TweTriS: Twenty Trillion-atom Simulation

Journal Title
XX(X):1–16

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Nikola Tchipev[1], Steffen Seckler[1], Matthias Heinen[2], Jadran Vrabec[2], Fabio Gratl[1], Martin Horsch[3,4], Martin Bernreuther[5], Colin W. Glass[6], Christoph Niethammer[5], Nicolay Hammer[7], Bernd Krischok[5] , Michael Resch[5,8], Dieter Kranzlmüller[7], Hans Hasse[9], Hans-Joachim Bungartz[1,7] and Philipp Neumann[10]

Abstract

Significant improvements are presented for the molecular dynamics code `ls1 mardyn` – a linked cell-based code for simulating a large number of small, rigid molecules with application areas in chemical engineering. The changes consist of a redesign of the SIMD vectorization via wrappers, MPI improvements and a software redesign to allow memory-efficient execution with the production trunk to increase portability and extensibility. Two novel, memory-efficient OpenMP schemes for the linked cell-based force calculation are presented, which are able to retain Newton's third law optimization. Comparisons to well-optimized Verlet list-based codes, such as LAMMPS and GROMACS, demonstrate the viability of the linked cell-based approach.

The present version of `ls1 mardyn` is used to run simulations on entire supercomputers, maximizing the number of sampled atoms. Comparing to the preceding version of `ls1 mardyn` on the entire set of 9216 nodes of SuperMUC, phase 1, 27% more atoms are simulated. Weak scaling performance is increased by up to 40% and strong scaling performance by up to more than 220%. On Hazel Hen, strong scaling efficiency of up to 81% and 189 billion molecule updates per second is attained, when scaling from 8 to 7168 nodes. Moreover, a total of twenty trillion atoms is simulated at up to 88% weak scaling efficiency running at up to 1.33 PFLOPS. This represents a fivefold increase in terms of the number of atoms simulated to date.

Keywords

molecular dynamics, SIMD vectorization, OpenMP, MPI, coloring, memory-efficient, SuperMUC, Hazel Hen

[1]Technical University of Munich, Boltzmannstr. 3, D-85748 Garching, Germany
[2]Thermodynamics and Process Engineering, Technical University Berlin, D-10623 Berlin, Germany
[3]Engineering Department, American University of Iraq, Sulaimani (AUIS), Kirkuk to Sulaimani Main Road, Raparin 46001, Sulaimani, Kurdistan Region, Iraq
[4]Scientific Computing Department, STFC Daresbury Laboratory, Warrington WA4 4AD, United Kingdom
[5]High Performance Computing Centre Stuttgart (HLRS), Nobelstr. 19, D-70569 Stuttgart, Germany
[6]Mechanical Engineering, Helmut Schmidt University, Holstenhofweg 85, D-22043 Hamburg, Germany
[7]Leibniz Supercomputing Centre, Boltzmannstr. 1, D-85748 Garching, Germany
[8]Institute for High-Performance Computing, University of Stuttgart, Nobelstr. 19, D-70569 Stuttgart, Germany
[9]Laboratory of Engineering Thermodynamics (LTD), TU Kaiserslautern, Erwin-Schrödinger-Str. 44, D-67663 Kaiserslautern, Germany
[10]Universität Hamburg, Bundesstr. 45a, D-20146 Hamburg, Germany

Corresponding author:
Philipp Neumann, Universität Hamburg, Bundesstr. 45a, 20146 Hamburg, Germany.
Email: philipp.neumann@uni-hamburg.de

Introduction

Motivation and Overview

Molecular dynamics (MD) simulation is an important tool in many fields, e. g., biology (Dror et al. 2012), life sciences (Karplus and Lavery 2014), thermodynamics (Niethammer et al. 2014), and materials science (Steinhauser and Hiermaier 2009). As a representative of N-body problems, it is computationally intensive. Thus, MD simulations are still limited to nano- and microscales. As hardware becomes increasingly powerful and laboratory experiments achieve higher resolution, a natural question is how to close the gap between MD and experimental work in the future. In light of this, important questions are:

- How large can a molecular simulation setup be chosen and executed on current supercomputers?
- Which code optimization and parallel programming techniques are most suited for this purpose?

The latter also needs to be considered in light of rapidly evolving hardware architectures which renders programmability, portability and extensibility of HPC software a similarly important challenge. This paper addresses these questions by describing the latest work on optimizing the performance of `ls1 mardyn`, which is an important tool for engineering applications (Niethammer et al. 2014).

`ls1 mardyn` was used in the past to establish a short-range MD world record simulation (Eckhardt et al. 2013), sampling the trajectories of more than four trillion atoms on the supercomputer SuperMUC, phase 1*. Excellent MPI-scalability on the entire machine, as well as an optimal molecule memory representation were demonstrated by Eckhardt et al. (2013). Nevertheless, several bottlenecks were outlined - too many MPI ranks, global collective MPI operations as well as the use of 128-bit SIMD. Here, we demonstrate how to further boost performance of this highly optimized code, by addressing these bottlenecks. This was done through the introduction of SIMD wrappers to easily switch between different vector lengths (that is 128-, 256- or 512-bit), as well as precision modes (single, double or mixed). Two novel, memory-efficient OpenMP shared-memory parallelization schemes for the linked cell method were introduced, which retain Newton's third law optimization. On the MPI side, nonblocking communication for global collectives was introduced as well as other minor improvements. Since our work is closely related to Eckhardt et al. (2013), we will refer to it as WR13. The present large scale runs were performed on the Hazel Hen machine†. In order to isolate the effect of our code changes from changes to hardware (Hazel Hen versus SuperMUC), we ran extensive experiments on SuperMUC, Phase 1, and compared to available data from WR13.

The remainder of the paper is organized as follows. In Section Short-Range MD, we introduce the short-range MD method and the algorithm in `ls1 mardyn`. Section Related Work: Short-Range Molecular Dynamics contains a literature review. Section Implementation and Optimization describes the aforementioned code improvements. Section Results discusses performance on the PetaFLOP platforms SuperMUC, Phase 1, Leibniz Supercomputing Centre (LRZ), and Hazel Hen, High-Performance Computing Center Stuttgart (HLRS). We provide detailed performance analyses at node- and multi-node level, up to full-machine size runs on Hazel Hen. Two main algorithms are most commonly used for short-range MD: linked cells and Verlet lists. We show that SIMD-optimized linked cells, despite their drawbacks, can compete with Verlet list implementations for certain simulation scenarios. With this scope, we provide a brief performance comparison with the well-established community frameworks Gromacs and LAMMPS. Section Conclusions and Outlook summarizes this work and outlines future activities.

Short-Range MD

In short-range MD, the translational and rotational equations of motion are numerically integrated. Considerations are restricted to small, rigid molecules with pairwise interactions that are explicitly evaluated within a specified cut-off radius r_c. Two well-established variants to implement the cut-off procedure are linked cells and Verlet lists (Rapaport 2004). Linked cells are used to sort molecules into a Cartesian grid with cell sizes $\approx r_c$; cf. Figure 1 (a). Only interactions between molecules in the same cell and in neighboring cells need to be tested and evaluated. In the Verlet approach, an interaction list is set up for every molecule, containing all molecules within a sphere that has a slightly larger radius $r = r_c + h$, where $h > 0$. Depending on the value of h and the conditions of the molecular system, this list needs to be rebuilt after some time interval that is larger than one time step Δt, e.g. $20\Delta t$. Both approaches reduce the molecule interaction complexity from $O(N^2)$ to $O(N)$. Verlet lists significantly reduce the overall volume for molecular interaction searches to the extended cut-off sphere's radius r, while yielding indirect molecule data accesses due to the list approach. Linked cells yield a larger volume for molecular interaction searches. However, sorting the molecules into cells is cheap, aligned data access is possible, and less memory is required because there is no need to store any additional molecule neighbor relations. A combination of both linked cells and Verlet lists is typically recommended (Brown et al. 2011). However, `ls1 mardyn` uses a linked cell approach only.

The leapfrog time integration scheme (Rapaport 2004) is used to solve Newton's equations of motion with a splitting

*SuperMUC, Phase 1, (S1) at LRZ, Garching/Germany, consists of 9,216 nodes, each built up by two hyper-threading-capable, 8-core Intel SandyBridge-EP Xeon E5-2680 processors. Running at a maximum of 2.7 GHz and using AVX, it provides a theoretical peak performance of 3.2 PFLOPS. For details, see Section SuperMUC Phase 1

†The Cray XC40 Hazel Hen machine (HH) at HLRS, Stuttgart/Germany, consists of 7,712 dual socket nodes, each featuring two 12-core Intel Haswell Xeon E5-2680 v3 processors. HH runs at a peak performance of 7.4 PFLOPS and provides 964 TB of memory. For details, see Section Hazel Hen

of the velocity update

$$v\left(t+\frac{\Delta t}{2}\right) = v(t) + \frac{\Delta t}{2m} f(t), \quad (1)$$

$$r(t+\Delta t) = r(t) + \Delta t v\left(t+\frac{\Delta t}{2}\right), \quad (2)$$

$$v(t+\Delta t) = v\left(t+\frac{\Delta t}{2}\right) + \frac{\Delta t}{2m} f(t+\Delta t), \quad (3)$$

where r denotes the position, v the velocity, f the force, m the mass and Δt the time step. The velocity update is split into two half-steps, so that the position and velocity can be sampled at the same physical time t, in order to evaluate macroscopic quantities such as kinetic or potential energy. This happens after step (3) and before step (1) of the next iteration. The force evaluation takes place after step (2). For multi-site molecules the rotational leapfrog variant is used (Fincham 1992).

Most molecular interactions in the present simulations are described by the Lennard-Jones (LJ) potential (Rapaport 2004)

$$U(r_{ij}) = 4\varepsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right), \quad (4)$$

with the energy and size parameters ε and σ of the molecular model and the distance between molecules i and j given by r_{ij}.

