Computer Physics Communications 290 (2023) 108760

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Code modernization strategies for short-range non-bonded molecular dynamics simulations ☆

James Vance [a,*], Zhen-Hao Xu [a], Nikita Tretyakov [a], Torsten Stuehn [b], Markus Rampp [c], Sebastian Eibl [c], Christoph Junghans [d], André Brinkmann [a,*]

[a] Zentrum für Datenverarbeitung, Johannes Gutenberg-Universität Mainz, Mainz, Germany
[b] Max Planck Institute for Polymer Research, Mainz, Germany
[c] Max Planck Computing and Data Facility, Garching, Germany
[d] Applied Computer Science Group, Los Alamos National Laboratory, Los Alamos, NM, USA

ARTICLE INFO

Article history:
Received 13 April 2022
Received in revised form 30 March 2023
Accepted 19 April 2023
Available online 3 May 2023

Keywords:
Molecular dynamics
High performance computing
HPX
MPI
ESPResSo++

ABSTRACT

Modern HPC systems are increasingly relying on greater core counts and wider vector registers. Thus, applications need to be adapted to fully utilize these hardware capabilities. One class of applications that can benefit from this increase in parallelism are molecular dynamics simulations. In this paper, we describe our efforts at modernizing the ESPResSo++ simulation package for molecular dynamics by restructuring its particle data layout for efficient memory accesses and applying vectorization techniques to benefit the calculation of short-range non-bonded forces, which results in an overall three times speedup and serves as a baseline for further optimizations. We also implement fine-grained parallelism for multi-core CPUs through HPX, a C++ runtime system which uses lightweight threads and an asynchronous many-task approach to maximize concurrency. Our goal is to evaluate the performance of an HPX-based approach compared to the bulk-synchronous MPI-based implementation. This requires the introduction of an additional layer to the domain decomposition scheme that defines the task granularity. On spatially inhomogeneous systems, which impose a corresponding load-imbalance in traditional MPI-based approaches, we demonstrate that by choosing an optimal task size, the efficient work-stealing mechanisms of HPX can overcome the overhead of communication resulting in an overall 1.4 times speedup compared to the baseline MPI version.

1. Introduction

As the growth of processor frequency continues to plateau, modern HPC systems increasingly rely on greater concurrency and parallelism to deliver more performance. This comes in the form of increasing core counts and providing wider vector registers. However, as core counts increase, traditional parallelization methods that rely on MPI contend with fewer available memory per core and their performance faces increased sensitivity to load imbalances and synchronization mechanisms. Full utilization of wider vector registers also means critical parts of applications need to be rewritten and often requires more complex solutions. Thus, applications have to be adapted and modernized in order to fully maximize new hardware capabilities and overcome memory constraints.

Molecular dynamics (MD) simulations play a significant role in the study and discovery of new materials especially in soft matter research. They also act as an ideal example to harness more recent hardware capabilities since the calculations involved offer different ways of parallelization, such as through force decomposition, atom decomposition and spatial decomposition schemes [1]. Consequently, many simulation packages have been developed over the past decades that can run on machines with up to hundreds of thousands of cores such as LAMMPS [2], GROMACS [3], and NAMD [4].

In this paper, we focus on ESPResSo++, an open-source software for performing molecular dynamics simulations of condensed soft matter systems [5,6]. It is built on a C++ backend with MPI-based communication enabling fast parallel execution. It also provides a Python frontend for convenient scripting and analysis. The foremost guideline in the design of ESPResSo++ is extensibility which allowed it to become a sandbox in which users can easily develop

☆ The review of this paper was arranged by Prof. Weigel Martin.

* Corresponding authors.
E-mail addresses: jnvance@outlook.com (J. Vance), brinkman@uni-mainz.de (A. Brinkmann).

https://doi.org/10.1016/j.cpc.2023.108760

new methods and algorithms [5]. However, this means that decisions taken in favor of extensibility may not always result in the best optimized code for performance on modern hardware.

To address this, we apply SIMD vectorization and multithreaded task-based parallelism to improve the performance of ESPResSo++ on modern multi-core CPUs. We specifically focus on molecular dynamics simulations involving short-range non-bonded forces whose calculations take up a significant portion of the total simulation time and can benefit the most from these additional layers of parallelism.

First, we maximize the utilization of wider vector registers through SIMD vectorization. SIMD, which stands for single instruction multiple data, allows a single instruction to simultaneously process multiple vector elements whose size depends on the bit length of registers known as the SIMD width [7]. For example, Intel has been developing extensions of the x86_64 instruction set over the years including Streaming SIMD Extensions (SSE) which support 128-bit registers, Advanced Vector Extensions 2 (AVX) for 256-bit registers, and AVX-512 for 512-bit registers. These are often provided as low-level functions called intrinsics which can be invoked directly from the source code but whose availability may vary among different architectures.[1] To benefit from these extensions while retaining code portability, other approaches can be used such as directives that give hints to the compiler about vectorizable loops such as `#pragma vector` provided by Intel[2] and `#pragma omp simd` from OpenMP,[3] and libraries that implement generalized high-level abstractions to the underlying intrinsics such as the Vc library [8]. SIMD optimizations can improve the performance of MD simulations independent of the application domain and we have evaluated their impact based on a standard Lennard-Jones fluid simulation and a polymer melt simulation running on a single node as benchmarks.

Next, we address the need for thread-level parallelism. Many applications running on computing clusters, including ESPResSo++, rely on the MPI programming paradigm that explicitly synchronizes calculation steps through communication. However, as the number of cores per node increases, applications become more sensitive to load imbalances and operating system noise causing wait times to increase. Such scenarios prevent good application scaling for inhomogeneous systems unless a good load-balancing strategy is employed [9–11]. For a more in-depth overview of load-balancing strategies in MD see sec. 5.3 of [2].

An alternative approach to the MPI programming model are asynchronous many-task runtime systems [12]. The number of tasks in many-task computing is typically significantly bigger than the number of cores. The many-task scheduler can therefore simply resolve load-imbalances by moving some tasks from busy to idle cores or even by moving tasks across nodes. One example is the Charm++ asynchronous programming model [13] which serves as the underlying runtime system of the MD simulation package NAMD [4] and of a highly scalable implementation of the finite element method [14]. Another emerging candidate library is HPX, which is a C++ standards-compliant library that provides wait-free asynchronous execution of tasks and synchronization through futures [15].

We have integrated HPX into ESPResSo++ for load-balancing on a single node, whereas we currently kept MPI to communicate between nodes. The HPX evaluation starts by again investigating balanced a Lennart-Jones fluid and a polymer melt to understand the overheads of HPX. We then investigate an unbalanced spherical configuration in which all particles are within a circle. The evaluation shows that the HPX overhead can be bounded by 5% for perfectly balanced simulations and that HPX can improve performance by a factor of 1.4 for our unbalanced setting.

In this paper, we therefore present our efforts at modernizing ESPResSo++ by optimizing its data layout, enabling SIMD vectorization and integrating the capabilities for fine-grained parallelism offered by HPX. Our aim is to evaluate the benefits of using HPX to provide node-level parallelism compared to traditional MPI-based parallelism.

This paper is structured as follows: In Section 2, we introduce some of the general concepts and calculations involved in molecular dynamics simulations and the particular design principles of ESPResSo++. We also describe the details of the HPX library that are relevant to our work. In Section 3, we illustrate the serial optimizations we performed on the data layout and critical loops of ESPResSo++ and show how we extended the pure MPI-based parallelization in ESPResSo++ to an MPI+HPX parallelization model. Then, we evaluated the performance of these optimizations and show the results in Section 4. We list related publications in Section 5. Finally, we conclude our work in Section 6 and discuss possible extensions to distributed parallelism in Section 7.

2. Background

In this section, we will recall the standard methods of performing molecular dynamics simulations on parallel machines. We will also discuss particular features and design principles of the target MD application, ESPResSo++. Finally, we will introduce the asynchronous many-task programming paradigms that will be used in the parallel optimizations of ESPResSo++.

2.1. Molecular dynamics simulations

Molecular dynamics (MD) simulations are used to model the dynamical properties of interacting particles in a system [16]. The central task of MD simulations is to integrate Newton's equations of motion

$$m_i \cdot \ddot{\mathbf{r}}_i = \mathbf{F}_i \tag{1}$$

where m_i is the mass of the particle i, $\mathbf{r}_i$ is its position vector and $\mathbf{F}_i$ is the total force on i from interactions with all other particles in the simulation. These forces can be derived from a potential energy U that depends on the coordinates of all N particles $\mathbf{r}^N$ such that $\mathbf{F} = -\nabla U$. The forces may include pairwise interactions, three-body interactions and higher order many-body interactions.

