Computer Physics Communications 271 (2022) 108171

Contents lists available at ScienceDirect

Computer Physics Communications

www.elsevier.com/locate/cpc

Feature article

LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales ☆,☆☆

Aidan P. Thompson [a,*], H. Metin Aktulga [b], Richard Berger [c], Dan S. Bolintineanu [a], W. Michael Brown [d], Paul S. Crozier [a], Pieter J. in 't Veld [e], Axel Kohlmeyer [c], Stan G. Moore [a], Trung Dac Nguyen [f], Ray Shan [g], Mark J. Stevens [a], Julien Tranchida [a], Christian Trott [a], Steven J. Plimpton [a,*]

[a] *Sandia National Laboratories, Albuquerque, NM 87185, USA*
[b] *Michigan State University, East Lansing, MI 48824, USA*
[c] *Temple University, Philadelphia, PA 19122, USA*
[d] *Intel Corporation, Hillsboro, OR 97124, USA*
[e] *BASF SE, Ludwigshafen am Rhein, Germany*
[f] *Northwestern University, Evanston, IL 60208, USA*
[g] *Materials Design Inc., San Diego, CA 92131, USA*

ARTICLE INFO

Article history:
Received 4 June 2021
Received in revised form 2 September 2021
Accepted 14 September 2021
Available online 22 September 2021

Keywords:
Molecular dynamics
Materials modeling
Parallel algorithms
LAMMPS

ABSTRACT

Since the classical molecular dynamics simulator LAMMPS was released as an open source code in 2004, it has become a widely-used tool for particle-based modeling of materials at length scales ranging from atomic to mesoscale to continuum. Reasons for its popularity are that it provides a wide variety of particle interaction models for different materials, that it runs on any platform from a single CPU core to the largest supercomputers with accelerators, and that it gives users control over simulation details, either via the input script or by adding code for new interatomic potentials, constraints, diagnostics, or other features needed for their models. As a result, hundreds of people have contributed new capabilities to LAMMPS and it has grown from fifty thousand lines of code in 2004 to a million lines today. In this paper several of the fundamental algorithms used in LAMMPS are described along with the design strategies which have made it flexible for both users and developers. We also highlight some capabilities recently added to the code which were enabled by this flexibility, including dynamic load balancing, on-the-fly visualization, magnetic spin dynamics models, and quantum-accuracy machine learning interatomic potentials.

Program Summary
Program Title: Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS)
CPC Library link to program files: https://doi.org/10.17632/cxbxs9btsv.1
Developer's repository link: https://github.com/lammps/lammps
Licensing provisions: GPLv2
Programming language: C++, Python, C, Fortran
Supplementary material: https://www.lammps.org
Nature of problem: Many science applications in physics, chemistry, materials science, and related fields require parallel, scalable, and efficient generation of long, stable classical particle dynamics trajectories. Within this common problem definition, there lies a great diversity of use cases, distinguished by different particle interaction models, external constraints, as well as timescales and lengthscales ranging from atomic to mesoscale to macroscopic.
Solution method: The LAMMPS code uses parallel spatial decomposition, distributed neighbor lists, and parallel FFTs for long-range Coulombic interactions [1]. The time integration algorithm is based on the Størmer-Verlet symplectic integrator [2], which provides better stability than higher-order non-symplectic

☆ The review of this paper was arranged by Prof. N.S. Scott.
☆☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).
* Corresponding authors.
E-mail addresses: athomps@sandia.gov (A.P. Thompson), sjplimp@sandia.gov (S.J. Plimpton).

https://doi.org/10.1016/j.cpc.2021.108171

methods. In addition, LAMMPS supports a wide range of interatomic potentials, constraints, diagnostics, software interfaces, and pre- and post-processing features.
Additional comments including restrictions and unusual features: This paper serves as the definitive reference for the LAMMPS code.

References

[1] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics. J. Comp. Phys. 117 (1995) 1–19.

[2] L. Verlet, Computer experiments on classical fluids: I. Thermodynamical properties of Lennard–Jones molecules, Phys. Rev. 159 (1967) 98–103.

1. Introduction

Classical molecular dynamics simulation (MD) is a powerful approach for modeling the thermodynamic, mechanical, and chemical behaviors of solids and fluids in a rigorous manner [1] at a level of accuracy determined by an interatomic potential (force field) which defines the potential energy of the system as a function of atom positions and optionally other properties. Early pioneering applications included radiation effects in solids [2], as well as dynamics and statistical mechanics of simple fluids [3,4]. Since then, the use of MD has become pervasive in myriad areas of physics, chemistry, biology, materials science, and related disciplines. In MD, the system is modeled as a large collection of particles (often individual atoms) whose time evolution is computed by numerically integrating Newton's equations of motion over a large number of timesteps to track their positions and velocities. Forces on atoms are calculated as the derivative of the analytic equations defining the potential function. Typical potentials are computationally inexpensive and accurately capture the basic physics of electron-mediated atomic interactions for important classes of materials, such as molecular liquids and solid-state metals. Efficient MD codes running on commodity workstations can be used to simulate systems with $N = 10^4 - 10^6$ atoms for times from ps to µs, sampling length and time scales at which many important physical and chemical phenomena emerge.

One reason for the increasing popularity of MD simulations is that they are computationally well-suited to take advantage of the dramatic hardware speed-ups afforded by Moore's Law and large-scale parallelism over the last few decades, both for conventional floating-point processors (CPUs) and more recently graphical processing units (GPUs). Consider that in 1988 an 8-processor Cray YMP ran the Linpack dense matrix inversion benchmark at a speed of two gigaflops (10^9 floating point operations/s). In 2012 a single IBM Blue Gene/Q CPU (16 cores) ran Linpack at 175 gigaflops; the largest BG/Q machine (Sequoia at Lawrence Livermore National Laboratory) had nearly 100,000 CPUs. In the next year or two, there will likely be several GPU-based supercomputers that run the Linpack benchmark at exaflop (10^{18}) speeds. This represents a half billion-fold speed-up in a little more than 30 years for the largest supercomputers; there have been concomitant increases in the speed of desktop machines and smaller clusters available to the broader scientific computing community.

For MD, the per-timestep computational cost for models with short-range interactions scales linearly as $\mathcal{O}(N)$ in the number of atoms N, due to the bounded number of neighbor atoms within a cutoff distance. In practice, long-range Coulombic interactions for MD scale nearly as well: $\mathcal{O}(N \log N)$ for FFT-based methods such as particle-mesh Ewald or $\mathcal{O}(N)$ for multipole methods, albeit with a larger pre-factor for the latter. And the fundamental MD operations for computing forces, building neighbor lists, and time integration are all inherently parallelizable over atoms [5].

In practice this means speed increases in computer hardware have enabled similar increases in accessible MD length or time scales or a product of the two. For systems which do not have enough atoms to fully exploit the parallelism of increasingly large machines, a variety of novel algorithms have been developed that run multiple replicas of small systems concurrently to either extend timescales, increase sampling efficiencies, or explore free energy landscapes. These include accelerated MD methods [6], Markov state models [7], and enhanced sampling methods [8] including metadynamics [9].

For materials science applications, a second reason for MD's popularity is that the availability of faster hardware has spurred development of more complex and computationally intensive potentials, which in turn have enabled more accurate predictive modeling of material properties. Notably, this includes development of manybody potentials, such as bond-order or reactive potentials, with computational costs up to a few orders of magnitude larger than simpler, pairwise additive potentials [10]. In recent years, this also includes so-called machine learning (ML) potentials, which replace physics-based equations fit to a relatively small set of experimental data or quantum calculations, with generic equations that describe only the geometry of a neighborhood of atoms and are trained on increasingly large volumes of data from quantum calculations; ML models in LAMMPS are discussed later in this paper.

These factors have motivated the development of a variety of parallel MD codes, many of which have attracted large user communities. These include codes focused on biomolecular modeling, such as Amber [11], CHARMM [12], GROMACS [13], and NAMD [14], and also codes with a focus on materials modeling, such as DL_POLY [15], HOOMD [16], and LAMMPS [5]. Nearly all these codes, including LAMMPS, are open-source and support parallel computations on both CPUs and GPUs.

One difference between biomolecular versus materials MD codes is in the kinds of models they support, though there is often overlap. While biological systems exhibit arguably a richer variety of complex behavior and phenomena than solid-state or even soft non-biological materials, the interatomic potentials (force fields) used to model materials are more numerous and diverse than for biomolecular systems, due to the wider range of chemical and physical interactions, thermodynamic conditions, and states of matter. This includes a variety of coarse-grained models that are not used at atomic scales, but at meso to continuum scales.

A second difference is in the size of systems and time scales of interest, though again there is considerable overlap. For biological MD, there are countless interesting systems with one or a few solvated biomolecules (tens to hundreds of thousands of atoms), and many of the most interesting phenomena, such as large conformational changes (e.g. protein folding), happen at timescales that are immense for MD modeling (µs to ms to even s). By contrast, for materials MD, many millions or even billions of atoms are sometimes needed to capture interesting phenomena. Such increases in the atom count decrease the accessible timescales proportionally and may also increase the physically-relevant timescale, yet there are interesting materials phenomena that can be observed experimentally at relatively short ps to ns timescales (e.g. shock response of solids).

In this paper, we describe several aspects of LAMMPS that have helped to make it a powerful tool for materials modeling. After giving a brief history of LAMMPS development in Section 2, Section 3 describes its parallel algorithms which enable scalable performance for both CPUs and GPUs. Section 4 explains how the source code is structured to enable users to easily customize their simulations by modifying or adding code. This includes methods to enable coupling LAMMPS with other simulation and analysis tools, either in a multiphysics, multiscale, or workflow sense, all of which are increasingly common in materials modeling.

Section 5 outlines the strategy for optimizing the large number of models and features within LAMMPS for different hardware and highlights some performance results. This includes discussions of load-balancing and scalability of various LAMMPS operations to very large machines. Finally, Section 6 gives brief overviews of the range of material models and other options available in LAMMPS, including some that are recent additions and some that are novel in an MD context. It includes discussions of manybody and coarse-grained potentials, charge-equilibration methods, spin dynamics models, and machine learning potentials.

2. A brief LAMMPS history

The development of LAMMPS began in the mid-1990s as a partnership between two US Department of Energy laboratories (Sandia National Laboratories and Lawrence Livermore National Laboratory) and three companies (Cray, DuPont, and Bristol-Myers Squibb). The goal was to create a parallel MD code which could leverage the spatial parallelism algorithms described in [5], to effectively use what were then large supercomputers with 100s to a few 1000 processors (cores today) for either materials or biomolecular modeling. The initial version was written in Fortran and was available at no cost, but required users to read and sign a perfunctory license agreement. In ten years, only ∼100 users did so, which we attribute to the repulsive and attractive interactions of paperwork with users and lawyers respectively.

We then re-wrote LAMMPS in C++, primarily because it was difficult in the Fortran-based code to flexibly add new models and features, such as interatomic potentials, particle properties, constraints, diagnostic computations, etc. For performance, we retained the low-level Fortran-like data structures (vectors, arrays, simple structs); critical kernels were written in C-style code. The object-oriented aspects of C++ were (and still are) used at a high-level to help structure and organize the code; in that sense the code is written mostly in “object-oriented C”. We have found that makes it easy for users from a variety of coding backgrounds to understand and modify LAMMPS. In 2004 we released the new version as open source code under the GNU General Public License (GPL); in a month it was downloaded more times than in the first ten years.

In the ensuing 17 years, the lines of LAMMPS source code have grown from 50 thousand to a million, which includes contributions from several hundred users. The increasing volume of contributions motivated migration of the code repository and development workflow to GitHub in 2016, which has substantially enhanced our ability to integrate and test code contributions. It has also made the code more robust by taking advantage of GitHub facilities for reporting bugs as well as automating configuration, regression, and unit testing.

Currently, each year there are tens of thousands of tarball downloads of the code, similar numbers of GitHub accesses, and several thousand mailing list and forum postings; the latter two are the chief mechanisms for answering user questions. In recent years, we have also allowed companies or other interested parties to license LAMMPS under the GNU Lesser Public License (LGPL). This allows development of proprietary code that invokes LAMMPS as a library, such as GUI, analyses, or workflow tools, without the requirement to distribute the proprietary code openly. As an example, DCS Computing builds on top of LAMMPS for particle simulations within its multiphysics simulation software [17].

From its beginning LAMMPS was designed for distributed-memory parallelism using MPI. In a weak scaling sense, with at least a few hundreds of atoms/core, most of its models are scalable to millions of CPU cores. In the last ten years, we have added GPU support to LAMMPS, initially with CUDA and OpenCL code, and more recently via the Kokkos library [18,19] which has back-ends for GPUs from different vendors as well as OpenMP. We describe the various options for GPUs and multithreading generally in Section 5.1.

Over time we have continuously re-factored portions of LAMMPS to make it as easy as possible for users to understand, modify, and extend. This has had the benefit of increasing the scope of what LAMMPS can model and has also enabled creative people to think outside the traditional MD box. As we highlight in Section 6, some of the contributed capabilities include features we never imagined would make sense or even be possible in a classical MD code.

We now view LAMMPS (and MD) in a broader context as a computational engine for modeling interacting particles at any length scale, so long as the interactions are primarily short-range, and particle densities are moderately bounded (unlike in N-body gravitational or plasma PIC codes). These characteristics make it advantageous to evaluate forces using neighbor lists, which is central to the design of LAMMPS.

3. Parallel algorithms

In this section, we describe how various MD algorithms are formulated in LAMMPS to enable parallelism across CPUs (via MPI). Versions of some of these algorithms for GPUs are discussed in Section 5.1 or the papers cited there. In this and subsequent sections, we often refer to particles as atoms, though as explained in Section 1, particles can also be finite-size coarse-grained or even continuum-scale objects. In order to help distinguish terms specific to LAMMPS from their more general meanings, we use the *italic* font style. Common examples of this are the LAMMPS *atom*, *pair*, *fix*, and *compute* styles.

3.1. Partitioning

The underlying spatial decomposition strategy used by LAMMPS for distributed-memory parallelism via MPI was described in the original LAMMPS paper [5] when the code was being initially developed. Most of the ideas from that paper still persist in the current code, but have been refined in various ways, as we explain here.

The LAMMPS simulation box is a 3d or 2d volume, which can be orthogonal or triclinic in shape, as illustrated in Fig. 1 for the 2d case. Orthogonal means the box edges are aligned with the x, y, z Cartesian axes, and the box faces are thus all rectangular. Triclinic allows for a more general parallelepiped shape in which edges are aligned with three arbitrary vectors and the box faces are parallelograms. In each dimension box faces can be periodic, or non-periodic with fixed or shrink-wrapped boundaries. In the fixed case, atoms which move outside the face are deleted; shrink-wrapped means the position of the box face adjusts continuously to enclose all the atoms.

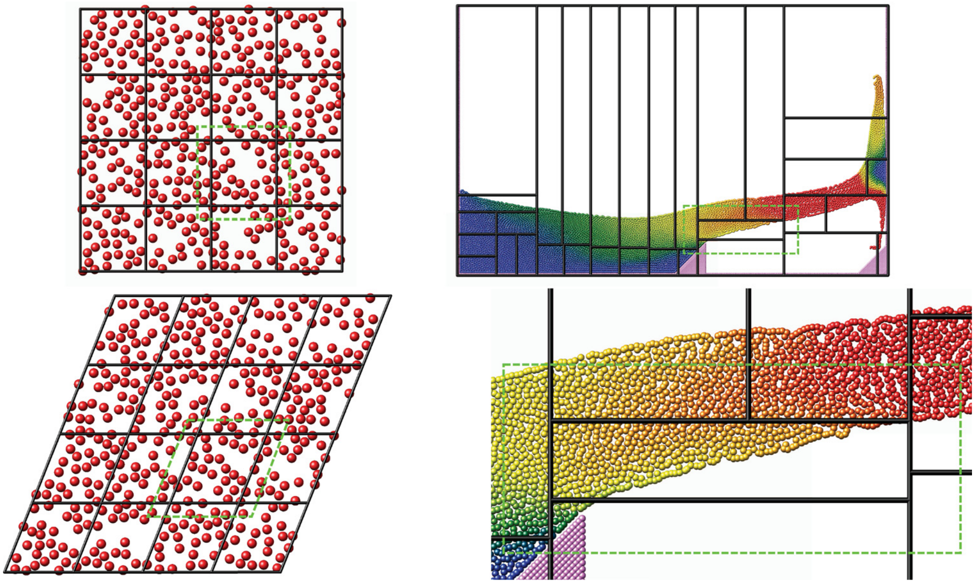

Fig. 1. Three kinds of spatial decomposition (partitioning) of the simulation box for MPI parallelism. The black solid lines are subdomain boundaries; the green dashed lines extend one processor's subdomain to include its ghost atoms (see Section 3.2). (Upper left) An orthogonal box with a roughly uniform density of particles, partitioned into a regular grid of 16 equal-sized processor subdomains. (Lower left) A triclinic box partitioned similarly, but with subdomain edges (faces in 3d) parallel to the box edges (faces). (Upper right) An orthogonal box partitioned into variable-sized subdomains such that the same number of particles are owned by each of 32 processors. This is a smoothed particle hydrodynamics (SPH) model of water flowing over a dam (purple triangle at the bottom center), resulting in a non-uniform and time-varying particle density. (Lower right) A close-up view of one SPH processor subdomain.

For distributed-memory MPI parallelism, the simulation box is spatially decomposed (partitioned) into non-overlapping subdomains which fill the box. These are delineated with solid lines in the figure. A unique MPI rank or process is assigned to each subdomain.[1]

The default partitioning, suitable when atom density is roughly uniform, is shown in the left-side images of Fig. 1. The subdomains comprise a regular grid and all subdomains are identical in size and shape. Both the orthogonal and triclinic boxes can deform continuously during a simulation, e.g. to compress a solid or shear a liquid, in which case the processor subdomains likewise deform.

For models with non-uniform density, the number of particles per processor can be load-imbalanced with the default partitioning. This reduces parallel efficiency, as the overall simulation rate is limited by the slowest processor, i.e. the one with the largest computational load. For such models, LAMMPS supports alternate partitionings where the size and shape of each subdomain can vary, so long as the faces of each subdomain are parallel to the simulation box faces. All the parallel algorithms described in Section 3 operate within this context.

The right-side images in Fig. 1 illustrate a partition produced by a dynamic load-balancing algorithm (recursive coordinate bisectioning or RCB) discussed in Section 5.3. This snapshot is from a smoothed particle hydrodynamics (SPH) simulation of water flowing over a dam. Initially, the water was in a tall column on the left side of the box which then collapsed due to gravity. The flowing particles are colored by their kinetic energy. Purple particles along the left/right/bottom edges of the domain and in the two triangles are stationary. The RCB algorithm can produce partitionings with equal numbers of particles in each subdomain.

3.2. Communication

Each processor stores information (positions, velocities, etc.) for the subset of atoms within its subdomain, called *owned* atoms. It also stores copies of some of that information for *ghost* atoms within a cutoff distance of its subdomain, which are owned by nearby processors. This enables each processor to calculate short-range interactions which involve atoms it owns. The dashed-line boxes in Fig. 1 illustrate the extended ghost-atom subdomain for one processor. For triclinic domains, the cutoff distance between the solid and dashed lines is in the direction perpendicular to the subdomain faces (right side of Fig. 3). Atoms that lie within the cutoff distance across a periodic boundary are also stored as ghost atoms. Thus in a self-similar sense, each processor stores its owned atoms plus nearby ghost atoms, whether a small simulation is run on a single processor or a huge simulation is run on millions of processors.

