` version 18.0.5 with the following compiler flags which influence performance: `-O0`, which disables any compiler optimization, `-no-vec`, which inhibits vectorization and `-check uninit,bounds`, which tells the compiler to add extra instructions to the program which perform explicit checking for uninitialized variables and out-of-bounds access of arrays.

Benchmarking the original version reveals that it takes on average 1270 seconds for all five supersteps to complete. A single superstep takes on average 254 seconds to complete. If one subtracts the sum of the individual timings of all five supersteps from the overall time, one gets the time spent doing the file output. The IO time lies at a quarter of a second for the original version, which is 0.02% of the overall runtime. IO time is more or less constant (depending on the workload of the Lustre file system) so it is omitted in the following evaluation of the program due to its insignificance. The new Fortran 90 version

```
ifort -g -O0  -check uninit,bounds -no-vec -fpp
-o ../bin/old_bench control.f MD.o util.o

./coord.inc(25): remark #6375: Because of COMMON, the alignment of
object is inconsistent with its type - potential performance
impact. [WIND]

      DOUBLE PRECISION wind(Ndim)
-----------------------^
```

Figure 1: Output from `ifort` when compiling the original version of the program. The constant vector $\vec{w}$ is not aligned.

of the program takes only 1110 seconds on average to complete. The average superstep time lies at 222 seconds. While the focus of the first phase of the optimization actually was about enhancing maintainability and setting the right foundation for the next phases, the performance was already increased by 13%.

4.2 Second Phase

The second phase was about using the more common compiler flags to enhance the performance of the program from phase one. The compiler flags used in phase one not only hinder compiler optimizations with `-O0`, they even make the code perform worse by adding the out-of-bounds access and uninitialized variables flags. Therefore the first step to better performing code was to utilize the compiler. This phase was not about finding a definitive set of flags, but only a first step to see how much the compiler can achieve using the more common compiler flags for optimization. It was more motivated by the still horrible benchmark time of 1110 seconds, which hinders rapid development during the following phases. Table 1 shows all the different flags tried and how they impacted performance.

The first act was to remove the unnecessary checks for out-of-bounds access of arrays and using uninitialized variables. The current version of the code is riddled with loops, so removing the checks for each should have quite the impact on performance. As it turned out it did. Removing the checks improved the average overall time by 33%. That means 1/3 of the time was spent checking for out-of-bounds access and the use of uninitialized variables.

The next step was to gradually increase the level of compiler optimization from `-O0` to `-O3`. The first level of optimization `-O1` enables speed optimizations that do not enlarge binary size. Optimizations done include data-flow analysis, test replacement and instruction scheduling. `-O1` is designed for large codes with many branches that are not loops (Intel, 2019g). While this description does not fit to the program at all, which has only one significant branch (code structure is discussed below) and spends most of its time in

loops, `-O1` still increases the performance by 44%. Average overall time is reduced from 734 seconds to 413 seconds.

`-O2` is the recommended level of compiler optimization. It performs basic loop optimizations like interchanging, unrolling or scalar replacements. Furthermore inlining, intra-file ipo (interprocedural optimization), dead code elimination and many more optimizations are enabled (Intel, 2019g). Enabling `-O2` reduced the average overall time down to 85 seconds, which is 79% better than the program compiled with `-O1`.

`-O3` enables more aggressive optimizations concerning loops and memory access transformations, additionally to the optimizations done using `-O2`. Optimizations include loop fusion and collapsing if-statements. It is the recommended level of optimization for floating point operation heavy programs that spend a lot of time in loops (Intel, 2019g). This exactly describes the molecular dynamics program and `-O3` actually increases the performance further. The average overall time was further reduced from 85 seconds down to 62 seconds. This is an additional 27% improvement over `-O2`.

At this point, vectorization was still disabled with the `-no-vec` flag. Enabling vectorization on the yet unoptimized Fortran 90 program from phase one resulted in a regression of the average overall time of 33% (see Table 1). Like described above, the second phase is only about finding a set of compiler optimizations that would enable a more rapid analysis of the performance in the following phases dealing with the rewrite of the program. This is the reason why the drop in performance was not analyzed further. For the unoptimized Fortran 90 version of the program somehow vectorization seems to cancel optimizations from `-O3`. The guess at this point was that the program profits more from optimizations from `-O3` (like loop unrolling) than it does from being vectorized.