As shown in Fig. 1, the simulation evolves in an iterative process in which the positions are used to calculate the forces, which are in turn used to update the velocities and positions for the next time step. For the latter, known as the INTEGRATE step, numerical integrators such as the Velocity Verlet method are commonly employed [17,18].

2.1.1. Short-range non-bonded potentials

Short-range force models comprise the most common class of interactions that are used in MD simulations. They physically result from electronic screening effects of long-range interactions and they also reduce computational load by restricting force contributions to a smaller region around each particle [1].

One commonly used example is the Lennard-Jones (LJ) potential which is given by

$$V(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2}$$

[1] https://software.intel.com/sites/landingpage/IntrinsicsGuide.
[2] https://software.intel.com/content/dam/develop/external/us/en/documents/cpp_compiler_classic.pdf.
[3] https://www.openmp.org/spec-html/5.1/openmp.html.

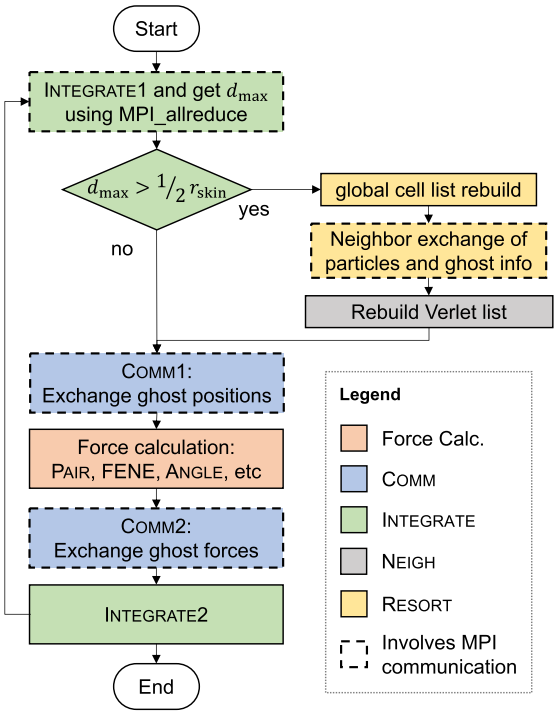


Fig. 1. Velocity Verlet integration scheme used by ESPResSo++ which updates particles' velocities and positions in two half-steps (INTEGRATE1 and INTEGRATE2) and uses a skin size r_{skin} to determine the frequency of particle re-binning done by RESORT. Colors indicate groupings of different sections together while dashed line borders (- - -) indicate sections that involve MPI communication. (For interpretation of the colors in the figures, the reader is referred to the web version of this article.)

where σ and ε are parameters that describe the size of the particle and the depth of the potential well, respectively. If, like in the example of Equation (2), the interaction decreases sufficiently rapidly to zero, the search for interacting particle pairs can be confined to within a distance $r < r_{cut}$ in order to calculate the total force on every particle.

2.1.2. Neighbor lists

Determining which particles are within cutoff of each other can be facilitated by binning the particles into cubic cells of minimum length r_{cut}. Then the search for possible interaction pairs could be restricted to its cell and the 26 surrounding neighbor cells. For interactions that obey Newton's third law, the symmetry of force calculations, $\vec{F}_{ij} = -\vec{F}_{ji}$, can be used to reduce the search to 13 neighbor cells. The interacting particles may then be recorded in a list of pairs known as a Verlet list [17]. To reduce the computational cost of building the Verlet list, an additional buffer of thickness r_{skin} is added to the cell size such that $r_{cell} \geq r_{cut} + r_{skin}$. This allows some particles to move out of the neighborhood for some time steps without needing to reassign particles into cells and to recompute the Verlet lists frequently.

2.1.3. Spatial decomposition

The most common way to parallelize molecular dynamics simulations on distributed-memory machines is by subdividing the simulation box into smaller *nodes*, each assigned to one processor [1]. Every processor takes care of computing the forces and updating positions and velocities of particles in its own node. Particles are then allowed to enter and exit a node by reassigning them to a different cell and node during the so-called RESORT step.

Each node is composed of *real cells* that belong to the spatial domain of that node and a surrounding layer of *ghost cells* that contain copies of particles from neighboring nodes [5]. The ghost cells are needed to correctly account for all force contributions at the node boundaries and they are updated at each time step using local message-passing communication, referred to as the COMM step.

2.2. ESPResSo++

Started in 2008 as a joint collaboration between the Fraunhofer SCAI Institute and the Theory Group of the Max Planck Institute for Polymer Research, ESPResSo++ has become a central tool to perform coarse-grained simulations and to provide a sophisticated and versatile framework for the development of new methods and algorithms [5,6]. In order to simplify data exchange in complex workflows in combination with other software packages such as LAMMPS [2], GROMACS [3] and VOTCA [19,20], interfaces to these programs have been implemented. ESPResSo++ supports a variety of standard MD algorithms (e.g. Velocity Verlet Integrator in NVE, NVT, NPT Ensembles) as well as several advanced methods (e.g. AdResS, H-AdResS, equilibration of polymer melts, Lattice Boltzmann, 3-body non-bonded Stillinger-Weber or the Tersoff potential) and modern data structures (e.g., H5MD) in parallel file I/O environments. More recent and ongoing developments include the integration of the ScaFaCoS library[4] for long range interactions and Lees-Edwards boundary conditions [21]. Collaborative development within an international group of scientists, including continuous integration (CI) and unit testing, happens on the GitHub platform.[5]

2.3. Asynchronous many-task programming models

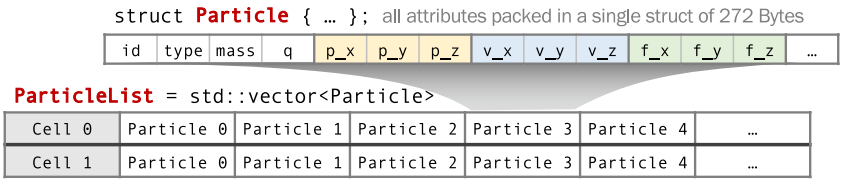
Most applications that run on modern parallel architectures rely on the MPI+X programming model [22] in which MPI handles inter-node communication and "X" is a hardware-dependent node-level programming paradigm such as OpenMP,[6] CUDA,[7] and OpenACC,[8] or a performance-portability framework such as Kokkos [23] and RAJA [24]. Solutions like QUO have also been implemented to handle thread-level heterogeneity while ensuring full utilization of CPU cores [25].

An emerging class of new parallel programming paradigms are asynchronous many-task (AMT) runtime systems which can provide fine-grained parallelism that can handle a large number of threads and efficiently distribute work across a node. One such example is HPX. The High Performance ParalleX (HPX) library is a C++ runtime system that can handle fine-grained parallelism in modern architectures [15]. It allows users to write code based on a task dependency graph and takes care of the thread scheduling and execution of tasks and communication between distributed compute nodes.

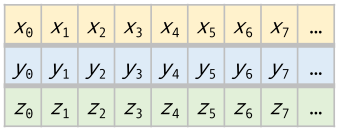
We specifically focus on the thread management aspects of HPX to write and execute multithreaded code. This is achieved through the concept of *futurization*. HPX provides an asynchronous return type `hpx::future<T>` which hides the execution of function calls and returns immediately even though the function has not completed its execution. The futures can also be consumed by local control objects (LCOs) such as `hpx::future<T>::then`, `hpx::wait_all` and `hpx::shared_future`, which enable the flow of task dependencies. The API also provides high-level parallel algorithms aligned to current and proposed C++ standards and extends them with asynchronous versions [15].

In the distributed case, HPX also provides an Active Global Address Space (AGAS) which registers objects with global identifiers

[4] http://www.scafacos.de.
[5] https://www.github.com/espressopp/tree/hpx4espp.
[6] https://www.openmp.org/wp-content/uploads/HybridPP_Slides.pdf.
[7] https://developer.nvidia.com/blog/introduction-cuda-aware-mpi/.
[8] https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf.



(a) Original layout in which each `Particle` struct contains 272 bytes and every cell has a `vector` of these structs.



(b) Improved layout using structure of arrays (SoA) in which each attribute is stored as a separate array.