The diagrams of Fig. 2 illustrate how ghost-atom communication is performed in two stages for a 2d simulation (three in 3d) for both a regular and irregular partitioning of the simulation box. For the regular case (left), corresponding to either of the left-side images in Fig. 1, atoms are exchanged first in the x-direction, then in y, with four neighbors in the grid of processor subdomains.

In the x stage, processor ranks 1 and 2 send owned atoms in their red-shaded regions to rank 0 (and vice versa). Then in the y stage, ranks 3 and 4 send atoms in their blue-shaded regions to rank 0, which includes ghost atoms they received in the x stage. Rank 0 thus acquires all its ghost atoms; atoms in the solid blue corner regions are communicated twice before rank 0 receives them.

For the irregular case (right), corresponding to the lower-right image in Fig. 1, the two stages are similar, but a processor can have more than one neighbor in each direction. In the x stage, ranks 1,2,3 send owned atoms in their red-shaded regions to rank 0 (and vice versa). These include only atoms between the lower and upper y-boundary of rank 0's subdomain. In the y stage, ranks 4,5,6 send atoms in their blue-shaded regions to rank 0. This may include ghost atoms they received in the x stage, but only if they

[1] In this paper, we often refer to each MPI rank or process as a "processor" meaning a CPU core, but more generally there can be multiple MPI ranks per core via simultaneous multithreading (SMT), one MPI rank for several CPU cores when also using OpenMP threads, or one or a few MPI ranks per GPU when a multicore CPU hosts one or more GPUs (discussed further in Section 5.1).

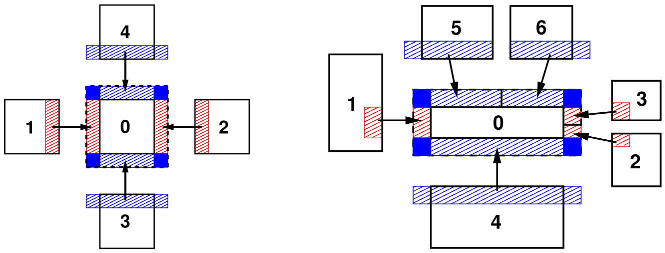

Fig. 2. Processor subdomains (black boxes) within a 2d simulation for a regular (left) and irregular (right) partitioning of the domain for a central processor 0 and its neighbors. Here the subdomains are drawn spatially separated for clarity. The dashed-line box is the extended subdomain of processor 0 which includes its ghost atoms. The red- and blue-shaded boxes are the regions of communicated ghost atoms, as described in the text.

are needed by rank 0 to fill its extended ghost atom regions in the $\pm y$ directions (blue rectangles). Thus in this case, ranks 5 and 6 do not include ghost atoms they received from each other (in the x stage) in the atoms they send to rank 0. The key point is that while the pattern of communication is more complex in the irregular partitioning case, it can still proceed in two stages (three in 3d) via atom exchanges with only neighboring processors.

When attributes of owned atoms are sent to neighboring processors to become attributes of their ghost atoms, LAMMPS calls this a "forward" communication. On timesteps when atoms migrate to new owning processors and neighbor lists are rebuilt (discussed in Section 3.3), each processor creates a list of its owned atoms which are ghost atoms in each of its neighbor processors. These lists are used to pack *per-atom* coordinates (for example) into message buffers in subsequent steps until the next reneighboring.

A "reverse" communication is when ghost atom attributes are sent back to the processor who owns the atom. This is used (for example) to sum partial forces on ghost atoms to the complete force on owned atoms. The order of the two stages described in Fig. 2 is inverted and the same lists of atoms are used to pack and unpack message buffers with per-atom forces. When a received buffer is unpacked, the ghost forces are summed to owned atom forces. As in forward communication, forces on atoms in the four blue corners of the diagrams are sent, received, and summed twice (once at each stage) before owning processors have the full force.

These two operations are used many places within LAMMPS aside from exchange of coordinates and forces, e.g. by manybody potentials to share intermediate per-atom values, or by rigid-body integrators to enable each atom in a body to access body properties. Here are additional details about how these communication operations are performed in LAMMPS:

- When exchanging data with different processors, forward and reverse communication is done using MPI_Send() and MPI_IRecv() calls. If a processor is "exchanging" atoms with itself, only the pack and unpack operations are performed, e.g. to create ghost atoms across periodic boundaries when running on a single processor.
- For forward communication of owned atom coordinates, periodic box lengths are added and subtracted when the receiving processor is across a periodic boundary from the sender. There is then no need to apply a minimum image convention when calculating distances between I, J atom pairs when building neighbor lists or computing forces.
- The cutoff distance for exchanging ghost atoms is typically equal to the neighbor cutoff R_n discussed in the next Section 3.3. But it can also be longer if needed, e.g. half the diameter of a rigid body composed of multiple atoms as discussed in Sections 4.1 and 6.2. It can also exceed the periodic box size. For the regular communication pattern in Fig. 2, if the cutoff distance extends beyond a neighbor processor's subdomain, then multiple exchanges are performed in the same direction. Each exchange is with the same neighbor processor, but buffers are packed/unpacked using a different list of atoms. For forward communication, in the first exchange a processor sends only owned atoms. In subsequent exchanges, it sends ghost atoms received in previous exchanges. For the irregular pattern in Fig. 2, overlaps of a processor's extended ghost-atom subdomain with all other processors in each dimension are detected.

3.3. Neighbor lists

To compute forces efficiently, each processor creates a Verlet-style neighbor list [4] which enumerates all pairs of I, J atoms ($I =$ owned, $J =$ owned or ghost) with separation less than a cutoff distance. In LAMMPS these are stored in a multiple-page data structure; each page is a contiguous chunk of memory which stores vectors of neighbor atoms J for many I atoms. This allows pages to be incrementally allocated or deallocated as needed; neighbor lists typically consume the most memory of any data structure in LAMMPS. The neighbor list is rebuilt (from scratch) once every few timesteps, then used repeatedly each step for force or other computations. The neighbor cutoff distance is $R_n = R_f + \Delta_s$, where R_f is the force cutoff defined by the interatomic potential for computing short-range pairwise or manybody forces and Δ_s is a skin distance that allows the list to be used for multiple steps. Typically the code triggers reneighboring when any atom has moved half the skin distance since the last reneighboring; other options can also be selected.

On steps when reneighboring is performed, atoms which have moved outside their owning processor's subdomain are first migrated to new processors via communication. Periodic boundary conditions are also (only) enforced on these steps to ensure each atom is re-assigned to the correct processor. After migration, the atoms owned by each processor are stored in a contiguous vector. Periodically each processor spatially sorts owned atoms within its vector to reorder it for improved cache efficiency in force computations and neighbor list building [20]. Atoms are binned in a spatial sense and then reordered so that atoms in the same bin are adjacent in the vector. A bin size of half R_n and a sorting frequency of 1000 timesteps gives good performance for many models, including liquids where atoms can spatially rearrange relatively quickly; these parameters can be altered by the user.

To build a local neighbor list in linear $O(M)$ time, where M in this case is the number of owned plus ghost atoms on a processor, the simulation domain is overlaid (conceptually) with a regular 3d (or 2d) grid of neighbor bins, as in Fig. 3 for 2d models and a single processor's subdomain. Each processor stores a set of neighbor bins which overlap its subdomain extended by the neighbor cutoff distance R_n. As illustrated, the bins need not align with processor boundaries; an integer number in each dimension is fit to the size of the entire simulation box, allowing them to be smaller than if they were fit to each processor's subdomain.

Most often LAMMPS builds what it calls a "half" neighbor list where each I, J neighbor pair is stored only once, with either atom I or J as the central atom. The build can be done efficiently by using a pre-computed "stencil" of bins around a central origin bin which contains the atom whose neighbors are being searched for. A stencil is simply a list of integer offsets in x, y, z of nearby bins surrounding the origin bin which are close enough to contain any neighbor atom J within a distance R_n from any atom I in the origin bin. Note that for a half neighbor list, the stencil can be asymmetric since each atom only need store half its nearby neighbors.

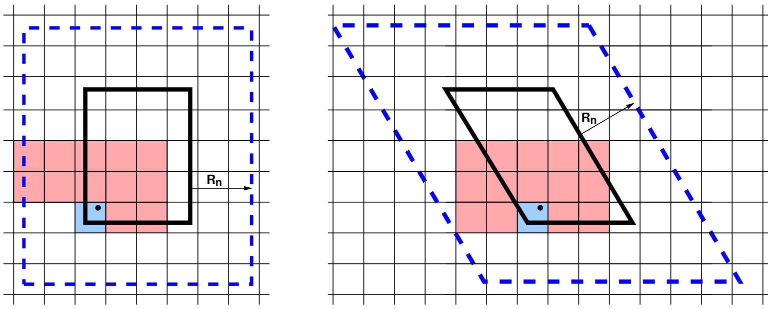

Fig. 3. One processor's subdomain for a 2d simulation, outlined with thick lines for both orthogonal (left) and triclinic (right) domains. The dashed-line box is augmented by the neighbor cutoff distance R_n and contains all the owned+ghost atoms for the processor. A regular grid of neighbor bins (thin lines) overlays the entire simulation domain and need not align with subdomain boundaries; only the portion overlapping the augmented subdomain is shown. In the triclinic case it overlaps the orthogonal bounding box of the tilted dashed-line parallelogram. The blue- and red-shaded bins are explained in the text and represent a stencil of bins searched to find neighbors of the black atom. The stencil is altered slightly in the triclinic case.

These stencils are illustrated in the figure for a half list and a bin size of $1/2R_n$. There are 13 red+blue stencil bins in 2d (for the orthogonal case, 15 for triclinic). In 3d there would be 63, 13 in the plane of bins that contain the origin bin and 25 in each of the two planes above it in the z direction (75 for triclinic). The reason the triclinic stencil has extra bins is because the bins tile the bounding box of the entire triclinic domain and thus are not periodic with respect to the simulation box itself. The stencil and logic for determining which I, J pairs to include in the neighbor list are altered slightly to account for this.

To build a neighbor list, a processor first loops over its M owned plus ghost atoms and assigns each to a neighbor bin. This uses an M-length integer vector to create a linked list of atom indices within each bin. It then performs a triply-nested loop over its owned atoms I, the stencil of bins surrounding atom I's bin, and the J atoms in each stencil bin (including ghost atoms). If the distance $r_{IJ} < R_n$, then atom J is added to the vector of atom I's neighbors.

Here are additional details for neighbor list build options that LAMMPS supports:

- The choice of bin size is an option; a size half of R_n is typically close to optimal. Smaller bins incur additional overhead to loop over a larger stencil; larger bins require more distance calculations for atoms beyond the cutoff distance. Note that for smaller bin sizes, the 2d stencil in the figure would be more semi-circular in shape (hemispherical in 3d), with bins near the corners of the square eliminated due to their distance from the origin bin.
- Depending on the interatomic potential(s) and other commands used in an input script, multiple neighbor lists and stencils with different attributes may be needed. This includes lists with different cutoff distances, e.g. for force computation versus occasional diagnostic computations such as a radial distribution function, or for the r-RESPA time integrator [21] which can partition pairwise forces by distance into subsets computed at different time intervals. It includes "full" lists (as opposed to half lists) where each I, J pair appears twice, stored once with I and J, and which use a larger symmetric stencil. It also includes lists with partial enumeration of ghost atom neighbors. The full and ghost-atom lists are used by various manybody interatomic potentials. Lists may also use different criteria for inclusion of a pair interaction. Typically this simply depends only on the distance between two atoms and the cutoff distance. But for finite-size coarse-grained particles with individual diameters (e.g. polydisperse granular particles), it can also depend on the diameters of the two particles.
- As explained in Section 4.2.2, so-called hybrid models can use multiple interatomic potentials to compute forces between different materials. In this case a master neighbor list for the full system may be partitioned into multiple sub-lists, each of which contain only the I, J pairs needed to compute interactions between subsets of atoms for an individual potential. This means not all I or J atoms owned by a processor are included in a particular sub-list.
- Some models use different cutoff lengths for pairwise interactions between different kinds of particles which are stored in a single neighbor list. One example is a solvated colloidal system with large colloidal particles where colloid/colloid, colloid/solvent, and solvent/solvent interaction cutoffs can be dramatically different. Another is a model of polydisperse finite-size granular particles; pairs of particles interact only when they are in contact with each other. Mixtures with particle size ratios as high as 10-100× may be used to model realistic systems. Efficient neighbor list building algorithms for these kinds of systems are available in LAMMPS. They include a method which uses different stencils for different cutoff lengths and trims the stencil to only include bins that straddle the cutoff sphere surface [22]. More recently a method which uses both multiple stencils and multiple bin sizes was developed [23]; it builds neighbor lists efficiently[2] for systems with particles of any size ratio, though other considerations (timestep size, force computations) may limit the ability to model systems with huge polydispersity.

3.4. Long-range interactions

For charged systems, LAMMPS computes long-range Coulombic interactions via the FFT-based particle-particle/particle-mesh (PPPM) method [24] as formulated in [25]. PPPM is closely related to the more commonly used particle-mesh Ewald (PME) method [26], differing only in the shape function used to interpolate charge to the mesh (grid). For a desired level of accuracy, PPPM can generally use a slightly coarser grid than PME [27]. LAMMPS also has options to compute long-range Coulombic interactions via the multilevel summation method (MSM) [7] or the fast multipole method (FMM), the latter by linking to the ScaFaCos package [28,29]. The LAMMPS implementation of MSM [30] is generally slower than PPPM unless a low-accuracy result is acceptable, but it has the advantage of allowing for fully non-periodic systems; PPPM in LAMMPS is limited to either fully periodic systems or slab geometries which are non-periodic in a single dimension.

In PPPM (as in PME), a regular FFT grid is overlaid on the simulation domain and Coulombic interactions are partitioned into

[2] Joel Clemmer (Sandia National Laboratories) helped implement the newest polydisperse neighbor list build methods in LAMMPS.

short- and long-range components. The short-ranged portion is computed in real space as a loop over pairs of charges within a cutoff distance, using a neighbor list as described in the preceding Section 3.3. The long-range portion is computed in reciprocal space and proceeds in several stages: (a) each atom's point charge is interpolated to nearby FFT grid points, (b) a forward FFT is performed, (c) a convolution operation is performed in reciprocal space, (d) one or more inverse FFTs are performed, and (e) electric field values from grid points near each atom are interpolated to compute the force on it. A single inverse FFT can be used for smoothed PPPM, similar to smoothed PME [31]; three inverse FFTs (one per electric field component) are used for the analytic differentiation formulation of PPPM.

For any of the spatial-decomposition partitionings illustrated in Fig. 1 for regular or irregular tilings, each processor owns the brick-shaped portion of FFT grid points contained within its subdomain. The two interpolation operations use a stencil of grid points surrounding each atom. To accommodate the stencil size, each processor also stores a few layers of ghost grid points surrounding its brick. Forward and reverse communication of grid point values is performed similar to the communication described in Section 3.2 for per-atom values. In this case, electric field values on owned grid points are sent to neighboring processors to become ghost point values. Likewise charge values on ghost points are sent and summed to values on owned points.

For triclinic simulation boxes, the FFT grid planes are parallel to the box faces, but the mapping of charge and electric field values to/from grid points is done in reduced coordinates where the tilted box is conceptually a unit cube, so that the stencil and FFT operations are unchanged. However the FFT grid size required for a given accuracy is larger for triclinic domains than it is for orthogonal boxes.

As is well known, parallel 3d FFTs require substantial communication relative to their computational cost. The so-called pencil-decomposition algorithm LAMMPS uses to interleave computation and communication is illustrated in Fig. 4; an early version was described in [32]. Initially (upper left), each processor owns a brick of same-color grid cells (actually grid points) contained within in its subdomain. A brick-to-pencil communication operation converts this layout to 1d pencils in the x-dimension (upper right). Again, cells of the same color are owned by the same processor. Each processor can then compute a 1d FFT on each pencil of data it wholly owns using vendor-provided libraries (e.g. MKL or cuFFT) or FFTW [33]. A pencil-to-pencil communication then converts this layout to pencils in the y dimension (lower left) which effectively transposes the x and y dimensions of the grid, followed by 1d FFTs in y. A final transpose of pencils from y to z (lower right) followed by 1d FFTs in z completes the forward FFT. The data is left in a z-pencil layout for the convolution operation. One or more inverse FFTs then perform the sequence of 1d FFTs and communication steps in reverse order; the final layout of resulting grid values is the same as the initial brick layout.

Each communication operation within the FFT (brick-to-pencil or pencil-to-pencil or pencil-to-brick) converts one tiling of the 3d grid to another, where a tiling in this context means an assignment of a small brick-shaped subset of grid points to each processor, the union of which comprise the entire grid. The parallel fftMPI library [34] written for LAMMPS allows arbitrary definitions of these tilings so that an irregular partitioning of the simulation domain (see right side of Fig. 1 and Section 5.3) can use it directly. Likewise 2d slabs of grid cells (instead of 1d pencils) can be an intermediate tiling when performing an FFT on fewer processors than a dimension of the grid. The latter allows for use of on-processor 2d FFTs as part of the 3d FFT, reducing the number of communication operations. Transforming data from one tiling to another is implemented in fftMPI using point-to-point communication, where each processor sends data to a few other processors, since each tile in the initial tiling overlaps with a handful of tiles in the final tiling.

The transformations could also be done using collective communication across all P processors with a single call to *MPI_Alltoall()*, but this is typically much slower. However, for the specialized brick and pencil tilings illustrated in Fig. 4, collective communication across the entire MPI communicator is not required. In the figure an 8^3 grid with 512 grid cells is partitioned across 64 processors; each processor owns a $2 \times 2 \times 2$ 3d brick of grid cells. The initial brick-to-pencil communication (upper left to upper right) only requires collective communication within subgroups of 4 processors, as illustrated by the 4 colors. More generally, a brick-to-pencil communication can be performed by partitioning P processors into $P^{2/3}$ subgroups of $P^{1/3}$ processors each. Each subgroup performs collective communication only within its subgroup. Similarly, pencil-to-pencil communication can be performed by partitioning P processors into $P^{1/2}$ subgroups of $P^{1/2}$ processors each. This is illustrated in the figure for the $y \Rightarrow z$ communication (lower left to lower right). An eight-processor subgroup owns the front yz plane of data and performs collective communication within the subgroup to transpose from a y-pencil to z-pencil layout.

LAMMPS invokes point-to-point communication by default, but also provides the option of partitioned collective communication. In the latter case, the code detects the size of the disjoint subgroups and partitions the single P-size communicator into multiple smaller communicators, each of which invokes collective communication.[3] Testing on the large IBM Blue Gene/Q machine at Argonne National Labs showed a significant improvement in FFT performance for large processor counts; partitioned collective communication was faster than point-to-point communication or global collective communication involving all P processors.

Here are additional details about FFTs for long-range and related grid/particle operations LAMMPS support:

- The fftMPI library allows each grid dimension to be a multiple of small prime factors (2,3,5), and allows any number of processors to perform the FFT. The resulting brick and pencil decompositions are thus not always as well-aligned as in Fig. 4, but the size of subgroups of processors for the two modes of communication (brick/pencil and pencil/pencil) still scale as $O(P^{1/3})$ and $O(P^{1/2})$.
- For efficiency in performing 1d FFTs, the grid transpose operations illustrated in Fig. 4 also involve reordering the 3d data so that a different dimension is contiguous in memory. This reordering can be done during the packing or unpacking of buffers for MPI communication.
- At large scale, the dominant cost for parallel FFTs is often communication, not the computation of 1d FFTs, even though the latter scales as $M \log(M)$ in the number of grid points M. As discussed in Section 4.1, LAMMPS has an option to partition the allocated P processors into two subsets, say 1/4 and 3/4 of P in size. In this context the smaller subset can perform the entire PPPM computation, including the FFTs, so that the FFTs run more efficiently (in a scalability sense) on a smaller number of processors. The larger subset is used for computing pairwise or manybody forces, as well as intramolecular forces (bond, angle, dihedral, etc). The two partitions perform their computations simultaneously; their contributions to the total energy, force, virial are summed after they both complete.
- LAMMPS also implements PPPM-based solvers for other long-range interactions, dipole [35] and dispersion (Lennard-Jones)

[3] Paul Coffman (Argonne National Laboratory) devised and implemented this partitioned collective communication algorithm within LAMMPS.