Lastly other common flags for compiler optimization were considered. Two recommended options besides `-O3` are `-ipo` and `-xHOST` (User389, 2020). Neither would improve the performance of the program. `-ipo` enables interprocedural optimizations between files (Intel, 2019e). The program only consists of a single file, so `-ipo` would not improve performance. `-xHOST` forces the compiler to generate instructions from the highest instruction set supproted by the host (Intel, 2019o). The host is a frontend node of Cirrus in this case. The frontend nodes of Cirrus are the same as its compute nodes. The highest instruction set supported is AVX2 (see Section 3). `-no-vec`, which is still enabled at this point, cancels out `-xHOST`.

Lastly `-Ofast` was tested. `-Ofast` is a shorthand compiler flag, which combines `-O3` with a faster floating point model than the default one. It sets `-O3`, `-no-prec-div` and `-fp-model fast=2` (Intel, 2019h). `-no-prec-div` increases the speed of floating point divisions. The cost of this flag is a reduction in precision (Intel, 2019f). `-fp-model fast=2` works the same way. It increases the performance on the cost of less precise results of floating point operations (Intel, 2019b). Enabling `-Ofast` does not result in less accurate test results. The program's correctness is still given, even though floating point precision was lowered. `-Ofast` does not increase the performance compared to `-O3` (see Table 1).

Optimization	∅ overall time	∅ superstep time	$+/-$ %	Status
Removed checks	734s	147s	33%	Improvement
`-O1`	413s	83s	44%	Improvement
`-O2`	85s	17s	79%	Improvement
`-O3`	62s	12s	27%	Improvement
Removed `-no-vec`	82s	16s	-33%	Regression
`-Ofast`	62s	12s	0%	Invariant

Table 1: Compiler flags tried during the second phase of optimization. The $+/-$ % column displays the variation in average overall time from the best version of the program so far. For example, the best version for the removal of the extra checks was the Fortran 90 version from phase one. For `-Ofast` the best version of the code was the one compiled without the checks and with `-O3`.

Phase two was terminated at this point. The best compiler flags determined were `-O3` with `-no-vec`. The average overall time after phase two is 62 seconds. This is an improvement of a staggering 94% over the results after phase one (1110 seconds), simply by enabling compiler optimization and removing unnecessary checks.

4.3 Third Phase

While the second phase already improved the performance by 94%, the quality of the program is still bad. Both in consideration of performance and more importantly maintainability. Phase three of the optimization efforts therefore tackles this problem by rewriting the critical section of the program, which spans the computations for updating the particles at each time step.

The common workflow for optimizing for speed normally consists of iterations, where the program is profiled, the bottleneck determined and then optimized. The workflow used here is less vigorous concerning profiling. Profiling after each change in order to identify and fix bottlenecks is not necessary, because it is well established where the bottleneck of molecular dynamics programs is: computing the pairwise forces between the particles (Chiu et al., 2011).

This assumption was validated for this program. The update operation of the simulation was split into two parts: (i) computing pairwise forces between the particles and (ii) computing the other forces and updating the position and velocity of each particle. Inlining was disabled and the program run with Intel's VTune 19 profiler (Intel, 2020). The profiler revealed that over 99% of the runtime is spend computing the pairwise forces.

Furthermore, as described above, the guiding principle for rewriting of the program was increased maintainability and elegance rather than pure speed. The codebase is not very big (approximately 250 lines of code) so not a lot of time could be wasted on rewriting parts of the code that are not part of the critical section.