The interaction between two point charges is used in the current work in the context of simulations of water and is given by

$$U_{qq} = \frac{1}{4\pi\varepsilon_0}\frac{q_i q_j}{r_{ij}}, \quad (5)$$

where q_i, q_j are the charge magnitudes, r_{ij} is the distance between them and $\frac{1}{4\pi\varepsilon_0}$ is the Coulomb constant. The interaction between two point-quadrupoles (Gray and Gubbins 1984) was used in simulations of benzene in the current work:

$$U_{QQ} = \frac{1}{4\pi\varepsilon_0}\frac{3}{4}\frac{Q_i Q_j}{r_{ij}^5} \left(1 - 5\left[\cos^2\chi_1 + \cos^2\chi_2\right] - 15\cos^2\chi_1\cos^2\chi_2 + 2\left[\cos\psi\sin\chi_1\sin\chi_2 - 4\cos\chi_1\cos\chi_2\right]^2\right), \quad (6)$$

where Q_i, Q_j are the quadrupole magnitudes, r_{ij} is the distance between them and χ_1, χ_2 and ψ are angles, which describe the orientation of the point quadrupoles w.r.t. the line connecting them. Dipole and mixed interactions (charge-dipole, charge-quadrupole, dipole-quadrupole) are also supported. For molecules with zero net charge (which is the case in most `ls1 mardyn` application scenarios), the Reaction Field method is used (Barker and Watts 1973; Allen and Tildesley 1989). An implementation of the Fast Multipole Method (Greengard and Rokhlin 1987) for `ls1 mardyn` is under development.

Related Work: Short-Range Molecular Dynamics

HPC and Related Software Packages A pre-search process to improve neighbor list performance at SIMD level and a slicing scheme for OpenMP parallelism were addressed by Hu et al. (2017b) and Wang et al. (2016), focusing on the software Crystal MD and Intel Xeon/Xeon Phi systems. Domain slices, however, need to be sufficiently thick to enable OpenMP parallelism. This restricts the efficient application of this method to locally large domains. A SIMD approach for Intel architectures using a reduced version of the LAMMPS package was presented by Pennycook et al. (2013), showing speedups of up to 5 for single precision runs on 256-bit SIMD devices. Vectorization of the cut-off check is performed via blending/masking: several interactions are evaluated in a single vector instruction. If one of them needs to be excluded because the cut-off condition is not fulfilled, its result is masked to zero. A short-range MD implementation for host-accelerator devices using LAMMPS with speedups in LJ simulations ($r_c = 2.5\sigma$, $\rho\sigma^3 = 0.84$) of 3-4 was described by Brown et al. (2011). To improve SIMD performance in Verlet list implementations, lists of particle clusters were introduced by Páll and Hess (2013) and incorporated into the Gromacs software. Improving data reuse and tuning the cluster size to the SIMD hardware properties, speedups of 1.5-3 compared to the traditional lists were obtained. Parallelism in Gromacs at all levels (vectorization, shared and distributed parallelism) was discussed by Abraham et al. (2015). Gromacs further supports the use of GPUs, yielding speedups of 3-5 compared to CPUs[‡]. Both LAMMPS and Gromacs replicate force storage for OpenMP parallelization of the force calculation, leading to T times the memory overhead for running on T threads. An alternative approach to directly computing the forces for every pair of particles is given by the use of table lookups, followed by interpolation. This approach was investigated, amongst others, by Eckhardt (2014), who found it to be inferior for the potentials used in this work.

The `ls1 mardyn` Package Vectorization leveraging the linked cell approach has been developed in `ls1 mardyn` for single-site (Eckhardt and Heinecke 2012) and multi-site molecules (Eckhardt 2014). To enable memory-efficient MD for extremely large MD scenarios, a sliding window method was developed by Eckhardt and Neckel (2012), which compresses and decompresses molecule data on-the-fly during the linked cell traversal. A multi-dimensional, OpenMP-based coloring approach **`c08`** that operates on the linked cell data structure was discussed by Tchipev et al. (2015), showing good scalability on Intel Xeon and Intel Xeon Phi architectures. k-d tree-based load balancing within `ls1 mardyn` has recently been used by Seckler et al. (2016) to evenly distribute the computational load of the compute-intensive particle simulations on heterogeneous hardware systems.

World Record History A simulation consisting of 5 billion molecules was performed by Roth et al. (2000), pointing at the limits of MD in the year 2000. In 2006, Kadau et al. (2006) reported on a 320 billion atom run with the MD code SPaSM. The first trillion atom run followed in 2008 (Germann and Kadau 2008), which was outperformed by

[‡] www.gromacs.org/GPU_acceleration, as of Nov 2017

a 4.125 trillion atom run (WR13) using `ls1 mardyn` on the supercomputer SuperMUC, Phase 1 (Eckhardt et al. 2013). A two trillion atom simulation was recently carried out using Crystal MD by (Hu et al. 2017a) on the second fastest supercomputer TianHe-2[§], as well as a four trillion atom simulation on the fastest system Sunway TaihuLight (Li et al. 2018). Comparing memory requirements of three MD codes (Crystal MD, LAMMPS, IMD), Hu et al. found that Verlet lists have a significant share of the total MD memory requirements.

WR13 was achieved with a specifically optimized and simplified branch of `ls1 mardyn`, which is a hindrance to maintainability. Molecules were locally converted inside the sliding window from a primary array-of-structures (AoS) format to a structure-of-array (SoA) format in each time step before the force calculation to exploit 128-bit AVX instructions. In the AoS format, only $8+12+12$ Bytes were required to store an `unsigned long` unique ID, single-precision position and velocity per molecule. In the SoA format, $12+12$ Bytes for position and force buffers were allocated and efficiently reused during temporary conversion. Non-symplectic explicit Euler time stepping was employed. OpenMP parallelism was only used at the hyperthreading level, with two threads working spatially close within a slightly extended sliding window—at the cost of frequent, but cheap on-core OpenMP synchronization. MPI parallelization was based on a classical Cartesian domain decomposition. Forces acting on molecules across subdomain boundaries were computed by both processors.

Overall, 16 MPI ranks were started per node, which meant that when running on the entire machine, around 25% of the RAM needed to be reserved for MPI buffers. Global collective operations for computing the total potential energy or temperature of the system were also found to be a bottleneck. Employing still nascent technology at the time, the force calculation was vectorized by hand via 128-bit SSE intrinsics. This, however, proved to be cumbersome when doing reimplementations for 256-bit AVX which had been supported by the SuperMUC, Phase 1, SandyBridge architecture already at that time.

The present work uses a fully functional integrated code base of `ls1 mardyn`, including Verlet time integration, three OpenMP schemes which can be changed at runtime, a reduced memory mode (**RMM**) that can be switched on/off at compile time, global collective operations and SIMD vectorization for several particle interaction kernels.

Implementation and Optimization

Data Layout

Structure-of-Arrays In the non-**RMM** mode of `ls1 mardyn` (referred to as **Normal**), molecules are stored in AoS format to ease programmability for users of the code, who wish to program new application features themselves. Unfortunately, this is suboptimal for SIMD. Moreover, the hyperthreaded sliding window approach from WR13 with SoA storage is not suited for a larger number of threads. Low synchronization approaches, such as **c08**, require threads to work on disjoint or spatially distant regions of a MPI subdomain, requiring each thread's data in SoA format in order to exploit SIMD.

Therefore, permanent storage was switched to SoA format in the **RMM** mode. To hide the internal structure and to preserve code modularity, molecule data can be accessed using iterators whose interface is independent of the storage mode. Since multiple parts of the code base and the **Normal** mode assume AoS storage and interfaces, we construct—where needed—AoS objects in the **RMM** mode on-the-fly. Other more time-critical routines, e. g., resorting molecules which propagate from one linked cell to another, were modified to make use of the permanent SoA storage.

*Reduced Memory Mode (**RMM**)* In `ls1 mardyn`, several molecular properties, being necessary for the multi-site molecules and multi-component applications, are stored in the `Molecule` class. All properties that can be neglected for single-site Lennard-Jones particles were removed in WR13, yielding 32 Bytes per molecule. We follow the same approach, but allow to switch between **RMM** and **Normal** mode at compile time instead of using a specialized branch. The memory requirements are listed in Table 1. The **RMM** mode was implemented via polymorphism and conditional compilation, the latter was kept to a minimum.

One SoA structure to hold the molecules is allocated per linked cell and stored by reference in the cell (`RMMCell`). The variables of the molecules contained in the same `RMMCell` (position, velocity and unique ID) are stored contiguously in a dynamic `std::vector<char>` with a custom C++11 allocator to ensure proper alignment. Each array is 64 Byte aligned and padded to fill a multiple of 64 Bytes. This ensures full cache lines at all times, prevents false sharing and allows some tolerance for appending molecules, which have propagated to the current cell. Within the array, the variables are stored in the following order: `x, y, z, vx, vy, vz, uID`. Padding with zeros at the end of the `x` coordinates is inserted so that the `y` coordinates can be loaded via aligned loads of the employed SIMD width. Care was taken to reduce the total memory footprint of one linked cell (`sizeof(RMMCell)`) down to 64 Bytes.

Time Integration

Since forces are not permanently stored in **RMM** mode and we are no longer making use of a sliding window, a new solution for implementing the leapfrog time integration is required. Listings 1 and 2 illustrate our **RMM** solution and the **Normal** approach for the leapfrog method.

The variables `r_i`, `v_i` and `f_i` represent position, force and velocity of particle `i`, while `f_ij` denotes the force between molecules `i` and `j`. `dt_m` denotes the integration factor $\Delta t/m$. The scheme of Listing 2 leads to a larger number of multiplications in the merged force calculation-velocity update.

If `fused multiply-add` instructions are supported, however, this can be executed efficiently and does not involve more floating point rounding errors than in Listing 1. In standard implementations of the leapfrog algorithm, the velocity update is usually split into two halfsteps (factor `0.5` in Listing 1) so that macroscopic quantities such as energy can be evaluated at the same time. The **RMM** scheme cannot support this without additional computational effort.

[§] www.top500.org, as of Nov 2017

Type	Normal		RMM	
	double	single	double	single
Position (x,y,z)	24	12	24	12
Quaternion (q_0,q_1,q_2,q_3)	32	16		
Velocity (x,y,z)	24	12	24	12
Angular momentum (x,y,z)	24	12		
Forces (x,y,z)	24	12		
Torsional moment (x,y,z)	24	12		
Virial (x,y,z)	24	12		
Mass	8	4		
Moment of inertia (x,y,z)	24	12		
Inverse moment of inertia (x,y,z)	24	12		
Unique ID (unsigned long)	8	8	8	8
Component pointer	8	8		
SoA pointer	8	8		
SoA index	32	32		
Total	288	172	56	32

Table 1. Memory requirement (in Bytes) for one `Molecule` object in the modes **`Normal`** and **`RMM`**.

