591 TFLOPS Multi-trillion Particles Simulation on SuperMUC

Wolfgang Eckhardt[1], Alexander Heinecke[1],
Reinhold Bader[2], Matthias Brehm[2], Nicolay Hammer[2], Herbert Huber[2],
Hans-Georg Kleinhenz[2], Jadran Vrabec[3], Hans Hasse[4], Martin Horsch[4],
Martin Bernreuther[5], Colin W. Glass[5], Christoph Niethammer[5],
Arndt Bode[1,2], and Hans-Joachim Bungartz[1,2]

[1] Technische Universität München, Boltzmannstr. 3, D-85748 Garching, Germany
[2] Leibniz-Rechenzentrum der Bayerischen Akademie der Wissenschaften, Boltzmannstr. 1, D-85748 Garching, Germany
[3] University of Paderborn, Warburger Str. 100, D-33098 Paderborn, Germany
[4] Laboratory of Engineering Thermodynamics (LTD), TU Kaiserslautern, Erwin-Schrödinger-Str. 44, D-67663 Kaiserslautern, Germany
[5] High Performance Computing Centre Stuttgart (HLRS), Nobelstr. 19, D-70569 Stuttgart, Germany

Abstract. Anticipating large-scale molecular dynamics simulations (MD) in nano-fluidics, we conduct performance and scalability studies of an optimized version of the code `ls1 mardyn`. We present our implementation requiring only 32 Bytes per molecule, which allows us to run the, to our knowledge, largest MD simulation to date. Our optimizations tailored to the Intel Sandy Bridge processor are explained, including vectorization as well as shared-memory parallelization to make use of Hyperthreading. Finally we present results for weak and strong scaling experiments on up to 146016 Cores of SuperMUC at the Leibniz Supercomputing Centre, achieving a speed-up of 133k times which corresponds to an absolute performance of 591.2 TFLOPS.

Keywords: molecular dynamics simulations, highly scalable simulation, vectorization, Intel AVX, SuperMUC.

1 Introduction and Related Work

MD simulation has become a recognized tool in engineering and natural sciences, complementing theory and experiment. Despite its development for over half a century, scientists still quest for ever larger and longer simulation runs to cover processes on greater length and time scales. Due to the massive parallelism MD typically exhibits, it is a preeminent task for high-performance computing.

An application requiring large-scale simulations is the investigation of nucleation processes, where the spontaneous emergence of a new phase is studied [8]. To enable such simulations, we optimized our program derived from the code `ls1 mardyn`. A description of `ls1 mardyn` focusing on use cases, software structure and load balancing considerations can be found in [1]. Based on the further development of the memory optimization described in [3], an extremely low memory requirement of only 32 Bytes per molecule has been achieved, which allows us to carry out the to our knowledge largest MD simulation to date on SuperMUC at Leibniz Supercomputing Centre. In order to run these large-scale simulations at satisfactory performance, we tuned the implementation of the molecular interactions outlined in [2] to the Intel Sandy Bridge processor and added a newly developed shared-memory parallelization to make use of Intel Hyperthreading. Thereby, this contribution continues a series of publications on extreme-scale MD. In 2000, Roth [18] performed a simulation of $5 \cdot 10^9$ molecules, the largest simulation ever at that time. Kadau and Germann [4, 10] followed up, holding the current world record with 10^{12} particles. These simulations demonstrated the state of the art on the one hand, and showed the scalability and performance of the respective codes. More recent examples include the simulation of blood flow [15] as well as the force calculation of $3 \cdot 10^{12}$ particles by Kabadshow in 2011 [9], however without calculating particle trajectories.

The remainder of the paper is organized as follows: this Section describes the computational model of our simulation code. Section 2 describes the architecture of SuperMUC, Section 3 details the implementation with respect to vectorization and memory-efficiency, and Section 4 presents the results.