Algorithm 1 : original computation per time step

1: **for** i=1,...,n **do**
2: $\vec{F_i} := -vis_i\vec{v_i}$ {Compute viscosity force for particle i}
3: **end for**
4: **for** i=1,...,n **do**
5: $\vec{F_i} := \vec{F_i} - vis_i\vec{w}$ {Compute wind force for particle i}
6: **end for**
7: **for** i=1,...,n **do**
8: $r_i := ||\vec{p_i}||_2$ {r_i is used in loop below for the denominator in $\vec{F}_{i\leftarrow central}$}
9: **end for**
10: **for** i=1,...,n **do**
11: $\vec{F_i} := \vec{F_i} + \vec{F}_{i\leftarrow central}$ {see Equation 2}
12: **end for**
13: Compute pairwise forces with Algorithm 2
14: **for** i=1,...,n **do**
15: $\vec{p_i} := \vec{p_i} + \Delta_t\vec{v_i}$ {Update position vector of particle i (see Equation 5)}
16: **end for**
17: **for** i=1,...,n **do**
18: $\vec{v_i} := \vec{v_i} + \Delta_t\vec{F_i}/m_i$ {Update velocity vector of particle i (see Equation 6)}
19: **end for**

The guiding tool for the environment specific optimizations was the optimization report generated by the compiler with the `-qopt-report=5` flag (Intel, 2019l). Otherwise the benchmark output and a Python script for comparing different benchmarks with each other were used to determine the impact a change to the source code had on the performance of the program. The script basically replaced using an advanced profiler like VTune after each change made to the code.

The first step rewriting the code was to remove the utility functions and subroutines and inline them by hand (see Section 4.1). This was done not in consideration of performance, but to better enable the rewriting. The utility functions were obscuring loops, making it harder to properly see loop nests which are crucial for vectorization. Looking at the optimization report revealed, that they were inlined by the compiler, so no performance difference was measured after the hand inlining.

Algorithm 1 shows the update operation per time step, as it was originally implemented. As stated above, computing $\vec{F_i}$ without the pairwise forces (Equation 4 without $\sum_{i\neq j}^{n} f_{i\leftarrow j}$) is not the critical section. Nonetheless the code is still badly written and not optimal. Algorithm 1, lines 1–12 are all needed, simply for computing Equation 4 without the pairwise forces. While `-O3` enables loop fusion, only the first two loops (the wind and viscosity forces) are fused together. So there are three iterations needed over n: one for the wind and viscosity forces, one for computing r and one for computing $\vec{F}_{i\leftarrow central}$. The first

```
! This is how the old version of the program computes r
do k = 1, Nbody
  r(k) = 0.0
end do
do j = 1, Ndim
  do i = 1, Nbody
    r(i) = r(i) + pos(i,j) * pos(i,j)
  end do
end do
do k = 1, Nbody
  r(k) = sqrt(r(k))
end do

! The new version uses Fortran's array syntax to convert this
! to a 1-liner
r(:) = sqrt(sum(pos(:,:) ** 2, dim=2))
```

Figure 2: Computing r for each particle in Fortran, as done in the old version of the program and the one using array syntax.

step was to reduce the $3n$ iterations to $2n$ iterations by fusing the viscosity, wind and central force computations all into a single loop setting $\vec{F}_i := -vis_i(\vec{v}_i + \vec{w}) + \vec{F}_{i \leftarrow central}$. Fusing the loops together by hand resulted in a regression concerning the program's performance. While the best version after phase two took on average 62 seconds overall, the version with the fused loops resulted in 64 seconds on average. This is a drop in performance of approximately 3%. The reason for this drop are reduced efficiency of the pipelines of the CPU caused by data hazards. The data hazards are caused by output dependencies (write after write) when computing $\vec{F}_i$ (see e.g. Patterson, 2014).

The next step was to beautify the computation of r (Algorithm 1, lines 7–9). Figure 2 shows the original code and the one-liner it was transformed to. Using Fortran's array syntax makes the code much more readable by being more concise and removing ten lines of code. Unfortunately, making this change results in another performance regression of about 1%. The reason for the performance drop is the fact that `pos(:,:)` copies the content of the `pos` matrix to a new temporal one. `pos` is the $n \times 3$ double precisision matrix storing the particle vectors $\vec{p}_i$. It occupies 96 Kb of memory. Copying 96 Kb is a costly operation which is why the code is slower than before.