Listing 1: **`Normal`** mode

```
// velocity half-step 1
v_i = v_i + dt_m * 0.5 * f_i
...
// position update
r_i = r_i + dt * v_i
...
// force calculation
f_i = f_i1 + f_i2 +...
...
// velocity half-step 2
v_i = v_i + dt_m * 0.5 * f_i
...
```

Listing 2: **`RMM`** mode

```
// position update:
r_i = r_i + dt * v_i
...
// force calculation and velocity full-step:
v_i = v_i + dt_m * f_i1 + dt_m * f_i2 +...
```

SIMD Wrappers

The handwritten intrinsics kernel of `ls1 mardyn` was realized via wrapper classes. The arithmetic operations `+,-,*,/` were implemented via class operators. Aligned loads, stores, horizontal additions, etc. were implemented as member functions. Taking care that all operations are inlined by the compiler, the use of the wrappers results in efficient and convenient coding. The current wrappers support single and double precision using SSE3, AVX, AVX2, KNC and AVX512 instruction sets as well as a "no-vec" mode (referred to as "SOA" here).

Figure 1(a) illustrates the linked cell-based force calculation. For every molecule, all interacting molecules within a cut-off radius r_c have to be identified and the individual force contributions added. Molecules are loaded in aligned chunks of the SIMD vectorization width (4 for SSE, 8 for AVX). If the center of mass of a given molecule falls within the cut-off radius, the force kernel is computed for the whole chunk. Interactions beyond the cut-off radius are masked out, cf. Listing 3 for a LJ example.

Listing 3: LJ kernel using SIMD wrappers

```
...
RealCalcVec r2_inv =
    RealCalcVec::reciprocal_mask(c_r2, forceMask);
RealCalcVec lj2 = sig2 * r2_inv;
RealCalcVec lj4 = lj2 * lj2;
RealCalcVec lj6 = lj4 * lj2;
RealCalcVec lj12 = lj6 * lj6;
RealCalcVec lj12m6 = lj12 - lj6;
RealCalcVec eps24r2inv = eps_24 * r2_inv;
RealCalcVec lj12lj12m6 = lj12 + lj12m6;
RealCalcVec scale = eps24r2inv * lj12lj12m6;
f_x = c_dx * scale;
f_y = c_dy * scale;
f_z = c_dz * scale;
...
```

OpenMP-Parallel Force Calculation

An optimization of the force calculation that halves computational effort is to exploit Newton's third law ($f_{ij} = -f_{ji}$). However, race conditions occur if two threads operate on neighboring linked cells.

Scheme `c08` Scheme **`c08`** (Tchipev et al. 2015) avoids race conditions using eight colors (i.e. synchronization steps) in 3D, cf. Figure 1(b).

Threads work in parallel on "packages" of cells, whose lower left corner is of the same color. In Figure 1(b), while processing cell 0, the interactions within the package of cells 0, 1, 9 and 10, which are marked with arrows, are computed. After the work on one color is done, threads synchronize at a barrier, before proceeding to the next color. After all colors have been processed, every cell has interacted with all of its direct neighbors, e. g., cell 48.

Using a `schedule(dynamic, 1)` scheduling to assign packages to threads, the algorithm is somewhat load imbalance tolerant at the cost of predictable memory access patterns and NUMA-friendliness. The scheme has the drawback that it is not cache-efficient because a package of four/eight cells is discarded immediately after use and a new, disjoint package of cells needs to be fetched. Effectively, molecule data are streamed through the CPU eight times.

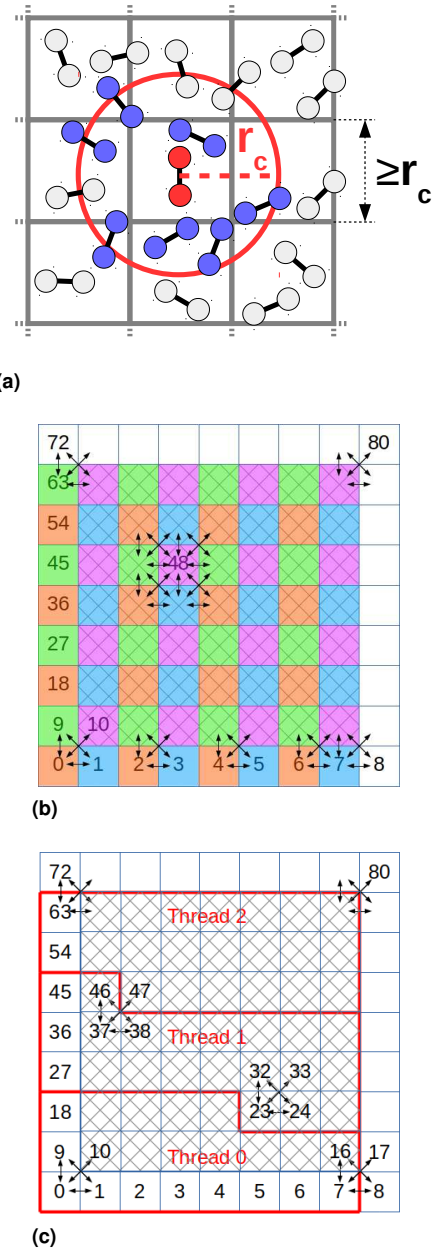


Figure 1. (a) Linked cell force calculation, (b) **c08** scheme, (c) **sli** scheme. Shaded cells in (b) and (c) represent inner cells, white cells represent halo cells. Interactions between two halo cells are not computed.

This is often nonetheless acceptable because the force computation is FLOP intensive.

Scheme ***sli*** To run at a single MPI process per dual-socket node efficiently, we introduce a novel slicing scheme **sli**. It essentially employs a one-dimensional domain decomposition, requiring a thread to obtain only one OpenMP lock from a neighboring thread, entailing extremely low synchronization cost. The 3D cell-iteration space is serialized into 1D and cut into T (number of threads) equal chunks, see Figure 1(c). At the start of the iteration, $T-1$ locks are generated and thread t sets lock number $t-1$. As soon as thread t has processed the first D $-$ 1 dimensional "slice", it unsets lock $t-1$ so that thread $t-1$ can obtain it for completing the last part of its iteration space. As soon as thread $t-1$ wants to access cells from t's iteration space, it attempts to set the lock number $t-1$ and possibly waits until it is released by thread t. Figure 1(c) shows an example for a 9×9 cell domain with 3 threads. Thread 1 works on cells 23 to 46 and sets lock 0 to prevent race conditions to cells 23-32. It waits for lock 1 to be released by thread 2, before computing forces for cells 37-46.

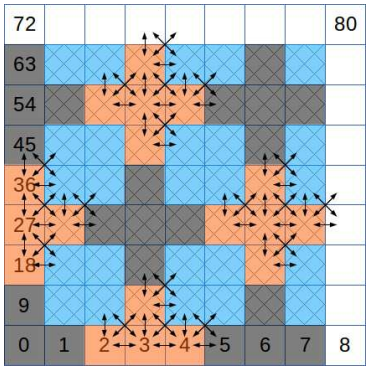
This scheme assumes that the workload is constant in all cells, although more load-balanced extensions are conceivable. It is cache-efficient and NUMA-friendly. At least $2T$ slices along the longest dimension of the domain of each MPI rank are required. Moreover, a load imbalance arises when a thread's domain covers many halo cells (cf. Figure 1(c)), as these interactions are skipped. This load imbalance is largest for thread 0 because it skips the largest number of halo-halo interactions. For nodewise, large-scale runs (e.g., 447^3 cells for 48 threads on Hazel Hen in the present work), this provides ample amounts of work without issues. For smaller and strong scaling scenarios, however, this requirement sometimes proves to be severe. We increased the number of MPI ranks per node in those cases, although switching to other schemes is supported at runtime.

Extending the one-dimensional splitting to two or three dimensions, while retaining the cheap synchronization of one lock per thread, is not easily conceivable. It would likely result in a need for much more locks per thread, which would probably be more efficient to be done with a global barrier. We point out that **sli** should also be applicable to other MD packages, even those employing Verlet lists (unlike **c08**). These codes usually sort molecules into bins to efficiently construct neighbor lists (Brown et al. 2011). Due to this spatial ordering, it can be computed when thread t begins to access molecules in the bins of thread $t+1$ in order to apply **sli**.

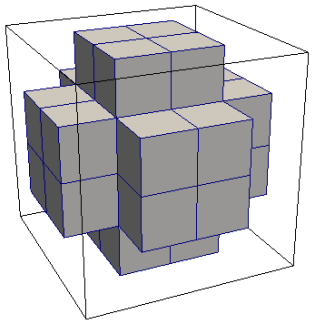
Scheme ***c04*** We now introduce a $D+1$-coloring variant for $D \leq 3$ dimensions and present some considerations for $D > 3$. This is an important distinction versus the 2^D colors of **c08** and suggests that potential generalizations to higher dimensions might be even more beneficial. This scheme can be understood in the following way: in one dimension, it is the well-known red-black coloring. In Figure 2, we show the implementations for $D = 2$ and $D = 3$. The $D = 2$ case in Figure 2(a) can be thought of as a two-dimensional red-black coloring on a Cartesian grid, with the diagonal links being broken by the third color. The crux lies here in the fact that the patches of the third color need to be "thick enough" to break the diagonal links.

It turns out that the three-dimensional variant of the scheme can be implemented again via "uniform" colors. Figure 2(b) shows the shape of the building block for all four colors. It represents 32 cells within a 4^3 bounding box, removing all cells which touch an edge of the bounding box. The centers of all building blocks in Figure 2(c) form a body-centered cubic (bcc) lattice. If one considers the centers of the bcc lattice as two three-dimensional Cartesian grids, the **c04** scheme can be interpreted as two, interleaved Cartesian grids, each colored in a red-black fashion.

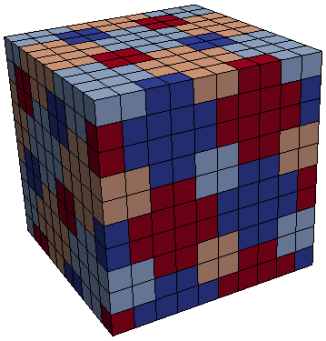
Comparing **c04** to **c08** and **sli**, **c04** has an intermediate number of synchronization steps and is cache friendly. Here, we implemented **c04** with a `schedule(dynamic, 1)` scheduling, to also allow



(a)



(b)



(c)

Figure 2. (a) Three-color, two-dimensional variant of the **c04** scheme, (b) one element of the **c04** scheme contained in its bounding box of 4^3 cells, (c) the **c04** scheme.

tolerance for load-imbalance. This, of course, comes at the cost of a potentially better NUMA-friendliness. Since **c04** schedules a whole building block of 32 packages of cells, the overhead of dynamic scheduling is, however, lower than in **c08** (which schedules only one package at a time).