Fig. 2. Original and improved layout for particle data.

that allows remote function calls while hiding explicit message passing. However, since we are focusing on node-level optimizations, AGAS will not be used in this work and inter-node communication will be done through MPI.

3. Optimizations

In this section, we discuss the key steps taken to improve the performance of ESPResSo++. We start with baseline optimizations that involve transforming the data layout and ensuring that use of SIMD vectorization is maximized by the compiler. Then, we implement thread-level parallelism by integrating the HPX runtime system into ESPResSo++.

The standard implementation of ESPResSo++ is heavily dependent on the `Particle` data structure. Since we want to maintain compatibility with the analysis tools in the current implementation while taking advantage of the performance improvements, we have chosen to implement the following optimizations as additional submodules within the `espressopp` Python module. The `vec` submodule contains the data structures for improved particle data layout described in Section 3.1 and the vectorized routines for force calculation and integration described in Section 3.2. On the other hand, the `hpx4espp` submodule contains the corresponding routines that use the node-level parallelism provided by HPX described in Section 3.3. This means that once the corresponding optimized versions of subroutines have been implemented, users only need to modify their python scripts by indicating the submodule. For example, the original version of the FENE potential is instantiated by `espressopp.interaction.FENE`, while for the vectorized version the `vec` submodule is specified resulting in `espressopp.vec.interaction.FENE`. The introduction of these submodules is facilitated by the extensibility and object-oriented design of ESPResSo++.

3.1. Improved data layout

Fig. 2a shows the particle data layout in the standard implementation of ESPResSo++. All data for a single particle are stored in a large struct of 272 bytes called `Particle` including basic attributes such as type, position, velocity, force, and other attributes required for more complex simulations. The particles are further grouped into cells, each containing its own `std::vector<Particle>`. This layout was chosen since it provides the most straightforward way of accessing particle data. This makes it compatible with the goal of extensibility and the object-oriented design of ESPResSo++, and it is well suited for algorithms working on nearly all particle data [5]. However, most of the time-consuming operations, such as force calculation and neighbor search, rely on only a few attributes. Performing those operations with this layout requires strided memory accesses through different attributes and operations with this layout usually cannot be auto-vectorized by compilers.

In order to optimize these operations, we employ an alternative layout in the form of a structure of arrays (SoA) in which every attribute of type `T` is stored in its own separate `std::vector<T>` as shown in Fig. 2b. SoA is known to improve cache re-use and is amenable to compiler-assisted vectorization of time-critical loops [7].

To maintain cache coherence and prevent false sharing during multithreaded accesses, the member arrays are allocated aligned to 64-byte boundaries. For `std::vectors` this was done by using the `aligned_allocator` provided by the Boost.Align library.[9] The entries between the end of one cell and the start of the next cell are also padded with dummy particles that lie far away from the simulation box to ensure that the entries for the next cell are also properly aligned.

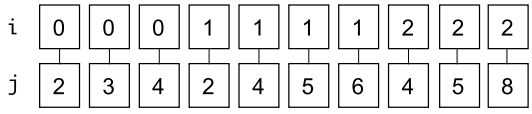
The SoA layout is utilized only during the INTEGRATE and force calculation steps while the original layout is used in the RESORT stage since it requires all particle attributes. Thus, to ensure accuracy, the data between them are synchronized at the beginning and end of the simulation, and before and after the intermediate RESORT stages.

3.2. Vectorization

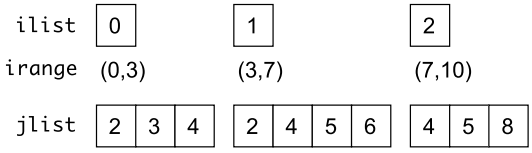
In this paper, we are specifically interested in optimizing for the vectorization capabilities provided by AVX-512 on Intel Skylake and Ice Lake processors. However, we also want to keep critical loops portable to other compilers and CPU architectures while taking advantage of the SoA data layout. Thus, we mainly rely on compiler auto-vectorization to ensure that SIMD instructions are efficiently utilized. Further directives were added in the code to inform the compiler of alignment and to assert the absence of data dependencies among vector entries [26].

The most time-consuming sections of typical short-range non-bonded MD simulations are the neighbor list rebuild and the force calculation. Particularly for pair potentials, these steps require an efficient representation of the Verlet list. In the current version of ESPResSo++, it is represented as a list of pairs of pointers (i, j) to the corresponding `Particle` structs as shown in Fig. 3a.

[9] https://www.boost.org/.



(a) List of pairs representing the indices of or pointers to interacting particles used in the standard implementation of ESPResSo++.



(b) SortedList based on [7] which groups together the pairs that have the same first particle.

Fig. 3. Possible representations of a Verlet list.

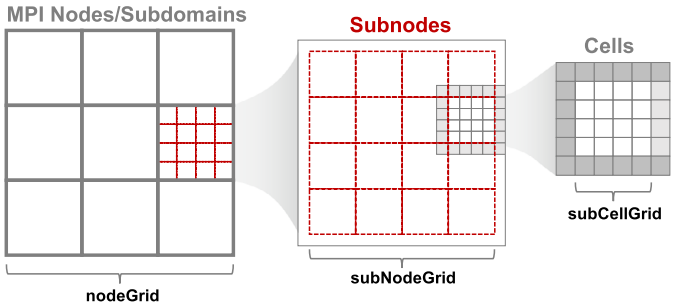


Fig. 4. MPI nodes are further divided into subnodes which determine the task granularity.

In our work, we use a more compact and vector-friendly representation known as a SortedList [7]. Here the j-particles that interact with the same i-particle are grouped together and stored in a contiguous list as shown in Fig. 3b. Thus, during force calculations the SortedList is traversed using a double for-loop: first, through the array of indices `ilist` and ranges `irange`, and then through the entries of `jlist` that are contained within the range. Since every i comes with distinct j entries, this structure leads to the possibility of applying vectorization on the inner for-loop over j.

3.3. Node-level parallelism using HPX

To enable multithreaded parallelism with HPX, we further divide the nodes into smaller *subnodes* as shown in Fig. 4. All computation needed for one subnode is packaged into one HPX task. Each subnode contains only the particle data of the cells that belong to it. This allows some operations such as the integration of positions and velocities to be done in parallel for each subnode without needing explicit locking of resources. However, force calculation requires modifying the force values of two or more particles that may reside in different subnodes. To avoid race conditions in the multithreaded version, Newton's third law is not used for interactions between particles belonging to different subnodes so that their force values are calculated separately. This allows the force calculation for each subnode to be run concurrently but at the expense of some redundant force calculations at the subnode boundaries.

The number of cells within the subnode grid determines the task granularity. If the subnode is too small, execution would be dominated by overheads, in particular ghost layer exchange. In contrast, a large grid size results in fewer subnodes which reduces HPX's work stealing capabilities. Thus, the number of subnodes per core, known as an oversubscription factor, has to be tuned in order to find the optimal point between overheads and starvation [27,28]. This tuning procedure could be done manually by performing several runs of a few time-steps while varying the number of subnodes at each run, starting with the number of threads per MPI locality until no further decrease in elapsed time is recorded. This optimal number of subnodes will vary for every simulation system depending on the task size and relative load imbalances among the resulting subnodes with the assumption that the relative load distribution does not vary too much during the simulation. With this approach we can leverage the HPX work stealing capabilities to achieve load balancing between the threads of one MPI rank. Since the number of tasks is in general higher than the number of threads new tasks get assigned to a thread as soon as it finishes its work.

To execute the MD operations in parallel across multiple threads, we rely on the C++ standards-compliant parallel algorithms provided by HPX. For the parallel execution of plain loops over subnodes we use `hpx::parallel::for_loop` and for those involving reductions we use `hpx::parallel::transform_reduce`. Their signatures are similar to the C++ Standard Template Library (STL) algorithms of the same name except that the first argument in HPX requires an execution policy which indicates whether and how to perform the algorithm in parallel [15]. Execution policies are gradually being adopted into the C++ standard, so that calls to functions in the `hpx` namespace may be replaced with `std` in the future.

4. Performance evaluation

Evaluation was performed on three different computing facilities: the Mogon II supercomputer at the Johannes Gutenberg University Mainz which is equipped with Intel Skylake processors, the Raven HPC System at the Max Planck Computing and Data Facility (MPCDF) which has the newer Intel Ice Lake processors, and an AMD EPYC 7452 cluster also at the MPCDF. Hardware specifications and compilers that were used are described in Table 1.