In order to further concise the program, reduce the amount of loops and the memory footprint, r was removed from the program completely, moving the computation of $||p_i||_2$

into the actual denominator of $\vec{F}_{i \leftarrow central}$. Problematic for this operation was the memory layout of the particle vectors. All particle vectors involved (force $\vec{F}_i$, position $\vec{p}_i$ and velocity $\vec{v}_i$) are actually represented as a $n \times 3$ matrix of all particles. Fortran stores matrices column-wise. This means all particle vectors are actually strided with a separation of n and not contiguous in memory. This does not leverage locality needed for utilizing the cache for fast memory access when doing particle-wise operations, rather than dimensional-wise (outer loop over the three dimensions of the particle vectors). The current version of the program actually computes $\vec{F}_i$ dimensional-wise, utilizing locality. r_i can only be computed particle-wise. So in order to remove r from the program the loops for computing $\vec{F}_i$ were exchanged to being particle-wise (outer loop over the particles instead of over the dimensions of each particle). While this change removes leveraging locality, removing r actually reduces the pressure on the cache. Like stated in Section 3, the whole program uses double precision floats. For $n = 4096$, this means r alone takes up 32 Kb—the whole L1 cache. The performance report reveals that the inner loop over the three dimensions of each particle is actually unrolled by the compiler. Removing r from the program resulted in a performance improvement of approximately 2%, still worse than the version of the program after phase two.

The last change made to the part of the simulation not dealing with the computation of the pairwise forces was fusing the loop computing $\vec{F}_i$ with the last two loops (Algorithm 1, lines 14–19), which were already fused. $2n$ were reduced to n iterations by fusing the two loops by hand. Again this reduction in the constant factor of n comes with the cost of locality when doing particle-wise operations instead of dimensional-wise ones. Concerning the maintainability, the whole rewrite of the program to this point saved approximately 30 lines of code and increased the readability drastically. Only a single loop is needed for computing $\vec{F}_i$ without the pairwise forces and updating $\vec{p}_i$ and $\vec{v}_i$. Fusing the two loops is invariant concerning the performance.

Next the critical section of the program—computing the pairwise forces $f_{i \leftarrow j}$—was optimized. How the original program computes the pairwise forces is shown in Algorithm 2. First two minor simplifications of the loop nest computing the pairwise forces (Algorithm 2, lines 11–16) were conducted. Two particles i and j collide, if $||\vec{p}_i - \vec{p}_j||_2 < \tau_{i,j} := radius_i + radius_j$ (see Section 2). $radius_i$ is never read from the file containing the data for each particle. Instead $\forall i : radius_i := 1/2$ is set in the routine reading the particle data from file. This way $\forall i, j : \tau_{i,j} = 1$, so we can remove $radius$ from the program (saving another 32 Kb of memory) and replace it with a single constant. The second minor change concerns an unnecessary branch. The program counts particle collisions. Whenever $||\vec{p}_i - \vec{p}_j||_2 < 1$ a counter is increased. Instead of increasing the counter in the if-statement of Equation 3, a boolean variable is set to true and another if-statement is needed for incrementing the collision counter. The second if-statement was replaced with incrementing the collision counter directly. Again the changes increased readability but the program is 5% slower than the fastest version.

Algorithm 2 : original pairwise forces computation

```
1: for i=1,...,n do
2:    for j=i+1,...,n do
3:       p⃗_{i,j} := p⃗_i − p⃗_j {p⃗_{i,j} is used in loop below and for the numerator in f_{i←j}}
4:    end for
5: end for
6: for i=1,...,n do
7:    for j=i+1,...,n do
8:       Δp_{i,j} := ||p⃗_{i,j}||_2 {Δp_{i,j} is used in loop below for the denominator in f_{i←j}}
9:    end for
10: end for
11: for i=1,...,n do
12:    for j=i+1,...,n do
13:       F⃗_i := F⃗_i + f_{i←j} {see Equation 3}
14:       F⃗_j := F⃗_j + f_{j←i} {see Equation 3}
15:    end for
16: end for
```