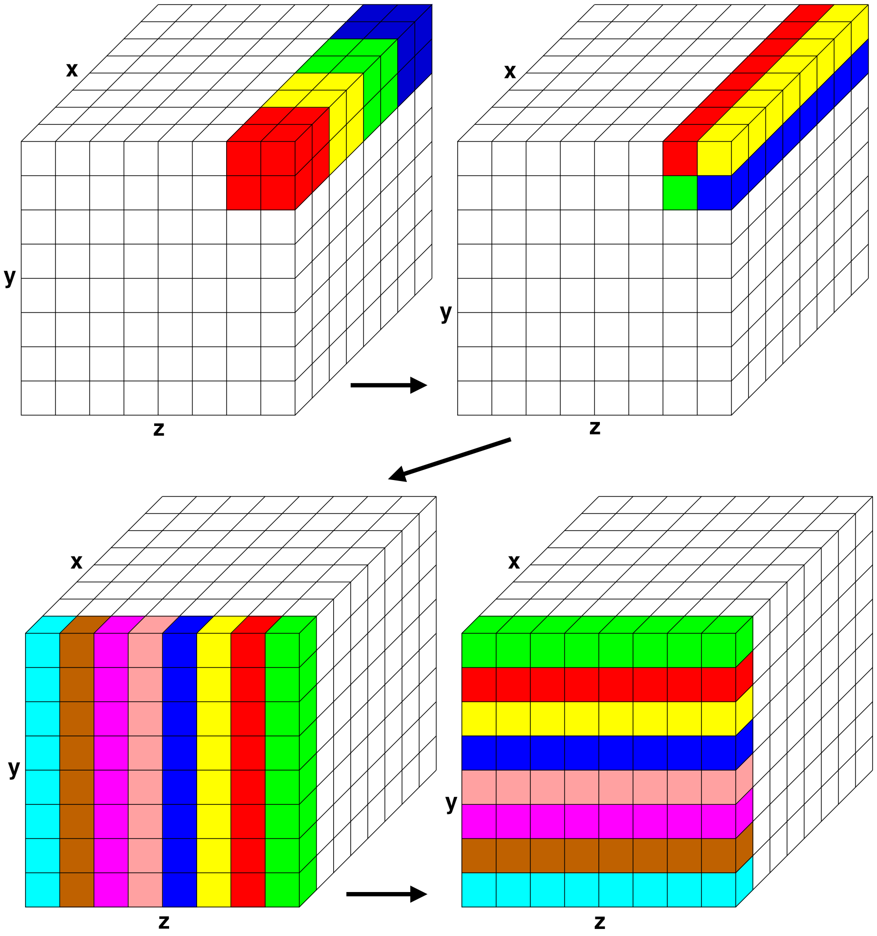

Fig. 4. Stages of a parallel FFT for a simulation domain overlaid with an 8 × 8 × 8 3d FFT grid, partitioned across 64 processors. Within each of the 4 diagrams, grid cells of the same color are owned by a single processor; for simplicity only cells owned by 4 or 8 of the 64 processors are colored. The top two images illustrate brick-to-pencil communication. Initially each processor owns a 2 × 2 × 2 3d brick of cells; after communication each owns one or more 1d pencils of cells in the *x* dimension. The bottom two images illustrate pencil-to-pencil communication, which in this case transposes the *y* and *z* dimensions of the grid. Both operations are described more fully in the text.

[36], which can be used in conjunction with long-range Coulombics for point charges.

- LAMMPS implements a *GridComm* class which overlays the simulation domain with a regular grid, partitions it across processors in a manner consistent with processor subdomains, and provides methods for forward and reverse communication of owned and ghost grid point values. It is used for PPPM as an FFT grid (as outlined above) and also for the MSM algorithm which uses a cascade of grid sizes from fine to coarse to compute long-range Coulombic forces. The *GridComm* class is also useful for models where continuum fields interact with particles. For example, the two-temperature model (TTM) defines heat transfer between atoms (particles) and electrons (continuum gas) where spatial variations in the electron temperature are computed by finite differencing the discretized heat equation on a regular grid [37,38]. The TTM was originally implemented in LAMMPS via a *fix ttm* command [39] and a new *fix ttm/grid* variant now uses the *GridComm* class internally to perform its grid operations in parallel.

4. Design of LAMMPS for flexibility

In materials modeling, it is common for a user of an MD code like LAMMPS to want to customize a model by altering or enhancing it in some manner, potentially for a one-time simulation. One of our chief goals with LAMMPS was to make it easy to do this, even for users without extensive programming experience. Here we discuss this from three perspectives. The first is what can be customized without programming from within a LAMMPS input script. The second involves writing code to add to LAMMPS, to extend its capabilities. The third is to use LAMMPS as a library rather than as a stand-alone application, which may require writing new code or a script external to LAMMPS. Appendix A describes support mechanisms we offer users to help with these options.

Some of these ideas were discussed at a high level in [40] in the context of creating a community of users who both use and enhance a materials modeling code. Here we give more technical details as to how the ideas are implemented.

4.1. Flexibility via input script options

A LAMMPS input script (text file) is simply a series of lines each beginning with a command name followed by one or more whitespace separated arguments. Programming-like commands are included which define variables, perform conditional tests, execute loops, or invoke shell commands, e.g. to launch a program external to LAMMPS. The input script is parsed and executed one line at a time which means a single script can be used to run a simulation in stages, altering one or more parameters between the stages, or to run a series of independent simulations where the entire system is reinitialized multiple times.

Variable values can be set in the input script or via the command line when LAMMPS is invoked to set simulation parameters at run-time. Variables can be defined, generated, or read from a file as a sequence of numbers or strings, which makes it easy to loop over a sequence of simulations that run with different parameters, e.g. a list of temperatures. Variables can also define mathematical expressions which are evaluated later and produce either a single value or a value for each atom. These expressions can use math operators, intrinsic functions, and simulation output keywords for quantities like temperature or pressure. They can also access *per-atom* data (coordinates, velocities, charge, etc.), or the output of various LAMMPS commands; see the discussion of *fix* and *compute* styles in the next Section 4.2.

Variables can be evaluated in the input script to create an argument for a command. Alternatively, names of variables which store formulas can be passed as an argument to a LAMMPS command which evaluates the variable internally to alter the command's behavior, either once or periodically as a simulation runs. This makes it easy for the user to customize the behavior of a command, e.g. to make it time- or spatially-dependent. For example, a thermostat command can use a variable to specify a time-dependent target temperature for a complex annealing cycle. As another example, a command that imposes an external electric field on charged atoms can apply it in a spatially-dependent manner.

When LAMMPS is launched in parallel it runs on P processors (MPI tasks). A command-line option can be used to partition the processors into M subsets, with a specified number of processors per subset. Within LAMMPS, this splits the top-level MPI communicator *MPI_COMM_WORLD* into M sub-communicators. All the MPI calls within the code use the processor's sub-communicator as their argument, which effectively means M independent simulations are running. The only exception is a handful of MPI calls used by commands that perform loosely coupled multi-replica simulations, discussed in Section 6.6, where inter-replica communication is needed.

The input script can define how the M simulations differ by using per-partition variables and, if needed, by commands that are specific to one or more of the simulations. This enables an allocation of time on a cluster or supercomputer for P processors to be used in a variety of ways, all from a single input script. A single long simulation, using all the processors, can run until the allocation time expires, and then later be continued (from a restart file) when a new allocation starts up. Alternatively, M long simulations can be run simultaneously, each on a different subset of processors. Finally, a large number $N >> M$ of simulations can be run, one after the other. Each can run on all P processors, or on one of the M subsets of processors. When one of the N simulations completes, LAMMPS assigns the next one to the (sub)set of processors that is now idle.

Many commands operate on what LAMMPS calls *groups* of atoms which can be defined statically or dynamically. An atom can be assigned to one or more groups. For example, one group could consist of boundary atoms, another group could be atoms currently inside a geometric region whose size or shape varies with time. Atoms can also be partitioned into *chunks*, where for a set of chunks, each atom is assigned to a single chunk. For example, a chunk could be a molecule or a spatial bin. The commands that operate on groups or chunks can be invoked multiple times to operate on different subsets of atoms. For example, a thermostat command can be used twice in different regions of the simulation domain with a different target temperature to create a temperature gradient. Diagnostic commands can likewise be invoked multiple times on different groups of atoms or on a chunk partitioning to produce a collection of per-chunk values.

Many commands produce output (internally) that can be used as input to other commands. This means the input script can perform a variety of on-the-fly processing tasks with simulation data. This can reduce the need to write large amounts of output to the file system and can also save the user from having to write and debug custom post-processing analysis code. Commands can produce *global* data, meaning all processors have a copy of a scalar, vector, or array. Or they can produce *per-atom* data, meaning each processor stores one or more values for only the atoms it owns. This data can be operated on directly using the variables described above that store math formulas. There are also commands to time- and/or spatially average this data, including commands that generate histograms or calculate time correlations.

Another source of flexibility is input script commands which create and name one or more user-defined *per-atom* properties. The property values can be set in the input script or read from a file. Internally, *per-atom* properties are stored with each atom, which means they persist for the duration of the simulation. These properties can be accessed by existing commands that use them as arguments. For example, the command that defines a set of chunks (by assigning each atom to a chunk), can partition the atoms based on a user-defined *per-atom* property. Another command treats each *chunk* of atoms as a rigid body for time integration. This means a user can effectively customize how these commands operate, i.e. choose which atoms are in what rigid bodies, and accumulate statistics on each body as a simulation runs. More generally, these user-defined properties can be accessed and altered as a simulation runs by new code written and added to LAMMPS, as described in the next Section 4.2.

Finally, Python code can be easily integrated into a LAMMPS input script by assigning a Python function to an input script variable. The code for the function can be embedded in the input script or exist in a separate file which can contain many functions. Each time the variable is evaluated, either by the input script, or periodically during a simulation by a command that uses the variable as an argument, the Python function is invoked with arguments specified in the input script. These can be numeric or string values or other input script variables. The Python function returns a value that can be used by the input script or by the command that invoked it.

The Python function can also use the LAMMPS library interface, discussed in Section 4.3, in a variety of ways. It can access internal LAMMPS data such as *per-atom* properties. It can also invoke LAMMPS commands which perform a computation and retrieve the results, e.g. to compute the temperature of the system. Alternatively, it can reset *per-atom* values, e.g. implement a charge-relaxation algorithm to adjust the charge on each atom. When LAMMPS is running in parallel, each processor invokes the Python function, which means the function can operate in a parallel manner on only the subset of atoms owned by that processor. The function can also use the *mpi4py* Python module [41] which wraps the MPI library to enable communication of data between processors.

4.2. Flexibility via source-code styles

An important consideration for making it easy for users to extend a code like LAMMPS is to make the source code easy to read and understand. LAMMPS is written in C++, but we intentionally do not use all the complexity of C++, such as operator overloading or excessive templating, because our audience is not exclusively expert programmers but domain scientists whose computational fluency may be in a variety of languages (Fortran, Python, Java, etc.). As mentioned in Section 2, LAMMPS is instead designed in more of an object-oriented C fashion where low-level data structures are generally simple (vectors, arrays, structs) and performance-critical kernels are written in a straightforward C-like syntax. We leverage C++ mostly at a high level within the code to provide flexibility and extensibility as discussed next or at a low level to implement functionality that users are expected to only use at a higher level through documented convenience functions and classes.

Table 1
One-line descriptions of many of the key styles defined as parent classes in LAMMPS. The numbers (in parenthesis) are the count of child class variants available in current LAMMPS. It does not include additional variants optimized for different hardware, e.g. for Intel CPUs, OpenMP threading, or GPUs.

atom	25	sets of *per-atom* properties
pair	230	pairwise and manybody potentials
bond/angle/dihedral bond/angle/dihedral/improper	15/20/15/10	molecular interactions
fix	225	operations while timestepping
compute	140	diagnostic calculations
region	8	geometric regions
dump	25	output of simulation snapshots
kspace	15	long-range Coulombic (dispersion, etc) solvers
minimize	9	energy minimization algorithms
npair/nstencil/nbin	75/20/5	neighbor list building operations
integrate	2	Verlet or r-RESPA algorithms
command	45	added input script commands

The core functionality of LAMMPS is encoded in just a dozen or so main classes. Examples include the *Domain* class which stores the geometry of the simulation box and its boundary conditions, the *Neighbor* class which stores the attributes and data for one or more neighbor lists, and the *Comm* class which stores information on how the domain is partitioned across processors and has methods for communicating information between processors in various patterns, like those described in Section 3.2. Pointers to the core classes are stored in a single base class inherited by all other LAMMPS classes, so their data and methods are accessible anywhere in the code, including from new code that a user adds.

In addition to core classes, LAMMPS defines about 20 C++ parent classes, called *styles*, that are broadly applicable to molecular dynamics, many of which are listed in Table 1. In C++ terminology, the parent styles are pure virtual classes which means they define a generic interface for how the rest of the code uses any child-class instance of the parent. The interface includes data structures common to all children as well as required and optional methods a child class defines. The rest of the code can access the data or call the methods without knowing the details of how a specific child class performs its operations. In practice, the author of a new child class often need only implement one or a few methods for it to work seamlessly with the rest of LAMMPS.

The mechanism for a user to add a child class to the code is to simply copy it into the LAMMPS source directory; it will then be included in a build of the code. The header file for each child class contains one line which associates a string with the class name. The compiler generates code from that line which instantiates the class if an input script command uses the matching string.

This model has proven popular with users for extending LAMMPS, either for their own use, or to contribute to the publicly released version. Child classes for the various styles now comprise over 90% of the million lines of LAMMPS source code. The remaining 75K lines are the core and parent classes described above, along with various utility classes. This model also enables modularity. Sets of related child class source files are grouped in sub-directories called *packages*; LAMMPS currently has more than 75 packages. Packages with classes a user does not need can be excluded from a LAMMPS build.

Here are more details on a few of the styles listed in Table 1, to indicate how they offer flexibility and provide a breadth of capabilities for LAMMPS.

4.2.1. Atom styles

Atom styles define the set of *per-atom* properties stored by a simulation and specify which are used in various communication or I/O operations. Because LAMMPS supports a variety of simulation models which operate at scales from the atomic to the continuum, the required properties for an atom or particle can vary greatly. Examples include moments of inertia, orientation, and torque for coarse-grained aspherical particles, radius and radial velocity for dynamic-size electron “particles” defined by the electron force field (eFF) [42], and internal energy, heat capacity, temperature, or chemical concentrations for smoothed particle hydrodynamics (SPH) or dissipative particle dynamics (DPD) models. If enabled, each property is stored in an allocated vector or array (e.g. an Nx3 array for atom coordinates); atom properties are thus conceptually stored by each processor as an SoA (struct of arrays).

Writing a new atom style can be as simple as defining a handful of strings, each of which list variable names for specific atom properties (e.g. x for a $N \times 3$ array of atom coordinates, q for a vector of N charges). The strings define which properties are used in specific operations, e.g. which properties to communicate to other processors (ghost atoms) every timestep, or on re-neighboring timesteps, or when atoms migrate to new processors, which properties are written to a restart file, or which properties are read from an initial data file. If needed, extra methods can be added, e.g. to convert a particle diameter read from an input file to an internally-stored radius. A *hybrid* atom style is also provided which effectively concatenates the atom properties defined by multiple atom styles, making it easier to formulate complex models, e.g. adding charges or dipole moments to DPD particles.

4.2.2. Pair styles

Pair styles are implementations of interatomic potentials (force fields) that use a pairwise neighbor list (Section 3.3). They compute pairwise or manybody interactions as discussed in Section 6.1. These are nonbonded interactions between nearby atoms. Long-range interactions (Coulombic, dispersion, dipolar) are computed by *kspace* styles; interactions between covalently bonded atoms are computed by *bond*, *angle*, *dihedral*, etc. styles.

Pair styles define a *compute()* method, invoked once per timestep, which takes as input the current atom properties for owned and ghost atoms plus one or more neighbor lists, and calculates the force on each atom as well as *global* and *per-atom* energy and virial tensors, if needed. Importantly, this method can be replaced by derived classes which implement optimized versions of the interatomic potential for different hardware, such as GPUs or Intel CPUs; see Section 5.1 for details. Depending on the complexity of the potential, a *compute()* method can be a few dozen lines of code or many thousands.

The core *Comm* class provides helper methods for forward and reverse communication (to/from ghost atoms) which a *pair* style can invoke and provide *pack* and *unpack* methods for. This is useful for manybody potentials, which often calculate intermediate *per-atom* quantities which need to be shared with or accumulated from ghost atoms.

Similarly, the parent *Pair* class provides several methods which can be called by any *pair* style to accumulate *per-atom* virial (stress) contributions from a manybody force calculation, using the methods outlined in [43]. There is also a customizable class (with its own styles) that can be invoked within the innermost loop of the *Pair::compute()* method to perform these tallies in alternate ways. For example, recent theoretical analysis has shown that the correct formulation of the heat flux for manybody potentials requires calculating *per-atom* contributions to a non-symmetric virial tensor and are defined in terms of the centroid of each manybody cluster [44].

A hybrid *pair* style is provided, which allows the user to list multiple individual *pair* styles to define a model. This can be done in one of two ways. One or more atom types (e.g. atomic elements) can be assigned to a specific *pair* style, as can cross-type interactions. An example is a model for water or polymer chains on a metal surface. The metal can be modeled with one *pair* style, the water or polymer with another, and water/metal interactions with a third. The second way is to overlay multiple *pair* styles to build up a more complex model. For example, the Streitz-Mintmire potential [45] has two terms: (1) a manybody Embedded Atom Method (EAM) *pair eam* style or other options available in LAMMPS, and (2) a Coulombic *pair coul/streitz* where the charge distribution around an atom is modeled as a point charge plus a Slater 1s orbital. Another example is the coarse-grained Derjaguin-Landau-Verwey-Overbeek (DLVO) potential for colloidal particle interactions [46,47]. It has two terms for van der Waals attraction and electrostatic repulsion. These are implemented as separate *pair colloid* and *pair yukawa/colloid* styles in LAMMPS which when overlaid become a DLVO model.

A *pair python* style is also provided, which allows the compute() method for a simple pairwise model to be implemented in Python code which calculates the energy and force between a single pair of atoms. As discussed in Section 4.1, the Python code is imported from a specified Python module that has to implement a Python class following a specific structure. This enables quick implementation of an interatomic potential for testing or debugging purposes, before making the effort to code a more performant C++ version. Alternatively, a *python* pair style can also be used to output a tabulation of energy/force versus distance to a file via the *pair_write* command. The file can be then be used by the *pair table* style which has performance on CPUs or GPUs comparable to coding the pairwise analytic formulas in C++.

