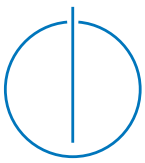
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Interdisciplinary Project (IDP)

Performance Modelling for Auto-Tuning of Molecular Dynamics Simulations

Sascha Sauermann



DEPARTMENT OF INFORMATICS

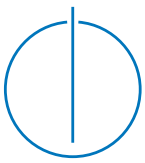
TECHNICAL UNIVERSITY OF MUNICH

Interdisciplinary Project (IDP)

Performance Modelling for Auto-Tuning of Molecular Dynamics Simulations

Performanz Modellierung für Auto-Tuning von Molekulardynamiksimulationen

Author:	Sascha Sauermann
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	M.Sc. Fabio Gratl
Submission Date:	January 20, 2019



I confirm that this interdisciplinary project (idp) is my own work and I have documented all sources and material used.

Munich, January 20, 2019 Sascha Sauermann

Abstract

Computing pairwise short-range particle interactions is a time-consuming and unavoidable part of Molecular Dynamics simulations. For the parallelization of the Linked Cell Algorithm there exist different traversal patterns having varying performance depending on the parameters and state of the simulation.

We use Extra-P to create performance models from benchmarks and use them to apply auto-tuning in ls1-mardyn. Thus we predict the optimal traversal given the current state of the simulation and use it until reevaluation.

For single-parameter models the prediction is quite good but the prediction quality declines fast for multi-parameter models. Additionally, the time required for benchmarks also gets infeasible quickly when adding more and more dimensions to the parameter space because the number of samples required grows exponentially.

We thus model the runtime depending only on the density. Auto-tuning with this simple model on each MPI rank separately already delivers close to optimal performance.

Contents

1. Introduction

Molecular dynamics (MD) simulations require a huge computational effort and thus take a lot of time even on large clusters. It is therefore very important to optimize the performance as even small improvements can accumulate for a simulation that takes millions of steps. The most computing-intense part of a MD simulation is the calculation of particle interactions between them.

There exist many different methods for traversing the simulation domain and calculating these interactions in parallel. As the performance of those traversals is depending on the parameters and state of the simulation, there is no optimal traversal for all possible simulations.

Our goal was to model the performance of traversals depending on the simulation parameters and state. We want to predict the optimal traversal at any time within the simulation and switch to it. This is called auto-tuning.

The following chapters start with a brief description of the theory of generating performance models for software. Then follows an overview over the tools that were used or created for this work. We conclude with an evaluation of the model quality and a performance comparison of the auto-tuning implementation in ls1-mardyn versus fixing a traversal for the whole simulation.

2. Theory

2.1. Performance Models

A performance model is an analytical model of a system or software that defines a relation between input parameters and performance metrics. Possible metrics are for example the speedup of computations, time to solution, the consumed resources or response times. In the area of high-performance computing, typical parameters include thread-, process- and node counts.

Performance models can be used for detecting bottlenecks and scalability bugs in parallel software or for kernel selection. A kernel is the performance dominating part of the software for increasing scale, like the force calculation in molecular dynamics simulations.

The main problem for generating models is the infinite search space of possible model functions. Fortunately, most performance models can be sufficiently represented by only finite number of powers and logarithms of the parameters. This follows from the way computer algorithms are constructed. [1]

Thus we can express all performance models in the so-called *performance model normal form* (PMNF):

$$f(x) = \sum_{k=1}^{n} c_k \cdot x^{a_k} \cdot \log_2^{b_k}(x)$$

Therefore we only have to find good values for the $c_k \in \mathbb{R}$ and the exponents $a_k \in A$ and $b_k \in B$ (with $A, B \subset \mathbb{Q}$). According to the authors of [1], neither the number of terms n, nor the sets A and B need to be very large or random to get a good fitting model.

By creating a set of plausible values for A and B we can thus limit the number of possible *model hypotheses* for a fixed number of terms n. We get one hypothesis $h_n^{(a,b)}$ where $a = (a_0, a_1, \ldots, a_n)$ (equally for b) for each possible selection of our values from A, B and therefore a set of hypotheses H_n.

2.2. Hypothesis Fitting

Fitting hypotheses to the performance data and then selecting the best hypothesis is an important step in modeling. For this we first need measurements y of the performance metric, that we want to model, for different values x of our parameter space. We get therefore a collection of measurements (x_i, y_i) we call our data.

The first step is to split the data into a training and an evaluation set. One way to assign the data to those sets is randomly selecting a fixed percentage for the evaluation set and keep the remaining points as the training set.

We fit the hypotheses on the training data and compare them by their performance on the evaluation data. When we just train on the complete data, the model might fit to perfectly and will not generalize well to unseen data. This is called *overfitting*. To decide which is the better model we evaluate them on the evaluation set and take the one that performs the best.

Fitting a hypothesis to the data involves minimizing the residuals $r_i = y_i - f_c(x_i)$, the differences between the estimated values $f_c(x_i)$ and the observed values y_i. This can be done by various different methods, for example by minimizing the *sum of least squares*:

$$\sum_{i=1}^{m} r_i^2 = \sum_{i=1}^{m} (y_i - f_c(x_i))^2$$

For the most non-linear hypotheses there exists no closed form solution and thus results in an numerical optimization problem. [4]

Next we validate the performance of the fitted model by calculating the error between the estimated values and the observed values on the evaluation set. One possible error estimator is the *mean squared error* (MSE):

$$\frac{1}{m} \sum_{i=1}^{m} (y_i - f_c(x_i))^2$$

For a more accurate error estimate some form of cross-validation can be used. For this the hypothesis is fitted multiple times, choosing a different part of the data for the evaluation set each. The error estimate of the different validation steps are then averaged.

A popular form of cross-validation is *k-fold cross-validation*. For this the data is split into k equal sized chunks. For each step, one of them is chosen as the evaluation set and the remaining chunks as the training set. After k steps, every chunk was once the evaluation set and we can average the error estimates.

Special forms of *k-fold cross-validation* are *leave-one-out cross-validation* ($k = m$) and *hold-out cross-validation* ($k = 2$).

The former is best used for a small amount of data points as every single point will be the evaluation set once. This allows for a high accuracy but also takes the longest time.

The latter uses just two chunks which results in fast evaluation bur lower accuracy. The Authors of [1] suggest that this method provides the best accuracy and time trade off. The two chunks can easily be created by assigning adjacent points to different chunks.

The best model hypothesis is then chosen as the one with the smallest error on the evaluation set.