The next change was the first major change of phase three where performance was significantly improved. Instead of computing $\vec{p}_{i,j}$ and $\Delta p_{i,j}$ in their own loops (Algorithm 2, lines 1–10) for each particle and having to save the result in memory for later use in computing $f_{i\leftarrow j}$, they were reduced to two temporal variables computed in the last loop (Algorithm 2, lines 11–16). There are $n(n-1)/2$ particle pairs. This means removing the two loops results in a reduction of $n(n-1)$ iterations. The original program actually saved $\vec{p}_{i,j}$ and $\Delta p_{i,j}$ not in a matrix/array with $n(n-1)/2$ elements but in oversized containers with n^2 elements. This means the array for $\Delta p_{i,j}$ alone takes up 128 Mb of memory. The matrix which saves $\vec{p}_{i,j}$ takes up 384 Mb of memory. The fastest version took on average 62 seconds to complete the whole simulation. The new version only takes on average 42 seconds to completion, a performance improvement of 32%.

The next two changes tried to reduce the impact of the branch in Equation 3. Computing $f_{i\leftarrow j}$ was changed to a particle-wise operation (see above). This way the branch was moved up one loop in the nest and the inner loop over the dimensions was again unrolled. This resulted in a 2% performance improvement. At this point $f_{i\leftarrow j}$ and $f_{j\leftarrow i}$ were still both computed, even though $f_{j\leftarrow i} = -f_{i\leftarrow j}$. Readability is increased by using $\vec{F}_j := \vec{F}_j - f_{i\leftarrow j}$ instead of $\vec{F}_j := \vec{F}_j + f_{j\leftarrow i}$. This change has no effect on the performance.

4.4 Fourth Phase

The rewrite in phase three was necessary not only for increasing maintainability, but also to make the goal of this phase easier to achieve. This phase concerns itself with

Algorithm 3 : computation per time step after phase three

1: **for** i=1,...,n **do**
2: **for** j=i+1,...,n **do**
3: $\vec{F_i} := \vec{F_i} + f_{i \leftarrow j}$ {see Equation 3}
4: $\vec{F_j} := \vec{F_j} - f_{i \leftarrow j}$ {see Equation 3}
5: **end for**
6: **end for**
7: **for** i=1,...,n **do**
8: $\vec{F_i} := \vec{F_i} - vis_i(\vec{v_i} + \vec{w}) + \vec{F}_{i \leftarrow central}$
9: $\vec{p_i} := \vec{p_i} + \Delta_t \vec{v_i}$
10: $\vec{v_i} := \vec{v_i} + \Delta_t \vec{F_i}/m_i$
11: **end for**

hardware and environment specific optimizations, mainly enabling vectorization. As will be shown below, idiomatic and modern Fortran code makes it easy for the compiler to auto-vectorize and only minor efforts were needed to enforce vectorization where the compiler could not auto-vectorize.

Algorithm 3 shows how the particles are updated for each time step after the optimizations in phase three. Up to this point vectorization was still disabled with the `-no-vec` flag. The first step of phase four was to enable vectorization. The compiler vectorizes the code when the `-no-vec` flag is not passed. But in order to guide the compiler to fully utilize vectorization, `-no-vec` was replaced with the `-xCORE-AVX2` and the `-vec-threshold0` flags. `-xCORE-AVX2` tells the compiler to generate AVX2 instructions when possible and optimize the code for this instruction set (Intel, 2019n). `-vec-threshold0` tells the compiler to vectorize whenever possible (Intel, 2019m).

The second loop (Algorithm 3, lines 7–11) is automatically vectorized by the compiler. The inner loops over the three dimensions of the vectors are unrolled. The compiler tries to vectorize the inner most loop in a loop nest. It is unable to vectorize the inner loop for computing the pairwise forces (ranging over $j = i+1, \ldots, n$), because of data dependencies (flow and anti dependency for updating $\vec{F_i}$). Like stated in Section 4.3, the pairwise forces loop takes over 99% of all the runtime of the program. Therefore it is not surprising that the vectorized second loop does not speed-up the program. Without making the pairwise loop vectorize, enabling vectorization is not increasing the performance.