4.2.3. Fix styles

Fix styles implement operations performed during a dynamics timestep or an energy minimization iteration. Specifying a *fix* in the input script or implementing a new one, thus enables customization of a model. This is illustrated in Fig. 5 where the sequence of fundamental operations for an MD timestep to run in parallel are shown unshaded. Optional fix computations which interleave between the required ones are shown shaded in blue; examples of such fixes are shaded in red. Internally, each *fix* sets flags which determine where in the timestep one or more of its methods will be invoked. In a fuller description of the timestep there are ∼10 such flags to choose from. Most fixes implement just one or two of these methods. As with the *pair python* command, *fix python* commands are provided which allow implementation of several of these methods in Python code. Finkelstein et al. used this approach to compare advanced Langevin time integrators [48].

The *initial_integrate()* and *final_integrate()* methods of *fix* styles can be used to implement portions of the velocity-Verlet algorithm for various ensembles at the appropriate points in the timestep. The *pre_migrate()* method can be used to add or delete atoms from the system, e.g. via Monte Carlo insertion. The *pre_force()* method is used by the *fix adapt* command to alter force field parameters for alchemical free energy calculations, or to alter the diameter of coarse-grained particles, e.g. to model swelling or an explosion (at the continuum scale) due to rapid expansion of one or more particles. The *post_force()* method can alter the forces just computed by the interatomic potential by addition of constraints or boundary conditions. The *end_of_step()* method is used by *fix* styles which alter the simulation box or perform time-averaging of various diagnostic quantities requested by the user.

As mentioned in Section 4.1, most *fix* styles act on specified groups of atoms and the same *fix* can be invoked multiple times. For example, a rigid-body integrator *fix* can be used for rigid bodies (each a collection of atoms) along with a traditional NVE or NVT integrator *fix* for solvent atoms or flexible molecules that surround the rigid bodies. Or a *fix* command that imposes boundary conditions can be used multiple times with different settings for different portions of the simulation domain.

Fix styles also allow information to persist from timestep to timestep, either for time averaging purposes or to enable *per-atom* values stored by the *fix* to migrate with atoms as they move to new processors. For example, for accelerated MD algorithms used to more efficiently model rare-event systems, a *fix* is used to store the state of the system (atom coordinates and velocities) before an energy minimization (quench) is performed (which may migrate atoms to new processors) to detect if an event has occurred. The pre-quench state can then be restored so dynamics can continue unaltered. As another example, interatomic potentials for granular materials often need information from previous timesteps, e.g. to accumulate strain between two frictional particles which are sliding past each other and remain in contact for multiple timesteps. A *fix* is used to store this contact history for pairs of particles (similar to a neighbor list) so it can migrate with particles as they move to new processors.

Like *pair* styles, *fix* styles can use neighbor lists or invoke methods in the core *Comm* class to perform forward and reverse communication of *per-atom* values the *fix* calculates. An example is several *fix* styles which perform dynamic charge equilibration (QEq), as discussed in Section 6.4. The QEq equations include a sparse matrix whose sparsity pattern is due to interactions between pairs of charged atoms within a finite cutoff. A parallel iterative solution to the matrix equation performs both forward and reverse communication at each iteration to compute a matrix-vector multiply that includes interactions with ghost atoms.

Fix styles can also be used to wrap external libraries, which perform complex computations. One example is the atom-to-continuum (ATC) library [49], included in the LAMMPS distribution, which stores a finite-element mesh which overlays and/or surrounds the volume of MD atoms. Each timestep, atom information is passed to the library which computes coupling terms between the atoms and elements. The library computes partial differential equations (PDEs) which may either deform the elements or update per-element quantities. And it computes resulting forces on atoms which need to be returned to LAMMPS. A *fix atc* command is effectively the interface between LAMMPS and the ATC library.[4]

A second example is the LATTE density-functional tight-binding (DFTB) code [50], which can be called as a library from LAMMPS to compute quantum forces (within the tight-binding approximation) between a collection of atoms. To couple the two codes, a *fix latte* command was written with a *post_force()* method which calls LATTE each timestep with the current simulation box and atom coordinates. LATTE solves the DFTB equations and returns forces on each atom. LAMMPS can enforce boundary conditions and invoke time integration or energy minimization to perform *ab initio* MD or molecular statics.[5]

[4] Reese Jones, Jeremy Templeton, and Jon Zimmerman (Sandia National Laboratories) developed the ATC library and the *fix atc* command in LAMMPS.

[5] Christian Negre (Los Alamos National Laboratory) helped implement the *fix latte* command to couple LATTE to LAMMPS.

```
Loop over timesteps:
    fix initial_integrate          NVE/NVT/NPT, rigid-body integration
    communicate ghost-atom coordinates
    if time to build neighbor list(s):
        fix pre_migrate            insert or remove particles
        migrate particles to new processors
        build neighbor list(s)
    fix pre_force                  alchemical or particle size changes
    compute forces
    communicate ghost-atom forces
    fix post_force                 SHAKE, walls, springs, gravity
    fix final_integrate            NVE/NVT/NPT, rigid-body integration
    fix end_of_step                alter box volume/shape, time-average diagnostics
    output to screen and/or files
```

Fig. 5. Simplified outline of a parallel MD timestep in LAMMPS. The unshaded lines are fundamental operations that all simulations perform. The blue-shaded operations are optional and are implemented by *fix* styles, selected (or implemented) by the user, which customize the model. The red-shaded text gives examples of such operations.

4.2.4. Compute styles

Compute styles in LAMMPS calculate a property of the system at the moment in time they are invoked. As explained in Section 4.1, they operate on specified groups or chunks of atoms, and they produce output which is stored internally for use by other commands. Adding a new *compute* style typically only requires implementing a single method, setting a few flags, and insuring the calculated result is stored in the proper variables for the rest of LAMMPS to access and use.

Like *pair* and *fix* styles, *compute* styles can use neighbor lists or invoke methods in the core *Comm* class to perform forward and reverse communication if needed. If a *compute* style also needs to store persistent *per-atom* state, it can use a *fix* to do so. For example, the *compute msd* command calculates mean-squared displacements for diffusion coefficients. It uses an instance of *fix store* to store the time=0 position of each atom so that as atoms diffuse and migrate to new processors during a long simulation, each processor can compute the displacements of its current owned atoms.

Like *fix* styles, *compute* styles can also wrap external libraries to perform complex diagnostic calculations. An example in LAMMPS is the *compute voronoi/atom* command which wraps the Voro++ library [51,52] to compute Voronoi tessellations of a collection of atoms. These are useful in several contexts in materials science, e.g. for estimating *per-atom* volumes or categorizing defects in solid-state systems. In parallel, if each processor passes coordinates of its owned and ghost atoms to its instance of the Voro++ library, then it computes a tessellation for its owned atoms which is consistent with the global tessellation of the entire simulation domain.

4.2.5. Npair, Nstencil, Nbin styles for neighbor list building

As indicated above, *pair*, *fix*, and *compute* styles may use neighbor lists. Each style does this by requesting one or more neighbor lists with the specific attributes it needs; Section 3.3 discusses relevant attributes. The core Neighbor class processes the requests and schedules the building of as many different lists as needed, each with their own set of attributes.

Lists requested by *pair* styles need to be rebuilt frequently, since they are used every timestep. Lists requested by *fix* or *compute* styles may be required only occasionally, when the fix or compute is invoked, so they can be re-built on demand. For efficiency, one list may simply be a pointer to an existing list, or be derived from another. For example it is computationally cheaper to create a half list from an already built full list, than it is to build it from scratch. Likewise sub-lists which only contain I, J atom pairs for a subset of atoms types can be derived quickly from a master list for all types.

As outlined in Section 3.3, the building of a neighbor list involves binning atoms (*nbin* style in Table 1), a stencil of bins (*nstencil* style) to check for possible neighbors, and a procedure to loop over the stencil and assemble the neighbor list (*npair* style) according to some pairwise criterion. By defining each of these three operations as a parent style class, it is relatively easy in LAMMPS to write one or more new child classes which account for a new attribute or which implement a new algorithm for building a neighbor list.

4.3. Flexibility via use of LAMMPS as a library

Modeling for materials science applications often needs to perform multiphysics or multiscale simulations which use more than one technique or model and thus more than one code. The growing use of workflows that generate large volumes of data for machine learning or other purposes also means that scripts or tools may want to use simulation codes as a black-box service. In such scenarios it is helpful for a code like LAMMPS to be usable as a library, and not just as a stand-alone executable.

A decade ago we re-designed the top level of LAMMPS to convert it into a library [53]. From the perspective of a calling code written in C++, LAMMPS is a single C++ class which when instantiated gives access to the entire code. A C-style API was added which can be called directly from other languages, e.g. C, Fortran, and Python (using *ctypes*). LAMMPS also includes a SWIG [54] interface file, that enables building wrapper modules to call the library from additional programming languages like Java, Lua, Perl, Ruby, Tcl, and many others. To build LAMMPS as a stand-alone executable, a tiny *main.cpp* file is added which simply initializes MPI, instantiates the LAMMPS library with the *MPI_COMM_WORLD* communicator, and makes a library call to pass it the input script.

To make the library flexible, many of the methods and useful data within LAMMPS classes are publicly accessible, either directly or via function calls which return values or pointers to data. The latter means a Fortran program or Python script can call a method that returns a pointer to a contiguous $N \times 3$ array of per-atom coordinates or forces, which is usable by the caller as a Fortran or Numpy array. No copying of the data is performed; the data can be directly accessed or overwritten. If the calling program is running in parallel, each of its processors accesses the portion of data that is spatially decomposed across processors within LAMMPS.

As part of the re-design process, three steps were taken to make LAMMPS a well-behaved library.[6] First, it was encapsulated within a C++ namespace, to avoid collisions with other codes or li-

[6] Depending on which optional packages are included, a LAMMPS build can also link to external libraries, some of which do not follow all the well-behaved guidelines outlined here.

braries it might be coupled with. Second, nearly all global (static) variables were eliminated[7] to allow the library to be instantiated multiple times by the same process. And third, when the calling program instantiates the library it passes in an MPI communicator; internally LAMMPS make no reference to the global *MPI_COMM_WORLD* communicator. This enables the calling program to choose how many and which processors to run one or more instances of LAMMPS on. We note that the LAMMPS library is not yet fully thread-safe which would enable additional modes of parallel use; this requires additional refactoring that is ongoing.

If a calling program needs control returned to it periodically at a specific point within the timestepping, LAMMPS provides a *fix external* command which allows the caller to set the name of a callback function. An example of its use is the Fortran density-functional tight-binding program DFTB+ [55,56] which has been coupled to LAMMPS by Nir Goldman (Lawrence Livermore National Laboratory). One science application this enables was the modeling of the formation of prebiotic organic molecules under extreme thermodynamic conditions, such as when a comet strikes the Earth [57]. DFTB+ triggers LAMMPS to perform an MD simulation on a small number of atoms initialized by DFTB+. Each timestep LAMMPS calls back to DFTB+ with the current atom coordinates at the *post_force()* point in the timestep illustrated in Fig. 5. DFTB+ solves the DFTB equations and returns the forces on each atom. Note that this mode of coupling is the inverse of the way LAMMPS wraps LATTE, another DFTB code which is itself a library, as described in Section 4.2.3.

Aside from the multiphysics, multiscale, and workflow motivations mentioned above, two other library use cases are as follows. For multi-replica simulations, a calling program can instantiate multiple instances of LAMMPS, define a simulation for each instance to run, and either collect the results or orchestrate communication of appropriate data between the replicas. A simple example with two replicas would be a calling program that implements the Gibbs Ensemble Monte Carlo method [58] requiring separate MD simulations of two bulk fluid states and exchanging atoms and box sizes between them to enforce equality of chemical potential and pressure. Each replica could be run in parallel on all the processors. An example with hundreds or thousands of MD replicas, each running on a subset of processors is explained in Section 4.3.2.

A second use case is a calling program simulating material response at the meso or continuum scale while invoking multiple instances of an all-atom or coarse-grained MD simulation at a finer scale within different subdomains of the caller's large simulation domain. An example is the work of Martinez-Saez and Caro which coupled their kinetic Monte Carlo (kMC) code with multiple instances of LAMMPS to model diffusion-drive microstructural evolution (grain growth) [59]. The kMC code can perform a sequence of diffusive events chosen with statistical accuracy from an enumeration of possible events if it knows their respective rates. LAMMPS can compute the barrier height (and thus the rate) for a diffusive event within a small region of surrounding atoms (∼1000) via a multi-replica nudged-elastic band (NEB) calculation as mentioned in Section 6.6. In [59], one instance of LAMMPS was partitioned into processor sub-groups to perform multiple NEB calculations simultaneously; a second instance was used to perform a global relaxation of the entire system after each kMC event was selected and performed.

4.3.1. Use of the LAMMPS library from Python

In addition to a Python wrapper for the C-style API described in 4.3, the LAMMPS distribution also includes additional Python classes which wrap the low-level API to create higher-level objects for easier access and manipulation. With either wrapper a Python script can be used to essentially replace the LAMMPS input script with a sequence of commands composed and issued by Python; this provides more options and programming-like control than the simple input script syntax offers. The Python script can perform complex pre- and post-processing before and after it runs a simulation. It can invoke LAMMPS diagnostic calculations directly, as well as extract information periodically as a simulation runs. It can also launch multiple isolated LAMMPS instances and coordinate their execution. These instances can either run in serial, use multithreading OpenMP options available inside LAMMPS, or run in parallel using MPI if Python is used with the *mpi4py* module [41].

Because there are Python interfaces to a huge variety of software tools, Python can also be used to couple LAMMPS to real-time plotting or visualization or to a GUI with widgets that dynamically control a running simulation. It also enables LAMMPS inputs and outputs to be embedded in web-based Jupyter notebooks for pedagogical or record-keeping purposes. Writing a Python script is also a quick and lightweight way to experiment with coupling multiple codes together for multiphysics or multiscale applications.

A particularly noteworthy example of a powerful Python application which wraps LAMMPS (and many other codes) is the Atomic Simulation Environment (ASE) [60,61]. It allows structures created in ASE to be passed to LAMMPS via its library interface. Results of the simulation can then be accessed by ASE for post-processing. ASE can use LAMMPS either as a stand-alone executable or through its library interface. Another example is the *FitSNAP* Python package [62] for training the SNAP machine-learning interatomic potentials discussed in Section 6.3. It uses the LAMMPS library interface to execute tasks at different stages of the training workflow: for evaluation of descriptor gradients with respect to the training loss function, verification of generated LAMMPS potential files, and execution of reference simulations for hyperparameter optimization.

4.3.2. Library extensibility

Similar to the motivation for styles and their parent/child classes discussed in Section 4.2, the LAMMPS library API is meant to be easy for a user to extend by adding one or more methods that perform a needed task.

An example where this was done is the parallel ParSplice code which implements accelerated MD (AMD) algorithms for modeling rare-event solid-state systems for long timescales [63]. It instantiates LAMMPS multiple times to manage a (potentially huge) ensemble of MD simulations running on a large parallel machine. Each instance can be a single replica of an entire small system or a subdomain of a large system. ParSplice launches each simulation on one or more processors (or GPUs) with its atoms in a particular potential energy basin (conformation), runs them for short periods of time (a few ps), and monitors them via quenches (energy minimizations) to detect if transition events have occurred. Based on the results, ParSplice can build a complex model for the system's multiple-basin energy landscape and compute transition rates between basins. It uses this information to initiate additional simulations to efficiently search for new events from different basins. It can then splice together the appropriate short simulations into time-accurate long-timescale trajectories for the system. Examples of its use with LAMMPS include [64] and [65].

To enable ParSplice to use LAMMPS as its MD engine in this manner, we added a library method which takes a simulation box and list of atoms as input and *scatters* them across processors to conform with the LAMMPS parallel spatial decomposition. Par-

[7] A few static class variables are used to enable callbacks to class methods, but in practice they do not restrict the usage modes described above.

Splice invokes this method whenever it starts a new simulation within any of the many LAMMPS instances it manages. A similar *gather* method was written to return a post-event snapshot from LAMMPS to ParSplice for it to archive in its database of energy basins.

4.3.3. Client/server coupling

Another mode of coupling two (or more) simulation codes is the client/server model where a client code sends a message (data) to a server code, and the server responds with a message, looping until the coupled calculation is complete.

Three useful attributes of the client/server model are first, that it does not require linking one code to the other (i.e. neither code needs to be a library); they can both be stand-alone executables. Second, the two codes can easily be run on different numbers of processors. Third, if a messaging protocol is defined to run a particular kind of coupled model, e.g. *ab initio* MD (AIMD) using quantum (QM) forces calculated by a QM code, then neither code needs to know the specific other code it is coupling with, e.g. LAMMPS can be coupled to any QM code which supports the protocol.

To enable LAMMPS to be used as either a client or server code in this manner, we originally wrote a lightweight, low-level, open-source client/server library called CSlib [66] and included it in the LAMMPS distribution. However it was recently replaced with the MolSSI Driver Interface (MDI) library [67,68] created by the Molecular Sciences Software Institute (MolSSI). MDI provides a higher-level of functionality for application codes like LAMMPS, enables easy definition of protocols useful for materials science (and biomolecular) codes, and is already supported by a growing number of codes. Note that the MDI library is not specific to MD or QM codes as we discuss here; it allows many kinds of scientific codes to be coupled to each other.

Using the MDI library and lingo, one driver code (the client) can communicate with one or more instances of one or more engine codes (servers). The driver and engines can be written in any language, e.g. C/C++, Fortran, or Python; each code links to the MDI library. Messages can be sent via MPI or sockets. The latter means that the driver and engine(s) can run on different machines, communicating via IP addresses and port numbers.

MDI support within LAMMPS for it to operate as an engine, was added via a new *mdi* command,[8] a child class of the command style listed in Table 1. When this command appears in an input script, LAMMPS enters a loop to wait on and read messages sent by the driver, which tell it to either return requested data or invoke a simulation, e.g. run dynamics or perform energy minimization. As an example, a Monte Carlo (MC) driver code could make a series of trial changes in atomic conformations and send them one at a time to LAMMPS. LAMMPS evaluates energies for the old and new conformations, including optionally performing a short dynamics run to relax the new system. It then sends the results back to the MC driver so it can accept or reject the change according to the Metropolis criterion.

LAMMPS can also act as an MDI driver and send messages to one or more servers. For example, to perform AIMD, LAMMPS sends atom coordinates to a QM code once per timestep and the QM code returns forces on each atom. This is done in the middle of each timestep, at the point where LAMMPS would normally compute forces via a classical interatomic potential. MDI already has support for about a dozen QM codes to perform AIMD in this manner, though currently most do not support MDI directly but only indirectly via a QCEngine wrapper which communicates with the QM codes via files [69].

Similarly, if LAMMPS is used to perform a nudged elastic band (NEB) calculation [70], e.g. to compute the energy barrier for a diffusive hop or other conformational change, it can use MDI to communicate with multiple instances of a QM engine. Each engine calculates quantum-accurate forces within a single replica of the system. See section 6.6 for a description of how the overall NEB calculation is performed in a multi-replica context.

5. Performance and scalability

LAMMPS implements versions of performance-critical kernels for different hardware, namely GPUs and multithreaded CPUs. This is described in Section 5.1, along with performance and scalability benchmarks for different models. Section 5.2 explains how a variety of operations in LAMMPS are implemented scalably via rendezvous algorithms, while Section 5.3 describes how the recursive-coordinate bisectioning (RCB) algorithm is used for load-balancing by LAMMPS. A scalable visualization method useful for producing on-the-fly images of simulated systems is described in Section 5.4.