The fluid under consideration is modeled as a system of N discrete particles. Only particles i and j separated by a distance r_{ij} that is smaller than a cut-off radius r_c interact pairwise through the truncated and shifted Lennard-Jones potential [19], which is determined by the usual Lennard-Jones-12-6 potential (LJ-12-6) $U_{LJ}\left(r_{ij}\right)$ with the potential parameters ϵ and σ:

$$U_{LJ}\left(r_{ij}\right) = 4\epsilon \cdot \left(\left(\frac{\sigma}{r_{ij}} \right)^{12} - \left(\frac{\sigma}{r_{ij}} \right)^{6} \right).$$

The interactions with all neighbors results in a force $F_i = \sum_{j \in particles} F_{ij}(r_{ij})$ on each of the particles, which is evaluated only once per particle pair, due to Newton's law $F_{ij} = -F_{ji}$.

In MD, the most time-consuming step is the force calculation. To efficiently search for neighboring particles, the linked-cell algorithm is employed in a similar way as in [10]. The computational domain is subdivided into cubic cells with an edge length r_c. Consequently, for a given particle, the distances to all other particles contained in the same cell as well as in the (in 3D) 26 adjacent cells have to be computed. This results in a linear complexity of the force calculation. The particles' data are stored in dynamic arrays, i.e. contiguous memory blocks, per cell, to avoid additional memory for pointers. Thus, the organization of the linked-cells data structure causes only small overhead.

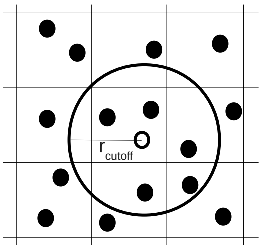


Fig. 1. Schematic of the linked-cell algorithm (2D)

In accordance with preceding large-scale simulations [4], single-precision variables are used for the calculation. For a particle we store only its position ($3 \cdot 4$ Bytes), velocity ($3 \cdot 4$ Bytes) and an identifier (8 Bytes), i.e. 32 Bytes in total.

The force vector does not need to be stored permanently, because the time integration of the equations of motion is carried out on the fly, as detailed in Section 3.

To evaluate our implementation, single-center Lennard-Jones particles were distributed on a regular grid according to a body-centered cubic lattice, with a liquid-like number density of $\rho\sigma^3 = 0.78$, and the cut-off radius was specified to be $r_c = 3.5\sigma$. The time step length was set to 1 fs.

For the MPI parallelization, we employ a spatial domain decomposition scheme. For n processes, the domain is divided in n equally-sized sub-domains, which are assigned to one process each. Each sub-domain is surrounded by a layer of ghost cells residing on neighboring processes, so the particles at the process boundaries have to be exchanged at the beginning of each time step.

2 SuperMUC – The World's Largest x86 Machine

2.1 System Topology

We optimized our MD code on the micro-architecture level for a specific processor: the Intel Sandy Bridge EP driving SuperMUC operated at the Leibniz Supercomputing Centre in Munich. This system features 147456 cores and is at present the biggest x86 system worldwide with a theoretical double precision peak performance of more than 3 PFLOPS, placed #6 on the current Top500 list. The system was assembled by IBM and features a highly efficient hot-water cooling solution. In contrast to supercomputers offered by Cray, SGI or even IBM's own BlueGene, the machine is based on a high-performance commodity network: a FDR-10 infiniband pruned tree topology by Mellanox. Each of the 18 leafs, or islands, consists of 512 nodes with 16 cores at 2.7 GHz clock speed (turbo mode is disabled) sharing 32GB of main memory. Within one island, all nodes can communicate at full FDR-10 data-rate. In case of inter-island communication, four nodes share one uplink to the spine switch. Since the machine is operated diskless, a significant fraction of the nodes' memory has to be reserved for the operation environment.

2.2 Intel Sandy Bridge Architecture

After a bird's eye view on the entire system, we now focus on its heart, the Intel Sandy Bridge EP processor that was introduced in January 2012, featuring a new vector instruction set called AVX. In order to execute code with high performance and to increase the core's instructions per clock, major changes to the previous core micro-architecture code-named Nehalem have been applied. These changes are highlighted by italic characters in Fig. 2. Since the vector-instruction width has been doubled with AVX (AVX is available with two vector widths: AVX128 and AVX256), also the load port's (port 2) width needs to be doubled. However, doubling a load port's width would impose tremendous changes to the entire chip architecture. In order to avoid this, Intel changed two ports by additionally implementing in each port the other port's functionality as shown for