Obviously, a drawback of **c04** is the rather complicated traversal pattern. Also, the handling of vertical boundaries is not trivial, since any plane, orthogonal to the primary axes, intersects building blocks of all colors in potentially four different configurations. This was handled by always traversing full building blocks and checking via `if`-statements what should be computed and what not.

More significant gains of **c04** over **c08** could be expected in applications with a higher dimensionality. The current coloring scheme should extend to D dimensions by comparing Figure 2(a) to a plane from (c) and observing that the "+"-like elements have "grown" and expecting alternating behavior for even and odd D. However, alternative colorings with the same number of colors in the same number of dimensions exist and may be more obvious to generalize to higher dimensions. One possibility is to increase the size of the blue squares in Figure 2(a) to 3×3, but retain the size of the "+"-like elements. All "+"-like elements then become disjoint and can, thus, be colored with the same color, leaving the third color for the newly formed 1×1 gaps along all edges of the blue squares. In $D > 2$ dimensions, the first color is used for a blue hypercube, with the observation that the size of the hypercube needs to increase with D, in order to break all emerging links. The remaining cells can then be colored similarly to the – already constructed – coloring in $D - 1$ dimensions (again, observing that higher dimensional analogues of lower dimensional shapes may need to be made "thicker").

*Turning **c08** and **c04** into Memory-Buffer Schemes*

Before proceeding, we highlight that both **c08** and **c04** can be turned into memory-buffer schemes, yielding two further possibilities. Let C be the number of colors and note the following: if threads are allowed to work on different colors independently (but still requiring only one thread per building block), then, at most C threads may access the same cell concurrently. The arising race conditions can then be resolved by writing to C different buffers, giving a constant memory overhead, independent of the number of threads. This can potentially be much smaller than the total number of threads, which can reach the order of one hundred for recent Xeon Phi or even Xeon architectures. An implementation of the resulting schemes is of interest, but beyond the scope of the present paper.

Resorting Particles in Cells

Apart from the force calculation, most other routines are relatively simple to be parallelized with OpenMP. The position update is embarrassingly parallel. Packing molecules from separate cells into contiguous buffers for MPI exchange can be done with a parallel-prefix-sum-like scheme. Resorting molecules, which have propagated from one cell to another, however, is not straightforward. Even after assuming that molecules can only propagate from one cell to one of its neighbors, this operation has the same read-write dependencies as the force calculation. Realizing this without additional memory buffers (in the **RMM** mode) is not straightforward. When applicable, we reused the **sli** traversal for this purpose. Otherwise, **c08** or **c04** were used. Because the resorting of molecules into cells is not compute intensive, the involved 8 traversals in **c08**, however, lead to a noticeable overhead compared to resorting through **sli** (not shown here). **c04** fares better in that aspect.

MPI Communication

The present domain composition uses the same partitioning algorithm as WR13. As all considered scenarios (see Section Scenario Description) are homogeneous, a Cartesian grid is the best option to partition the simulated MD volume and data.

Point-to-Point Communication Point-to-point communication was optimized by reducing the size of the transferred messages. It is differentiated between molecules which propagate from the domain of one process to another, and molecules which are just copied to enable the local force calculation with the halo regions. For the former, the full molecule data need to be communicated. For the latter, only information that is relevant for the force calculation (halo atom position/12 Bytes) needs to be transferred.

Global Collectives Collective communication is required in MD to gather statistical properties, such as temperature or energy. These values tend to change rather slowly. Therefore, we integrated a nonblocking scheme in `ls1 mardyn` based on MPI-3 in which these values are not based on the current, but the previous time step. A collective call, for which a value of the previous time step is allowed, has to be identified by a specific tag. The behavior of the collective calls of one specific tag is as follows: in the first time step, no previous collective call with that tag is available. Therefore, a normal blocking collective operation is performed using the current local values. Starting at the second time step, a collective call with that tag has already completed. Thus, the current local values are used to initiate a nonblocking collective call. Starting at the third time step, a nonblocking collective call from the previous time step is ongoing. The program waits upon its completion and updates the global values accordingly.

Results

We present analyses in FLOPS (floating point operations per second), and MMUPS (million molecule updates per second). The first metric is excellent to assess hardware utilization and compare with results of WR13. The second one is a system size independent metric frequently used in MD.

Compute Systems

SuperMUC Phase 1 SuperMUC, Phase 1, (S1) at LRZ, Garching/Germany, consists of $9,216$ nodes, each built up by two hyper-threading-capable, 8-core Intel SandyBridge-EP Xeon E5-2680 processors. Running at a maximum of 2.7 GHz and using AVX, it provides a theoretical peak performance of 3.2 PFLOPS. The nodes are connected by an Infiniband FDR10 interconnect in a fat tree topology. A total of 288 TB of RAM is available, while only 216 TB are typically usable for compute applications. All simulations were performed with disabled turbo mode. Instead, a fixed clock frequency was used.

Hazel Hen The Cray XC40 Hazel Hen machine (HH) at HLRS, Stuttgart/Germany, consists of $7,712$ dual socket nodes, each featuring two 12-core Intel Haswell Xeon E5-2680 v3 processors. Each core runs at 3.3 GHz at maximum and is capable of 2-way hyperthreading as well as AVX2. HH is currently the 19th fastest supercomputer (as of November 2017)[¶] with a peak performance of 7.4 PFLOPS (5.6 PFLOPS LINPACK) and provides 964 TB of memory (128 GB per node). The nodes are connected through the Aries interconnect. All runs in this work were performed with enabled turbo boost, unless mentioned otherwise.

Scenario Description

Except where noted, simulations were carried out with the same simulation setup as in WR13, namely a system consisting of single-site molecules, whose interactions were modeled by the LJ potential. Molecules that can be simulated with this kind of model include, e. g., argon. We used the same density of $\rho\sigma^3 = 0.78$, the same cut-off radius of $r_c/\sigma = 3.5$ and the same time step of 1 fs as in the world record run in 2013. Starting domains (before MPI decomposition) were always cubic and periodic boundary conditions were applied.

Node–Level Experiments

Sequential Performance Sequential performance was analyzed for a modest MD system of $18 \cdot 10^6$ LJ molecules (ca. 750 MB of memory). Figure 3(a) shows that the SIMD speedups increase with increasing r_c. The number of molecules per linked cell for $r_c/\sigma = 2.5$, 3.5 and 5.0 are 12, 33 and 98. (Single precision) Vectorization width is 4/8 for SSE/AVX. This explains why the SIMD gains are higher with increasing r_c: the loop trip count increases considerably. As noted in WR13, the kernel in Listing 3 features an addition-to-multiplication imbalance, chained multiplication operations, and—if used—floating point division. This inhibits the use of superscalarity, but can be mitigated by hyperthreading. For $r_c/\sigma = 5.0$, we attained more than 20% of the theoretical S1 peak (8.7 out of 43.2 GFLOPS). SSE performance was in very good agreement with Eckhardt (2014). On HH, AVX2 only yields marginal improvement over AVX, since the LJ model cannot exploit fused-multiply-add.

Due to considerable vectorization gains, the MMUPS decrease only slightly when going from $r_c/\sigma = 2.5$ to $r_c/\sigma = 3.5$, despite a $\left(\frac{3.5}{2.5}\right)^3 \approx 2.7$ fold increase of the number of floating point operations.

Strong Scaling OpenMP Fig. 4 shows the excellent strong scaling behavior on one node in an experiment with $47 \cdot 10^6$ LJ molecules and $r_c/\sigma = 3.5$. Turbo mode on HH was switched off in this run. Hyperthreading was investigated in a single-core experiment, pinning two threads on one core and running **`sli`**. The gains of hyperthreading are two-fold: hiding memory latency and helping instruction-level-parallelism in Listing 3. Over 22% performance gains were achieved on both S1 and HH, which is nearly two times higher than in WR13.

Considering **`sli`**, the strong scaling efficiency is 99% at 8 cores on S1 (on the same socket) and drops to 97% when going to 16 threads on both sockets due to emerging NUMA accesses. When using the estimate of the hyperthreading

[¶] www.top500.org, as of Nov 2017

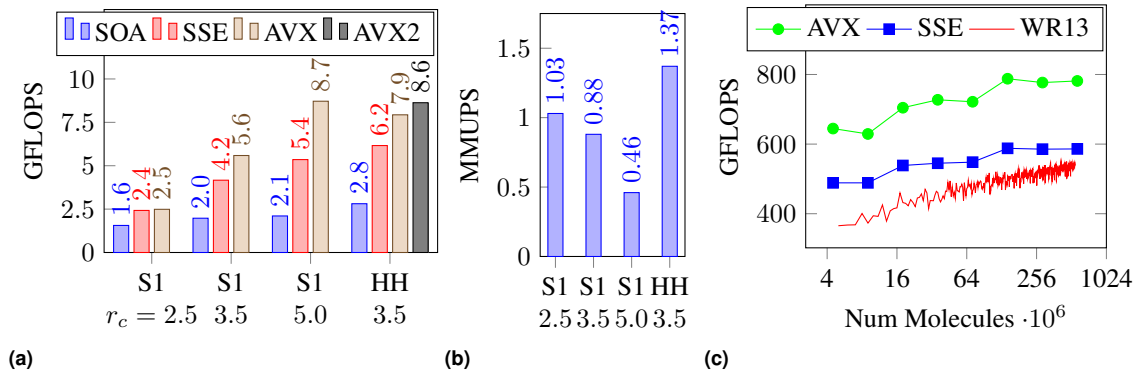


Figure 3. (a) GFLOPS performance for varying r_c (in σ) and SIMD modes. (b) MMUPS performance for varying r_c. (c) Comparison to WR13 on 8 nodes for $r_c/\sigma = 3.5$.