The inner loop of computing the pairwise forces was vectorized using OpenMP 4.5's simd directive (OpenMP Architecture Review Board, 2015). The outer loop could not be properly vectorized, because of the irregularity of the inner loop, it depending on i. This hindered OpenMP to successfully collapse the inner and outer loops. In order to cope with the flow and anti dependencies $\vec{F_i}$ was removed from the inner loop and replaced with a temporal reduction variable $\vec{F_i}'$. After the inner loop completed the temporal variable was integrated into $\vec{F_i}$ by setting $\vec{F_i} := \vec{F_i} + \vec{F_i}'$. Using the reduction clause of the simd directive

over $\vec{F}_i'$, the inner loop was vectorized successfully. The average overall runtime of the program was reduced from 42 seconds to 37 seconds, an increase in performance of 12%.

Looking at the performance report revealed that the compiler thinks accessing the vectors $\vec{p}_j$ and $\vec{F}_j$ is unaligned. Telling the compiler that these are actually aligned accesses (with a stride of n) with the `!dir$ vector aligned` compiler hint did not increase the performance. All other array and matrix accesses are automatically aligned.

One major part of the optimization strategy was reducing loops. The last step to reduce the amount of loops in the program to the minimal amount was to fuse the outer loop of computing the pairwise forces (Algorithm 3, line 1) with the update loop (Algorithm 3, line 7). This destroys the vectorization of the update loop while reducing the amount of iterations performed by the program. But instead of unrolling the dimensional-wise vector operations (Algorithm 3, lines 8–10), they were vectorized by the compiler. Fusing the two loops did not increase the performance. Algorithm 4 shows the structure of the final version of the update operation executed for each time step of the simulation.

At this point all dimensional-wise operations were replaced with particle-wise. The dimensions of the matrices containing the particle vectors are still $n \times 3$, so the dimensions of the particle vectors are stored contiguously in memory. This means particle-wise operations are strided with a stride of n, reducing the utilization of the cache (see Section 4.3).

The last part of the rewriting process tried to increase the cache utilization by transposing the matrices containing the particle vectors to remove the stride from particle-wise operations. Transposing the matrices was done after vectorization was enabled. First enabling vectorization was done, because this way possible alignment issues with the transposed version are exposed. Vectorization works best with aligned base pointers. For Cirrus's Broadwell processor (see Section 3) the data is best aligned on 64 byte boundaries (Krishnaiyer, 2015). A particle vector is only 24 bytes long. Accessing every particle except the first is not aligned based on the 64 byte boundary. As expected, making particles contiguous in memory reduced the performance by 5%, from 38 seconds to 40 seconds, because accessing the particles is not aligned anymore. The processor can handle strided loads and stores better than it can handle unaligned accesses.

Algorithm 4 : final computation per time step

1: **for** i=1,…,n **do**
2: **for** j=i+1,…,n **do**
3: $\vec{F}_i' := \vec{F}_i' + f_{i \leftarrow j}$ {see Equation 3}
4: $\vec{F}_j := \vec{F}_j - f_{i \leftarrow j}$ {see Equation 3}
5: **end for**
6: $\vec{F}_i := \vec{F}_i + \vec{F}_i' - vis_i(\vec{v}_i + \vec{w}) + \vec{F}_{i \leftarrow central}$
7: $\vec{p}_i := \vec{p}_i + \Delta_t \vec{v}_i$
8: $\vec{v}_i := \vec{v}_i + \Delta_t \vec{F}_i / m_i$
9: **end for**

4.5 Fifth Phase

The fifth and last phase was again about using the compiler to increase the performance. To this point the program was compiled with the following flags: `-fpp`, `-qopenmp`, `-O3`, `xCORE-AVX2` and `-vec-threshold0`. Only the last three flags are relevant for performance, while the first two just enable features used in the program (Fortran preprocessor and OpenMP). Some more aggressive compiler optimizations were tried to further increase the performance. Table 2 lists the compiler flags tested and their impact on the program's performance.