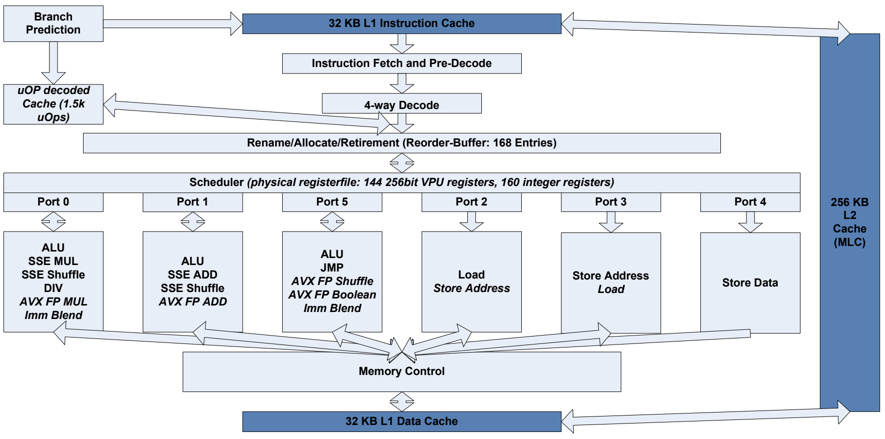

Branch Prediction
32 KB L1 Instruction Cache
Instruction Fetch and Pre-Decode
uOP decoded Cache (1.5k uOps)
4-way Decode
Rename/Allocate/Retirement (Reorder-Buffer: 168 Entries)
Scheduler (physical registerfile: 144 256bit VPU registers, 160 integer registers)
Port 0
Port 1
Port 5
Port 2
Port 3
Port 4
ALU SSE MUL SSE Shuffle DIV AVX FP MUL Imm Blend
ALU SSE ADD SSE Shuffle AVX FP ADD
ALU JMP AVX FP Shuffle AVX FP Boolean Imm Blend
Load Store Address
Store Address Load
Store Data
Memory Control
32 KB L1 Data Cache
256 KB L2 Cache (MLC)

Fig. 2. Intel Sandy Bridge, changes w.r.t. Intel Nehalem are highlighted with italic characters: trace-cache for decoded instructions, AVX support and physical register file

ports 2 and 3. Through this trick, the load bandwidth has been doubled from 16 Bytes to 32 Bytes per cycle at the price of reduced instruction level parallelism. Changes to the ALUs are straightforward: ports 0, 1 and 5 were simply doubled, and they provide classic SSE functionality for AVX instructions and extensions for blend and mask operations. However, this bandwidth improvement still does not allow for an efficient exploitation of AVX256 instructions as this would require a 64 Bytes per cycle load and 32 Bytes per cycle store bandwidth. This increase will be implemented with the up-coming Haswell micro-architecture [14]. Due to 32 Bytes load bandwidth and the non-destructive AVX128 instruction set, AVX128 codes can often yield the same performance as AVX256 on Sandy Bridge but much better than SSE4.2 on an equally clocked Nehalem chip. This can also be attributed to the fact that 16 Bytes load instructions have a three times higher throughput (0.33 cycles) than 32 Bytes load instructions (here ports 2 and 3 have to be paired and cannot be used independently). According to experiments we did with different applications and kernels using AVX256 on Sandy Bridge, the full performance enhancement of 2× speed-up can be just exploited for kernels which can be perfectly register-blocked, e.g. DGEMM [7]. If in contrast only a standard 1D-blocking is possible, roughly a 1.5-1.6 × speed-up can be achieved in comparison to AVX128 [6].