We used the following software versions: HWLOC 2.2, Boost 1.72.0, Python 3.8 and HPX 1.5.1 [29]. Binaries were compiled with Intel C++ Compiler with the flags "-O3 -xHost -qopt-zmm-usage=high -restrict" to enable full AVX-512 optimizations on Mogon and Raven, and with GCC compiler with flags "-O3 -march=native" on the AMD cluster.

Two simulation systems were used to evaluate performance – Lennard-Jones and polymer melts. The quantities used here will be expressed in Lennard-Jones dimensionless units with $m = \varepsilon = \sigma = 1$ which also results in reduced units of time since $(\varepsilon/m\sigma^2)^{1/2} = 1$ [16].

Table 1
Specifications of the compute nodes used for performance evaluation.

Name	Mogon	Raven	AMD
Processor	Intel Gold 6130 (Skylake)	Intel Xeon IceLake-SP 8360Y	AMD EPYC 7452
Base frequency	2.10 GHz	2.40 GHz	2.35 GHz
Sockets per node	2	2	2
Cores per socket	16	36	32
Vector Extensions	AVX-512	AVX-512	AVX2
Compiler	ICC 19.1.1.217	ICC 2021.3.0	GCC 11.2.0

The Lennard-Jones simulation was initialized with particles in a cubic lattice at a density $\rho = 0.8442$, and a cutoff distance of $r_{\text{cut}} = 2.5$ and a buffer thickness of $r_{\text{skin}} = 0.3$ were set. A Langevin thermostat was introduced to equilibrate the particles to some target temperature T.

We also prepared a polymer melt simulation containing ring polymers of chain length 200 and density $\rho = 0.85$. A repulsive Lennard-Jones interaction exists between all monomers within a cutoff distance $r_{\text{cut}} = 2^{1/6}$ and skin size $r_{\text{skin}} = 0.4$. Aside from the short-range non-bonded interactions, the polymer melt simulation includes bonded interactions composed of a FENE potential between pairs along the chains and a cosine potential on triples that form angles [30].

The simulations were executed with a fixed step size $\Delta t = 0.005$. We measured the elapsed time of the integrator loop and excluded any initialization steps such as setting up classes and reading particle data. Within this period, the timings of the following key sections were also collected:

- Forces - calculating non-bonded (LJ/Pair) and bonded (FENE, Angle) interactions
- Comm - communicating positions and collecting forces for ghost layers
- Integrate - updating positions and velocities
- Neigh - rebuilding neighbor lists
- Resort - sorting particles to cells and nodes, and copying particle data between original and SoA layout

These correspond to the groupings of sections described in Fig. 1 in which timers for sections with the same color are combined (e.g. Comm1 + Comm2 = Comm). The values presented are averages across all MPI ranks.

4.1. Baseline optimizations

We first evaluate the performance benefits following the transformation of the particle data layout to SoA and the vectorization of critical loops. To do this, we compare three cases:

- orig - the original implementation,
- soa - the implementation optimized with SoA particle data layout but with auto-vectorization disabled using the flags "-no-vec -no-simd" on ICC and "-fno-tree-vectorize" on GCC, and
- vec - the fully vectorized implementation.

We performed the measurements on a single node on each of the three machines listed in Table 1 using the Lennard-Jones fluid and polymer melt simulation as benchmark cases. We present the timing results in Fig. 5 and Figure S1.

As our first benchmark case, we simulate a Lennard-Jones fluid containing $N = 262,144$ particles running for 1000 time steps and equilibrated to $T = 1.0$. The overall elapsed time and the speedup with respect to orig are shown in Figs. 5a-5b. On Mogon, we observe a 2× speedup from orig to soa due to the change in data layout, and a further 1.5× speedup from soa to vec due to the vectorization of the Lennard-Jones interaction and the neighbor list rebuild. This is reduced on the AMD node to 1.5× and 1.09× respectively since the processor has AVX2 extensions which has a smaller vector width and reduced vectorization capabilities.

Table 2
Actual speedups (S) from soa to vec and corresponding ideal speedups (S_{max}) calculated from (3).

	W	S^{LJ}	$S^{\text{LJ}}_{\text{max}}$	S^{PM}	$S^{\text{PM}}_{\text{max}}$
Mogon	8	1.55	2.39	1.09	1.24
Raven	8	1.65	2.07	1.09	1.25
AMD	4	1.09	1.86	1.04	1.26

The elapsed time of different sections on a full node is shown in Figs. 5e-5f. Speedups can be observed in all sections that use the SoA layout. There is an overhead in the Resort section due to the additional copying of data from the SoA layout to the unoptimized layout on which the re-binning is performed, but this is compensated by the larger speedups in the other sections. Among the different code sections, speedups resulting from vectorization are more significant in the force calculation and neighbor list rebuilds because these sections involve a traversal of the SortedList which has a higher trip count than the iteration through the particles in the other sections.

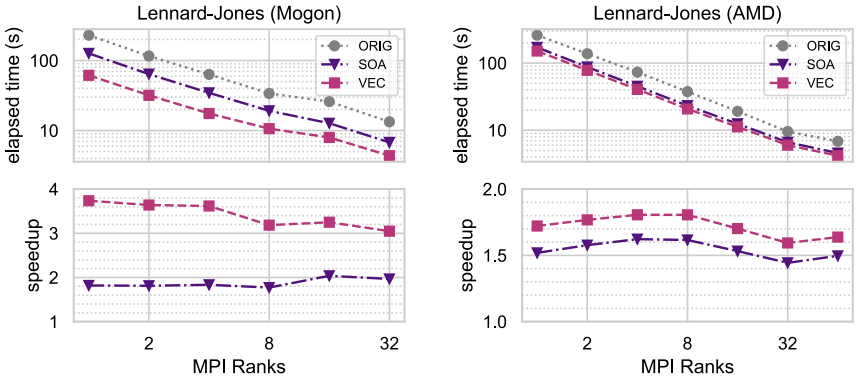
We also performed the same comparison on a polymer melt simulation with $N = 320,000$ particles and show the results in Figs. 5c-5d. On Mogon, a 2× speedup is achieved from orig to soa, similar to the LJ simulation. However, the speedup from soa to vec was reduced to only 1.07×. This is because the smaller cutoff for the pair potential results in fewer neighbors per particle at an average of 9.4 compared to 41.2 for the LJ setup, which in turn results in fewer iterations for the vectorized inner loop of the SortedList traversal in Pair and Neigh. Also, the other forces — FENE and Angle — were not vectorized in the same manner since they require mechanisms for conflict detection which are not yet supported by auto-vectorization. For the simulation on Raven, speedup benchmarks for both systems are similar to those on Mogon and corresponding data are given in the supplementary information.

The speedups we obtained from the soa implementation to the vec implementation can be compared with the ideal speedup which can be calculated by assuming perfect vectorization of the Pair and Neigh operations. If we denote the execution time for the non-vectorized sections as t_{rest}, we can quantify the maximum speedup as

$$S_{\text{max}} = \frac{t_{\text{rest}} + t_{\text{Pair}} + t_{\text{Neigh}}}{t_{\text{rest}} + (t_{\text{Pair}} + t_{\text{Neigh}})/W} \tag{3}$$

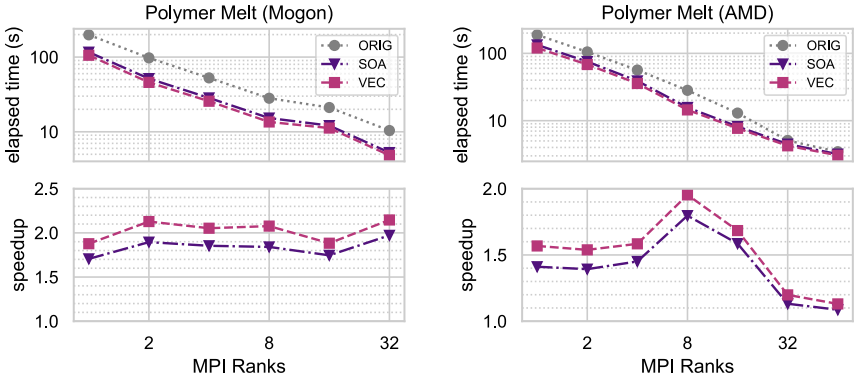
where the execution times t_{Pair} and t_{Neigh} are those obtained with soa, and W denotes the vector width or how many double-precision floating point numbers can fit in the register of the vectorized instructions (8 for AVX-512 and 4 for AVX2).