2.3. Coefficient of Determination: Adjusted R^2

For regression models the R^2 coefficient is a measure that indicates what proportion of the variation in the data of sample size m is explained by the model. It is calculated as the fraction of the *sum of squared errors* (SSE) and the *total sum of squares* (SST):

$$R^2 = \frac{\text{SSE}}{\text{SST}} = \frac{\sum_{i=1}^{m}(y_i - f(x_i))^2}{\sum_{i=1}^{m}(y_i - \sum_{j=1}^{m} y_j)^2}$$

The problem with using R^2 for comparing the fit of two models is that it monotonously approaches 1 when adding more and more terms. This is fixed by adjusting it to the number of free variables p. This gets us the adjusted R^2:

$$\overline{R}^2 = 1 - \frac{(1 - R^2)(m - 1)}{m - p - 1}$$

It is thus a measure in $[0, 1]$ how good a model fits the given data adjusted for the number of variables in the model, where a value of 1 is the perfect fit.

Increasing the number of terms therefore can have two effects on the value of the $\overline{R}^2$:

- If the additional term does improve the model, the $\overline{R}^2$ will increase.
- If the new term does not increase the explanatory power of the model, the $\overline{R}^2$ will decrease.

2.4. Extra-P

Extra-P is a tool for performance modeling which primary goal is detecting scalability bugs. It is designed to model performance measurements like e.g. FLOPS or communication time depending on the number of processes. Creating models with other

parameters is possible, though. It can either be used as an extension to Scalasca using the automatic creation of performance traces or stand-alone with self-supplied data points.

Extra-P creates models by an iterative refinement approach using *hold-out cross-validation*. We start with the PMNF that has only one term, i.e. $n = 1$ and perform the following steps (assuming $\overline{R}_0^2 = 0$):

1. Create the model hypotheses H_n from a set of plausible values
2. Find the best hypothesis h_n^* by computing the best fitting values for c and performing k-fold cross validation
3. Compute the $\overline{R}_n^2$ for the best hypothesis h_n^*
4. Compare $\overline{R}_n^2$ with $\overline{R}_{n-1}^2$
   - If fit increased ($\overline{R}_n^2 > \overline{R}_{n-1}^2$), increase the number of terms $n = n + 1$ and do another iteration
   - If fit decreased ($\overline{R}_n^2 <= \overline{R}_{n-1}^2$), choose h_n^* as the final model

3. Implemented Tools

The next sections contain a high-level description of the used and implemented tools. Detailed explanations and user guides can be found in the respective Github repositories[10][9].

3.1. Benchmarks of ls1-mardyn with Jube

To create performance models of ls1-mardyn we, first of all, need some performance measurements. We use the benchmarking environment Jube[5] to automate the creation of input files and job scripts as well as extracting the measurements from ls1-mardyn's output.

To start simply we model only up to three parameters: density (ρ), cutoff radius (cr) and the number of Lennard-Jones centers (lc). We then chose five to ten values in a sensible range for each one. Our parameter space is then the cartesian product of those values i.e. a full grid. Additionally, we vary the applied cell traversal and perform each measurement multiple times. This increases our parameter space by another two dimensions.

For each of those grid points, Jube creates a ls1-mardyn input file and a job script for the specified cluster.

After running all jobs of the benchmark, we use the parsing feature of Jube to get a CSV file containing different performance measurements taken by ls1-mardyn. We focus on the "Time to solution" measure for our models.

3.2. Job Combination

Using Jube we create a huge number of jobs with a very short runtime. As short-running jobs produce a large scheduling overhead on shared clusters, they should be combined to fewer long-running jobs.

For this purpose, a job combination script[9] was developed. It parses existing job files and combines similar jobs considering restrictions of the cluster like wall time or the maximum number of queued jobs. Combined scripts are then executed sequentially

within a single run. If sub-jobs fail or combined jobs time-out, a partial restart of the unfinished sub-jobs with adapted wall times is possible.

3.3. Converter for CSV to Extra-P Input

The results of the benchmarks are stored as a CSV file and need to be converted to an input file for Extra-P to create models. Our converter allows selecting which parameters should be part of the model and which of the available metrics to model. The parameters that are not used for modeling can be fixed to a specific value e.g. to select the traversal that should be modeled.

This allows to create multiple models from one benchmark run and reuse as many data points as possible.

3.4. Automatic Model Creation from CSV Files

As we want to create multiple models for the different traversals automatically, we created yet another script for this. It takes the benchmark CSV file, filters and converts it to Extra-P input and then runs the matching Extra-P modeling process.

The output of Extra-P consists of the model function, the adjusted R^2 and depending on the input of the RSS and some means. The notation of the model does not follow typical conventions though and thus cannot be processed by many popular parsers for mathematical expressions. Problematic are for example notations like $log^2(p)$ or $-1 + -2x^1 + -3x^2$. We therefore standardize the notation and simplify it by removing unnecessary exponents, etc.

As Extra-P only uses a single thread for modeling, different models are created in parallel to minimize the time needed.

3.5. Model Visualization and Comparison

As we now have an easy way to create different models we want to evaluate and compare them. For this a web-based visualizer was created using Dash and Plotly.[8] It can be run remotely while forwarding the port via ssh to connect to it with the local browser. This allows a responsive interface while still using fast CPUs for the modeling process.

In this visualizer (see Figure 3.1) the parameters for the conversion described in section 3.3 can easily be selected from the values occurring in the benchmarks. The created models are then plotted against the real data to allow a visual evaluation of the fit and comparison of the different traversals.

Multi-parameter models are also compared with a combination of single-parameter models that are created assuming independence of the parameters.