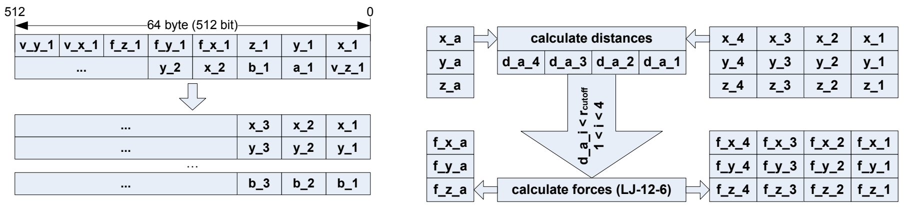
Up to Nehalem, each unit had dedicated memory for storing register contents for executing operations on them. A so-called out-of-order unit took care of the correctness of the execution pipeline. With AVX, a register allocation in each compute unit of the core would be too expensive in terms of transistors required, therefore a so-called *register file* was implemented: Register contents are stored in a central directory. Shadow registers and pointers allow for an efficient out-of-order execution. Furthermore, a general performance enhancement was added to the Sandy Bridge architecture: a cache for decoded instructions. This trace-cache like cache boosts the performance of kernels with small loop bodies, such as the force calculation in MD. Furthermore, the Sandy Bridge EP cores feature Intel's SMT implementation called *Hyperthreading Technology* which helps to increase the core's utilization in workload scenarios where the instruction mix is not optimal or the application is suffering from high memory latencies.

3 Implementation

3.1 Vectorization of the Compute Kernel

Since our simulation code is written in C++ and therefore applies standard object-oriented design principles with cells and particles being single entities, we follow an approach of memory organization and vectorization, first sketched in [2]. That work describes, by using a simple proxy application and not the entire `ls1 mardyn` code base, how the LJ-12-6 force calculation inside a linked cell algorithm can be vectorized on x86 processors. That prototype implementation does not feature important statistical measurements such as virial pressure and potential energy which we added in this work.

The object-oriented memory layout is cache-efficient by design because particles belonging to a cell are stored closely together. However, implementing particles in a cell as a so-called *array of structures (AoS)* forbids easy vectorization, at least without gather and scatter operations (see [5]) which, unfortunately, are not available on Intel Sandy Bridge. Only in simple cases (e.g., updates of one member, etc.) this drawback does not matter, because prefetch logic inside the hardware loads only cache-lines containing data which have to be modified.



(a) *AoS to SoA conversion*: In order to allow for efficient vectorization, corresponding elements have to be stored for data streaming access.

(b) *Kernel vectorization*: The vectorization of the LJ-12-6 force calucation is optimized by duplicating one particle and streaming four other particles.

Fig. 3. Optimizing LJ-12-6 force calculation by SoA storage scheme and vectorization

Implementing the LJ-12-6 force calculation on AoS-structures poses major challenges: The upper part of Fig. 3a shows elements scattered across several cache-lines. Taking into account that only a small portion of the members is needed for the force calculation, a temporary *structure of arrays (SoA)* can be constructed in order to address cache-line pollution and vectorization opportunities, illustrated in the lower part of Fig. 3a. Figure 3b sketches the applied vectorization of the LJ-12-6 calculations. In contrast to other methods which vectorize across the spatial coordinates [11–13], the present approach can exploit vector-units of arbitrary length.

In this work, single-precision AVX128 instructions were employed. The calculation is performed on particle pairs, therefore we broadcast-load the required data of one particle in the first register (a), the second register is filled by data from four other particles (1, 2, 3 and 4). Dealing with four particle pairs at once, we can theoretically reduce the number of operations by a factor of four. Since the force calculation may be required for all, some or none of the pairs in the vector register, we need to apply some pre- and post-processing performed by regular logical operations: It has to be determined, if for any particle pair the distance is smaller than r_c (pre-processing), because only then the force calculation has to be executed. If the force calculation has been executed, the calculated results need to be zeroed by a mask for all particle pairs whose distance is larger than r_c (post-processing). In order to ensure vectorization of the kernel we employed intrinsics. Due to the cut-off radius `if`-condition inside the inner-most loop, current compilers (gcc and icc) deny to vectorize the loop structure iterating over particles in cell-pairs. For the chosen simulation scenario (cut-off radius $r_\mathrm{c} = 3.5\sigma$) a speed-up of $3\times$ is possible on a single core by using the proposed SoA-structure and vectorization.

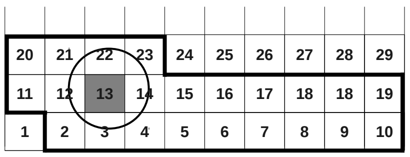
With increasing vector length, this masking technique becomes the major bottleneck. Here, it can easily happen that more elements are being masked than elements which have to be computed. Therefore, moving to a wider vector-instruction set may result in more instructions being executed. However, if the vector-instruction set features *gather* and *scatter* instructions, this issue can be overcome because only the particle pairs taking part in the interaction are processed, which has been successfully demonstrated by Rapaport with the layered-linked-cell algorithm [16, 17]. The first x86 processor which offers full gather/scatter support is the so-called Xeon Phi coprocessor. Enabling `ls1 mardyn` for Xeon Phi is ongoing research.