We apply Equation (3) to the data in Figs. 5e-5f, and present the results in Table 2. On Mogon, we find that for the Lennard-Jones simulation $S^{\text{LJ}}_{\text{max}} = 2.39$ compared to our measured $S^{\text{LJ}} = 1.55$ and for the polymer melt simulation $S^{\text{PM}}_{\text{max}} = 1.24$ whereas we obtained $S^{\text{PM}} = 1.09$ in our implementation. This difference is mainly due to the fact that the j-particles in the SortedList are not contiguous and have to be gathered before being simultaneously processed by the SIMD instructions. Similar values were obtained on Raven (Figure S1c) since its Ice Lake processor has the same AVX-512 vectorization features as Skylake on Mogon. On the other hand, the speedups obtained on the AMD machine were relatively reduced due to the shorter SIMD width and the fewer vectorization capabilities of the AVX2 instruction set.



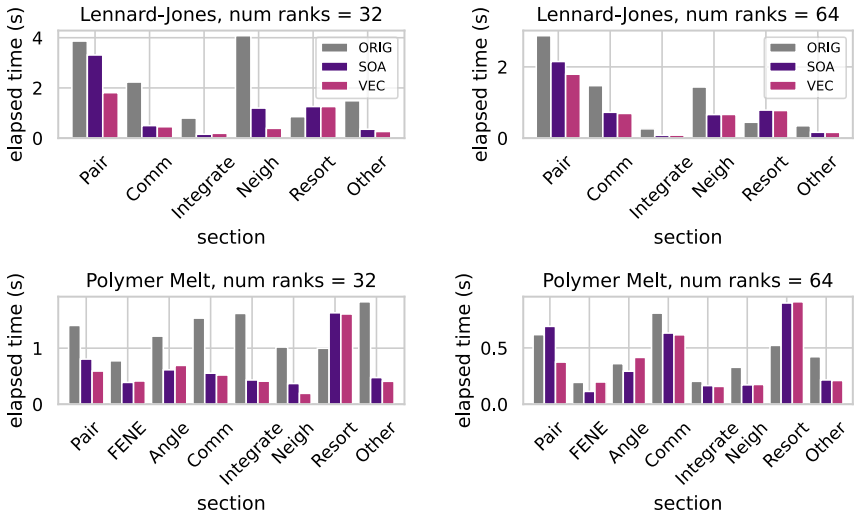
(a) LJ fluid on Mogon

(b) LJ fluid on AMD node



(c) Polymer melt on Mogon

(d) Polymer melt on AMD node



(e) Sectional time on Mogon.

(f) Sectional time on AMD Node

Fig. 5. Strong scaling behavior on one node of Mogon and AMD. Speedups shown are calculated with respect to ORIG at the same number of MPI ranks. A higher speedup is obtained for the LJ fluid due to its pair potential having a larger cutoff distance than the one used for the polymer melt simulation. For elapsed time, lower is better.

We also compare our vectorized implementation with LAMMPS which provides an Intel-optimized USER-INTEL package [31]. We ran a strong scaling comparison against the original implementation and fully vectorized version of ESPResSo++ and present the results in Fig. 6. We find a similar performance of LAMMPS USER-INTEL and our optimized version of ESPResSo++ with the latter being faster up to 128 MPI ranks. However, scalability is eventually limited by the RESORT stage which is still based on the original implementation and could be further improved in future implementations. Nevertheless, we have shown that even our SIMD optimizations already resulted in comparable overall performance with LAMMPS.

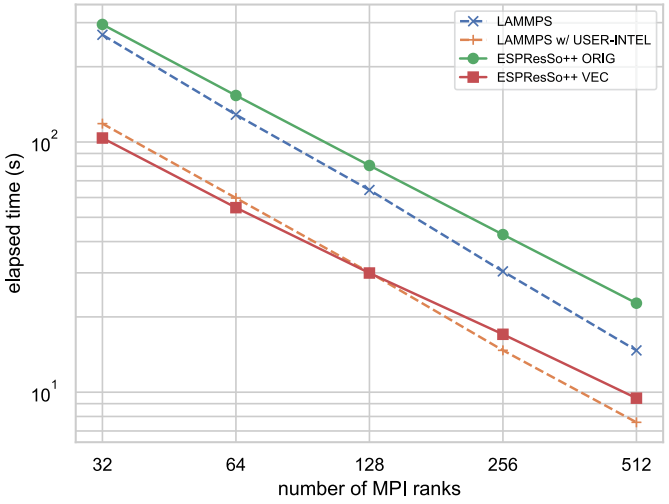


Fig. 6. Strong scaling comparison with LAMMPS on Mogon.

4.2. MPI vs HPX comparison

In this section, we evaluate our molecular dynamics implementation with HPX shared-memory parallelism in comparison to traditional MPI-based intra-node parallelism. We specifically investigate two cases: a spatially homogenous simulation to measure the overheads resulting from the HPX implementation, and a spatially inhomogeneous simulation to demonstrate the load-balancing capabilities of HPX. In particular, we focus on single node performance since we want to investigate the benefits from the work-stealing capabilities of HPX. The MD simulations were executed on the compute nodes listed in Table 1 equipped with two sockets. For the HPX version, one MPI rank is assigned to each socket and the number of HPX threads for each rank is equal to the number of cores per socket. For the baseline MPI version, one MPI rank with one thread is assigned to each physical core, which corresponds to the fully vectorized implementation evaluated in the previous section.

We start with a spatially homogenous case in which we expect the simulation to be dominated by implementation overheads. We use the same Lennard-Jones setup from the previous section which contains $N = 262,144$ particles and forms a cell grid of $(24, 24, 24)$. For the HPX version, we perform the simulations for different numbers of subnodes per MPI rank, n_{sub}, starting from the number of cores on each socket and gradually increasing by a factor of two until no further subdivision is possible. This autotuning procedure allows us to find the optimal value of n_{sub}. For MPI version, we run only one n_{sub} per MPI rank.

We first performed the evaluation on a single socket to isolate our measurements from the effects of MPI communication. From the results shown in Fig. 7a, an immediate drop in elapsed time can be seen from one to two n_{sub} per core followed by a steady slowdown as n_{sub} continues to increase. This means that executing two subnodes per thread is already sufficient since the Pair section does not improve anymore after this point. On the other hand, as n_{sub} increases, the number of force calculations that do not use Newton's third law also increases, which in turn increases the time for the Pair section. At the optimal point for Mogon at two n_{sub} per core, we find that the HPX implementation is 5% slower compared to the traditional MPI implementation where the overhead originates from task sizes of the Integrate section being too small and the increase in necessary but redundant force calculations in the Pair section. For AMD and Raven, we see from Figs. 7b and S2a that the optimal point is at four n_{sub} per core and there is higher overhead from the Pair force calculation.

The same simulation was then performed on two MPI ranks assigned to two sockets with the same number of HPX threads for each rank. From Figs. 7c-7d we see similar behavior except for a uniform increase in communication time. On Mogon, this results in a 37% slowdown compared to running one MPI task per core on the entire compute node. This is due to the fact that in the HPX implementation only the main thread in each MPI locality is performing communication so the number of communication calls is reduced and the data volume for each call is increased. This can be overcome by future implementations of the distributed parallelism provided by HPX, which is beyond the scope of this paper.

To investigate a spatially inhomogeneous case, we construct a simulation using Lennard-Jones particles that mimics the load distribution arising in adaptive resolution simulations where particles in a spherical region are treated with full atomistic resolution, while the particles in the rest of the domain are coarse-grained [32] or treated as ideal gas [33] resulting in a locally reduced computational load. As shown in Fig. 8, a spherical region in the center of a simulation box is filled with Lennard-Jones particles with density $\rho_{\text{in}} = 0.8442$ similar to the bulk simulation described earlier.