5.1. Performance acceleration options

The LAMMPS design goal of being a flexible code that is easy to understand and extend can be at odds with the challenge of providing optimal performance for a specific model on a variety of different hardware. This is compounded by the fact that LAMMPS supports over 200 *pair* styles (forces between particles), as well as hundreds of *fix* and *compute* styles, any of which may be used for simulation of a specific model and which can vary markedly in their computational cost. For example, the number of neighbor particles contributing to short-range forces may be less than five for a model of finite-size particles with contact forces, or greater than 1000 for models with long cutoffs. The arithmetic intensity of individual force calculations can also vary dramatically between models (up to 1000× on a per-atom per-timestep basis) and users can combine multiple models into a single hybrid model. Users may have access to CPU-based machines with sophisticated data caches and fast scalar computation or to GPUs that require a high degree of simultaneous multithreading (SMT) and vectorized computation for high performance.

All these factors can impact the best strategy for exploiting fine-grain parallelism at the single CPU or GPU (node) level in modern systems, separate from the MPI-parallel spatial decomposition strategies described in Section 3. Additionally, the optimal choice of programming model and language can depend on vendor support; many developers consider low-level direct programming languages (such as CUDA for NVIDIA GPUs) to be essential for performance on some hardware. Finally, LAMMPS uses deterministic algorithms which can be parameterized to produce identical results from run to run; this has advantages for debugging and identifying issues on machines with new hardware and software. However, the best performance on some hardware may require atomic operations resulting in non-deterministic algorithms.

Our current solution for these issues is to leverage the *package* and *style* capabilities supported by LAMMPS for optimizations that would otherwise decrease the simplicity, readability, and extensibility of the original code for a large user base, many of whom are domain scientists, not computer or even computational scientists. As explained below there are four accelerator packages (OPENMP, INTEL, GPU, KOKKOS) which include optimized versions of various LAMMPS style classes and can therefore coexist in the same build. Typically each derived style overrides the main computational kernel in the unoptimized style and inherits the remaining methods,

[8] Taylor Barnes (MolSSI), the developer of the MDI library, implemented engine support for MDI within LAMMPS.

e.g. each accelerated *pair* style provides a new *compute()* method (see Section 4.2.2).

Each package defines a suffix for naming its styles: the original, unoptimized Lennard-Jones potential is *lj/cut*; the four accelerated variants are *lj/cut/omp*, *lj/cut/intel*, *lj/cut/gpu*, and *lj/cut/kk*. A command-line suffix switch, e.g. "-sf intel", triggers LAMMPS to use any accelerated style variant (from the INTEL package in this case) that is available without needing to change the input script. Likewise another command-line switch can be specified to control settings for a particular accelerator package, e.g. the number of OpenMP threads or number of GPUs to use.

While the accelerator packages and their styles aim to provide good performance across a range of simulation parameters (impacting some optimization choices), they also allow use of alternative algorithms, data layouts, floating point precision modes, and programming languages, to achieve better performance on specific hardware. Of course, this approach is not without disadvantages. It requires multiple variants of LAMMPS styles to be maintained when changes are needed. Likewise some higher-level optimizations are not easily implemented at the level of individual style classes.

The most important algorithmic changes introduced by the accelerator packages affect how LAMMPS pair styles compute forces between particles, in order to mediate write conflicts on particle force arrays introduced by thread-level parallelization. One of three approaches is used: (1) half neighbor lists are used and each thread has its own copy of the force array (data replication); the copies are combined after force computations, (2) half neighbor lists are used but all force updates are done with atomic operations, or (3) full neighbor lists are used which avoid any write conflicts at the cost of 2× more computation for pairwise potentials (larger for manybody potentials).

Generally speaking, (1) is best on architectures with low thread counts (e.g. CPUs, not GPUs), while (2) is best for compute-intensive force fields or when hardware floating point atomic support is available. Using either (1) or (3) has the advantage that results are deterministic. More specifically, the OPENMP and INTEL packages use approach (1), the GPU package uses (3), while the KOKKOS package supports all three and can switch transparently between (1) and (2) through a data structure designed especially for this purpose and available in the Kokkos library. Similar write-conflict issues occur in charge discretization algorithms for long range Coulombic solvers. Atomic operations and/or data duplication approaches are also used by the accelerator packages for this computation.

The four accelerator packages are described in more detail in the following bulleted list.

- The OPENMP package adds loop-level shared-memory parallelism to the algorithms discussed in Section 3 to enable multithreading on CPUs of many LAMMPS styles using the OpenMP standard. For inter-particle force calculations this includes privatization of the force arrays to avoid memory conflicts. Because spatial decomposition at the MPI level is efficient in use of shared data and increasing the number MPI tasks also improves data locality, it is typically still beneficial with this package to use a high number of MPI tasks on each CPU. However, the OPENMP package can provide a significant performance boost when it is used (1) for SMT within CPU cores that support it (i.e. hardware multithreading), (2) to decrease the number of MPI tasks involved in 3d FFTs for simulations using the long-range Coulombics PPPM method, or (3) to reduce the number of MPI tasks when there are either few atoms per task (which can reduce the cost of ghost atom communication) or when the particle count per MPI task is poorly load balanced, since multithreading effectively balances over particles directly instead of indirectly as a spatial decomposition does. The OPENMP package uses deterministic algorithms.
- The INTEL package exploits thread parallelism ideas originally developed in the OPENMP package with a heavy emphasis on optimizations for SIMD vector computation [71]. This includes options for single, mixed, and double floating-point precision modes, algorithm and data-layout changes to improve SIMD performance, support for SIMD-optimized random number generation, and improved support for runtime selection of how Newton's 3rd law (equal and opposite forces) is exploited in its algorithms. It also provides two methods for parallelization via SMT. In the first, loop-based parallelism is used selectively where instruction/memory latencies are exposed, increasing the instruction throughput on each core. In the second, task-based parallelism is used to perform parts of the long-range PPPM calculation on one thread while other threads calculate other components of the force. Although the INTEL package was optimized for Intel processors, the source code exploits directive-based vectorization for compiler-generated vector code and can thus also provide improved performance on CPUs from other vendors. Likewise, the package benefits significantly from Intel compilers but is compatible with non-Intel compilers. The INTEL package uses deterministic algorithms.
- The GPU package was initially written for NVIDIA and other GPUs which support either the CUDA or OpenCL programming languages and was also designed to support hybrid (CPU+GPU) nodes [72–74]. Recently, a ROCm HIP back-end was added to support AMD GPUs [75]. The package uses the Geryon library to allow the same high-level source code to compile with any of these back-ends. Optimized LAMMPS styles in the GPU package are limited to the *pair*, *neighbor*, and *kspace* styles, where the compute-intensive portion runs on the GPU and per-atom data is transferred between the CPU and GPU each time step. Remaining computations on the CPU can be overlapped with the GPU via task-based parallelism (e.g. performing bonded computations on the CPU while *pair* calculations are performed on the GPU). Additional performance gains can often be achieved by parallelizing CPU computational tasks, either by use of the OPENMP package or by having multiple MPI tasks share a single GPU. The GPU package also supports single, mixed, and double precision floating-point and uses deterministic algorithms.
- The KOKKOS package uses the Kokkos performance portability library [18,19,76]. Compute-intensive methods of KOKKOS-optimized styles are coded in a templated C++ manner using abstractions that allow the same code to be converted to different back-end programming models (e.g. OpenMP, CUDA, HIP, and SYCL). Likewise, multi-dimensional Kokkos arrays are abstracted in the C++ code, which allows them to be converted at compile time to different data layouts by the back-ends to optimize memory access patterns for different hardware. When all styles invoked within a timestep are supported, Kokkos can be used in "reverse-offload" mode where all per-atom data remains GPU-resident for multiple time steps before any of it is transferred back to the CPU (e.g. for I/O). The KOKKOS package also supports GPU-aware MPI implementations for remote direct memory access (using pointers in fast GPU memory directly in MPI calls). On heterogeneous systems (CPUs+GPUs) Kokkos supports running styles on either the CPU or GPU though this may require data transfer each step; each style is templated on the device type (host or accelerator). Kokkos sometimes enforces thread safety through atomics rather than data duplication which means its algorithms are not always deterministic.

The degree of speed-up provided by the accelerator packages depends strongly on which material model is used and the per-node atom count for the system being simulated. However, as general rules of thumb, (1) the OPENMP package can improve performance via SMT within each core of a CPU, (2) the INTEL package can provide significantly better performance on Intel CPUs, (3) the GPU package can perform best when mixed-precision is sufficiently accurate or portions of the per-timestep computation must be run on the CPU (no Kokkos-enabled style is available) or long-range Coulombics (PPPM) is used, and (4) the Kokkos package can perform best when all the per-timestep computations have support for execution on the GPU.

Fig. 6 shows performance of a standard Lennard-Jones model on both a CPU cluster and a GPU-based supercomputer. This is for a liquid state point (reduced density $\rho^* = 0.8442$) and a cutoff of 2.5σ for $\sim$55 neighbors/atom, reneighboring every 20 timesteps. This benchmark has been used to test MD performance on a variety of machines over many decades [5,77]. The CPU data is from a Sandia National Laboratories cluster with Intel Skylake Xeon Gold 6140 dual-socket CPUs on each node (36 cores or MPI tasks/node, with optional 2× hyperthreading) and an Intel Omni-Path network. The GPU data is from the Lawrence Livermore National Laboratory (LLNL) Lassen supercomputer with an IBM Power 9 dual-socket CPU (44 cores), 4 NVIDIA Tesla V100-SXM2-16GB GPUs on each node, and a dual-rail Mellanox EDR Infiniband network.[9] In all but the lower-right plot a single GPU/node was used to illustrate performance on more common hardware configurations. In all the plots of both Figs. 6 and 7, the following abbreviations for atom counts are used for the x-axes and legends: K = 1000 and M = 1024*1000. For the y-axes M = 10^6. All benchmarking runs were performed in double precision.

The upper-left plot shows single node performance as a function of system size. The CPU curve is the performance of unaccelerated LAMMPS. The optimized CPU curve is for the INTEL package using up to 2 OpenMP threads per core (SMT). The GPU curve has data points from either the GPU or KOKKOS packages, whichever ran faster for a particular problem size. In this plot, the GPU package was faster for the smallest problem sizes, KOKKOS was faster for the rest.

The y-axis is millions of atom-steps per second. Thus for a small simulation with 4K atoms, the INTEL package runs at $\sim$15000 timesteps/s. For a large simulation with 4M atoms, the Kokkos package runs at $\sim$90 steps/s. Plotting the performance in this manner means that ideal scaling on any hardware would be a horizontal line. The unoptimized CPU curve is closest to ideal, except for very small system sizes. The GPU curve shows the strongest dependence of performance on the number of atoms.

The upper-right and lower-left plots show performance as a function of node count for strong- and weak-scaled simulations. In these plots (and the lower-right one) the y-axis is millions of atom-steps per second per node so that a horizontal line again represents ideal scaling. For each curve in the strong-scaling plot, the atom count is the same on any number of nodes. In the weak-scaling plot the size of the simulation grows with the node count; data for 512K and 4M atoms/node are shown. The rightmost open-symbol points on the 4M atom/node weak-scaling curves are thus for simulations of a billion atoms (1024*1024*1000) on 256 nodes.

In the strong-scaling plot (upper-right), the CPU curves are nearly flat; performance for the smaller system begins to decrease slightly at 64 nodes where the atom count/MPI task is $\sim$1800. For the GPU curves, performance falls off more dramatically with increasing node count, particularly for the smaller system. This is primarily because the atom count/GPU is halved each time the node count doubles, and thus performance per GPU decreases as in the single node plot (upper left). In the weak-scaling plot (lower-left) the CPU curves are nearly flat; the GPU performance now decreases more slowly with increasing node count. Unevenness in the curves is due to the mix of results from both the GPU and KOKKOS packages as well as the GPU package launching kernels asynchronously which overlap.

The lower-right plot shows weak-scaling performance for the GPU-based Lassen machine using 1, 2, or 4 GPUs per node, for systems with 4M atoms/node or 4M, 2M, 1M per GPU. All of these runs were made with the KOKKOS package to reduce variation in the results. At any node count the speed-up for this system is 1.7-1.9× on 2 GPUs and 2.5-2.9× on 4 GPUs. Again the largest run was for 1 billion atoms on 256 nodes.

In Fig. 7 weak-scaling performance for two different size systems (atoms/node or GPU) is plotted for several manybody interatomic potentials (simulating materials appropriate to the potential), all of which are briefly described and cited later in Sections 6.1 and 6.3. Also shown is data for a widely-used bead-spring model for linear polymer chains. It uses a short-range (repulsive only) Lennard-Jones pairwise interactions with finite-extensible nonlinear elastic (FENE) bonds and is described in Section 6.2. As in most of Fig. 6, the GPU runs in these plots used a single V100 GPU per node of the Lassen supercomputer.

On CPUs (solid symbols) the per-atom computational cost of the four potentials varies by $\sim$1000x, which is representative of the range of expense for materials modeling in MD; see Fig. 10 in Section 6.1. The speed-up on a V100 GPU (open symbols) versus the dual-socket Intel Skylake CPU is from $\sim$2-12x, except for the cheapest model (bead-spring polymer) which is no faster for the small system and $\sim$2x faster for the large. The smaller bead-spring system fits in cache on the CPUs and the short cutoff means there is relatively little computation/atom. For the smaller systems (left plot) a decrease in parallel efficiency from one to 256 nodes is more evident, e.g. for the bead-spring model running on GPUs. For the larger systems (right plot), both the CPU and GPU curves are nearly horizontal.

All available accelerator packages were benchmarked for each model, with the fastest results shown in Fig. 7. On CPUs, the INTEL package was fastest for Tersoff, the OPENMP generally fastest for the bead-spring polymer model, and the OPENMP or KOKKOS packages for ReaxFF. There is not yet a CPU accelerator option for SNAP in LAMMPS. On GPUs, the KOKKOS package was fastest for Tersoff, and KOKKOS was also used for ReaxFF and SNAP as there is not yet a GPU package option for these models. Both KOKKOS and GPU packages were used for various bead-spring data points (whichever was faster).

We also highlight four production-scale simulations which illustrate the ability of LAMMPS to run very large systems scalably on large supercomputers:

- The GPU package was used to run a coarse-grained liquid-crystal model (Gay-Berne potential [78]) on the full Titan machine ($\sim$16000 nodes) at Oak Ridge National Laboratory to study thin film stability and rupture dynamics with up to 26M ellipsoidal particles [79]. It was one of the first GPU-accelerated materials simulations to be run on a petascale supercomputer.
- The embedded atom method (EAM) manybody potential was used to model dislocation-mediated plasticity in bulk tantalum on the Sequoia Blue Gene Q machine at Lawrence Livermore National Laboratory [80]. The largest simulations (not discussed in the paper) were of 2B (billion) atoms run for 0.5B

[9] Though LAMMPS supports it, all the GPU runs on Lassen were performed with the GPU-direct (GDR) driver-level communication mode disabled (while still using CUDA-aware MPI). When enabled (in CUDA 10.1) GDR produced uneven performance, sometimes much slower than runs with GDR disabled.

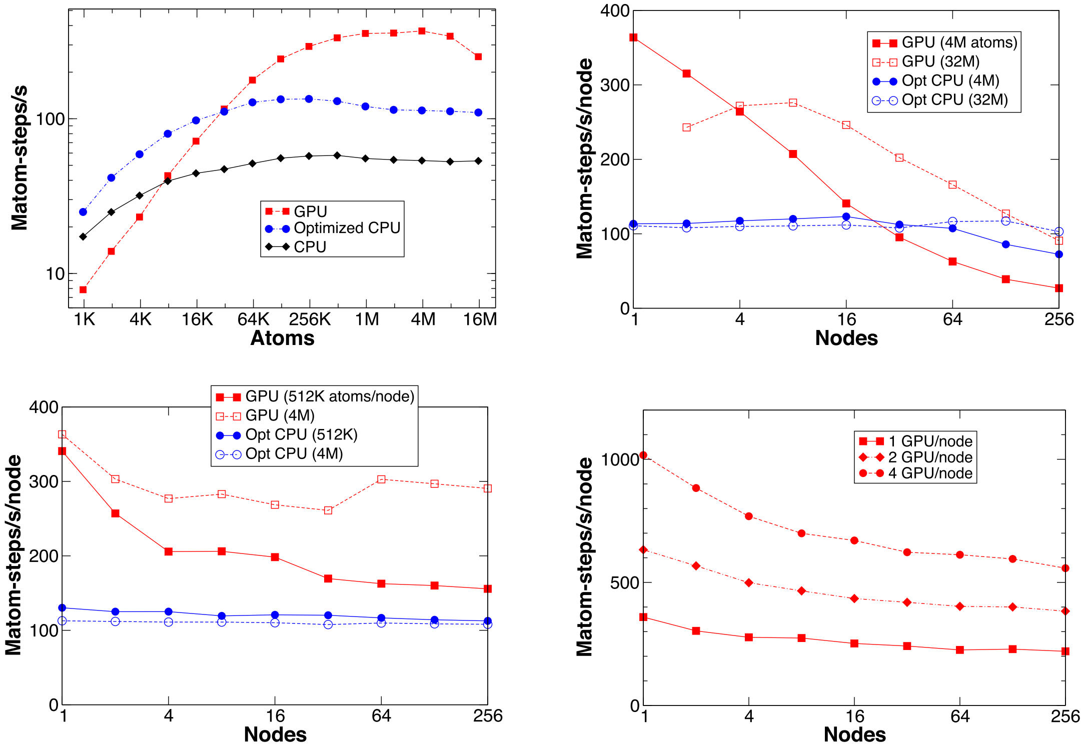

Fig. 6. Performance of a Lennard-Jones liquid benchmark on one or more nodes of a CPU cluster (Intel Skylake CPUs, 36 MPI tasks/node) and a GPU-based supercomputer (LLNL Lassen, with 4 NVIDIA V100 GPUs/node). (Upper left) Single-node runs on a single dual-socket CPU or single GPU. (Upper right) Strong-scaling runs of systems with 4M (million) and 32M atoms; one GPU/node was used. (Lower-left) Weak-scaling runs with 512K (thousand) and 4M atoms/node; one GPU/node was used. (Lower-right) Weak-scaling runs using different numbers of GPUs/node with 4M atoms/node. In the upper-left plot performance is in Matom-steps/s; in the other three plots it is in Matom-steps/s/node. Thus in all the plots a horizontal line would represent ideal scalability.

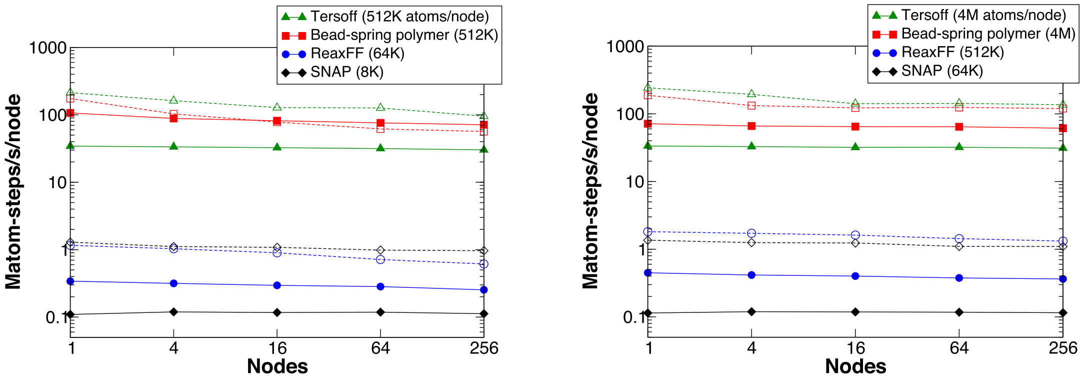

Fig. 7. Weak-scaling performance in millions of atom-steps/s/node for three manybody potentials (Tersoff, ReaxFF, SNAP) and a coarse-grained bead-spring polymer model with great variation in their per-atom computational expense. The solid symbols and lines are on a CPU-based cluster, the open symbols and dashed lines are on a GPU-based supercomputer; this is same hardware used in Fig. 6. The left plot is for smaller models (per node or GPU); the right plot is for 8× larger models. As in Fig. 6, a horizontal line would represent ideal scalability in either case.

timesteps (4 μs) on 32K nodes (524K cores) or 1/3 of the machine. The size of the system and strain rate imposed enabled observation of many tens of thousands of interacting dislocations and gave new insights into the atomic mechanisms underlying plastic response. At the time (2017) this represented one of the largest MD simulations ever performed (10^{18} atom-steps).