A different issue inhibiting the most efficient usage of the Sandy Bridge core is the lack of instruction level parallelism in the compute kernel. The evaluation of distance, potential energy and force on the particles requires significantly more multiplications than additions, thus the ADD unit cannot be fully utilized. Even worse, the calculation of the power-12-term of the LJ-12-6 requires a sequence of dependent multiplications. Therefore, the superscalarity of a Sandy Bridge core can not be exploited optimally, a fact we address by using Hypterthreading Technology as described below.

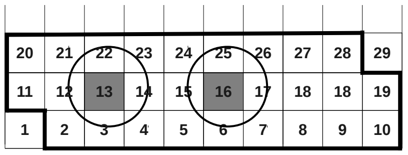
We restricted ourselves to AVX128 instructions for several reasons. In Section 2.2 we described that Intel Sandy Bridge is not able to handle AVX256 instructions at full speed. This fact would also forbid to use Hyperthreading efficiently as currently ports 2 and 3 inside the core can be used by different threads. Switching to AVX256, these ports are operated in paired mode, available to just one of both threads. Furthermore, we showed in the outlook of [2] that AVX256 instruction are only beneficial when increasing the cut-off radius. Last but not least we want to ensure that `ls1 mardyn` runs best on various x86 platforms. Besides Intel Sandy Bridge, AMD Interlagos plays an important role since this chip is used as processor in most of Cray's supercomputers. AMD Interlagos features two 128bit FPUs shared between two integer units. Therefore an AVX128 code is essential for best performance on Interlagos. With the current code base we only expect slight changes when moving to an Interlagos based machine.

3.2 Memory and Utilization Optimizations

In order to achieve the low memory requirement of only 32 Byte per molecule, we refined the linked-cells algorithm with the sliding window that was introduced in [3]. It is based on the observation that the access pattern of the cells can be described by a sliding window, which moves through the domain. After a cell has been searched for interacting particles for the first time in a time step, its data will be required for several successive force calculations with particles in neighboring cells. If the force calculation proceeds according to the cells' index as depicted in Fig. 4a, these data accesses happen within a short time period, until the interactions with all neighbors have been computed. While the cells in the window are accessed several times, they naturally move in and out of the window in FIFO order.



(a) Sliding window (cells in bold black frame) in 2D. Particles in cells in the window will be accessed several times, cells 2 through 23 are covered by the window in FIFO order. For the force calculation for the molecules in cell 13, cell 23 is searched for interacting particles for the first time in this iteration. The particles in cell 2 are checked for the last time for interactions.



(b) Extension of the sliding window for multi-threading. By increasing the window by 5 cells, two threads can independently work on three cells each: thread 1 works on cells 13, 14, 15; thread 2 works on cells 16, 17, 18. To avoid that threads work on same cells (e. g., thread 1 on the cell pair 15–25, thread 2 on 16–25), a barrier is required after each thread finished its first cell.

Fig. 4. Basic idea of the sliding window algorithm and extension for multi-threading

Particle data outside the the sliding window are stored in form of C++ objects in AoS-manner, only with position, velocity and an identifier. Per cell, particle objects are stored in dynamic arrays. When the sliding window is shifted further and covers a new cell, the positions and velocities of the particles in that cell are converted to SoA-representation. Additionally, arrays for the forces have to be allocated. The force calculation is now performed on the particles as described above. When a cell has been considered for the last time during an iteration, its particles are converted back to AoS-layout. Therefore, the calculation of forces, potential energy and virial pressure can be performed memory- and runtime-efficiently on the SoA, while the remaining parts of the simulation code can be kept unchanged according to their object-oriented layout. To avoid the overhead of repeated memory (de-)allocations when particle data in a cell are converted, we initially allocate dynamic arrays fitted to the maximum number of particles per cell for each cell in the window, and reuse that memory. Since the sliding window covers three layers of cells, these buffers consume a comparably small amount of memory, while the vast majority of the particles is stored memory-efficiently. At this point, it becomes apparent that the traversal order imposed by the sliding window also supports cache reusage: when particle data are converted to SoA-representation, that data are placed in the cache and will be reused several times soon after.