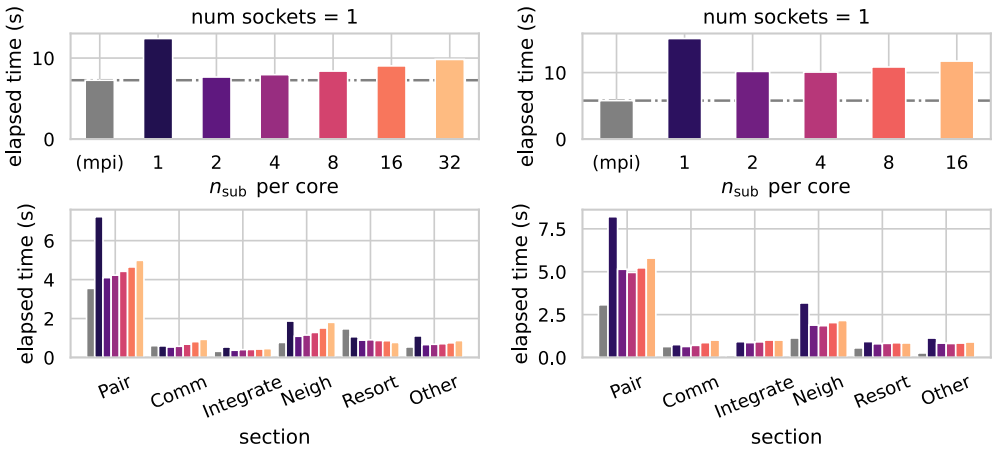
For this setup, we use a larger cubic simulation box of length $L = 271$ with a spherical region containing 2.58 million particles or 16% of the volume of the simulation box. To keep the spherical structure intact, the thermostat temperature was kept at $T = 0.1$ and the simulation was run for 100 time steps.

Results of the autotuning procedure for a full compute node are shown in Fig. 9. In contrast to the bulk simulation, the optimal number of subnodes for this setup on Mogon has shifted to $n_{\text{sub}} = 256$ or 16 subnodes per core with a $1.4\times$ speedup over the MPI version. On the other hand, the optimal n_{sub} for both AMD and Raven (Figure S3) clusters is 32 subnodes per core which indicates the need for the autotuning procedure since the same setup could behave differently on different systems.

Among the different code sections, the largest speedup can be observed in the communication which captures the MPI synchronization and takes the load imbalance into account. In the MPI version, the processors with fewer particles in their subdomain have to wait for other processors with more particles to complete their force calculations and synchronize via MPI. In the version with HPX multithreading, the work load is divided symmetrically between the two MPI ranks and the subnodes are executed concurrently by HPX threads.

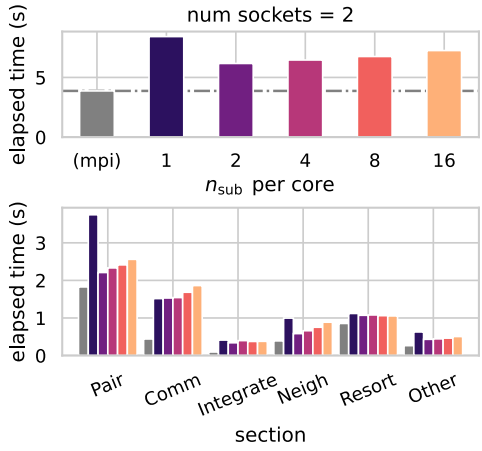
In order to put these speedups into perspective, we devise a performance model to estimate the minimum execution time that we can ideally obtain for our load imbalanced system. The theoretical minimum execution time τ_s for each section s is obtained when the workload is evenly distributed across all MPI ranks, which we can relate to the corresponding measured execution time t_s in the MPI version. We observe that the timers for the Pair, Neigh, and Other sections are not affected by load imbalances since they do not involve any inter-node communication. Thus, for these sections, $\tau_s = t_s$ since the measured execution times t_s are already averages across MPI ranks.

For the remaining sections which involve MPI communication within the timers, we estimate the ideal time by running a homogenous setup and scaling the result using the relative number of particles. We start with a bulk LJ setup formed by filling the entire simulation box described in Fig. 8 with LJ particles and measuring t_s^{bulk}, the execution time of each section. From this, we can estimate t_s by relating each section's workload to the relative number of particles in the two simulations (N_{bulk} and N_{sphere}). Since Integrate scales with the total number of particles in the simulation box, its ideal execution time is given by $\tau_{\text{Integrate}} = t^{\text{bulk}}_{\text{Integrate}} \left(N_{\text{sphere}}/N_{\text{bulk}}\right)$. On the other hand, the Re-

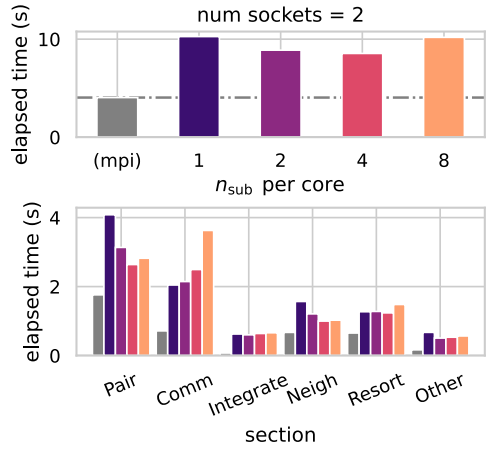


(a) Mogon (1 MPI rank, 16 HPX threads)

(b) AMD (1 MPI rank, 32 HPX threads)



(c) Mogon (2 MPI ranks, 16 HPX threads)



(d) AMD (2 MPI ranks, 32 HPX threads)

Fig. 7. Elapsed time for the homogenous Lennard-Jones fluid with the MPI version (- - -) and with HPX using different number of subnodes on Mogon. Lower time is better. On a single socket of Mogon, the performance of the HPX version at $n_{sub} = 32$ is similar to MPI. For the single node case, the performance is impacted by having only the main thread performing MPI calls. AMD also shows higher overhead for performing the PAIR force calculation.

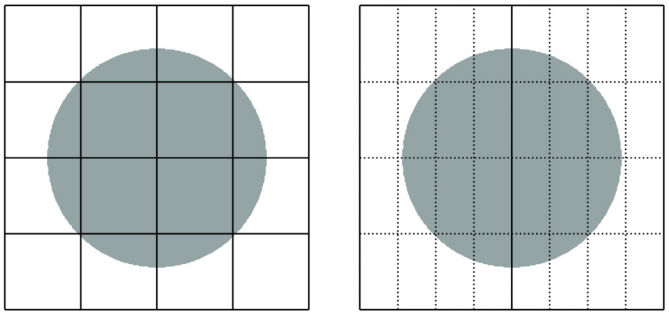


Fig. 8. Initial spherical configuration of Lennard-Jones particles showing the domain decomposition at the x-y plane with partitioning according to MPI ranks (—) and HPX subnodes (· · ·). Left: MPI-only partitioning into $(4, 4, 2)$ grid. Right: HPX-enabled partitioning into 2 MPI ranks and $(4, 4, 4)$ subnode grid.

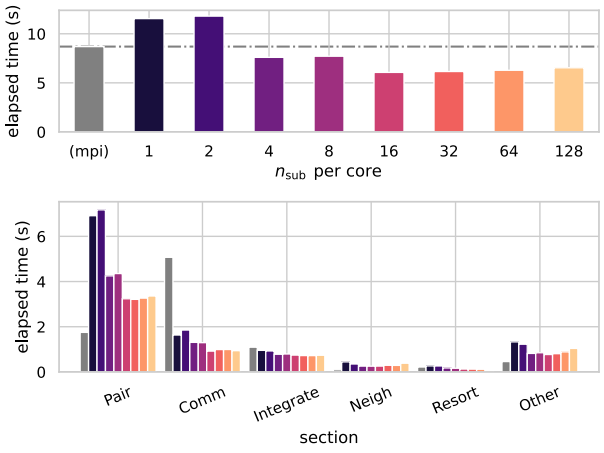
SORT and COMM sections are dominated by inter-node communication that happens only along the planes of the node boundaries which are proportional to the surface area of the simulation box. Since the length of the boundaries is $L = (N/\rho)^{1/3}$ for some constant number density ρ, the surface area of the simulation box is given by $6 \times (N/\rho)^{1/3} \times (N/\rho)^{1/3}$ which means that the communication workload is proportional to $N^{2/3}$. Using this, we can estimate the ideal execution time for these sections to be $\tau_s = t_s^{\text{bulk}} \left(N_{\text{sphere}}/N_{\text{bulk}}\right)^{2/3}$ for $s \in \{\text{RESORT}, \text{COMM}\}$. Thus, the total ideal execution time for the spherical setup is given by

$$\tau = t_{\text{PAIR}} + t_{\text{NEIGH}} + t_{\text{OTHER}} + t_{\text{INTEGRATE}}^{\text{bulk}} \left(\frac{N_{\text{sphere}}}{N_{\text{bulk}}}\right) + \left(t_{\text{COMM}}^{\text{bulk}} + t_{\text{RESORT}}^{\text{bulk}}\right) \left(\frac{N_{\text{sphere}}}{N_{\text{bulk}}}\right)^{2/3} \tag{4}$$