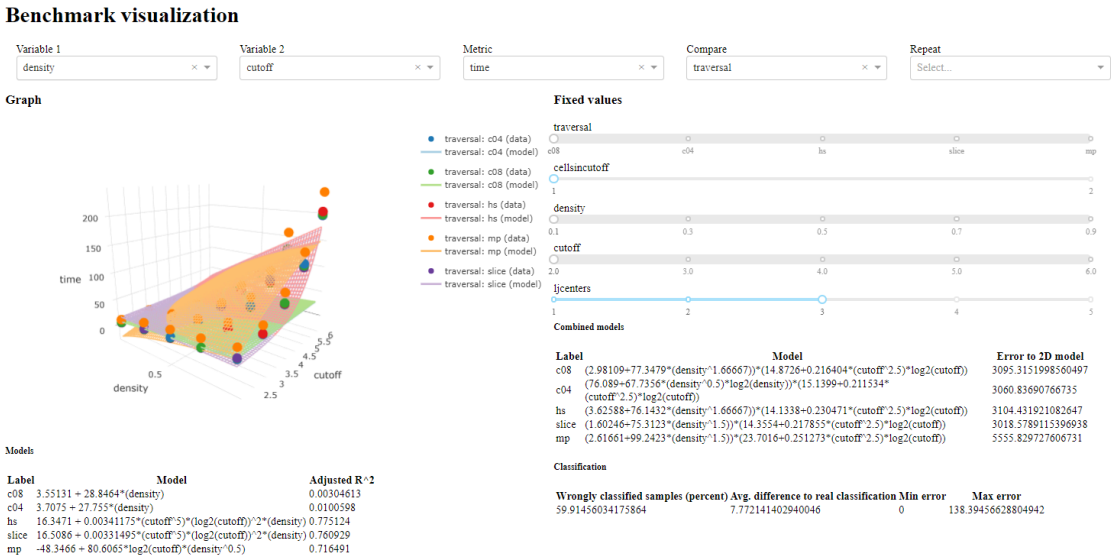


Figure 3.1.: **Screenshot of visualizer**: The visualizer is a simple Dash application that allows for an easy exploration of the benchmarks. It creates models for the selected parameters, metric, etc. and plots them against the original data.

3.6. Auto-Tuning in ls1-mardyn

As we can now automatically create models from benchmarks, we want to use them to predict the fastest traversal for a given state in ls1-mardyn[7]. ls1-mardyn already provides a traversal tuner class that allows switching the active one. This selection is currently read from the input file for the simulation.

We extended the options in the input for activating auto-tuning and selecting the number of steps between each reevaluation. At the start and at each reevaluation step, the models for each applicable traversal are evaluated and the traversal with the fastest predicted time is selected.

For traversals that have the same MPI communication scheme, different ranks can select different local traversals.

4. Results

The benchmarks of ls1-mardyn were performed on the CooLMUC3 cluster of the Leibniz Supercomputing Center (LRZ), if not specified differently, on the most recent version available at the time. The units of the parameters are according to Table 1 of [7].

All runs were performed with a static regular domain decomposition. The data points used for the modeling are gathered by running ls1-mardyn with each configuration from the parameter set and using the output from the internal timers as the result. Each run was performed three to five times to get multiple data points for each configuration. This reduces the influence that variance in the measurements has on the model.

4.1. Cluster Configuration

The LRZ has multiple linux clusters that are available for our computations. [6] We used the CooLMUC2 and the CooLMUC3 cluster for running the benchmarks. See Table 4.1

	CooLMUC2	CooLMUC3
Architecture	Haswell	Knights Landing
CPU	Intel Xeon E5-2697 v3	Intel Xeon Phi 7210F
Frequency	2.6 - 3.6 GHz	1.3 - 1.4 GHz
Cores per Node	28	64
Memory per Node	64 GB RAM	96 GB RAM / 16 GB HBM
Interconnect	FDR14 InfiniBand	Intel Omni-Path OPA1

Table 4.1.: Specifications of the CooLMUC2 and CooLMUC3 cluster

4.2. Single-Parameter Models

For a first evaluation of the approach we have a look at single-parameter models. The parameters for the benchmarks are chosen as follows:

- Cutoff radius (cr) in $[2.0, 6.0]$

- Density (ρ) in $[0.1, 0.8]$
- Lennard-Jones centers (lc) in $[1, 5]$
- Traversal in [Full Shell, Half Shell, Midpoint, C08, C04]

The resulting models for time-to-solution look similar this (example for C08, cr=5, ljc=2):

$$t(\rho) = 2.689 + 75.7957 \cdot \rho^{1.66667}$$

A first visual evaluation of the model against the data (see Figure 4.1) confirms that most of the single-parameter models fit very well with an adjusted R^2 of close to 1.

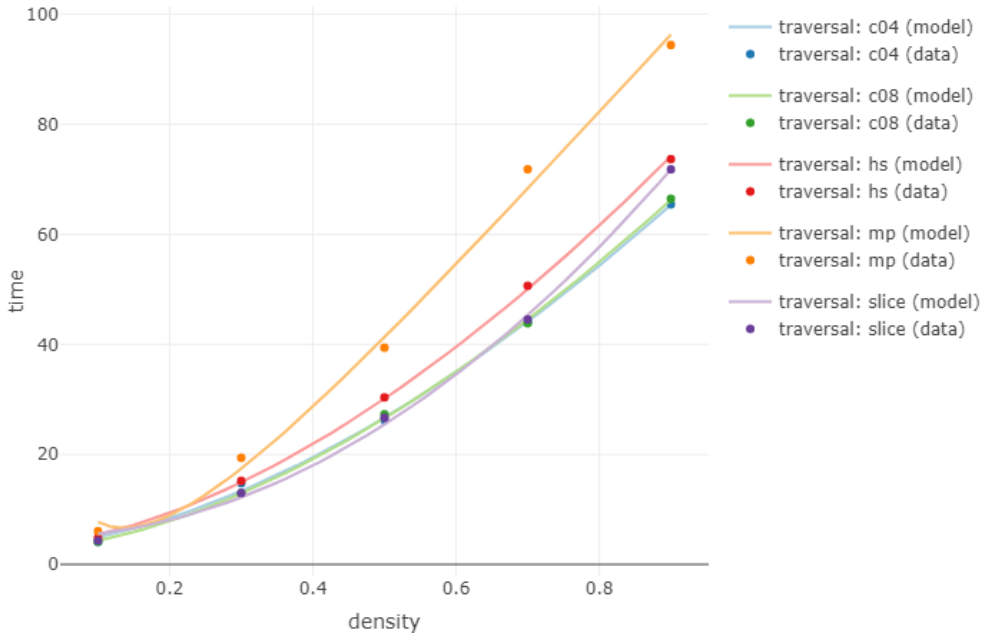


Figure 4.1.: **Single-parameter models for runtime for a given density**: This plot shows the data and the generated models for varying density. The cutoff is fixed to 5 and number of LJC to 2. The models fit the data quite good.

4.3. Multi-Parameter Models

Using the same measurements as in section 4.2, we can now use Extra-P to create multi-parameter models. Unfortunately most of them do not fit our data at all. The

respective adjusted R^2 can be as low as 0.01. Some models fit reasonably well with an adjusted R^2 of around 0.75 and look similar to this (HS, ljc=2):

$$t(\mathrm{cr}, \rho) = 13.846 + 0.00205082 \cdot \mathrm{cr}^5 \cdot (\log_2(\mathrm{cr}))^2 \cdot \rho$$

We can see this easily when plotting the data and the models (see Figure 4.2).