- The KOKKOS package with its OpenMP back-end was used for ReaxFF models of shock compression effects on energetic materials. The image in Fig. 8 is from a simulation of a 20M atom sample of a hexanitrostilbene (HNS) molecular crystal with randomly distributed voids representing a realistic pore microstructure. The uncompressed sample dimensions were 0.5 μm × 0.25 μm×3 nm and the impact velocity was 1 km/s, requiring a small timestep of 0.1 fs. Dynamic load balancing (see Section 5.3) was used to maintain a roughly equal number of atoms per processor, even in the presence of large spatial density variations around the shock front. Similar simulations were also run on both the Intel Knights Landing (KNL) Xeon Phi and Intel Haswell partitions of the Trinity machine at Los Alamos National Laboratory with up to 130M atoms with over 100,000 processors. These simulations provided new insight into shockwave energy localization due to pore collapse in realistic microstructures and enabled development of equation-of-state and material strength models for continuum hydrocode simulations of initiation in energetic materials [81].

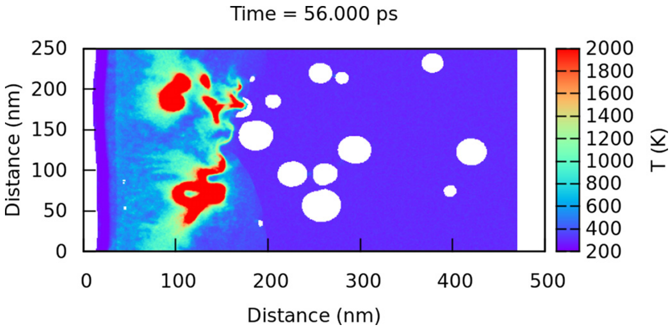

Fig. 8. Temperature map midway through a 20M atom simulation of shocked HNS molecular crystal containing a random distribution of voids using the ReaxFF reactive interatomic potential [81]. The uncompressed sample dimensions were 0.5 μm × 0.25 μm × 3 nm. The red regions indicate hot-spots caused by localization of the shockwave energy around collapsed voids. The KOKKOS package and dynamic load balancing were used to efficiently run on 500 nodes (18,000 Intel Broadwell cores) of a Sandia supercomputer. The simulation rate was about 1.4 timestep/s; 1.7M timesteps (0.17 ns) were required for the shockwave to sweep through the entire sample. Five different microstructures were simulated, requiring a total of 30M core-hours of compute time.

- The KOKKOS package with its CUDA backend was used for SNAP models of a variety of material systems, including plasma-material interactions for fusion energy [82] and carbon at pressures and densities of exoplanet cores [83]. Using 4,650 nodes (27,900 NVIDIA V100 GPUs) of the Summit machine at Oak Ridge National Laboratory, it was possible to simulate 20 billion carbon atoms at a rate of 5.92 Matom-steps/s/node [83]. On 256 nodes of the same machine, it is possible to run a 4 million atom tungsten/beryllium/helium system with a parallel efficiency of 50%, achieving a simulation rate of 30 ns/day, which is comparable to the maximum simulation rate achievable using a lower accuracy potential such as the embedded atom method (EAM).[10]

5.2. Scalability of other algorithms

As implemented in LAMMPS, the fundamental MD algorithms discussed in Section 3 all scale as $\mathcal{O}(N/P)$ in the number of atoms N and processor count P. An exception is the FFTs required for long-range interactions. Their computation scales as $\mathcal{O}(N\log(N)/P)$ and their communication costs can be performance bottlenecks for sufficiently large P.

When running very large simulations on large processor counts, we have also tried to ensure other more incidental operations in LAMMPS also scale as $\mathcal{O}(N/P)$ to avoid scalability issues. An example is the "lookup" of atom IDs to find partner atoms in a bond, angle, or dihedral computation for a molecular model. This lookup is done each time re-neighboring is performed and atoms migrate to new owning processors, since that alters each processor's list of owned atoms. LAMMPS uses a hash table local to each processor to map atom IDs (32- or 64-bit) to a local index, which enables an atom ID to be found in constant $O(1)$ time independent of the N/P number of (owned plus ghost) atoms stored by each processor.

As parallel supercomputers became larger in the last decade (tens of thousands of nodes, more than a million MPI tasks), we discovered other non-scalable operations in the code. Often these operations are performed only occasionally or at setup time for a large simulation. On smaller problems and node counts, their cost is not an issue. But at scale, they became bottlenecks. We have recently refactored several of these algorithms to make them much faster. They can now operate efficiently at the scale of billions of atoms and hundreds of thousands or millions of MPI tasks.

Examples include:

- Walking the bond-topology within molecules to identify neighbors which are excluded from (or weighted in) pairwise interactions for biomolecular force fields.
- Initializing the properties of large collections of small rigid bodies, especially for polydisperse systems where each body may be different in size, shape, or numbers of constituent atoms. This requires identifying the atom nearest the geometric center of the body so it can store information about the body, and insuring all other atoms in the body know the ID of that central atom.
- Creating huge numbers of atoms either by populating lattice sites or replicating a smaller system many times.
- Enabling a large simulation to be "rerun" by reading a large number of snapshots one at a time and computing new diagnostic properties of the system. Each snapshot may have been written to 100s or 1000s of files, and the new simulation may be running on a different processor count than the original.

To perform these operations efficiently at scale we now use so-called *rendezvous* algorithms. As described in [84], they are useful when processors do not know who to send data to or receive data from in order to distribute independent computations evenly across a large parallel machine. They define an intermediate (rendezvous) decomposition of the data, which allows both sending or receiving processors to identify the appropriate intermediate rendezvous processor. Two stages of all-to-all communication are used to send data to and from the rendezvous decomposition, which can leverage the large bisection bandwidth of large parallel machines. Application of rendezvous algorithms to each of the LAMMPS operations in the list above are described in [84].

In the case of the "rerun" operation, having an efficient way to re-process data from a previous simulation enabled rapid analysis of thousands of snapshots from the 2B (billion) atom simulation of dislocation formation and interactions in a deformed metal sample [80] mentioned in Section 5.1. The post-analysis was run on several thousand nodes of a large IBM BG/Q machine (tens of thousands of cores); the initial simulations were run on 32K nodes (half a million cores) of the same machine.

5.3. Load balancing

As mentioned in Section 3.1 LAMMPS uses spatial decomposition for distributed-memory MPI parallelism. The simulation box is partitioned into non-overlapping subdomains which fill the box, one per MPI task. MD models at the atomic scale do not typically involve large variations in atom density; a regular partitioning of the simulation box into equal size subdomains is often adequate for load-balancing purposes. However, as illustrated in Fig. 1 of Section 3.1, for coarse-grained models at the meso or continuum

[10] Strong scaling this 4M atom simulation with EAM provides little speedup beyond one node (6 GPUs).

scale, particle densities can vary widely both spatially and in time. In the SPH simulation shown on the right of that figure, there are large empty regions in the system, which would result in poor load-balancing and parallel efficiency with the default partitioning.

To load-balance such models, LAMMPS uses a recursive coordinate bisectioning (RCB) algorithm [85], which allows the size and shape of each subdomain to be adjusted freely with the constraint that the faces (edges in 2d) of all subdomains remain parallel to the orthogonal or triclinic simulation box faces. Rebalancing can be performed once when a relatively static simulation geometry is initialized, or periodically as a simulation runs. An imbalance threshold can be defined which triggers rebalancing only when necessary.

In the upper-right sub-figure of Fig. 1 an RCB partitioning for 32 processors is shown by the solid black lines in the figure; in this case each processor's subdomain contains the same number of particles. More generally, particles can be weighted; each processor is assigned a collection of particles that sum to (nearly) the same weight. This can be useful for hybrid models (e.g. water on a metal surface), when the cost of computing the interatomic potential for metal atoms is more expensive than for water molecules. A weighting can also be assigned based on CPU time (force computation plus neighbor list building) accumulated by each processor since the previous rebalance. Processors which have accumulated more time than others assign proportionally more weight to each of their currently owned particles.

As its name implies, the RCB algorithm works recursively; the partitioning is computed in $log_2 P$ stages where P is the number of processors. At the first stage, a plane (3d) or line (2d) is positioned to split the particle weight for the entire simulation box equally into two halves in one dimension of the box. The position of the plane is computed by iterative bisectioning (median search). Half the processors are assigned to one of the new subdomains, the other half to the other subdomain. This procedure is repeated recursively in subsequent stages until each subdomain is assigned to a single processor. If at one stage the number of processors assigned to a subdomain is odd (say seven), then at the next stage the particle weight and processor counts are divided proportionally (3/7 and 4/7) between the two new subdomains. At each stage, each processor communicates once per median search iteration to exchange information with a partner processor in the other half of its new partition. Likewise *MPI_Allreduce()* operations which check for median convergence each iteration are performed only within the subset of processors assigned to the partition being split.

One advantage of using the RCB algorithm for load-balancing in LAMMPS is that the faces (edges in 2d) of the final processor subdomains are parallel to the faces of the simulation box, as illustrated on the right side of Fig. 1. This means only modest changes to the code were needed to use subdomains generated by RCB. Alternative load-balancing strategies produce subdomains with more complex shapes and boundaries [86,87].

A second advantage is that the geometry of the entire partitioning can be stored in a P-1 length vector which indexes a binary tree of split plane positions and dimensions. Even for a million-processor simulation, this data structure is small enough that each processor can own a copy. This is useful in two contexts. When a particle moves outside a processor subdomain, the new owning processor can be identified from the particle coordinates by traversing the tree in $log_2 P$ steps. A similar traversal can be used to generate the list of all nearby processors overlapping the ghost subdomain of one processor (dashed box in the lower-right image of Fig. 1). These are the processors needed to perform the ghost atom exchanges illustrated on the right side of Fig. 2. Note that to minimize communication of ghost atoms, the subdomains generated by RCB should be as square (2d) or cubic (3d) as possible. At each stage, this depends on the distribution of particles in the current partition subdomain. LAMMPS computes trial split planes in the x, y, and z directions to check which produces the best result.

Table 2
Scaling of the relative computational cost for the RCB load-balancing algorithm with number of processors and problem size. Relative cost is the time for one rebalance operation divided by the time for one MD timestep. Simulations of a Lennard-Jones liquid were run on 1 to 1024 nodes of the ALCF Theta machine (64 MPI tasks/node) for 3 weak-scaled system sizes (32K, 256K, 2M atoms/node). The largest system benchmarked was thus 2.1B atoms runs on 65,536 processors (cores). The parenthesized value in the last column is the actual time (in s) for a single rebalance operation.

Atoms/node	– Nodes –					1024 (time)
	1	4	16	64	256	
32K	16	42	70	108	147	242 (0.62)
256K	5.3	9.8	16	25	31	55 (0.76)
2M	0.15	1.1	2.0	5.3	7.3	12 (1.3)

A final advantage is that the RCB algorithm is fast and parallel to compute, as well as reasonably scalable to very large numbers of processors. The data in Table 2 show performance of RCB on up to 1024 nodes (65,536 processors); we have used the same algorithm in another particle code (SPARTA for Direct Simulation Monte Carlo (DSMC) low-density gas modeling [88]) on the LLNL Sequoia Blue Gene Q machine mentioned previously, running on more than a million MPI tasks.

To measure RCB performance, short simulations of a uniformly dense Lennard-Jones liquid were performed at a reduced density and temperature of 0.84 and 0.73 respectively with a pairwise cutoff of 2.5σ. Though no load-balancing is needed to run this equilibrated model efficiently, load-balancing was forced to occur every 100 timesteps. Small density fluctuations at the current timestep versus the previous timestep induced split planes in a different sequence of dimensions. This meant that after rebalancing the majority of processors owned different subdomains, which induced a larger-than-usual volume of migrated atoms. For the smallest node counts, from 50 to 70% of atoms migrated to new processors at each rebalance; for the largest node counts, 90% migrated.

The tests were run on 1 to 1024 nodes of the Argonne National Laboratory ALCF Theta machine. The nodes are Intel KNL CPUs, each with 64 physical cores. These were weak-scaling tests; the problem size increased proportional to node count. Three different per-node sizes were tested, with 32K, 256K, and 2M atoms/node (K = 1000, M = 1024*1000 in this case). Thus the largest system run was ~2.1B (B = billion = 1024*1024*1000) atoms on 1024 nodes.

Table 2 shows the relative cost of RCB rebalancing for each simulation reported in timesteps, i.e. the time for one rebalancing operation (RCB plus atom migration) divided by the time for one MD timestep. One trend is the increase in relative balancing cost on larger node counts (across a row in the table). This is due in part to the $\mathcal{O}(\log P)$ scaling of the RCB algorithm. On one node six levels of recursion are needed (64 processors); on 1024 nodes 16 levels are needed. Another trend is the reduction in relative cost for larger per-node atom counts (down a column). This is primarily due to the increased per-timestep cost while the RCB algorithm communication costs remain fixed.

It is also worth noting that the Lennard-Jones model used in this benchmark is computationally cheap. More complex coarse-grained models can be 10× to 100× more expensive on a per-atom per-timestep basis. In which case the relative cost of rebalancing would decrease by 10× or 100×.

Fig. 9 shows three simulation snapshots of models which require load balancing to run efficiently in parallel. On the left is a coarse-grained model of a planar membrane interacting with a large spherical virus-like particle. With the RCB load-balancing algorithm this 3.6M (million) particle model ran scalably on up to 256 nodes (8192 cores) of the Cray Blue Waters supercomputer at

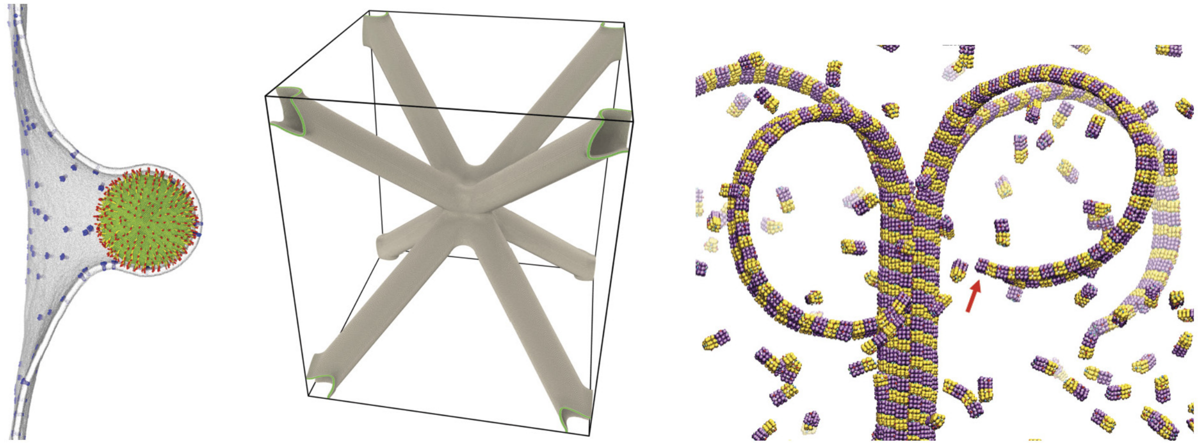

Fig. 9. Three models where load balancing is required for good parallel performance. (Left) Coarse-grained model of a 100 nm virus-like particle budding through interaction with a cell membrane. Transmembrane proteins are shown in blue. (Middle) A hollow metal strut (∼200 nm on a side) used to measure the mechanical strength and stiffness of ultra-lightweight nanoengineered materials. (Right) Coarse-grained simulation of catastrophic depolymerization of $\alpha\beta$-tubulin. The yellow and purple particle wedges represent the α and β monomer subunits [89]. Images courtesy of John Grime and Gregory Voth, University of Chicago (left), Alexander Stukowski, Technische Universität Darmstadt (middle) and Jonathan Bollinger and Mark Stevens, Sandia National Laboratories (right).

NCSA; it was still 50% efficient with 440 atoms/core. In the middle is a model of a metal strut of Ni atoms; the cylindrical projections can be hollow or solid. A 21M atom solid strut model was run on 1024 nodes (16K cores) of an IBM Blue Gene Q machine. Without load-balancing some processors have 18× more atoms than the average atoms/core. With load-balancing it ran 13× faster. On the right is a snapshot from a coarse-grained molecular dynamics simulation of catastrophic depolymerization of $\alpha\beta$-tubulin due to hydrolysis-driven shape changes in the subunits [89]. The initially intact microtubule was 320 dimers long, each dimer composed of an α subunit and a β subunit. The red arrow indicates the dynamic tip of the ram's horn where the dimers detach from the polymer.

5.4. On-the-fly visualization

LAMMPS provides a *dump* style (see Table 1) to create on-the-fly snapshots (JPEG or PNG files, or a movie that concatenates them) as a simulation runs. Like other dump styles a subset of particles can be selected for output. Image attributes can also be specified, such as the view direction and a zoom factor. This can be useful for debugging, e.g. to verify a model was defined and initialized correctly. All four of the simulation images in Fig. 1 were created in this manner.

This capability can also be used to reduce the volume of output data for large simulations run for long timescales. Obviously, it does not provide the interactivity of a post-processing visualization tool like OVITO [90] or VMD [91], but it can create images of very large systems quickly by leveraging the same parallelism used to run the simulation. The low-level code used in LAMMPS for this operation has also been used by the SPARTA Direct Simulation Monte Carlo (DSMC) particle code [88] to create images of tens of billions of gas particles undergoing turbulent flow; see [92] for modeling of a 2d Richtmyer-Meshkov instability with images of up to 100B particles, and Figure 8 of [88] for images of 20B particles in a model of a 3d Rayleigh-Taylor instability. The implementation for SPARTA, including visualization of triangulated objects embedded in the flow, was described in [88]; here we briefly explain how it works in LAMMPS for MD.

Snapshot images are created in parallel using the methodology outlined in Kadau et al. [93]. Each processor renders the particles it owns into an empty JPEG (or PNG) buffer the size of the desired image (e.g., 1024 × 1024). Particles are rendered pixel-by-pixel as small shaded spheres; bonds between particles are optionally rendered as small-diameter cylinders.[11] Either can be colored using any per-atom property that LAMMPS stores for the system or by a per-atom fix, compute, or variable specified in the input script, e.g. the result of a diagnostic computation. The first time a pixel is drawn into the image buffer, a depth value for that pixel is also stored in a companion buffer, which is the perpendicular distance from that pixel to the plane of the image. When the same pixel is drawn again (likely with a different color), its depth value is compared to the stored value. If the new pixel is in front of the old one, it replaces it in the JPEG buffer, and the stored depth is updated. If it is behind the old one, the new pixel is discarded.