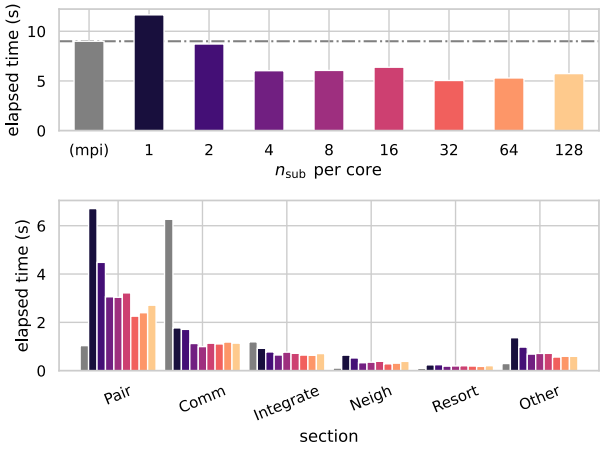
We performed the bulk simulation on the three machines and used Equation (4) to calculate the ideal execution times shown in Table 3. Also shown are the actual execution times using the HPX and MPI implementations from Fig. 9. We find that on Mogon, our HPX implementation is 1.71× slower compared to the ideal, while the base MPI implementation is 2.45× slower.

5. Related work

Maximizing SIMD capabilities of recent hardware architectures has been thoroughly explored and applied to molecular dynamics simulations. For example, optimizations have been made in LAMMPS that enable full use of the Intel Knights Landing architec-



(a) Mogon at 16 subnodes per core



(b) AMD at 32 subnodes per core

Fig. 9. Comparison of sectional elapsed times with the MPI version (-·-·-) and with HPX using different number of subnodes for the spherical Lennard-Jones. Lower time is better.

Table 3
Ideal execution time τ calculated from (4) compared with the actual execution time with HPX and MPI.

	τ (s)	t_{HPX} (s)	t_{HPX}/τ	t_{MPI} (s)	t_{MPI}/τ
Mogon	3.55	6.07	1.71	8.69	2.45
Raven	1.75	3.12	1.78	5.39	3.08
AMD	2.34	5.05	2.16	8.99	3.84

ture with AVX-512 which is still relevant to recent Intel CPUs [26]. SIMD vectorization techniques have also been investigated specifically for the Lennard-Jones potential on Intel CPUs with AVX2 and AVX-512 instructions [7].

Different forms of the hybrid MPI+X programming model have been widely applied in molecular dynamics simulations. For example, LAMMPS has support for multi-core CPUs and accelerators through OpenMP, CUDA and Kokkos [26,31]. To reduce the complexity of maintaining code for different architectures, performance portability has also been applied to particle-based simulations through Cabana, which is a library built on Kokkos, and demonstrated specifically for molecular dynamics simulations through the CabanaMD proxy application [34]. Task-based parallelism and asynchronous mesh refinement approaches have previously been implemented for highly dynamic MD simulations parallelized using MPI and OpenMP [35]. It has also been shown that fully adopting a hybrid MPI+OpenMP programming model alleviates load balancing problems and communication pressure better than pure MPI implementation that allows the adjustment of domain boundaries [36].

HPX has been successfully applied to N-body simulations of stellar mergers in astrophysics [37]. The implication of task granularity on performance of HPX-based applications has also been studied [28]. Comparisons between the traditional MPI programming model and HPX-based implementations of the discontinuous Galerkin method for the two-dimensional shallow water equations have also been reported in [27] which found a 6% faster runtime with HPX parallelization which they attribute to better cache behavior due to overdecomposition.

6. Summary and discussion

In this work, we presented our code modernization efforts for the ESPResSo++ molecular dynamics simulation package. This was achieved by optimizing the particle data layout, implementing SIMD vectorization on critical loops and integrating node-level parallelism through HPX. We have optimized operations that require only a few particle attributes by using a structure of arrays layout which improved the memory access patterns of these operations resulting in an up to 2× overall speedup. The change in layout also allowed us to ensure SIMD vectorization on critical for loops used in short-range interactions. This results in an additional 1.5× speedup for Lennard-Jones simulations and 1.1× speedup for polymer melt simulations, suggesting that the greater benefits from vectorization of short-range interactions depend on having a larger cut-off distance among other factors.

These serial optimizations served as a baseline on which we implemented node-level parallelism through HPX. By decomposing the MPI node into smaller subnodes, we were able to modulate the granularity of the task sizes and obtain an optimum balance between resource starvation and overhead from additional force calculation between subnodes. We then compared the HPX-based implementation with the traditional MPI-based parallelization. For a spatially homogenous system, we found the HPX implementation to be 5% slower on a single socket running a single MPI rank, which rises to 37% slower on a full node running two MPI ranks resulting from the increase in communication volume per MPI call. Nevertheless, the benefit of using the work-stealing capabilities of HPX was demonstrated when running spatially inhomogeneous simulations such as an LJ system with spherical load distribution for which we obtained a 1.4× speedup.

To maintain compatibility with the original packages within ESPResSo++, we have chosen to implement our code modernization strategies as additional submodules in ESPResSo++ (`vec` and `hpx4espp`). This is due to the intrusive change in the particle data structure which all algorithms and analysis tools are dependent on. This design choice allows us to continuously develop other performance improvements which could eventually be adopted in the main branch of ESPResSo++.

Through this implementation and evaluation process, we have found that HPX can provide suitable features for implementing thread-level parallelism for MD applications involving short-range interactions while keeping MPI as the driver for inter-node communication. However, as with the introduction of any shared-memory programming model, certain care has to be taken in order to ensure thread-safety, which is not a concern in MPI-based implementations. We were able to solve this in our HPX implementation by introducing a node-level domain decomposition layer at the expense of some additional overhead. By autotuning for a few time steps, we can find an optimal task size that maximizes parallelism and minimizes this overhead.

7. Future work

Further performance improvements can be achieved by utilizing HPX also for the distributed parallelism aspect of our application. The current implementation, which relies on the multithreaded parallel algorithms of HPX, is still limited by implicit synchronization among the different MPI localities since they still utilize MPI communication to update their ghost cells with only the main thread handling MPI calls. This constraint can be further decoupled by allowing every subnode to independently perform ghost cell updates using asynchronous remote function calls. This can be done using the distributed parallelism framework of HPX through its Active Global Address Space (AGAS). Using global addressing, we can decouple the ghost communication among neighboring subnodes so that every direction can be updated independently through an API that is the same whether the neighboring subnode is in the same locality or not. This can then utilize the futurization capabilities of HPX to form a task-based dependency tree that will expose more parallelism and reduce the synchronization among MPI localities.

Although this work only addresses node-level load balancing through work-stealing, HPX can also be used to implement load balancing among different localities by converting the subnodes into migratable objects that can be moved between localities while maintaining their neighborhood topologies through their global addressability.

Funding

This work was funded by the German Research Foundation (DFG) through the collaborative research center TRR 146: Multiskalen-Simulationsmethoden für Systeme der weichen Materie (Project number 233630050) (Project G). ESPResSo++ is one of the central software packages of the TRR146 (https://trr146.de) and also part of the software pool of the European E-CAM project (https://www.e-cam2020.eu). Parts of this research were conducted using the supercomputer MOGON II and advisory services offered by Johannes Gutenberg University Mainz (https://hpc.uni-mainz.de), which is a member of the AHRP (Alliance for High Performance Computing in Rhineland Palatinate, https://www.ahrp.info) and the Gauss Alliance. Some of the development was also carried out with the HPC system Raven at the Max Planck Computing and Data Facility (https://www.mpcdf.mpg.de). LANL is operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA000001). This document is LA-UR-22-21943.

CRediT authorship contribution statement

James Vance: Conceptualization, Methodology, Software, Writing – original draft. **Zhen-Hao Xu:** Conceptualization, Writing – review & editing. **Nikita Tretyakov:** Conceptualization, Software, Writing – review & editing. **Torsten Stuehn:** Conceptualization, Funding acquisition, Software, Supervision, Writing – review & editing. **Markus Rampp:** Conceptualization, Resources, Writing – review & editing. **Sebastian Eibl:** Software, Writing – review & editing. **Christoph Junghans:** Formal analysis, Software, Writing – review & editing. **André Brinkmann:** Conceptualization, Funding acquisition, Supervision, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Appendix A. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cpc.2023.108760.