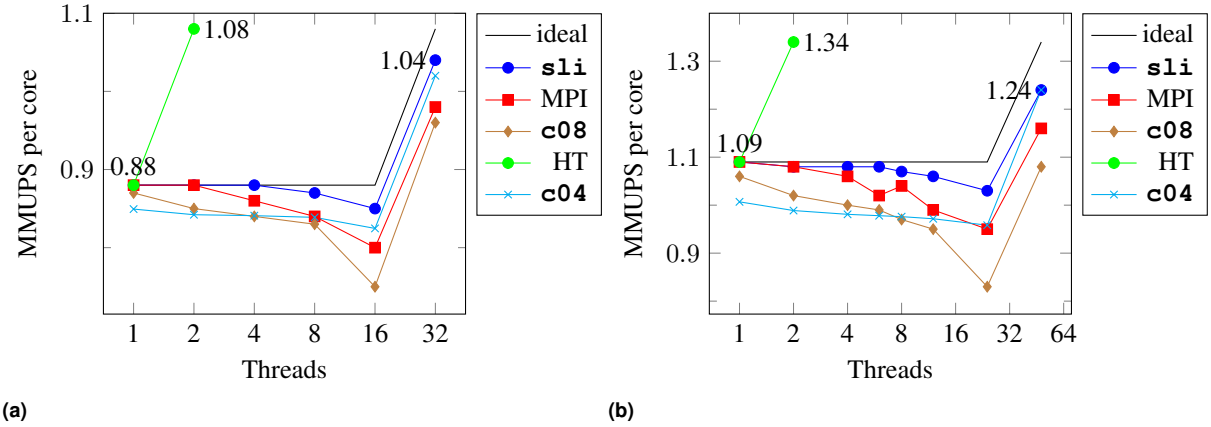


Figure 4. Strong scaling on one node, given in MMUPS per compute core. (a) S1, (b) HH. One thread per core is used, except for results achieved with 32/48 threads on S1/HH, corresponding to two threads per core (hyperthreading). Ideal performance behavior in the hyperthreading regime is extrapolated from single-core hyperthreading results, indicated by the green curves. Turbo mode was switched off on HH.

experiment to normalize parallel efficiency at 32 threads, we arrive at 96%. On HH, the values are 97% at 12 cores, 95% at 24 cores and 93% at 48 threads. When switched on, Turbo mode causes efficiency to drop to 86%, due to higher CPU boost at one thread (19%) than at 48 threads (9%). The gains of **sli** over **c08** are highest at 16 and 24 threads, reaching 13% and 25%. Because hyperthreading mitigates remote NUMA accesses to a certain extent, the gains drop to 8% and 15% at 32 and 48 threads. Finally, we point out that, unlike **c08**, **sli** is now able to beat our pure MPI implementation by 6-8%, which we consider a qualitatively significant achievement. While they may seem small, these differences are important when we push strong scaling to its limits in runs on entire supercomputers in the following.

For this configuration, the **c04** scheme delivers 4%/8% lower performance than **sli** on S1/HH sequentially. Section Scheme **c04** highlighted some drawbacks and potential reasons for this. **c04**, however, features the best scaling, which allows it to overtake both **c08** and the MPI variant on both architectures. Ultimately, on the full node, it delivers the second best performance - within 1% of the **sli** values.

Before proceeding, we briefly consider a smaller, load-imbalanced scenario, similar to scenarios arising in direct vapor-liquid equilibrium simulations (Eckelsbach and Vrabec 2015), namely a slab of liquid, surrounded by vapor on both ends. Figure 5(a) shows a visualization of the simulated domain. The ratio of liquid volume to vapor volume was one to one. The scenario featured slightly over 350 000 single-site LJ molecules at a cut-off radius of $r_c/\sigma = 3$ contained within $28 \times 55 \times 28$ linked cells. The density of the liquid phase of the domain was $\rho\sigma^3 = 0.6223$ and 0.06482 in the vapor phase.

Figure 5(b) shows the obtained strong scaling on one S1 node. Since the present implementation of **sli** always cuts along the longest dimension, a load imbalance arises in the **sli** curve for more than two threads. Cutting along one of the other dimensions would result in a perfect load balance, but further decrease the number of threads with which the scheme can be executed.

The low density vapor phase implies that the force calculation is not FLOP-intensive in this region, so both scheduling overhead, and data access become visible. At one thread, **sli** and **c04** fare better than **c08** (5.5% and 4.5%, respectively), which is due to data-reuse. With an increasing number of threads, the gap between **c04** and **c08** rises to 7% at 8 threads, which is now due to scheduling. At 16 threads, emerging NUMA accesses penalize **c08** harder, so that the gap becomes 21%, which is then (again) mitigated at 32 threads, bringing the gap down to 14%. Overall, the

speedup of **c04** is 13.7 at 16 threads and 14.3 at 32 threads, which we consider appropriate values for load-imbalanced scenarios simulated with the linked cell algorithm.

For the remainder of the paper we focus on **sli** and sometimes on **c08** when **sli** is not applicable.

Hybrid MPI-OpenMP Experiments Figures 6 and 7 show the results on S1 and HH for hybrid MPI-OpenMP scalings on 64 nodes. On both machines, we started from approximately full RAM utilization ($36 \cdot 10^9$ particles on S1 and $190 \cdot 10^9$ on HH) and considered geometrically scaled-down systems. Simulations on the chosen entire systems (2^0) using many MPI ranks could not be performed due to increasing halo-layer storage, highlighting the higher memory efficiency achieved through our OpenMP implementation.

The differences between the fastest and slowest scheme at 50% RAM are below 8% on both S1 and HH and rise only up to 18% for smaller sizes. On S1, the best performance was delivered by the configuration 8×4, while 6×8 worked best on HH, although 48×1 performed surprisingly well for large system sizes.

We further analyzed the performance of the force calculation for decreasing system size: it remained fairly constant, suggesting that performance degradation is a consequence of other operations (e.g., MPI communication and related network latency).

Since the difference between the fastest scheme and the "1 MPI rank per node" scheme at 100% memory utilization is only 6%/5% on S1/HH, the weak scaling experiments in the next sections made use of the latter (more memory efficient) configuration. The general purpose behind strong scaling is to solve problems as quickly as possible. Hence, we decided to use the 8×4 and 6×8 configurations for the strong scaling analyses.

Comparison to WR13 on 8 Nodes Figure 3(c) shows a comparison between the present implementation in the 8×4 configuration and WR13. A direct comparison can be drawn for SSE mode and WR13, which is only about 7% faster at high particle counts. The present implementation, however, sustains performance at low counts better, where SSE outperforms WR13 by about 33%. Comparing the AVX version, it is around 43% faster at high counts and 74% at low counts. Before proceeding, we point out that the performance for low counts will become important when considering strong scaling scenarios up to the entire machine. Then, roughly ten times lower molecule counts than shown in Figure 3(c) will come into play, suggesting even greater gains compared to WR13.

Performance Comparison to LAMMPS and Gromacs We compared the performance to LAMMPS, considering it as a representative of a well-optimized Verlet list-based MD code. The comparison in Figure 8 was carried out on one node of S1.

For LAMMPS, the latest stable version (11 Aug 2017) was used and compiled using `Makefile.intel_cpu_intelmpi`. The tests were run with the `USER-INTEL` package in single precision. The LAMMPS input file `in.intel.lj` was used (and—where noted—modified), featuring 512 000 single-site LJ atoms at $\rho\sigma^3 = 0.8442$ and $r_c/\sigma = 2.5$. For the Verlet lists, the default settings of "skin radius" of $h/\sigma = 0.3$ and the `neigh_modify` settings of `delay 0 every 20 check no` were used, meaning that the list is rebuilt every 20 iterations and no check is performed whether it needs to be updated more or less frequently. A similar input file was created for `ls1 mardyn` with slightly more (524 288) atoms.

A hybrid MPI-OMP analysis was performed to determine the best configuration to run both codes on this scenario. The MMUPS results are shown in Figure 8(a).

For this configuration, LAMMPS is the clear winner, delivering a threefold higher MMUPS rate. The best performance was delivered by the 16×2 configuration with Newton's third law optimization turned on. Both the "on" and "off" curves, however, deviate considerably from an ideally flat curve, suggesting that the MPI implementation outperforms the OpenMP one. These deviations are more pronounced for the "on" setting, as the "off" one should in fact be embarassingly parallel. For `ls1 mardyn`, the best results were obtained running at 4×8. This system, however, is almost "too small" for `ls1 mardyn`, as it features only $35 \times 35 \times 35$ cells; the scheme **sli** is, thus, not applicable in the 1×32 configuration. In `ls1 mardyn`, molecule storage required slightly more than 21 MB, which almost fits in the cache of one socket.

In Figure 8(b), we explore parameterizations, which affect MMUPS performance considerably: r_c, the number of atoms and Verlet list settings. Except where noted, other simulation parameters were kept as in Figure 8(a).

A LAMMPS run with the parameters `check yes` (not shown) revealed that all builds are marked as "dangerous", suggesting that the neighbor lists were not updated frequently enough. As advised by the LAMMPS documentation, the `neigh_modify` settings were modified, until the number of "dangerous builds" dropped to zero, which was at the settings `delay 0 every 5 check yes`. This resulted in 10 builds over the course of 100 time steps, instead of the previous 5 builds. These measurements are denoted with "check" in Figure 8(b). Comparing to the default settings, this results in roughly 16% less performance for LAMMPS.

Next, we investigated increasing the system size up to what each code can fit on a node. For LAMMPS, a run with $16 \cdot 10^6$ molecules exceeded the available RAM, so the "LAMMPS 8M check" data series was run with 8 192 000 molecules. The increased system size caused a minor deterioration of performance, except for $r_c/\sigma = 5.0$. With `ls1 mardyn`, we were able to store over $434 \cdot 10^6$ molecules and this resulted in up to 43% performance improvement. Comparing "LAMMPS 8M check" to "`ls1` 434M", the margins are no longer as drastic and decrease with increasing the cut-off radius. For $r_c/\sigma = 3.5$, which was used for the large scale runs, it is seen that the cost of memory efficiency is about 33%, which we consider tolerable.