In order to reduce the memory requirement to 32 Byte per particle and to further improve the hardware utilization, this algorithm needs two further revisions: the time integration has to be performed on the fly, and opportunity for multi-threading needs to be created. Since the forces are not stored with the molecule objects, the time integration has to be performed during that conversion, i. e., the particles' new positions and velocities have to be calculated at that moment. Nevertheless, the correct traversal of the particles is ensured, because cells that have been converted are not required for the force calculation during this time step any more and the update of the linked-cells data structure, i.e. the assignment of particles to cells, takes place only between two time steps.

As stated above, the LJ-12-6 kernel is not well instruction-balanced, impeding the use of the superscalarity of a Sandy Bridge core. In order to make use of Hyperthreading Technology, we implemented a lightweight shared-memory parallelization. By extending the size of the sliding window as shown in Fig. 4b, two threads can perform calculations concurrently on three independent cells. Exploiting Newtons third law $F_{ij} = -F_{ji}$ for the force calculation and considering cell pairs only once, it must be avoided that threads work on directly neighboring cells simultaneously. Therefore, a barrier, causing comparably little overhead on a Hyperthreading core, is required after each thread has processed the first of its three cells. This allows the execution of one MPI rank per core with two (OpenMP-)threads to create sufficient instruction level parallelism, leading to a 12% performance improvement.

4 Strong and Weak Scaling on SuperMUC

In order to evaluate the performance of the MD simulation code `ls1 mardyn`, we executed different tests on SuperMUC. With respect to strong scaling behavior, we ran a scenario with $N = 4.8 \cdot 10^9$ particles, which perfectly fits onto 8 nodes; 18 GB per node are needed for particle data. Fig. 5 shows that a very good

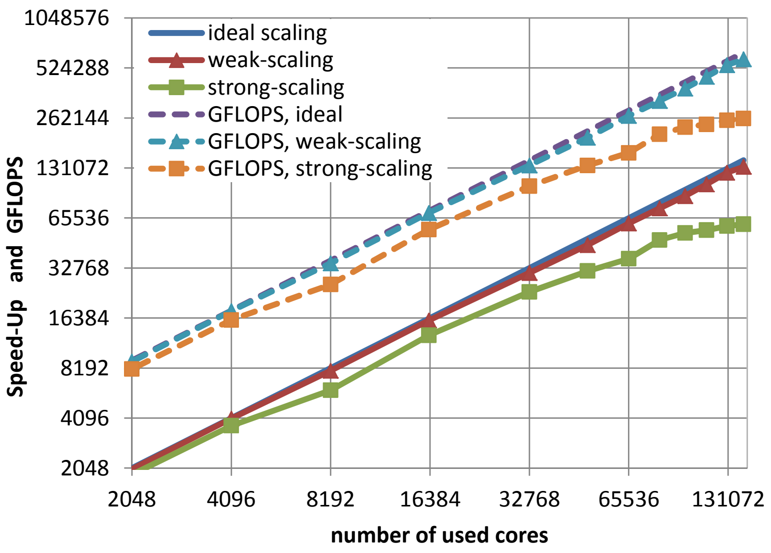


Fig. 5. Weak and strong scaling for 2048 to 146016 cores with respect to speed-up and GFLOPS on SuperMUC. Ideal scaling was achieved in case of weak scaling whereas as a parallel efficiency of 42% was obtained in the strong scaling tests. We cut off the plot at 2048 cores, here we obtained a parallel efficiency of 91.1% in case of strong scaling (compared to 128 cores) and 98.6% in case of weak scaling (compared to one core).

scaling was achieved for up to 146016 cores using 292032 threads at a parallel efficiency of 42 % comparing 128 to 146016 cores.