Once each processor has rendered an image of its particles, the image buffers are pairwise merged into a final image in $\log_2(P)$ iterations, where $P =$ the number of processors. At each iteration, half the processors send their current image and depth buffers to partner processors which perform the merge; the number of participating processors is halved at each iteration. Pixels are compared one-by-one between the two images; the in-front pixels become the new merged image and their depth values are retained. At the last iteration a single processor performs the merge which produces the final image containing all the particles; it then writes the image to disk. This operation scales well to very large processor counts because the initial rendering is perfectly parallel (assuming equal numbers of particles per processor), the iteration count is logarithmic in P, and the per-processor communication per needed at each iteration is limited to the size of a single image.

6. Material models and methods

This section gives examples of the range of material models, methods, and other features available in LAMMPS which have been enabled by the flexible design principles described in Section 4. Many of the capabilities mentioned here were implemented by users who contributed their code to LAMMPS. Several are recent additions.

6.1. Manybody potentials

The extent to which a classical MD simulation matches the behavior of a real material is limited primarily by the accuracy of

[11] Nathan Fabian (Sandia National Laboratories) wrote the graphics rasterization primitive methods for use in LAMMPS.

the potential energy model used to calculate interaction forces between particles at the atomic or coarse-grained scale. LAMMPS includes ∼230 *pair style* models with different analytic forms. They vary widely in physical realism, code complexity, and computational cost. The majority of these compute pairwise interactions, but ∼45 are manybody potentials where energy terms in the potential involve clusters of three or more nearby atoms. For manybody models neighbor lists are used to enumerate unique combinations of multiple neighbor atoms. Force computation for a manybody potential often requires first calculating intermediate quantities for each atom, such as bond order or local density. Forward and reverse communication operations, as described in Section 3.2 may be required to complete the intermediate calculations.

Notable manybody potentials in LAMMPS include bond-order (Tersoff [94], REBO [95], AIREBO [96][12]) and variable charge (COMB [97,98], ReaxFF [99]) models. The latter two are manybody potentials designed to treat chemical reactions. They include terms in their potential energy equation for the effects of covalent bonding, bond-order corrections, hydrogen bonding, conjugated bond dispersion interactions, and Coulombic interactions, among others. They also include a charge equilibration (QEq) operation each timestep, which is discussed in Section 6.4. Recently, a growing variety of machine learning manybody potentials have also been added to LAMMPS; these are discussed separately in Section 6.3.

Fig. 10 illustrates how the computational cost of manybody potentials has grown over several decades. This timing data is from benchmark calculations of systems with a few tens of thousands of atoms for materials appropriate to the various potentials. Because all the potentials compute short-range interactions within a cutoff distance, the run time of an MD simulation grows linearly with atom count, and the per-atom per-timestep CPU cost on the y-axis is a good metric for comparison. Several of the manybody potentials (red data points) were mentioned above. The machine learning potentials (blue points) are discussed in Section 6.3. The other acronyms are for these potentials: embedded atom method (EAM) [100], modified EAM (MEAM) [101], embedded ion method (EIM) [102], bond-order potential (BOP) originally due to Pettifor and Oleinik [103] and later updated [104,105], and the electron force field (eFF) which treats electrons as finite-size variable-radius particles [106].

The upward trend in the figure is evidence that materials scientists have leveraged faster computers not simply to run older, cheaper models at larger scale, but also to develop more accurate and costly models. This increased complexity also reflects a shift in the role played by MD in materials modeling. Rather than providing only qualitative insight into the general behavior of materials, MD is increasingly asked to provide accurate quantitative representations of processes occurring in specific materials, including large changes in local structure, as well as chemical structure and composition.

LAMMPS also has a *pair kim* style and series of *kim* commands which effectively wrap the Open Knowledgebase of Interatomic Models (KIM) library developed and maintained by the KIM project [108,109]. These allow any KIM model to be used in a LAMMPS simulation. In the KIM terminology, a “model” means an interatomic potential plus a set of parameters specific to a particular material; the KIM library currently has ∼500 such models. The library also includes ∼20 potentials (analytic equations in this case) which LAMMPS does not natively implement.

It is worth noting that the connection between LAMMPS and OpenKIM is richer than just running a LAMMPS simulation with a KIM model. A LAMMPS input script can perform queries of the KIM online database to obtain material properties (e.g. a zero-stress lattice constant computed using the KIM model), or to obtain a list of models which meet a specified criterion (e.g. all Al/Cu alloy models) and then execute a loop to perform simulations with each of them. Several of these operations can interact in real-time with the KIM website to retrieve information. Likewise the OpenKIM project uses LAMMPS internally as a tool to perform calculations of material properties and archive the results on their website.

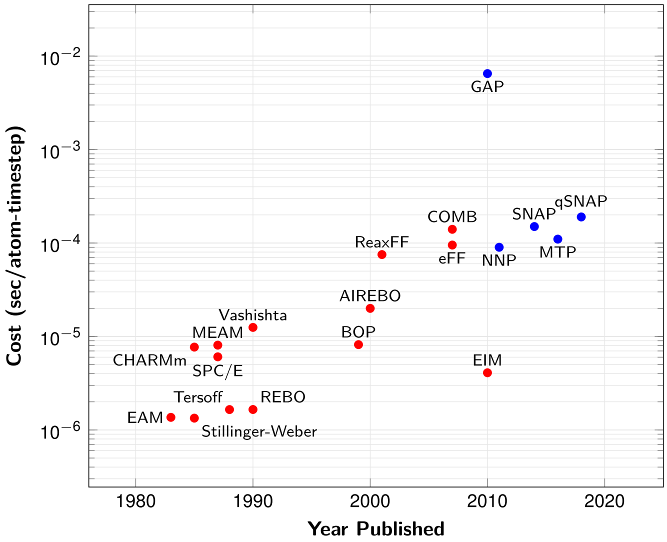

Fig. 10. Comparison of the computational cost on a per-atom, per-timestep basis of various manybody potentials (red) and several machine learning interatomic potentials (blue), as implemented in LAMMPS. The potentials are discussed in the text. The x axis is the year in which the potential was first published. The manybody potentials were run on a single Intel Broadwell core using the 21 July 2020 version of LAMMPS. The machine-learning potentials were run on a single Intel i7-6850k 3.6 GHz core using the 12 December 2018 version of LAMMPS [107].

6.2. Coarse-grained models

In order to reach longer length and time scales than is possible with all-atom MD, coarse-grained (CG) models are advantageous for many systems. This subsection highlights various CG models implemented in LAMMPS, extending in some cases to meso and continuum scale models.

6.2.1. Three kinds of CG models

(1) Idealized CG models of polymeric systems [110,111], surfactants and lipids [112–115], and DNA [116–118] are readily simulated with LAMMPS. Such models are simple by design, but include attributes that induce physical behaviors. For example, bead-spring polymer models define connectivity (bonds) and prohibit chain crossings by use of the finite extensible nonlinear elastic (FENE) bond potential [119]; both are essential attributes of all types of polymers. For lipids and surfactants, self-assembly is a key soft matter phenomenon, and can be induced by appropriate choice of repulsive and attractive potentials for different particle types. CG modeling of self-assembly processes has been one of the great successes of molecular dynamics simulations [115]. Many large-scale parallel simulations of such models have been performed using LAMMPS [120,121]. At much larger scales, models to simulate entire biological cells have been implemented in LAMMPS [122,123] using mesoscale capabilities described in the next subsection 6.2.2.

(2) In the last decade or so, higher-fidelity CG models have been developed which derive directly from atomistic simulation data and can incorporate chemical detail [124–127]. For many of these the ability to use pre-tabulated potentials via the *pair table* style in LAMMPS can be leveraged. For example, iterative

[12] Asegun Henry (MIT) implemented the *REBO* and *AIREBO* potentials in LAMMPS.

Boltzmann inversion (IBI) can derive CG potentials from the radial distribution statistics of an atomistic simulation [128]. The Votca package [129–131] implements IBI (and other methods) and can produce LAMMPS-compatible input script commands and tabulated potential files. LAMMPS itself has a *fix mscg* command[13] which wraps the multi-scale coarse-graining (MS-CG) library from Gregory Voth's group at the University of Chicago [132–134]. It can be used to post-process a series of all-atom simulation snapshots using the variational force-matching methods in MS-CG to produce a new CG potential, suitable for running further CG simulations with LAMMPS. Recent work by the Voth group used their MS-CG and other methods to develop a multiscale CG model of an entire SARS-CoV-2 virion with 165K (thousand) particles. It was simulated with LAMMPS for 10 million timesteps corresponding to 1 µs [135].

(3) Colloidal and similar systems may have µm-scale particles which are far too large to treat atomistically, and when solvated often require coarse-graining at two length scales (colloid and solvent) to create an effective CG model. The LAMMPS *pair colloid* and *resquared* styles treat finite-size spherical [136] and ellipsoidal [137] colloid particles conceptually as if filled with small Lennard-Jones (LJ) particles at a uniform density. The colloid/solvent and colloid/colloid pair potentials are the analytic forms obtained from integrating over the colloid particle volume(s) to calculate the total potential. The *pair yukawa/colloid* potential models Coulombic interactions between finite-size large particles screened by an electrolyte. As mentioned in Section 4.2.2, when *yukawa/colloid* is used in combination with *pair colloid*, it enables a model consistent with Derjaguin-Landau-Verwey-Overbeek (DLVO) theory [46,47].

Hydrodynamic interactions between colloidal particles arising from the presence of solvent which can be coarse-grained as LJ particles or modeled via multi-particle collision dynamics (MPCD), also known as stochastic rotation dynamics (SRD), where solvent particles do not interact directly with each other but only with solute particles [138,139]. MPCD is best for simulations where both thermal fluctuations and hydrodynamics are important. The solvent particle velocities are randomly rotated periodically within spatial bins to imbue the CG fluid with an effective viscosity. This model is implemented as a *fix srd* command within LAMMPS. Additional options for hydrodynamic models include (a) the Fast Lubrication Dynamics (FLD) approximation [140] to Stokesian dynamics, implemented as a *pair lubricate* command, (b) the LATBOLTZ package, which implements a lattice-Boltzmann model for a flowing fluid that interacts with CG particles or polymer chains [141], and (c) various options for dissipative particle dynamics (DPD), discussed below. Several of these methods are discussed and compared in [142].

6.2.2. *Four kinds of meso to continuum scale models*

At larger length scales particle-based mesoscale and continuum models are often MD-like in nature, enabling their implementation in LAMMPS. Many of the models listed here leverage the capability of Section 4.2.1 to add new atom styles that define additional per-particle attributes. It is worth noting that at the meso and continuum scales, particle models are typically dramatically more computationally expensive than their PDE-based counterparts solved on finite element (FE) meshes. However, particle methods are inherently meshless and their interaction models naturally encode non-local and nonlinear effects. As discussed below particles can thus be used to effectively model fluids or solids undergoing large deformations, including topological changes (e.g. crack formation). In such cases, particle models avoid complications with mesh entanglement and re-meshing associated with FE approaches.

(1) Several variants of dissipative particle dynamics (DPD) models have been implemented. These extend the original DPD formalism [143] for energy-conserving non-isothermal processes (eDPD) [144], for many-body interactions (mDPD, as compared to simple pairwise) for modeling vapor-liquid coexistence [145], or for including transport terms (tDPD) [146] to model advection-diffusion-reaction systems. Additionally, there is a DPD model for reactive energetic materials (explosives) [147] which can model the effect of material microstructure on shock response. Each particle stores a vector of species concentrations that represents its internal partially-reacted state. An ODE solver is invoked each timestep for each particle to evolve its state via a set of coupled rate equations.

(2) LAMMPS includes a variety of options for representing aspherical particles which can be used to model shape effects on packing and dynamics. As discussed in [148] (see Fig. 1 of that reference), this can be done in one of two ways. Individual particles can store parameters which define their shape and orientation when interacting with other particles. This can be as simple as ellipsoids with a stored quaternion which defines its current orientation. Or it can be as complex as what LAMMPS terms "body" particles which store their internal geometry: a polygon for 2d models or a polyhedron for 3d models described by polygonal faces connected by edges with corner points. Like other styles described in Section 4.2, body styles are extensible. In principle a new body style could be implemented to store a mesh which triangulates the surface of an arbitrary shaped particle, which could enable modeling of deformable particles. The second way is to define rigid bodies as collections of simpler particles as mentioned in Section 4.1. The interaction between two bodies is computed as the sum of pairwise inter-body interactions between their constituent particles. This kind of model allows for great freedom in defining irregular-shaped particles and inclusion of polydispersity effects.

(3) To model granular materials, LAMMPS has several discrete element models (DEMs) which compute frictional interactions between finite-size spherical particles.[14] The most general of these is the *pair granular* style which has options for various elastic, damped, and cohesive interactions, including several models for tangential, twisting, and rolling frictional forces [149]. As mentioned in Section 4.2.3, these models store the pairwise state (information from previous timesteps) for persistent contacts between a particle and each of its neighbors. Granular systems with large polydispersity can be modeled efficiently with the neighbor list building options described in Section 3.3. Using the aspherical methods discussed in the paragraph above, rigid bodies consisting of multiple finite-size spheres can be defined to model composite granular particles with irregular shape. This can be done with highly overlapped spheres to create bodies with geometric shapes and tunably roughened surfaces using the methodology and tools described in [150]. Likewise the aspherical polygonal (2d) or polyhedron (3d) body particles can be used with pair styles that implement DEM models for frictional interactions between polygons and polyhedra with rounded corners and edges [151,152].

A novel application of customized DEM models added to LAMMPS has been to study the seasonal formation, melting, and dynamics of the sea ice pack in the Arctic Ocean.[15] For the full global system, these are models with hundreds of thousands of

[13] Lauren Abbott (NASA Ames Reseach Center) implemented the *fix mscg* command in LAMMPS in collaboration with Gregory Voth's group (U Chicago).

[14] Leo Silbert (Central New Mexico Community College) along with Jeremy Lechman and Gary Grest (Sandia National Laboratories) wrote the initial Fortran granular models which were integrated into LAMMPS.

[15] See cover image for this issue.

particles, run at length scales of thousands of kilometers for multidecadal timescales. The DEM capabilities of LAMMPS have also been extended by DCS Computing, a European software company, which created the LIGGGHTS simulator [153] for modeling industrial processing of granular materials. Because LAMMPS was provided to DCS Computing under the LGPL open-source license, it has since enabled them to add proprietary software to create the commercial product Aspherix [17].

(4) LAMMPS has several particle-based models which can be used at the continuum scale to model fluid dynamics and solid mechanics. For fluids this includes several variants of smoothed particle hydrodynamics (SPH) models [154] as well as its thermalized variant smoothed DPD (SDPD) [155]. For solids, it includes a Smoothed Mach Dynamics (SMD) capability [156] with several material models. SMD enables hydrostatic interactions to be treated independently from material strength models.[16] The particle-based non-local continuum method known as Peridynamics (PD) [157] has also been implemented in LAMMPS for several constitutive models which capture elastic or plastic deformation, embrittlement, and viscoelastic behavior [158,159]. PD can be parameterized to model fracture and damage [160] at the meso to continuum scales in a variety of materials, including impact and penetration effects [161,162].

6.3. Machine learning interatomic potentials

As in many disciplines, machine learning (ML) techniques have become increasingly popular and powerful for MD, where they can be used to create interatomic potentials in an automated manner. This has the promise to address a critical need in materials modeling, namely on-demand generation of interatomic potentials that can be accurate for particular materials in the particular regimes relevant to a specific application. Success will enable the use of atomistic simulation in a wide range of research areas where currently the necessary interatomic potentials do not exist. This is particularly true in the case of computational design of new materials, compositionally or structurally complex materials, materials at extreme equilibrium and non-equilibrium conditions, as well as chemically active materials. Without an appropriate potential, MD or other classical atomic-scale material modeling computations cannot be performed. The goal for ML potentials is to achieve near-quantum accuracy, at least for some material properties and responses, with a computationally cheap potential which still scales as $O(N)$ in the number of atoms in the system – as compared to $O(N^3)$ scaling or higher for conventional density functional theory and higher-order quantum methods.

A key reason for this interest has been the rapid growth in availability of data from quantum electronic structure (QM) calculations for systems with up to a few hundred atoms. The abundant QM data has exposed the limited accuracy of many classical potentials. Moreover, the data has provided a means for systematically improving their accuracy via machine learning. Consequently, the last decade has seen a rapid exploration of many different approaches, referred to collectively as data-driven or machine learning interatomic potentials (MLIAP). Many of these MLIAPs are now available in LAMMPS, either as native implementations, or as external packages and libraries maintained by other groups. Fig. 10 includes benchmarks for several prominent MLIAPs that were included in a recent study by Zuo et al. [107]. It is clear that the increased accuracy of ML potentials need not require a large increase in computational cost.

While the rapid development of MLIAPs has resulted in a great diversity of approaches, most of the successful methods share a common structure. First, the total potential energy is decomposed into atomic energies summed over all atoms. This is an expedient choice that does not have a rigorous basis in theory, but it greatly simplifies the training process and also the implementation of MLIAPs in parallel MD codes like LAMMPS. The atomic energy model is trained on QM results for the total potential energy, forces, and stress tensor components for a large number of small atomic configurations. Atomic energy model forms draw on well-established data science techniques such as linear regression, Gaussian process regression, and artificial neural networks (ANN). Second, the atomic energy of each atom is assumed to depend only on the positions of other atoms in its local neighborhood. In addition, the atomic energy should vary smoothly and differentiably as atoms move in and out of the neighborhood, and should also be invariant to rotation and to permutation of atoms of the same element. This requires transforming the positions and chemical identities of the neighbor atoms into a set of *descriptors*. The most successful descriptors are specialized to satisfy these unique requirements, while at the same time providing enough flexibility to capture the important physics. Hence, while most MLIAPs share these common features, they differ in their choice of descriptors and atomic energy models.

Despite the diversity of approaches that have been developed, there is considerable evidence to suggest that many of the ML potentials are performing similarly both in computational cost and accuracy. The recent study by Zuo et al. [107] showed that four of the more prominent machine-learning approaches achieved similar computational performance at a given level of accuracy, and vice versa. For each of the four MLIAP approaches in the study, the potential that provided the optimal performance/accuracy trade-off has been added to Fig. 10 (blue points). These potentials appear to have disrupted the decades-long exponential growth in computational cost of increased-accuracy interatomic potentials. In addition, the low-level structure of the force kernels for MLIAPs lends itself to optimization for manycore and GPU processors, resulting in even greater performance gains relative to complex empirical potentials [163,164].

Here we highlight a few notable examples of ML potentials which are available for use in LAMMPS, and then discuss some of the opportunities and challenges that exist.

- **NNP** The first successful MLIAP was the Neural Network Potential (NNP) model for silicon developed by Behler and Parrinello [165]. It uses two-body and three-body rotationally invariant descriptors (symmetry functions) as inputs to a differentiable non-linear neural network model for the atomic energy of each atom. A multistreaming C++ implementation of NNP has recently been created by Singraber and Dellago in their N2P2 software [166]. They created an interface to N2P2 via the *pair hdnnp* command and *ML-HDNNP* package in LAMMPS.
- **GAP** Gaussian Approximation potentials (GAP) predict local atomic energy and force of each atom using Gaussian process regression on a sparsified set of training configurations. Proximity of prediction points to training points is measured using a similarity metric in the descriptor space. The original GAP descriptors were the $SO(4)$ bispectrum components [167]. Most GAP potentials now use the power spectrum of basis functions in $SO(3)$ combined with a radial basis. The GAP potentials have been implemented in the *QUIP/libatoms* software [168]. All potentials implemented in QUIP can be accessed via the *pair quip* command in the *ML-QUIP* LAMMPS package.
- **SNAP** Spectral Neighbor Analysis Potentials (SNAP) use $SO(4)$ bispectrum components for local atomic density as input to a linear model for local atomic energy [169]. The method has been extended to quadratic [170] and neural network

[16] Georg Ganzenmüller (Ernst Mach Institute) implemented both the SPH and SMD models in LAMMPS.