At first glance, the gain of Newton's third law optimization in Figure 8(a) is small. Running the combinations of Figure 8(b), however, (not shown) leads to increasing gains for larger cut-off radius reaching up to 34% for the "LAMMPS 8M check $r_c = 5.0\sigma$" configuration, suggesting that it should not be undererstimated. For `ls1 mardyn`, the

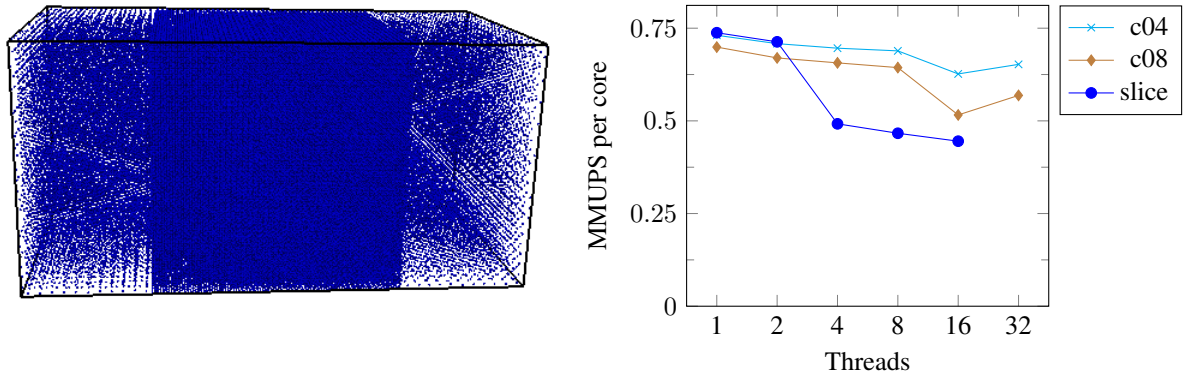


Figure 5. (a) Visualization of load imbalance scenario, (b) strong scaling on one node of S1 for the scenario visualized in (a). One thread per core is used, except for the case with 32 threads, which corresponds to hyperthreading (two threads per core).

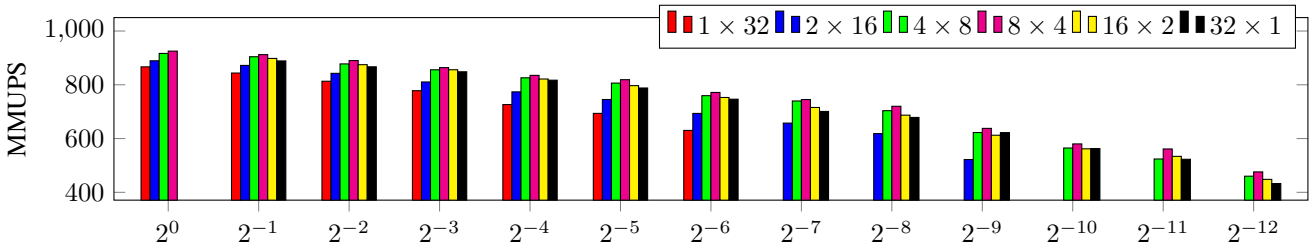


Figure 6. Hybrid MPI-OpenMP analysis on S1 for decreasing system size. 2^0 denotes 100% RAM utilization, 2^{-1} denotes 50% RAM utilization and so on. $N \times M$ denotes N MPI $\times$ M OMP.

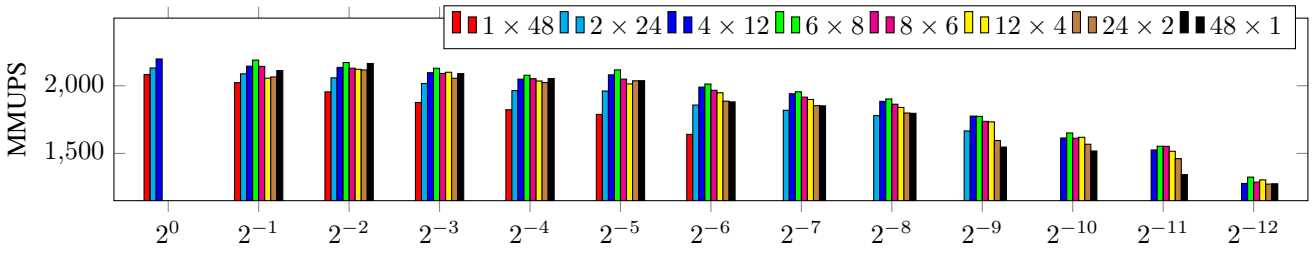


Figure 7. Hybrid MPI-OpenMP analysis on HH for decreasing system size. 2^0 denotes 100% RAM utilization, 2^{-1} denotes 50% RAM utilization and so on. $N \times M$ denotes N MPI $\times$ M OMP.

gains of Newton's third law optimization were investigated by Tchipev et al. (2015) who found it to be higher, i.e. between 30% and 50%.

The primary advantage of LAMMPS over `ls1 mardyn` in these scenarios is the reduced number of unnecessary cut-off condition checks between the Verlet list and linked cells implementations (28% vs 84%). For single-site LJ atoms, this is a serious overhead. However, the primary application area of `ls1 mardyn` are multi-site molecules (e. g., benzene: 6 LJ sites and 6 point quadrupoles), for which the cut-off condition check represents only a small fraction of the total number of FLOPs to be performed, especially since `ls1 mardyn` performs the cut-off condition check on a center of mass basis.

For this reason, a performance comparison with the important rigid water model TIP4P-2005 (Abascal and Vega 2005) was made, featuring 1 LJ and 3 charge sites per molecule. This implies that the force calculation between two molecules involves 1 LJ and $3 \times 3 = 9$ charge interactions, and is, thus, roughly ten times more expensive than the evaluation of the interaction between two single-site LJ molecules. The results in MMUPS (not *atom*-updates-per-second) are shown in Figure 9. The results were obtained for a system containing 512 000 TIP4P molecules at $T = 373.15$ K (100° C), $p = 1$ bar, timestep 2 fs, $r_c = 10$ Å, run in double precision. Long-range calculations were switched off or set to the reaction field mode, so that the non-bonded force calculation took more than 80% of the runtime for all codes. The **`Normal`** mode of `ls1 mardyn` was used. For LAMMPS, the package USER-OMP was used, as it features a specialized TIP4P calculation (as well as OpenMP support) via `pair_style lj/cut/tip4p/cut`.

We further included Gromacs in the performance comparison. We used the 2016.3d release, where d stands

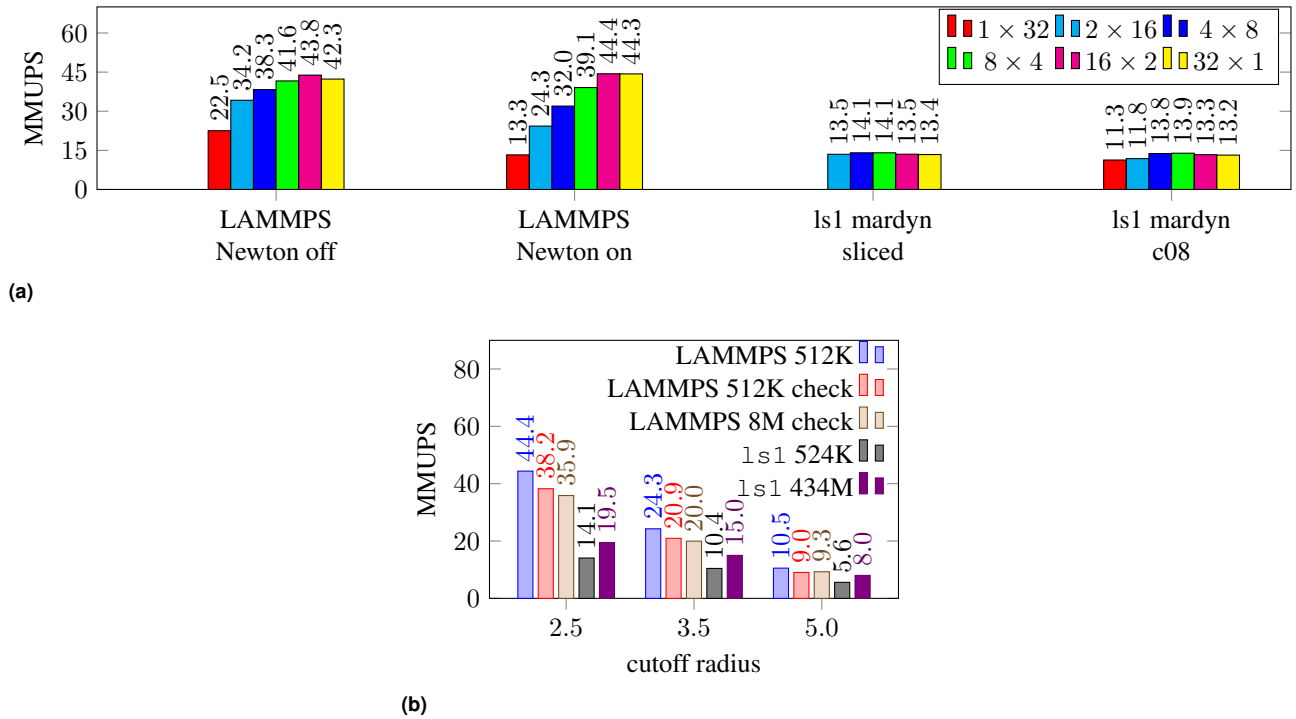


Figure 8. MMUPS comparison between `ls1 mardyn` and LAMMPS. (a) Hybrid MPI-OMP study of the Lennard-Jones benchmark. Newton "off" and "on" indicate whether Newton's third law optimization was turned on for LAMMPS. (b) Parameter investigation for LAMMPS 16×2 Newton on and `ls1 mardyn` 4×8.

for double-precision, which was available on S1[‖]. To create a suitable input, `gmx solvate` was used to create a box of 16913 water molecules employing the TIP4P water model which comes with Gromacs. This box was then subject to energy minimization, as well as temperature and pressure equilibration runs at $T = 373.15$ K, $p = 1$ bar. Subsequently, we used `gmx genconf` to enlarge the scenario to a box containing 456651 molecules through stacking of the previously equilibrated coordinate file. For the actual benchmark run, we used settings aiming for maximum comparability to `ls1 mardyn`. Temperature and pressure coupling was disabled and both `vdw-type` (Van der Waals) and `coulomb-type` settings were set to `Cut-off` with distances `rvdw`, `rlist` and `rcoulomb` all specified with 1 nm. The actual `cutoff-scheme` was set to `Verlet` which the documentation describes as a "pair list with buffering"[**]. Dispersion correction was enabled for both energy and pressure (`DispCorr = EnerPres`) and no further `vdw-modifiers` were employed.