References

[1] S. Plimpton, J. Comput. Phys. 117 (1) (1995) 1–19, https://doi.org/10.1006/jcph.1995.1039.
[2] A.P. Thompson, H.M. Aktulga, R. Berger, D.S. Bolintineanu, W.M. Brown, P.S. Crozier, P.J. in't Veld, A. Kohlmeyer, S.G. Moore, T.D. Nguyen, et al., Comput. Phys. Commun. 271 (2022) 108171, https://doi.org/10.1016/j.cpc.2021.108171.
[3] M.J. Abraham, T. Murtola, R. Schulz, S. Páll, J.C. Smith, B. Hess, E. Lindahl, SoftwareX 1 (2015) 19–25, https://doi.org/10.1016/j.softx.2015.06.001.
[4] J.C. Phillips, D.J. Hardy, J.D.C. Maia, J.E. Stone, J.V. Ribeiro, R.C. Bernardi, R. Buch, G. Fiorin, J. Hénin, W. Jiang, R. McGreevy, M.C.R. Melo, B.K. Radak, R.D. Skeel, A. Singharoy, Y. Wang, B. Roux, A. Aksimentiev, Z. Luthey-Schulten, L.V. Kalé, K. Schulten, C. Chipot, E. Tajkhorshid, J. Chem. Phys. 153 (4) (2020) 044130, https://doi.org/10.1063/5.0014475.
[5] J.D. Halverson, T. Brandes, O. Lenz, A. Arnold, S. Bevc, V. Starchenko, K. Kremer, T. Stuehn, D. Reith, Comput. Phys. Commun. 184 (4) (2013) 1129–1149, https://doi.org/10.1016/j.cpc.2012.12.004.
[6] H.V. Guzman, N. Tretyakov, H. Kobayashi, A.C. Fogarty, K. Kreis, J. Krajniak, C. Junghans, K. Kremer, T. Stuehn, Comput. Phys. Commun. 238 (2019) 66–76, https://doi.org/10.1016/j.cpc.2018.12.017, arXiv:1806.10841.
[7] H. Watanabe, K.M. Nakagawa, Comput. Phys. Commun. 237 (2019) 1–7, https://doi.org/10.1016/j.cpc.2018.10.028, arXiv:1806.05713.
[8] M. Kretz, V. Lindenstruth, Softw. Pract. Exp. 42 (11) (2012) 1409–1430, https://doi.org/10.1002/spe.1149.
[9] T. Ishiyama, K. Nitadori, J. Makino, in: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 1–10.
[10] J.-L. Fattebert, D. Richards, J. Glosli, Comput. Phys. Commun. 183 (12) (2012) 2608–2615, https://doi.org/10.1016/j.cpc.2012.07.013.
[11] S. Eibl, U. Rüde, Comput. Phys. Commun. 244 (2019) 76–85, https://doi.org/10.1016/j.cpc.2019.06.020.
[12] I. Raicu, I.T. Foster, Y. Zhao, in: 2008 Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS@SC), Austin, TX, USA, November 17, 2008, pp. 1–11.
[13] L. Kale, et al., Uiuc-ppl/charm: v7.0.0-rc1, https://doi.org/10.5281/zenodo.4988098, Jun. 2021.
[14] J. Bakosi, R. Bird, F. Gonzalez, C. Junghans, W. Li, H. Luo, A. Pandare, J. Waltz, Adv. Eng. Softw. 160 (2021) 102962, https://doi.org/10.1016/j.advengsoft.2020.102962.
[15] H. Kaiser, P. Diehl, A. Lemoine, B. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. Brandt, N. Gupta, T. Heller, K. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, T. Zhang, J. Open Sour. Softw. 5 (53) (2020) 2352, https://doi.org/10.21105/joss.02352.
[16] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, Oxford University Press, 2017.
[17] L. Verlet, Phys. Rev. 159 (1) (1967) 98–103, https://doi.org/10.1103/physrev.159.98.
[18] W.C. Swope, H.C. Andersen, P.H. Berens, K.R. Wilson, J. Chem. Phys. 76 (1) (1982) 637–649, https://doi.org/10.1063/1.442716.
[19] V. Rühle, C. Junghans, A. Lukyanov, K. Kremer, D. Andrienko, J. Chem. Theory Comput. 5 (12) (2009) 3211–3223, https://doi.org/10.1021/ct900369w.
[20] J. Wehner, L. Brombacher, J. Brown, C. Junghans, O. Çaylak, Y. Khalak, P. Madhikar, G. Tirimbò, B. Baumeier, J. Chem. Theory Comput. 14 (12) (2018) 6253–6268, https://doi.org/10.1021/acs.jctc.8b00617.
[21] Z.-H. Xu, J. Vance, N. Tretyakov, T. Stuehn, A. Brinkmann, Implementation and code parallelization of the Lees-Edwards boundary condition in ESPResSo++, arXiv preprint, https://doi.org/10.48550/arXiv.2109.11083, 2021.
[22] M. Wolfe, Compilers and more: MPI+ X, HPC Wire, https://www.hpcwire.com/2014/07/16/compilers-mpix, 2014.
[23] C.R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D.S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, J. Wilke, IEEE Trans. Parallel Distrib. Syst. 33 (4) (2022) 805–817, https://doi.org/10.1109/TPDS.2021.3097283.
[24] D.A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A.J. Kunen, O. Pearce, P. Robinson, B.S. Ryujin, T.R. Scogland, in: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), IEEE, 2019, pp. 71–81.

[25] S.K. Gutiérrez, K. Davis, D.C. Arnold, R.S. Baker, R.W. Robey, P. McCormick, D. Holladay, J.A. Dahl, R.J. Zerr, F. Weik, et al., in: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp. 469–478.
[26] J. Jeffers, J. Reinders, A. Sodani, Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition, Morgan Kaufmann, 2016, https://dl.acm.org/doi/10.5555/3050856.
[27] M. Bremer, K. Kazhyken, H. Kaiser, C. Michoski, C. Dawson, J. Sci. Comput. 80 (2) (2019) 878–902, https://doi.org/10.1007/s10915-019-00960-z.
[28] P. Grubel, H. Kaiser, J. Cook, A. Serio, in: 2015 IEEE International Conference on Cluster Computing, 2015, pp. 682–689, http://stellar.cct.lsu.edu/pubs/hpcmaspa2015.pdf.
[29] H. Kaiser, et al., STEllAR-GROUP/hpx: HPX V1.5.1: The C++ Standards Library for Parallelism and Concurrency, https://doi.org/10.5281/zenodo.4059746, Sep. 2020.
[30] K. Kremer, G.S. Grest, J. Chem. Phys. 92 (8) (1990) 5057–5086, https://doi.org/10.1063/1.458541.
[31] S. Plimpton, A. Kohlmeyer, A. Thompson, S. Moore, R. Berger, LAMMPS Stable release 29 October 2020, https://doi.org/10.5281/zenodo.4157471, Oct. 2020.
[32] A.C. Fogarty, R. Potestio, K. Kremer, J. Chem. Phys. 142 (19) (2015) 05B610, https://doi.org/10.1063/1.4921347.
[33] L.A. Baptista, R.C. Dutta, M. Sevilla, M. Heidari, R. Potestio, K. Kremer, R. Cortes-Huerto, J. Phys. Condens. Matter 33 (18) (2021) 184003, https://doi.org/10.1088/1361-648X/abed1d.
[34] S.M. Mniszewski, J. Belak, J.-L. Fattebert, C.F. Negre, S.R. Slattery, A.A. Adedoyin, R.F. Bird, C. Chang, G. Chen, S. Ethier, et al., Int. J. High Perform. Comput. Appl. 35 (6) (2021) 572–597, https://doi.org/10.1177/10943420211022829.
[35] R. Prat, T. Carrard, L. Soulard, O. Durand, R. Namyst, L. Colombet, Comput. Phys. Commun. 253 (2020) 107177, https://doi.org/10.1016/j.cpc.2020.107177.
[36] J. Morillo, M. Vassaux, P.V. Coveney, M. Garcia-Gasulla, J. Supercomput. (2022) 1–32, https://doi.org/10.1007/s11227-021-04214-4.
[37] D.C. Marcello, S. Shiber, O. De Marco, J. Frank, G.C. Clayton, P.M. Motl, P. Diehl, H. Kaiser, Mon. Not. R. Astron. Soc. 504 (4) (2021) 5345–5382, https://doi.org/10.1093/mnras/stab937.