In this case, less than 20 MB ($5.2 \cdot 10^5$ particles) of main memory per node, which fits basically into the processors' caches, are used. This excellent scaling behavior can be explained by analyzing Fig. 6. Here we measured achievable GFLOPS depending on the number of particles simulated on 8 nodes. Already for $N = 3 \cdot 10^8$ particles (approx. 8% of the available memory) we are able to hit the performance of roughly 550 GFLOPS which we also obtained for $N = 4.8 \cdot 10^9$.

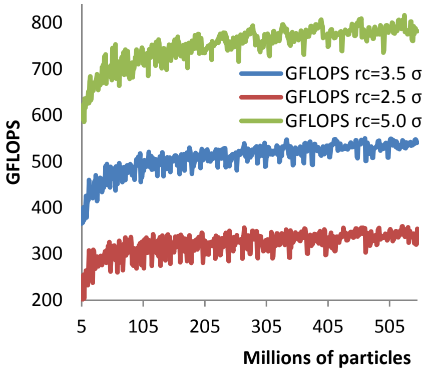


Fig. 6. GFLOPS dependeding on particle count and cut-off on 128 cores

It should be pointed out that the performance only decreases slightly for systems containing fewer particles (reducing the particle system size by a factor of 100): for $N = 10^7$ (which corresponds to the strong scaling setting in case of 146016 cores w.r.t. particles per node) we see a drop by 27% which only increases to the mentioned 58% when moving from 128 to 146016 cores. We have to note that the overall simulation time in this case was 1.5 s for 10 time steps, thereof 0.43 s were communication time. Since 0.43 s are roughly 29% of 1.5 s, it becomes clear that the biggest fractions of the 58% decrease are stemming from low particle counts per process and relatively high communication costs.

Moreover, we performed a weak scaling analysis which is, to our knowledge, the largest MD simulation to date. Due to MPI buffers on all nodes, we were not able to keep the high number of particles per node ($6.0 \cdot 10^8$) and were forced to reduce it to $4.52 \cdot 10^8$. Particularly, buffers for eager communication turned out to be the most limiting factor. Although we reduced them to a bare minimum (64 MB buffer space for each process), roughly 1 GB per node had to be reserved as we use one MPI rank per core. By keeping Fig. 6 in mind, we know that this slight reduction has no negative impact on the overall performance of our simulation. In case of 146016 cores we were able to run a simulation of $4.125 \cdot 10^{12}$ particles with one time step taking roughly 40 s. For this scenario, a speed-up of 133183 $\times$ (compared to a single core) with an absolute performance of 591.2 TFLOPS was achieved, which corresponds to 9.4% peak performance efficiency.

These performance numbers can be easily improved by increasing simulation parameters like the cut-off radius r_c which results in a higher vector-register utilization. However, preceding publications [18, 10, 4] used cut-off radii within the interval $2.5\sigma < r_c < 5.0\sigma$. Therefore we restricted ourselves to $r_c = 3.5\sigma$ in order to ensure fairness, please consult Fig. 6 for a performance comparison of `ls1 mardyn` for different cutoff radii in this interval.

5 Conclusions

In this paper we showed that MD simulations can be scaled up to more than 140000 cores and a multi-trillion ($4.125 \cdot 10^{12}$) number of particles on modern supercomputers. Due to the sliding window technique, only 32 Bytes are required per particle, and with the help of a shared memory parallelization and a carefully optimized force calculation kernel we achieved 591.2 TFLOPS, which is 9.4% of the system's theoretical peak performance.

We achieved not only perfect weak scaling, but also excellent strong scaling results together with a good performance of the kernel also for comparably small particle numbers per core. These properties are essential for the investigation of large inhomogeneous molecular systems. Such scenarios are characterized by highly heterogeneous particle distributions, which requires a powerful load balancing method implementation. Therefore, we are working on the incorporation of the load balancing from the original `ls1 mardyn` code.

As indicated during the force kernel's discussion, the current kernel implementation suffers from not fully exploited vector-registers. Increasing the net-usage of vector-registers is subject of ongoing research. The most promising instruction set is currently provided by the Intel Xeon Phi coprocessor which features a full blown gather/scatter implementation.

Beside tuning `ls1 mardyn` for better performance on emerging architectures, energy efficiency with focus on the energy to solution ratio is an additional research direction, especially when targeting MD scenarios with millions of time steps. Since SuperMUC is capable of dynamic frequency scaling, it provides an optimal testbed for such activities.