The best version of the program at this point took on average 37 seconds to complete the simulation. `-Ofast` was tried in phase two. It did not increase performance on the unoptimized version of the program. On the optimized version of the program on the other hand `-Ofast` results in improved performance. Ten seconds are cut off of the simulation's overall runtime. That is an improvement of 27%.

`-Ofast` makes additional performance optimization to the floating point model. Because they worked so well two additional compiler optimizations for floating point speed were tried: `-fp-speculation=fast` and `-fma`. `-fp-speculation=fast` enables floating point speculation which can relax the way floating point operations are carried out (Intel, 2019c). `-fma` tells the compiler to use vectorized fused multiply-add instructions where applicable (Intel, 2019a). Neither enhances the performance of the program.

Next the maximum level of software prefetching was enabled in order to increase the cache hit-rate. This is done with the `-qopt-prefetch=5` flag (Intel, 2019k). This resulted in a 5% performance improvement, taking away another second of average overall time.

Lastly the compiler was told to optimize the memory structure of the program. This was done with the `-qopt-mem-layout-trans=3` and the `-pad` flags (Intel, 2019j,i). Neither benefited the program.

Phase five was the last phase of the optimization process. Both maintainability and performance of the program were significantly improved. Approximately 20% of the source code was removed. A test setting which enables regression tests was added. The program was successfully vectorized and the memory usage was cut down drastically. The unoptimized version of the program took on average 1270 seconds to finish the simulation described in Section 3. The final version took 26 seconds on average. 98% of the time the original program took for the simulation was successfully removed.

5. Discussion

This section will discuss the most remarkable observations made during the optimization process. Most remarkably is the fact that well readable, idiomatic and modern Fortran code that uses high level features like array slicing results in a very well performing program which is also easy to maintain. The biggest performance gains were made by properly

Optimization	∅ overall time	∅ superstep time	+/ − %	Status
`-Ofast`	27s	5s	27%	Improvement
`-fp-speculation=fast`	27s	5s	0%	Invariant
`-fma`	27s	5s	0%	Invariant
`-qopt-prefetch=5`	26s	5s	5%	Improvement
`-qopt-mem-layout-trans=3`	26s	5s	0%	Invariant
`-pad`	26s	5s	0%	Invariant

Table 2: Compiler flags tried during the fifth phase of optimization (compare Table 1).

utilizing the compiler to make optimizations. The biggest performance benefit observed of well written code was the fact that it enables the compiler to properly optimize.

Table 3 shows the average overall performance of the best version of the program after each optimization phase. Clearly the biggest performance gains were made in phase two. Phase two enabled compiler optimization in the first place. In phases three and four the program was rewritten to be more maintainable and faster. A lot of memory was saved, as was runtime. Compared to the unoptimized version of the program which was used in phase two, another performance gain of 40% was achieved by rewriting the program and enabling vectorization. But more remarkably are the results of phase five. The fifth phase was again about utilizing the compiler to increase the performance further. `-Ofast` was tried in phase two. It did not increase the performance of the unoptimized version of the program (see Section 4.2). `-Ofast` on the optimized version on the other hand resulted in a performance gain of 27% (see Section 4.5). Combining the raw speed gains of the optimized version with the speed gains it enabled by allowing the compiler to properly optimize, the optimized version gains 58% on the unoptimized version of the program. This is a remarkable number and validates the rewriting of the program as successful from a performance perspective.

Like stated above, not only is raw performance considered during the optimization process, but also maintainability. Maintainability is not as easy to measure as is performance. The best measurement that can be presented is the fact, that the optimized version is 20% smaller concerning the lines of code than is the original program. The build process is also much simplified and only a single source file is needed to the previous four. Again does the optimized version leverage (relatively) modern Fortran features like modules over old-fashioned ones like `COMMON` blocks to increase its maintainability (see Section 4.1).