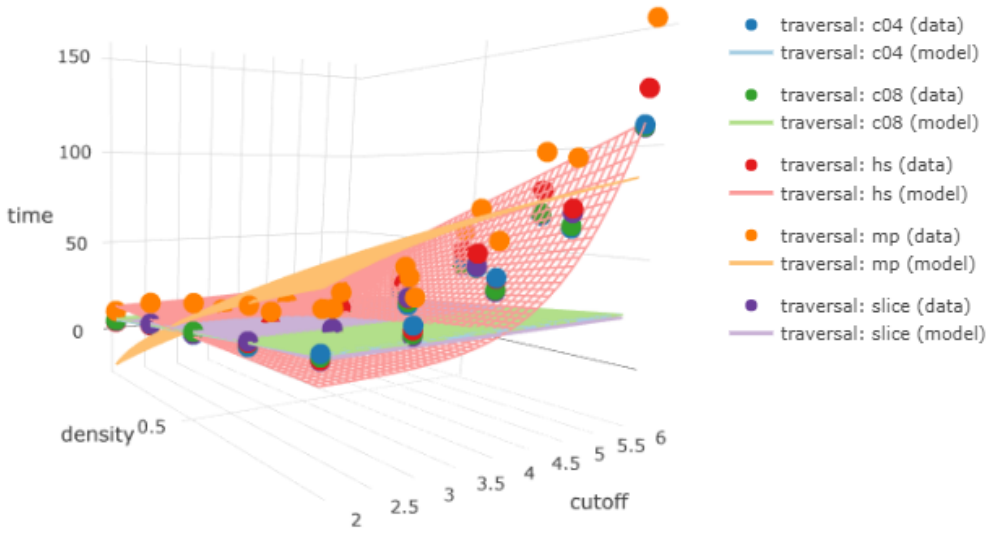


Figure 4.2.: **Multi-parameter models for time(density, cutoff)**: This plot shows the data and the generated models for varying density and cutoff. The number of LJC is fixed to 2. Some models like the the one for the HS traversal fit the data reasonably well, but others do not fit the data at all.

4.4. Missing Metric for Homogeneity

As discussed in the section 4.2 and section 4.3, we can create quite good models for one and sometimes even two parameters. The next step would be to use them for an automatic traversal selection in ls1-mardyn. We can consider two different approaches for a first implementation:

- Statically selecting a traversal at the start depending on time invariant parameters
- Re-selecting the traversal every n steps depending on time variant parameters

As the first approach would be quite simple and there would not be much to explore further, we chose the second one.

Neither the cutoff radius nor the number of Lennard-Jones centers change over time, but we expect the local density of the domain to change. Unfortunately, our density metric measures the global density and is thus the same over the whole simulation.

One may create a scenario that has an in- or outflow of particles and therefore a change in global density, but most real applications assume a closed or periodic domain.

We expect that the performance of the traversals varies for different distributions of the particles in the domain, e.g. uniform distribution against particles clumping together at one place. Thus, we looked at modeling a metric for the homogeneity of the distribution instead of the density.

Considering multiple options, we found that simple metrics like the variance of particles per cell do not work well. The best working metric we found is based on the Voronoi partition of the domain as this models the empty space between particles. This partitioning is basically the solution to the *closest post office problem*. The metric is calculated as follows:

- Compute the Voronoi partitioning of the domain such that each cell contains exactly one particle.
- Compute the area (volume for 3D) of each of those cells
- The metric for the homogeneity is then the variance of those areas

Figure 4.3 shows a Voronoi diagram of a uniform particle distribution and Figure 4.4 of a particle cluster for a 2-dimensional periodic domain (i.e. a torus). While the uniform distribution leads to cells with about the same area, the particle cluster produces a large variance of area sizes.

Although this works quite well as a metric, it is relatively expensive to compute as a 3-dimensional Voronoi partition has a worst case complexity of $\mathcal{O}(n^2)$ for n particles. [3] Implementing this in ls1-mardyn is also not a small task because periodic boundary conditions have to be considered.

4.5. Rank-Local Auto-Tuning

A different way to circumvent the problem of the fixed global density, is to use the fact that different MPI ranks process unique parts of the domain in ls1-mardyn. Therefore, we can have an in- and outflow of particles from the perspective of each rank and thus a changing rank-local density. Thus, we can select a different traversal for each rank that is time-variant on this density.

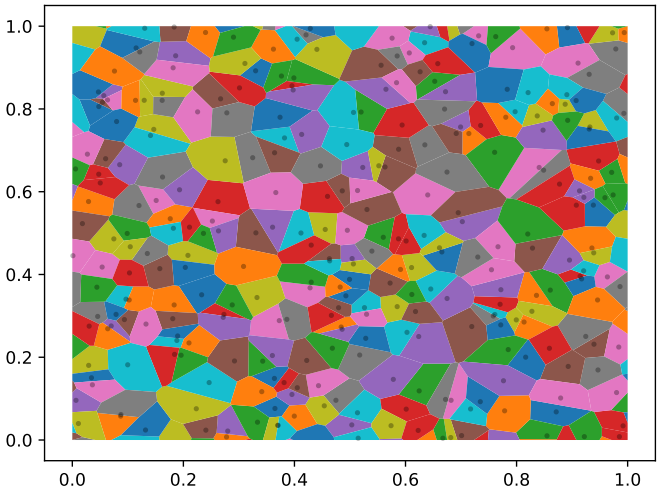


Figure 4.3.: **Voronoi partitioning for a uniform particle distribution**: This figure shows a uniform particle distribution for a periodic 2-dimensional domain $[0,1] \times [0,1]$. The variance of the areas around each particle is quite low and thus this distribution leads to a metric close to zero.