References

1. Buchholz, M., Bungartz, H.-J., Vrabec, J.: Software design for a highly parallel molecular dynamics simulation framework in chemical engineering. Journal of Computational Science 2(2), 124–129 (2011)
2. Eckhardt, W., Heinecke, A.: An efficient vectorization of linked-cell particle simulations. In: ACM International Conference on Computing Frontiers, Cagliari, pp. 241–243 (May 2012)
3. Eckhardt, W., Neckel, T.: 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, pp. 103–110. IEEE (2012)
4. Germann, T.C., Kadau, K.: Trillion-atom molecular dynamics becomes a reality. International Journal of Modern Physics C 19(09), 1315–1319 (2008)
5. Gou, C., Kuzmanov, G., Gaydadjiev, G.N.: SAMS multi-layout memory: providing multiple views of data to boost SIMD performance. In: Proceedings of the 24th ACM International Conference on Supercomputing, ICS 2010, pp. 179–188. ACM, New York (2010)
6. Heinecke, A., Pflüger, D.: Emerging architectures enable to boost massively parallel data mining using adaptive sparse grids. International Journal of Parallel Programming 41(3), 357–399 (2013)
7. Heinecke, A., Trinitis, C.: Cache-oblivious matrix algorithms in the age of multi- and many-cores. Concurrency and Computation: Practice and Experience (2013); accepted for publication
8. Horsch, M., Vrabec, J., Bernreuther, M., Grottel, S., Reina, G., Wix, A., Schaber, K., Hasse, H.: Homogeneous nucleation in supersaturated vapors of methane, ethane, and carbon dioxide predicted by brute force molecular dynamics. The Journal of Chemical Physics 128(16), 164510 (2008)
9. Kabadshow, I., Dachsel, H., Hammond, J.: Poster: Passing the three trillion particle limit with an error-controlled fast multipole method. In: Proceedings of the 2011 Companion on High Performance Computing Networking, Storage and Analysis Companion, SC 2011 Companion, pp. 73–74. ACM, New York (2011)
10. Kadau, K., Germann, T.C., Lomdahl, P.S.: Molecular dynamics comes of age: 320 billion atom simulation on bluegene/l. International Journal of Modern Physics C 17(12), 1755–1761 (2006)
11. Lindahl, E., Hess, B., van der Spoel, D.: Gromacs 3.0: a package for molecular simulation and trajectory analysis. Journal of Molecular Modeling 7, 306–317 (2001)
12. Olivier, S., Prins, J., Derby, J., Vu, K.: Porting the gromacs molecular dynamics code to the cell processor. In: IEEE International Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–8 (March 2007)
13. Peng, L., Kunaseth, M., Dursun, H., Nomura, K.-i., Wang, W., Kalia, R., Nakano, A., Vashishta, P.: Exploiting hierarchical parallelisms for molecular dynamics simulation on multicore clusters. The Journal of Supercomputing 57, 20–33 (2011)
14. Piazza, T., Jiang, H., Hammarlund, P., Singhal, R.: Technology Insight: Intel(R) Next Generation Microarchitecture Code Name Haswell (September 2012)

15. Rahimian, A., Lashuk, I., Veerapaneni, S., Chandramowlishwaran, A., Malhotra, D., Moon, L., Sampath, R., Shringarpure, A., Vetter, J., Vuduc, R., Zorin, D., Biros, G.: Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11. IEEE Computer Society, Washington, DC (2010)
16. Rapaport, D.C.: The Art of Molecular Dynamics Simulation. Cambridge University Press (2004)
17. Rapaport, D.C.: Multibillion-atom molecular dynamics simulation: Design considerations for vector-parallel processing. Computer Physics Communications 174(7), 521–529 (2006)
18. Roth, J., Gähler, F., Trebin, H.-R.: A molecular dynamics run with 5 180 116 000 particles. International Journal of Modern Physics C 11(02), 317–322 (2000)
19. Vrabec, J., Kedia, G.K., Fuchs, G., Hasse, H.: Comprehensive study of the vapour-liquid coexistence of the truncated and shifted lennard-jones fluid including planar and spherical interface properties. Molecular Physics 104(9), 1509–1527 (2006)