In this configuration, `ls1 mardyn` outperforms LAMMPS—at least when using the USER-OMP package—by almost a factor of two. A reason for this is the multi-site molecule-oriented SIMD vectorization of `ls1 mardyn`. Consecutive sites of a single molecule always lie in contiguous memory locations, increasing the gains due to SIMD vectorization. Gromacs still outperforms `ls1 mardyn`, but only by 22%. A reason for the excellent Gromacs performance could be the extra effort invested in intrinsics vectorization and clustering of the Verlet lists, as highlighted by Abraham et al. (2015). A LAMMPS run with the USER-INTEL package would, of course, be of interest, but the current version does not support the `pair_style lj/cut/tip4p/cut` interaction type.

To summarize this comparison, Verlet list-based codes, such as LAMMPS and Gromacs, certainly offer excellent alternatives to linked cell calculations and can outperform them in many cases. Linked cells, however, can still offer excellent performance when the interaction between two molecules is expensive and treated in a memory efficient way. Continuing the comparison between the codes to investigate the effects of density, skin radius, and multi-node performance would be of great interest in the future.

Multi-Node Results

Global Overlapping Collectives As described in Section Global Collectives, we have replaced blocking collectives with nonblocking collectives. Figure 10 shows the performance improvements gained through nonblocking collectives. The strong scaling results are shown for a simulation of $64 \cdot 10^6$ benzene molecules, which were modeled by 6 LJ and 6 quadrupole sites, at a liquid density $\rho = 11.4$ mol/l and a cut-off radius $r_c = 2.0139$ nm. The simulation was performed in double precision and **`Normal`** mode. We used the **`c08`** traversal and pinned one MPI process on each NUMA domain. A speedup of ca. 16% could be measured on 4096 nodes of SuperMUC, resulting in a performance of 191 TFLOPS (13.8% of the theoretical peak performance on 4096 nodes). Nonblocking collectives boost the performance of ls1 mardyn significantly. They were thus used in the multi-node simulations discussed in the following.

[‖] https://www.lrz.de/services/software/comp-chemistry/gromacs/

[**] http://manual.gromacs.org/documentation/5.1-current/user-guide/mdp-options.html?highlight=dispcorr#mdp-cutoff-scheme

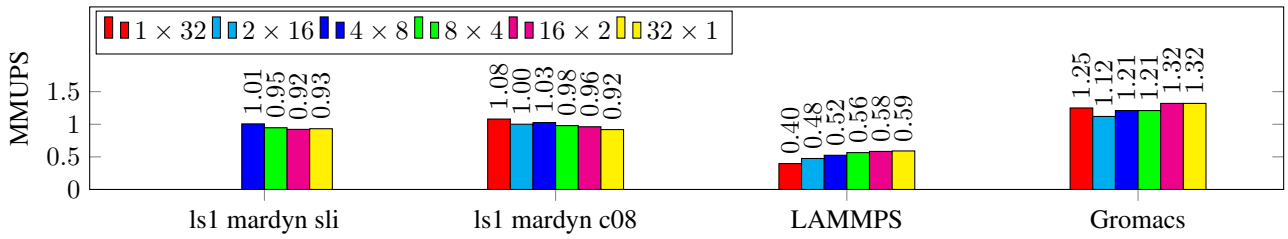


Figure 9. MMUPS comparison between `ls1 mardyn`, LAMMPS USER-OMP and Gromacs 2016.3d.

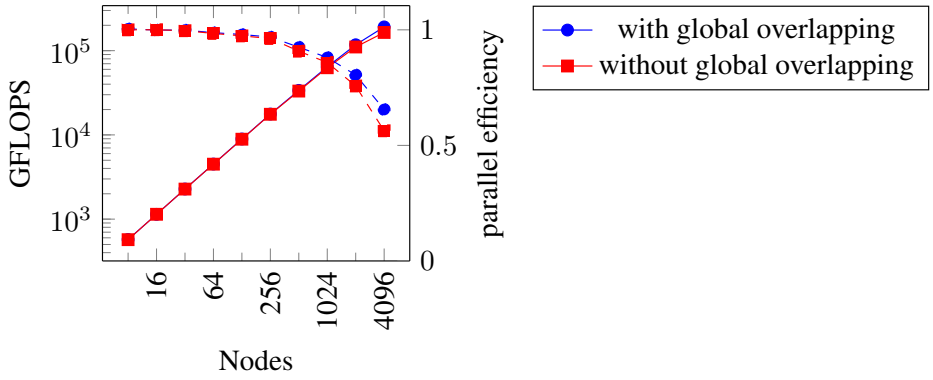


Figure 10. Comparison of the strong scaling behavior with and without the use of global overlapping collectives. A scenario with $64 \cdot 10^6$ benzene molecules was simulated on SuperMUC.

Results on SuperMUC, Phase 1 We conducted strong and weak scaling experiments on S1 and HH; results in terms of GFLOPS and parallel efficiency are shown in Figure 11. For the same scenario as in WR13 (LJ, $r_c/\sigma = 3.5$), a total of 5.2 trillion molecules could be simulated on S1, retaining a parallel efficiency of 87% and a performance of 720 TFLOPS (13.6% of the theoretical peak performance at 2.3 GHz) in the weak scaling scenario on 9216 nodes, using only 1 MPI rank per node, cf. Figure 11c. This is an increase of 27% in molecules and 21% in performance (WR13: $4.125 \cdot 10^{12}$ particles, 591 TFLOPS). However, we had to run these simulations at a frequency of 2.3 GHz, as technical difficulties prohibited an execution at 2.7 GHz. Scaling this up, our simulations yield a 40% increase in terms of performance. These improvements can mainly be attributed to increased node-level performance that shows a similar performance difference. For a scenario using $r_c/\sigma = 5.0$ and 2.3 GHz, a performance of 1.2 PFLOPS could be measured. A performance decrease of around 10% was measured for simulations on 8 nodes compared to simulations on 1 node. This is the consequence of a very efficient OpenMP parallelized intra-process exchange of molecules over the periodic boundaries, whereas the MPI communication, even though OpenMP parallelized, can not be optimized to the same degree.

Considering the performance on 8 nodes as a baseline and using 4.6 billion LJ particles at $r_c/\sigma = 3.5$, a strong scaling efficiency of 73% (WR13: 41.1%) was measured for runs on the entire machine in the 8×4 configuration, cf. Figure 11d. Compared to WR13 (260 TFLOPS, 2.7 GHz), we were able to more than double the performance (573 TFLOPS, 2.3 GHz, 10.8% of the theoretical peak performance), despite the lower frequency. If one scales the results to the proper frequency (2.7 GHz), a remarkable performance improvement of 258% can be measured for strong scaling experiments, even though a production-ready code version was used that, in contrast to the measurements from WR13, includes global collectives and the use of thermostats. The most important reasons for this are node-level performance and communication improvements. The former manifests itself in higher performance gains at low particle counts, cf. Figure 3(c). The latter was achieved through the improvements described in Section MPI Communication as well as the hybrid MPI-OpenMP parallelization and its accompanying reduction of the number of MPI processes. The strong scaling tests were performed such that we always chose the best values out of 11 runs. This was done to minimize the influence of the environment.

Results on Hazel Hen We were able to simulate systems with up to $2.1 \cdot 10^{13}$ particles on up to 7168 nodes of HH; for liquid xenon ($\sigma = 3.9450$ Å), this corresponds to a cube with a width of 11.8 μm. On up to 7168 nodes, we obtained a weak scaling efficiency of 88% (cf. Figure 11c) in the 1×48 configuration. On 7168 nodes, a performance of 1.33 PFLOPS was achieved, which represents around 9% of the single precision peak performance of HH. The value is lower than on S1 (13.6%) because the benefits from AVX2 are limited, cf Figure 3(a). The simulation rate could be measured at up to $189 \cdot 10^9$ MUPS.

The strong scaling tests were performed for a cut-off radius of $r_c/\sigma = 3.5$ and with a system of 23.85 billion particles. A performance of 1.18 PFLOPS with a parallel efficiency of 81% on 7168 nodes was observed in the 6×8 configuration. The simulation rate was measured as $178 \cdot 10^9$ MUPS. Like for the strong scaling runs on SuperMUC, we chose the best out of 11 runs on HH. A reason for the higher strong scaling efficiency—compared to the SuperMUC experiments—is that we began from 8