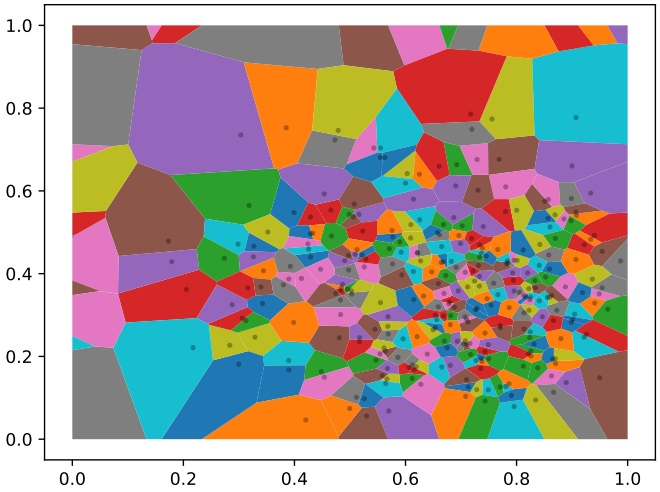


Figure 4.4.: **Voronoi partitioning for a particle cluster**: This figure shows Beta(α, β) distributed particles with $\alpha = (4,3)$ and $\beta = (2,5)$ for a periodic 2-dimensional domain $[0,1] \times [0,1]$. Such a particle cluster leads to higher variances in the areas of the Voronoi diagram and thus larger metrics.

To get measurements for the performance of a single MPI rank, we ran different benchmarks on the CoolMUC2 cluster with 8 OpenMP threads. The three selected traversals have the same MPI communication scheme.

The parameters were selected as follows:

- Cutoff radius (cr): 2.5
- Density (ρ) in $[0.01, 0.9]$
- Lennard-Jones centers (lc): 1
- Traversal in [C08, C04, Sliced]

Instead of measuring the overall time of the simulation, here we measured the time the force calculation in each time step took. We can now compare the performance of an auto-tuning run with the performance of the three traversals on a per rank basis. We chose a simple scenario for this purpose that results in large changes in the rank local densities:

- Cutoff radius (cr): 4.0
- Lennard-Jones centers (lc): 1
- Sphere of particles with a density of 0.85
- Rest of domain has density 0.0065
- 8 MPI ranks with 8 OpenMP threads each

The sphere is placed as such that it completely covers rank 0 with a few particles spilling into the adjacent ranks. Over time it diffuses until we reach a uniform distribution over the complete domain.

We now can plot the time each simulation step takes for each rank. For visual simplicity and to get rid of the outliers, we plot a generalized additive model (GAM) smoothing of the real data.

As seen in Figure 4.5, the time in each step is relative to the expected density distribution. Although the auto-tuning run only rarely selects the optimal traversal (see Figure 4.5 and Figure 4.6), the performance of the selected traversal is always very close to the optimal one (see Figure 4.7).

We can also look at the absolute (see Figure 4.7) and relative differences (see Figure 4.8) between the auto-tuning run and the optimal traversal at any step. This validates that the auto-tuning run is only up to 13ms slower each step. When also looking at the relative differences, we can see that such small differences can mean that

a step takes up to 23% longer. Comparing this result with earlier plots shows though that those huge relative differences only occur when the times per step get quite small anyways.

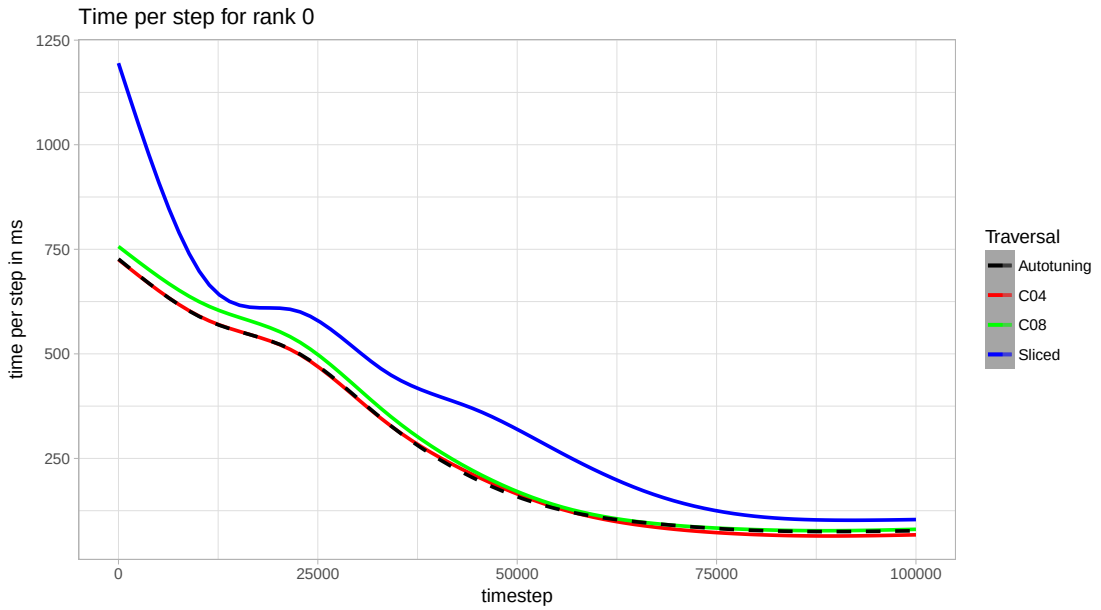


Figure 4.5.: **Time per timestep: Rank 0**: As the particles sphere starts out in this rank we can see that each step takes quite long in the beginning. As the particles diverge, the times normalize to about 100ms in the end for all traversals. The auto-tuning run starts with C04 and switches to C08 after about 55000 steps although C04 is still a little bit better.

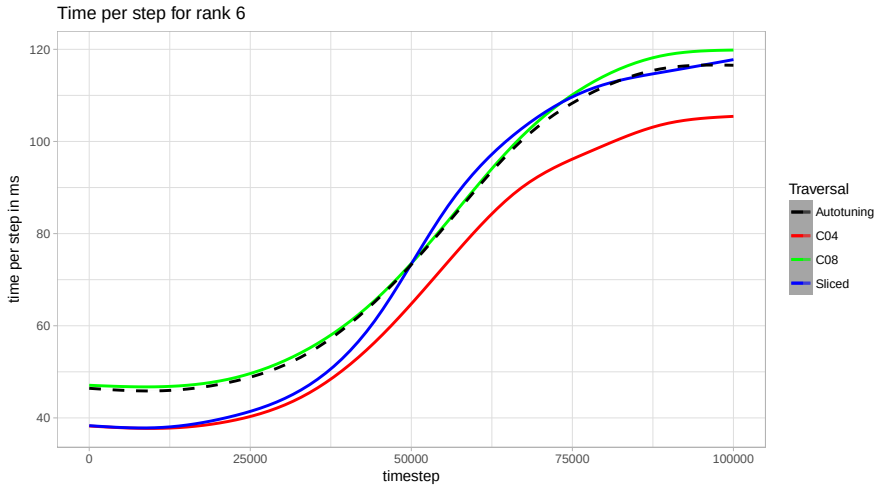


Figure 4.6.: **Time per timestep: Rank 6**: For other ranks we see the times per step increasing as they contain more and more particles. In case of rank 6, we interestingly never select the fastest traversal in the auto-tuning run but instead start with C08 as the slowest one and switch then to sliced at about step 75000. We are always quite close to the optimal traversal though.