6. Conclusion

The optimization process described in Section 4 is considered to be successful from both perspectives: performance and maintainability. The original version of the program took on average 1270 seconds to complete the simulation described in Section 3. The final

	Original	Phase one	Phase two	Phase three	Phase four	Phase five
$\emptyset$ overall time	1270s	1110s	62s	42s	37s	26s
$+/-$ %	-	13%	94%	32%	12%	30%

Table 3: Benchmark results after each optimization phase.

version of the program took only 26 seconds to complete the same simulation. That is a performance gain of 98%.

These gains were mainly due to the proper utilization of the compiler. The most remarkable observation made during the optimization process was the fact that idiomatic and modern Fortran code makes it easy for the compiler to optimize, while also increasing the maintainability of the program.

This paper only considered serial performance. The obvious next step to further increase the program's performance and enable much bigger simulations would be to parallelize it to multiple threads (using e.g. OpenMP) or even distribute it to multiple nodes (e.g. with MPI).

References

Mark Buxton. Haswell New Instruction Descriptions Now Available!, 2011. URL `https://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available/`.

Matt Chiu, Md Khan, and Martin Herbordt. Efficient calculation of pairwise nonbonded forces. pages 73 – 76, 06 2011. doi: 10.1109/FCCM.2011.34.

EPCC. Cirrus, 2020a. URL `https://www.cirrus.ac.uk`.

EPCC. Cirrus Hardware, 2020b. URL `http://www.cirrus.ac.uk/about/hardware.html`.

Richard P. (Richard Philipps) Feynman. *The Feynman lectures on physics*. Addison-Wesley Pub. Co., Reading, Mass. ; London, 1963. ISBN 0201020106.

David Harper and L. M. Stockman. COMMON Blocks, BLOCK DATA and EQUIVALENCE, 2020. URL `https://www.obliquity.com/computer/fortran/common.html`.

Intel. Intel Fortran Compiler 18.0 for Linux, 2018. URL `https://software.intel.com/en-us/articles/intel-fortran-compiler-180-for-linux-release-notes-for-intel-parallel-studio-xe-2018`.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – fma, 2019a. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-fma-qfma.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – fp-model, 2019b. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-fp-model-fp.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – fp-speculation, 2019c. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-fp-speculation-qfp-speculation.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – fpp, 2019d. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-fpp.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – ipo, 2019e. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-ipo-qipo.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – prec-div, 2019f. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-prec-div-qprec-div.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – O, 2019g. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-o.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – Ofast, 2019h. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-ofast.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – pas, 2019i. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-pad-qpad.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – qopt-mem-layout-trans, 2019j. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-qopt-mem-layout-trans-qopt-mem-layout-trans.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – qopt-prefetch, 2019k. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-qopt-prefetch-qopt-prefetch.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – qopt-report, 2019l. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-qopt-report-qopt-report.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – vec-threshold, 2019m. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-vec-threshold-qvec-threshold.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – x, 2019n. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-x-qx.

Intel. Intel Fortran Compiler 19.1 Developer Guide and Reference – xHost, 2019o. URL https://software.intel.com/en-us/fortran-compiler-developer-guide-and-reference-xhost-qxhost.

Intel. Intel VTune Profiler 19, 2020. URL https://software.intel.com/en-us/vtune.

Rakesh Krishnaiyer. Data Alignment to Assist Vectorization, 2015. URL https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.

Parviz Moin. Modules, 2002. URL https://web.stanford.edu/class/me200c/tutorial_90/09_modules.html.

OpenMP Architecture Review Board. OpenMP application program interface version 4.5, 2015. URL https://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

David A. Patterson. *Computer organization and design : the hardware/software interface*. Elsevier Science & Technology, Saint Louis, fifth edition.. edition, 2014. ISBN 9780124078864.

Adrian Sandu. Free vs. Fixed Formats, 2001. URL http://people.cs.vt.edu/~asandu/Courses/MTU/CS2911/fortran_notes/node4.html.

C. K. Shene. Fortran Formats, 2020. URL https://pages.mtu.edu/~shene/COURSES/cs201/NOTES/chap05/format.html.

User389. Intel Fortran Compiler: tips on optimization at compilation, 2020. URL https://scicomp.stackexchange.com/questions/265/intel-fortran-compiler-tips-on-optimization-at-compilation.