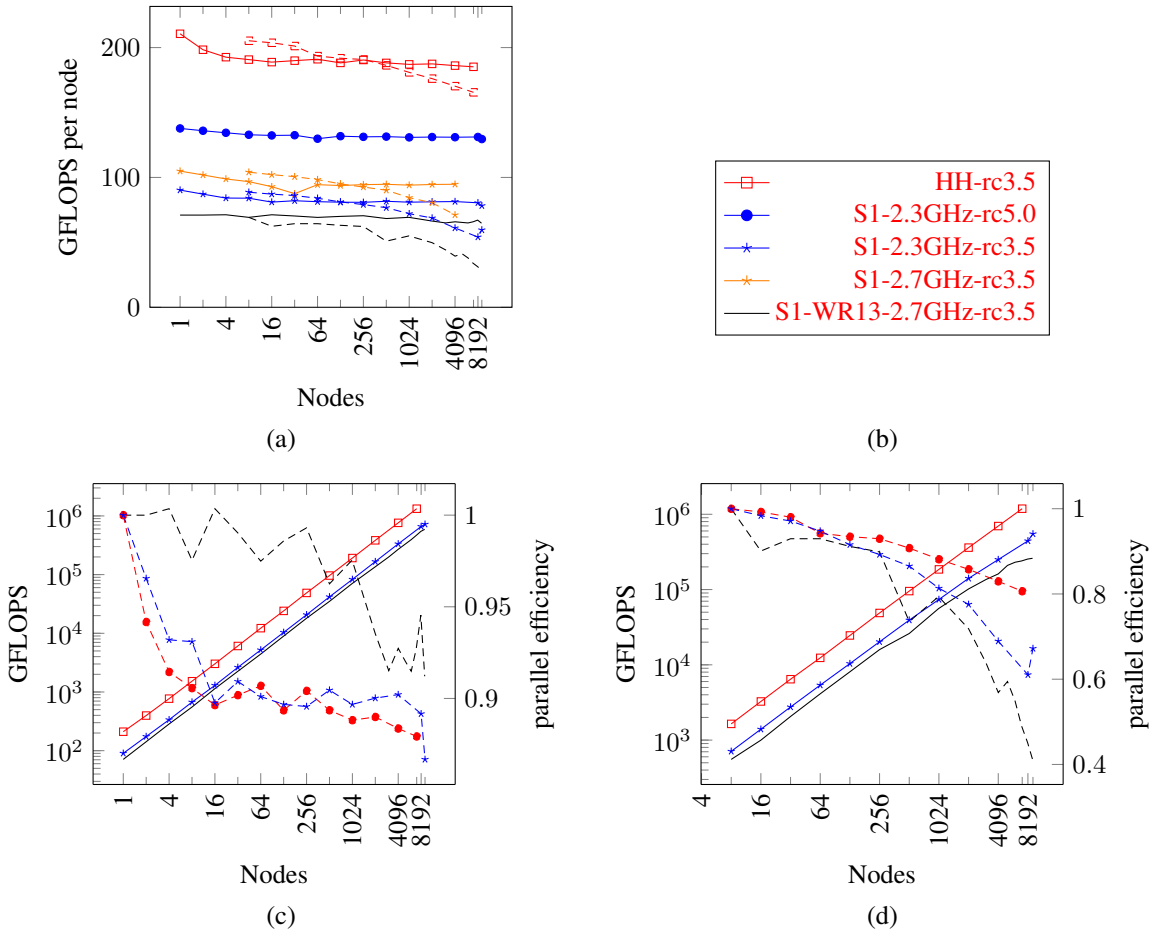


Figure 11. Scalability experiments on Hazel Hen (HH) and SuperMUC, Phase 1 (S1), using up to 9216 (S1)/7168 (HH) nodes. In (c) and (d) GFLOPS are shown by solid, the parallel efficiency by dashed lines. (a) GFLOPS per node in weak scaling, (b) legend, (c) total GFLOPS in weak scaling, (d) total GFLOPS in strong scaling.

full nodes, which means that the starting system contained almost five times more molecules on Hazel Hen.

On HH, we encountered hardware issues at large node counts, especially for the weak scaling simulations. Here, performance drops by up to 50% and failing nodes were observed. The cause of these performance decreases is under current investigation.

Conclusions and Outlook

We provided a detailed discussion of latest improvements of the MD package `ls1 mardyn`. Boosting node-level performance by enhanced hybrid MPI-OpenMP schemes, addressing programmability issues by SIMD wrappers and incorporating nonblocking global collectives, we could show that Peta-FLOP simulations and handling tens of trillions of atoms is feasible on current top supercomputers.

The OpenMP scheme **`sli`** is not only applicable to the linked cell, but also to Verlet list approaches. The scheme further yields minimal synchronization for large process-local domains. The **`c04`** scheme provides a load balance-tolerant alternative to **`c08`**, which was also demonstrated to perform very well, while bringing the number of colors and synchronization stages even further down. We further outlined the construction of two more schemes, based on additional memory buffers. Different OpenMP schemes were advantageous in one or the other case. It was not always easy to predict which one would deliver the best performance. Hence, an automatic choice of the scheme on-the-fly would be desirable to always achieve optimal performance. One way to achieve this is by iterating through available OpenMP schemes and selecting the one that delivers the lowest runtime. To take this a step further, different particle containers can also be iterated, including linked cells, Verlet lists and further possibilities such as full $\mathcal{O}(N^2)$ interactions (for very small systems) or tree containers (such as the ones used in astrophysics applications). Possibilities for this will be explored within the project "Task-based load balancing and auto-tuning in particle simulations" in the future. On the MPI side, even more performance may be attained in the hard strong scaling limits with our implementation by relying on enhanced domain decomposition methods, such as the eighth-shell approach—which is ongoing work. For large-scale runs, problems were observed that included nodes with very low performance as well as failing nodes. We currently investigate invasive parallelization techniques to timely exclude failing nodes and to recover from failures.

Due to the HPC improvements and having them integrated into the trunk (and thus in the production version) of `ls1 mardyn`, new scientific studies are enabled. For example, nucleation processes, which are the onset of phase change transitions, can now be sampled significantly

closer to (supersaturation) conditions that are accessible with experimental work; corresponding collaborative work is also in progress. The presented software version of `ls1 mardyn` is open-source. The version detailed in this contribution is available for download at www.ls1-mardyn.de/download.

Acknowledgements

We are grateful for financial support of the research underlying this work by the Intel Parallel Computing Center "ExScaMIC-KNL" and the Federal Ministry of Education and Research, Germany, project "Task-based load balancing and auto-tuning in particle simulations" (TaLPas), grant number 01IH16008. We thank the Gauss Centre for Supercomputing and its facilities LRZ and HLRS for support and computational resources (project GCS-MDDC).

References

Abascal J and Vega C (2005) A general purpose model for the condensed phases of water: TIP4P/2005. *The Journal of Chemical Physics* 123(23): 234505.

Abraham M, Murtola T, Schulz R, Páll S, Smith J, Hess B and Lindahl E (2015) GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX* 1–2: 19–25.

Allen M and Tildesley D (1989) *Computer Simulation of Liquids*. New York, NY, USA: Clarendon Press.

Barker J and Watts R (1973) Monte Carlo studies of the dielectric properties of water-like models. *Molecular Physics* 26(3): 789–792.

Brown W, Wang P, Plimpton S and Tharrington A (2011) Implementing molecular dynamics on hybrid high performance computers - short range forces. *Computer Physics Communications* 182(4): 898–911.

Dror R, Dirks R, Grossman J, Xu H and Shaw D (2012) Biomolecular Simulation: A Computational Microscope for Molecular Biology. *Annual Review of Biophysics* 41(1): 429–452.

Eckelsbach S and Vrabec J (2015) Fluid phase interface properties of acetone, oxygen, nitrogen and their binary mixtures by molecular simulation. *Physical Chemistry Chemical Physics* 17: 27195–27203.

Eckhardt W (2014) *Efficient HPC Implementations for Large-Scale Molecular Simulation in Process Engineering*. Munich: Dr. Hut. PhD thesis.

Eckhardt W and Heinecke A (2012) An efficient Vectorization of Linked-Cell Particle Simulations. In: *ACM International Conference on Computing Frontiers*. New York, NY, USA: ACM, pp. 241–243.

Eckhardt W, Heinecke A, Bader R, Brehm M, Hammer N, Huber H, Kleinhenz HG, Vrabec J, Hasse H, Horsch M, Bernreuther M, Glass C, Niethammer C, Bode A and Bungartz HJ (2013) 591 TFLOPS Multi-trillion Particles Simulation on SuperMUC. *Lecture Notes in Computer Science* 7905: 1–12.

Eckhardt W and Neckel T (2012) Memory-Efficient Implementation of a Rigid-Body Molecular Dynamics Simulation. In: *Proceedings of the 11th International Symposium on Parallel and Distributed Computing - ISPDC 2012*. Munich: IEEE, pp. 103–110.

Fincham D (1992) Leapfrog rotational algorithms. *Molecular Simulation* 8(3-5): 165–178.

Germann T and Kadau K (2008) Trillion-atom molecular dynamics becomes a reality. *International Journal of Modern Physics C* 19(09): 1315–1319.

Gray C and Gubbins K (1984) *Theory of Molecular Fluids: I: Fundamentals*. Oxford: Oxford University Press.

Greengard L and Rokhlin V (1987) A fast algorithm for particle simulations. *Journal of Computational Physics* 73(2): 325–348.

Hu C, Bai H, He X, Zhang B, Nie N, Wang X and Ren Y (2017a) Crystal MD: The massively parallel molecular dynamics software for metal with BCC structure. *Computer Physics Communications* 211: 73–78.

Hu C, Wang X, Li J, He X, Li S, Feng Y, Yang S and Bai H (2017b) Kernel optimization for short-range molecular dynamics. *Computer Physics Communications* 211: 31–40.

Kadau K, Germann T and Lomdahl P (2006) Molecular dynamics comes of age: 320 billion atom simulation BlueGene/L. *International Journal of Modern Physics C* 17(12): 1755–1761.

Karplus M and Lavery R (2014) Significance of Molecular Dynamics Simulations for Life Sciences. *Israel Journal of Chemistry* 54(8-9): 1042–1051.

Li S, Wu B, Zhang Y, Wang X, Li J, Hu C, Wang J, Feng Y and Nie N (2018) Massively Scaling the Metal Microscopic Damage Simulation on Sunway TaihuLight Supercomputer. In: *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018. New York, NY, USA: ACM, pp. 47:1–47:11.

Niethammer C, Becker S, Bernreuther M, Buchholz M, Eckhardt W, Heinecke A, Werth S, Bungartz HJ, Glass C, Hasse H, Vrabec J and Horsch M (2014) ls1 mardyn: The massively parallel molecular dynamics code for large systems. *Journal of Chemical Theory and Computation* 10(10): 4455–4464.

Páll S and Hess B (2013) A flexible algorithm for calculating pair interactions on SIMD architectures. *Computer Physics Communications* 184(12): 2641–2650.

Pennycook S, Hughes C, Smelyanskiy M and Jarvis S (2013) Exploring SIMD for Molecular Dynamics, Using Intel®Xeon®Processors and Intel®Xeon Phi™Coprocessors. In: *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE, pp. 1085–1097.

Rapaport D (2004) *The Art of Molecular Dynamics Simulation*. Cambridge: Cambridge University Press.

Roth J, Gähler F and Trebin HR (2000) A molecular dynamics run with 5 180 116 000 particles. *International Journal of Modern Physics C* 11(2): 317–322.

Seckler S, Tchipev N, Bungartz HJ and Neumann P (2016) Load Balancing for Molecular Dynamics Simulations on Heterogeneous Architectures. In: *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*. Hyderabad, India: IEEE, pp. 101–110.

Steinhauser M and Hiermaier S (2009) A Review of Computational Methods in Materials Science: Examples from Shock-Wave and Polymer Physics. *International Journal of Molecular Sciences* 10(12): 5135–5216.

Tchipev N, Wafai A, Glass C, Eckhardt W, Heinecke A, Bungartz HJ and Neumann P (2015) *Optimized Force Calculation in*

Molecular Dynamics Simulations for the Intel Xeon Phi. Cham: Springer International Publishing, pp. 774–785.

Wang X, Li J, Wang J, He X and Nie N (2016) *Kernel Optimization on Short-Range Potentials Computations in Molecular Dynamics Simulations*. Singapore: Springer, pp. 269–281.