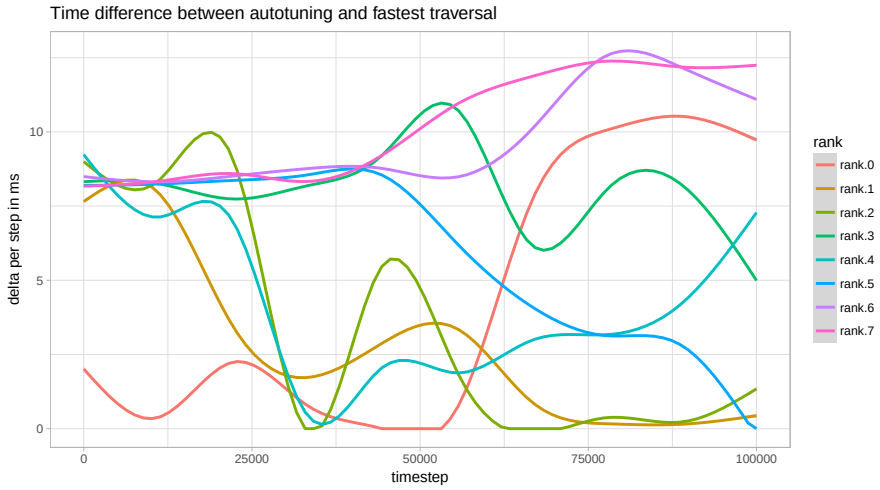


Figure 4.7.: **Time difference between auto-tuning and the fastest traversal**: Plotting the difference between the time per step for the auto-tuning run and the fastest traversal in each step, shows that the absolute difference is always quite small (less than 13ms). In some parts (e.g. around time step 50000) we also select the optimal traversal.

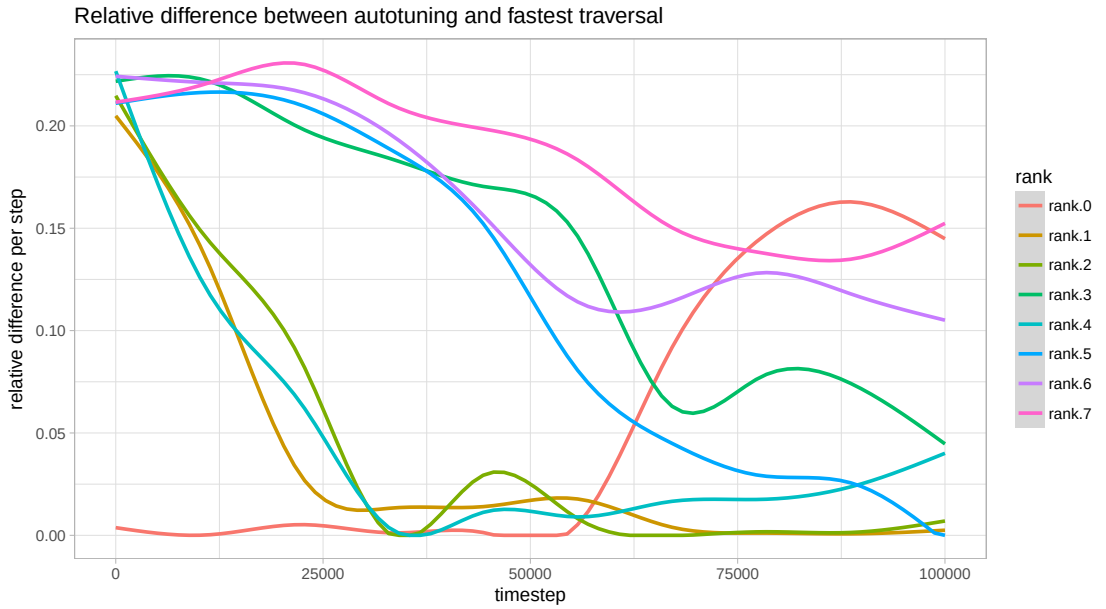


Figure 4.8.: **Relative difference between auto-tuning and the fastest traversal**: Similar to Figure 4.7, we can plot the relative difference to the fastest traversal. This shows that the auto-tuning run was in some parts up to 23% slower.

5. Conclusion and Future Work

The generation of performance models for ls1-mardyn with Extra-P works quite well for single-parameter models. When adding more parameters in the modeling process, the prediction quality of the resulting models starts to decline quite fast tough. Additionally, the number of benchmarks we must run to generate those also gets infeasible quite fast. Thus, we cannot utilize multi-parameter models to make meaningful predictions.

Using a performance model to predict the optimal traversal for each MPI rank (section 4.5) shows that this approach for auto-tuning can lead to promising results. Our prediction is often quite close to the optimal traversal but if we could model more parameters the prediction quality probably would increase significantly.

One should explore more sophisticated classification algorithm that works better with many parameters for predicting the optimal traversal. Additionally, the benchmarking should be improved before attempting to generate data for more than a few parameters as the benchmarks with four varying parameters for this paper already took multiple days.

The main problem here is that we take measurements for the time it takes to execute n steps. This n must be large enough so that the fastest runs (few particles, small cutoff) take at least some seconds. But this leads to the slowest runs (many particles, large cutoff) taking hours. The best way to solve this would be modeling a metric that can be benchmarked in a roughly fixed amount of time.

The metric for the homogeneity of the particle distribution based on the Voronoi partitioning of the domain described in section 4.4 looks promising although difficult to compute in 3 dimensions. There exist more efficient algorithms for 2-dimensional Voronoi partitions though, like Fortunes algorithm ($\mathcal{O}(n \log n)$). [2] An implementation of the metric would also have to take the periodic boundary conditions into account and thus calculate the partitioning on a 4D torus.

A. Appendix

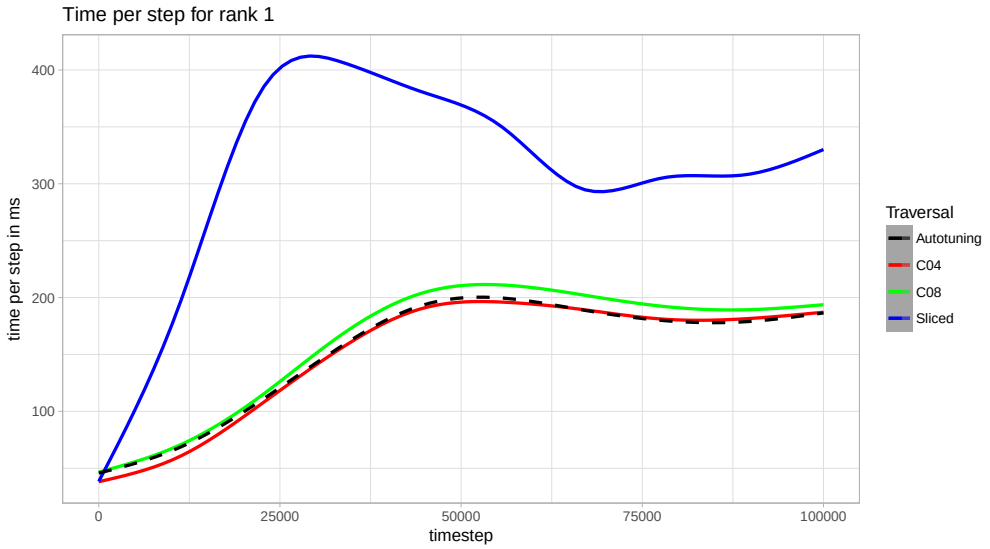


Figure A.1.: **Time per timestep: Rank 1**: see Figure 4.6

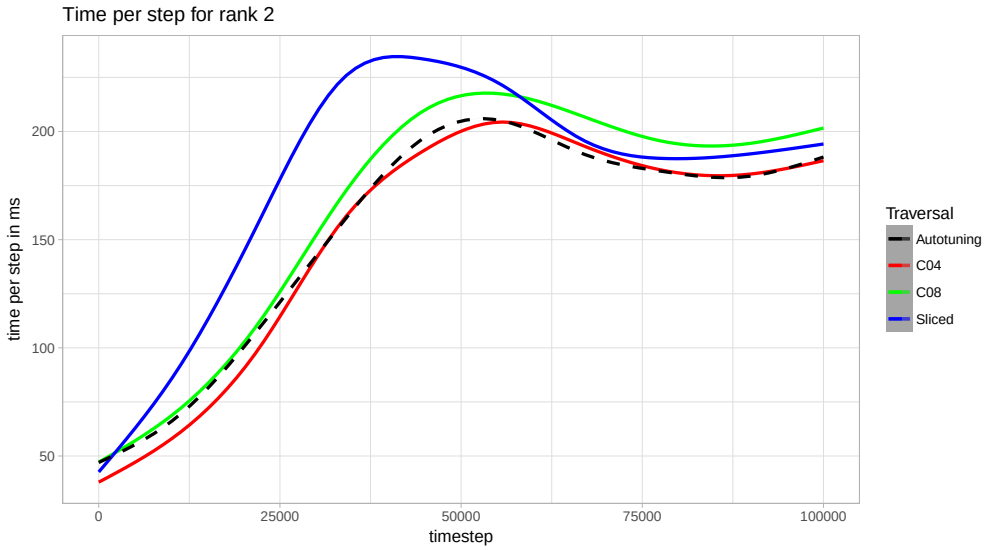


Figure A.2.: **Time per timestep: Rank 2**: see Figure 4.6

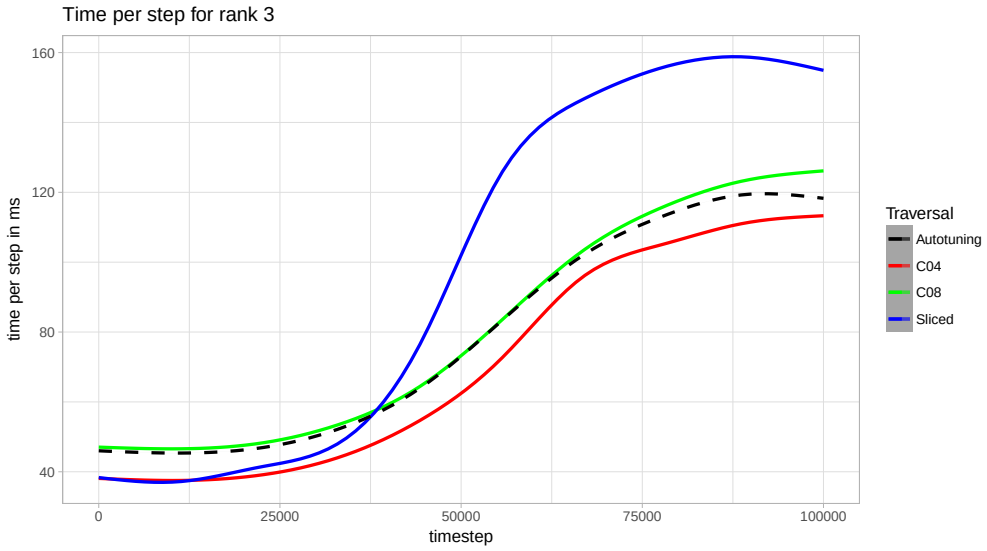


Figure A.3.: **Time per timestep: Rank 3**: see Figure 4.6

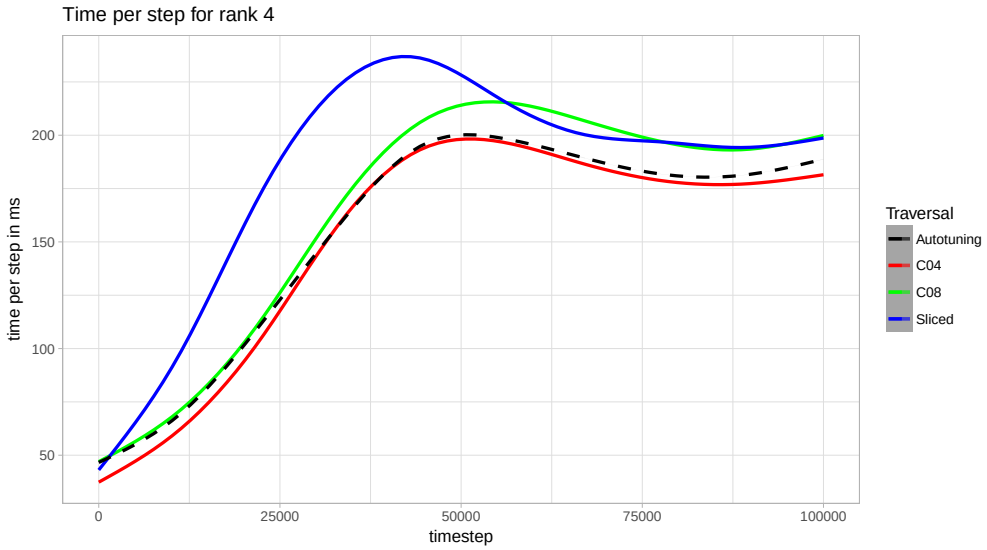


Figure A.4.: **Time per timestep: Rank 4**: see Figure 4.6

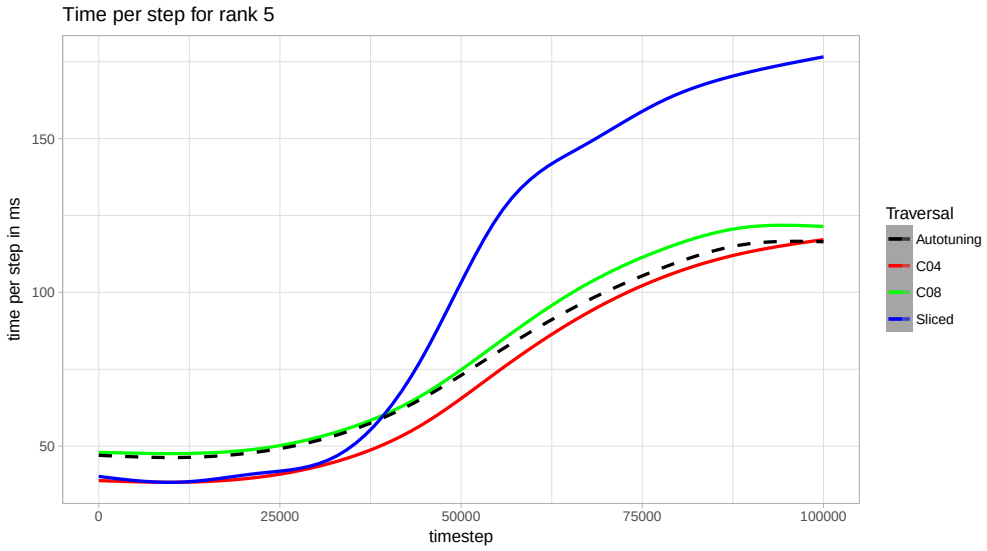


Figure A.5.: **Time per timestep: Rank 5**: see Figure 4.6

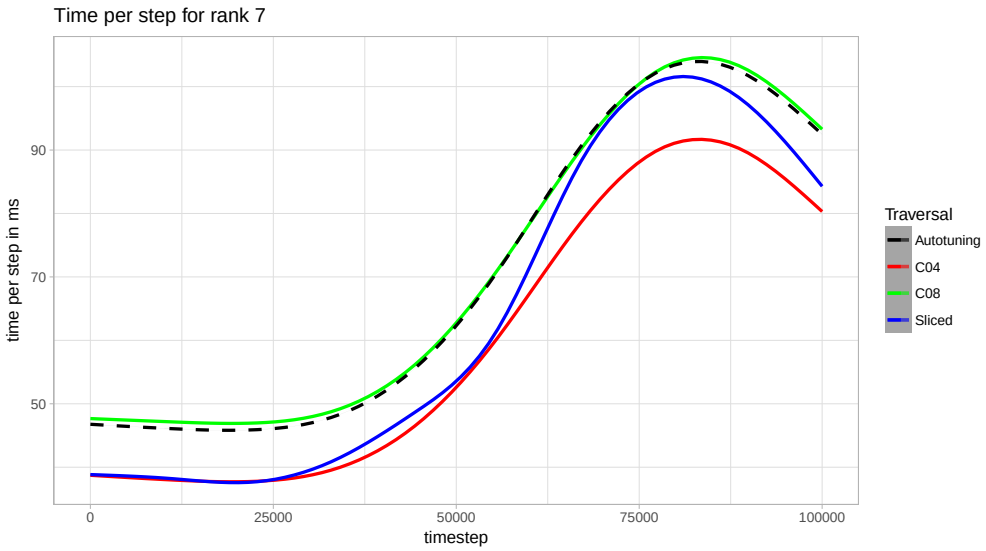


Figure A.6.: **Time per timestep: Rank 7**: See Figure 4.6

List of Figures

List of Tables

Bibliography

[1] A. Calotoiu, T. Hoefler, M. Poke, and F. Wolf. "Using automated performance modeling to find scalability bugs in complex codes." In: *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Nov. 2013, pp. 1–12. DOI: 10.1145/2503210.2503277.

[2] S. Fortune. "A sweepline algorithm for Voronoi diagrams." In: *Algorithmica* 2.1 (Nov. 1987), p. 153. ISSN: 1432-0541. DOI: 10.1007/BF01840357.

[3] M. J. Golin and H.-S. Na. "On the average complexity of 3D-Voronoi diagrams of random points on convex polytopes." In: *Computational Geometry* 25.3 (2003), pp. 197–231. ISSN: 0925-7721. DOI: 10.1016/S0925-7721(02)00123-2.

[4] I. Griva, S. Nash, and A. Sofer. *Linear and Nonlinear Optimization: Second Edition*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2009. ISBN: 9780898717730.

[5] Jülich Supercomputing Centre. *JUBE Benchmarking Environment*. Nov. 26, 2018. URL: https://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/JUBE/_node.html.

[6] Leibniz Supercomputing Centre. *Overview of the Cluster Configuration*. Dec. 15, 2018. URL: https://www.lrz.de/services/compute/linux-cluster/overview/.

[7] C. Niethammer, S. Becker, M. Bernreuther, M. Buchholz, W. Eckhardt, A. Heinecke, S. Werth, H.-J. Bungartz, C. W. Glass, H. Hasse, J. Vrabec, and M. Horsch. "ls1 mardyn: The Massively Parallel Molecular Dynamics Code for Large Systems." In: *Journal of Chemical Theory and Computation* 10.10 (2014). PMID: 26588142, pp. 4455–4464. DOI: 10.1021/ct500169q. eprint: http://dx.doi.org/10.1021/ct500169q.

[8] Plotly. *Plotly*. Dec. 16, 2018. URL: https://plot.ly/.

[9] S. Sauermann. *Repository job-combine*. URL: https://github.com/ssauermann/job-combine/.

[10] S. Sauermann. *Repository md-perfmod*. URL: https://github.com/ssauermann/md-perfmod/.