[171,172] energy models, as well as chemically-labeled descriptors [173]. All the code for running SNAP potentials is part of the LAMMPS *ML-SNAP* package. This includes a *pair snap* command, potential files for a range of materials, and code for generating information needed to train potentials (force gradients). A Kokkos version of SNAP, optimized for manycore and GPU architectures, is provided in the KOKKOS package [163]. Multiple open-source software codes for training SNAP potentials are also available to users who want to train a potential for their own MD application [62,174].

- **MTP** Similar to SNAP, the Moment Tensor Potential (MTP) describes the local energy of atoms in terms of rotationally invariant scalars derived from the local density of neighbors around a central atom [175]. The *USER-MLIP* package, distributed separately from LAMMPS, provides an interface to the MTP code [176].
- **ACE** The Atomic Cluster Expansion (ACE) method [177] is a rigorous generalization of the ideas at the core of several of the more successful ML potentials, including the GAP, SNAP, and MTP potentials described above. The local atomic energy is expanded in a series of progressively higher order scalar invariant products of $SO(3)$ and radial basis functions. The addition of higher order terms, while computationally expensive, makes it more straightforward to systematically improve accuracy. The PACE library provides a performant implementation of ACE for running large-scale molecular dynamics [178,179]. The *pair pace* command in the *ML-PACE* LAMMPS package provides an interface to the PACE library.
- **DeepPot** The DeepPot potentials combine the TensorFlow neural network library with several different rotationally invariant descriptors. An open source framework for both training and running DeepPot potentials has been interfaced to LAMMPS via the *USER-DEEPMD* package, distributed separately from LAMMPS [180,181].
- **KIM** The Open Knowledgebase of Interatomic Models (KIM) [108,109] library described in Section 6.1 provides access to several ML potentials, including DUNN [182], hNN [183], and PANNA [184].
- **Other** Additional notable examples of MLIAPs that have been implemented or interfaced to LAMMPS include ANI [185], AGNI [186], AENET [187], RANN [188], PINN [189], and MLIP [190].

The above list shows it is relatively straightforward for a developer to construct an effective interface between LAMMPS and their particular ML potential. However, these efforts have also exposed certain limitations. First, LAMMPS pair styles typically only support the calculation of forces needed for running MD in LAMMPS. The training of potentials is done in a separate code that may or may not use the same low-level methods as those used for calculating forces. This makes inconsistencies possible between the LAMMPS implementation for MD and the software used to train the potential. Second, the use of a single software package for both calculating descriptors, and predicting energy and forces makes it difficult to combine different methodologies. For example, to experiment with SNAP descriptors combined with Gaussian process, kernel ridge regression, or TensorFlow, a new, custom code would need to be written for each coupling. Even though open source code exists for each of these models, it is always tied to a particular descriptor. This limits the rate at which a group can experiment and innovate because of the need to integrate descriptors or models from other groups into their workflow.

To address this issue we recently developed a general, lightweight software framework for ML potentials that is now included in LAMMPS as the *ML-IAP* package and the *pair mliap* command. Whether training a potential or running MD, the choice of energy model is decoupled from the choice of descriptors, allowing any energy model to be combined with any set of descriptors. By contrast, in most MLIAP implementations, including the original SNAP implementation in LAMMPS, the calculation of descriptors and forces is performed together in a single function call. Using SNAP as a test case, we have shown it is relatively straightforward to convert an existing implementation of a ML potential to this new framework.

The decoupling is achieved by two pure virtual classes called *MLIAPModel* and *MLIAPDescriptor* that define the interface to a set of methods that new low-level derived classes must provide. Once a particular machine-learning potential is integrated into LAMMPS in this manner, it is possible to replace either the energy model or the descriptor with an alternate one by changing one line of the input script. To further simplify the refactoring, all the inputs and outputs from these methods are organized in a third low level class *MLIAPData*. This eliminates the need for *MLIAPModel* and *MLIAPDescriptor* classes to know anything about the rest of LAMMPS or each other.

The SNAP bispectrum component descriptors and the SNAP linear and quadratic energy models have been interfaced in this manner. These reproduce all variants of the original SNAP potential, and can both run MD via the *pair mliap* style as well as generate parameter gradients via the *compute mliap* style for use in training software like *FitSNAP* [62]. A variety of additional *MLIAPModel* and *MLIAPDescriptor* styles have been added by different groups. For example, the *mliappy MLIAPModel* style supports arbitrary Python-based energy models, including the *PyTorch* ANN library, enabling running potentials based on *PyTorch* neural network energy models.[17]

The following pseudocode shows how the decoupling of *MLIAPModel* and *MLIAPDescriptor* objects is achieved when calculating forces within the *pair mliap* style:

```
descs = Descriptor.computeDescs(atoms)
modelGrads = Model.computeGrads(descs
forces = Descriptor.computeForces(atoms, modelGrads)
```

Listing 1: Decoupled MLIAP interface for simulation.

Note that in the actual implementation, data is passed between the *MLIAPModel* and *MLIAPDescriptor* objects via a pointer to the *MLIAPData* object. The same approach can be also used to define a LAMMPS interface for training MLIAP models. The main challenge is to efficiently calculate the gradient of force predictions w.r.t. model parameters. This is true regardless of whether the energy model is linear, as in the case of SNAP, or a complex nonlinear function, such as an artificial neural network (ANN). For example, in the case of an ANN library that uses stochastic gradient descent to optimize the ANN weights, computing the gradient of the loss function w.r.t. weights requires first calculating the gradient of each atomic force component w.r.t. each descriptor and then passing this information to the ANN library. This information can now be efficiently calculated within LAMMPS, using SNAP or some other descriptor and then passed to an arbitrary energy model. The energy model will compute the gradient of the loss function and update its parameters accordingly, as described in the following pseudocode:

For N_{desc} descriptors and N_{param} model parameters, the efficacy of this approach depends on the size of the $N_{desc} \times N_{param}$ matrix *gradGrads*, which is the double gradient of the potential energy w.r.t. both descriptors and model parameters. For a general

[17] Nicholas Lubbers (Los Alamos National Laboratory) created the *mliappy MLIAPModel* style which provides an interface to arbitrary Python models, including the *PyTorch* ANN library.

```
descs = Descriptor.computeDescs(atoms)
gradGrads = Model.computeGradGrads(descs)
forceGrads = Descriptor.computeForceGrads(atoms, gradGrads)
Model.update(forceGrads)
```

Listing 2: Decoupled MLIAP interface for training.

non-linear atomic energy model this matrix can be prohibitively large to evaluate and store. However, for many important models, most entries will be zero, and the cost of calculating and storing *gradGrads* is negligible. For a linear model, the matrix is simply the identity matrix of rank N_{desc}, while for a quadratic model, the number of non-zero entries scales as $\mathcal{O}(N_{desc}^2)$. Conversely, in cases where *gradGrads* is large, the following alternative algorithm for obtaining the force gradient can be more efficient:

```
descGrads = Descriptor.computeDescGrads(atoms)
forceGrads = Model.computeForceGrads(atoms,descGrads)
Model.update(forceGrads)
```

Listing 3: Decoupled MLIAP interface for training, alternate.

The efficacy of this alternate approach depends on the size of *descGrads*, the derivative of each descriptor w.r.t. the position of each neighbor atom, which scales as $\mathcal{O}(3N_{neigh} \times N_{desc})$, where N_{neigh} is the average number of neighbors per atom. To first order, the relative performance of the two methods depends primarily on the complexity of the energy model relative to the size of the neighbor list. For example, the N2P2 implementation of the Behler-Parrinello neural network models uses an algorithm equivalent to Listing 3; see [164] for details.

All of these algorithmic options have been implemented in the new MLIAP package. Our hope is that this more generic and separable interface to ML potentials not only benefits user workflows and productivity, but will likewise support developers of MLIAPs by providing a path for incorporating current and future MLIAP strategies into LAMMPS which can then use descriptors or energy models from other MLIAPs.

6.4. Charge equilibration methods

Most interatomic potentials that explicitly treat electrostatic interactions assume a fixed value for the positive or negative charge assigned to each atom. In some cases, it can be useful to relax this assumption by allowing the charge values to varying continuously in response to the local environment. This is particularly useful for chemically reactive systems and systems containing metal atoms in both the pure elemental and oxidized states. Potential energy models have been developed in which the charges on atoms are treated as variables determined by solving an additional set of equations that depend on the relative positions and chemical identities of the atoms. The most successful methods for doing this are all based on variants of the electronegativity equalization method (EEM) [191,192] or the closely-related charge equilibration method (QEq) [193]. In LAMMPS, all of these variants are referred to as charge equilibration and the acronym QEq is used. The QEq algorithm invokes the electronegativity equalization principle [194,195]:

$$\frac{\partial E(\mathbf{r},\mathbf{q})}{\partial q_i} = -\mu \tag{1}$$

where E is the total electrostatic energy of the system, and q_i is the charge on one atom, and $\mathbf{r}$ and $\mathbf{q}$ are the positions and charges of all the atoms in the system, respectively. The principle states [194] that the electronegativity at every atom site i is the same, equal to the negative of the chemical potential (μ) of the electrons in the system. Given any small change in the positions of the atoms, charge quickly redistributes throughout the system so that the electronegativity at all atomic sites remains equal.

LAMMPS provides support for a range of variable charge potentials. Examples include the simple point charge model for water (SPC-Q) [196], the Streitz-Mintmire potential for Al_2O_3 [197] that couples a Slater-type orbital charge model to an embedded atom method (EAM) potential [198], and its modification [199] coupled to a modified EAM (MEAM) potential [200]. The variable charge Morse-stretch (MS-Q) potential [201,202] and variable charge Morse-Buckingham (MB-Q) potential [203] couple simple point charge models for TiO_2. Charge-optimized many-body (COMB) [204–206] and reactive force field (ReaxFF) [207,208] potentials design their own charge models for a wide variety of materials.

The simple point charge model uses point charges. The Slater-type orbital and COMB charge model use a Slater 1s-orbital to describe charge density. The ReaxFF charge model use a shielded-Coulomb charge density. Note that all the LAMMPS implementations of these charge models and associated variable-charge potentials treat Coulombic interactions as effectively short range; long-range Coulombic effects are not currently treated.

In all these methods, when computing energy, force, and virial contributions from the electrostatic interactions, the electronegativity equalization condition Eq. (1) must be satisfied. In many cases, the electrostatic energy $E(\mathbf{r},\mathbf{q})$ is quadratic in the charges, so that Eq. (1) is equivalent to a set of N linear equations, where N is the total number of charges, and standard iterative linear algebra solvers such as conjugate gradient (CG) and GMRES can be used [209].

In the more general case where $E(\mathbf{r},\mathbf{q})$ is an arbitrary non-linear function, obtaining a set of self-consistent charges that satisfy Eq. (1) can be quite challenging. In this case, it is expedient to treat the charges as dynamic variables that are allowed to evolve with the atom positions so as to approximately satisfy Eq. (1). This approach is often referred to as the extended Lagrangian (XL) and is similar to the Car-Parrinello method for approximating the Born-Oppenheimer potential energy surface in *ab initio* molecular dynamics [210]. The equations of motion for XL QEq are

$$\mathbf{s} \cdot \ddot{\mathbf{q}}(t) = \mathbf{M}(t) - \boldsymbol{\eta}(t) \tag{2}$$

where $\ddot{\mathbf{q}}$ is the acceleration vector of the charges, $\mathbf{s}$ is an effective mass matrix, $\mathbf{M}(t)$ is the electrostatic force on the charges, and $\boldsymbol{\eta}$ is the damping force. The electrostatic forces drive the charges toward a low energy state while conserving the total charge, and are given by

$$\mathbf{M} = -(\frac{\partial E}{\partial \mathbf{q}} - \frac{1}{N}\sum_i \frac{\partial E}{\partial q_i}\mathbf{1}) \tag{3}$$

LAMMPS supports two flavors of XL QEq that differ primarily in the definition of the damping force. The damped dynamics [204] equations of motion use a fixed damping factor of η_d. The FIRE [211] equations of motion use a dynamically changing damping factor that depends on the relative orientation of the charge velocity and the charge force.

To summarize, the following *fix qeq* styles are provided by LAMMPS, to assist in development of variable-charge interatomic potentials:

- *fix qeq/reaxff* Solves the variable charge equations using iterative linear algebra methods [209]. It can only be used with the ReaxFF potential [207,208].

Table 3
Summary of LAMMPS styles implemented for the SPIN package.

LAMMPS style	Style name	Style description
atom	spin	adds classical spins to atoms
pair	exchange	classical Heisenberg interaction
	exchange/biquadratic	Heisenberg interaction with biquadratic term
	dmi	Dzyaloshinskii-Moriya interaction
	neel	dipolar and quadrupolar Néel interaction
	magelec	magneto-electric interaction
	dipole/cut	short-range dipolar interaction
	dipole/long	long-range dipolar interaction
fix	nve/spin	coupled spin-lattice parallel symplectic time integrator
	precession/spin	external precession torques: Zeeman field, uniaxial and cubic anisotropies
	setforce	set a constant precession torque to a group of spins
	langevin/spin	couple spin to a Langevin bath for NVT dynamics
	neb/spin	atom forces and spin torques for GNEB calculations
compute	spin	global magnetization vector, magnitude, and temperature
minimize	spin	damped dynamics with adaptive time-step
	spin/cg	conjugate-gradient minimization^2
	spin/lbfgs	limited-memory BFGS minimization^2
kspace	ewald	Ewald summation for long-range spin-spin interactions
	pppm	particle-mesh method for long-range spin-spin interactions

- *fix qeq/comb* Solves the variable charge equation using the damped dynamics extended Lagrangian method [204]. It can only be used with the COMB and COMB3 potentials [205,206].
- *fix qeq/point* A generalized implementation of *fix qeq/reaxff* for solving the simple point charge model [209].
- *fix qeq/shielded* A generalized implementation of *fix qeq/reaxff* for solving the shielded charge model [209].
- *fix qeq/dynamic* A generalized implementation of *fix qeq/comb* using the damped dynamics extended Lagrangian method. It can be used with any *pair* style that treats atomic charges.
- *fix qeq/fire* A generalized implementation of *fix qeq/comb* using the FIRE extended Lagrangian method [211]. It also can be used with any *pair* style that treats atomic charges.

6.5. Coupled magnetic spin and lattice dynamics

Magnetism and magnetization dynamics are known to strongly influence material properties of magnetic metals and oxides. Thus for accurate computations, magnetic degrees of freedom must also be accounted for. This is the case, for example, for thermomechanical properties of magnetic transition metals and alloys (in particular close to their Curie or Néel transitions) [212–214], as well as for magnetostriction [215,216], magneto-structural phase transitions [217], or magnetocaloric effects [218,219]. The *SPIN* package in LAMMPS [220] provides a fully parallel model for either spin dynamics (SD) or coupled spin and lattice dynamics (SD-MD) [221–224]. Table 3 summarizes the different LAMMPS styles which were implemented in the package to enable such models.

Atom style *spin* allows magnetic particles to be defined (usually atoms, but coarse-grained macromagnetic particles can also be represented). It stores a unit vector $\vec{s}_i$ representing a classical magnetic spin with each particle. Combined with pair and fix styles, this allows a variety of spin-lattice Hamiltonians to be modeled. For example, in the case of a simple exchange interaction,

$$\mathcal{H}_{sl}\left(\vec{r},\vec{p},\vec{s}\right)=\sum_{i=1}^{N}\frac{|\vec{p}_i|^2}{2m_i}+\sum_{i,j=1}^{N}V\left(r_{ij}\right)-\sum_{i,j}^{N}J\left(r_{ij}\right)\vec{s}_i\cdot\vec{s}_j, \tag{4}$$

which combines kinetic energy with an interatomic potential (here a pair potential $V\left(r_{ij}\right)$)) and a magneto-elastic interaction (here a two-body Heisenberg Hamiltonian). More generally, any of the magneto-elastic pair styles in the table can be combined with any interatomic potential in LAMMPS by using the *pair hybrid* style, discussed in Section 4.2.2. Long-range magnetic dipolar contributions (leveraging Ewald or PPPM methods) to the energy, torque, and virial can also be included in the models.

Fix *nve/spin* integrates the spin-lattice equations of motion derived from the Hamiltonian as

$$\frac{d\vec{r}_i}{dt}=\frac{\vec{p}_i}{m_i} \tag{5}$$

$$\frac{d\vec{p}_i}{dt}=\sum_{j}^{N_i}\left[-\frac{dV\left(r_{ij}\right)}{dr_{ij}}+\frac{dJ\left(r_{ij}\right)}{dr_{ij}}\vec{s}_i\cdot\vec{s}_j\right]\vec{e}_{ij} \tag{6}$$

$$\frac{d\vec{s}_i}{dt}=\boldsymbol{\omega}_i\times\vec{s}_i \tag{7}$$

where $\vec{e}_{ij}$ is the unit vector along $\vec{r}_{ij}$ and N_i is the number of neighbors of atom i. Eq. (7) describes the precession of classical spins, where $\boldsymbol{\omega}_i$ is a precession vector. In the case of a simple exchange interaction, this is defined as:

$$\boldsymbol{\omega}_i=\frac{\gamma}{\mu_i}\sum_{j}^{N_i}J\left(r_{ij}\right)\mathbf{s}_j \tag{8}$$

with γ the gyromagnetic ratio ($\gamma\approx 0.176$ rad·THz·T^{-1}) and μ_i the atomic spin norm (in Bohr magneton) which yields a precession frequency in rad·THz.

Stable numerical time integration of the first-order (massless) equations for the spins requires the discretization be symplectic [225], but this property is violated by standard parallel time integration within the spatial-decomposition context of Section 3.1. The integrator in the *SPIN* package overcomes this by subdividing each processor 3d subdomain into 8 sectors and looping over them in a prescribed manner, interleaved with appropriate communication of updated spin values [220]. The resulting scheme was shown to be highly scalable, while retaining the stability and $\mathcal{O}(\Delta t^2)$ accuracy of the underlying velocity-Verlet discretization with timesteps commonly used for the lattice dynamics. For example, on 64 nodes of a commodity computing platform it is possible with the *SPIN* package to run an SD-MD simulation of 128 million magnetic iron atoms at a speed that is only 4× slower than running conventional MD with an EAM potential.

Other *fix* styles in the table enable application of external magnetic fields (torques) to the system and thermostatting of the spin degrees of freedom. When a spin thermostat is applied, Eq. (7) becomes the classical Landau-Lifschitz equation, as described by Evans and Ma et al. [224,226], and a magnetic damping constant can be defined. The *compute spin* command calculates diagnostic properties of the spin system. Three spin energy minimization algorithms were also implemented: a dissipative scheme with an adaptive time step, and orthogonal spin optimization (OSO) using either conjugate gradient (CG) or limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) algorithms [227,228]. They enable parallel energy minimization of large-scale spin textures in magnetic systems [229]. The Geodesic Nudged Elastic Band (GNEB) approach as developed by Bessarab et al. [230] was also implemented and works with the three minimization algorithms [228]. GNEB enables calculation of minimum energy pathways and energy barriers in spin systems. It has proved effective for study of magnetic phase transitions such as skyrmion lifetime, or domain-wall nucleation [231].

6.6. Multi-replica and accelerated MD methods

As mentioned in Section 4.1, LAMMPS allows P processors to be partitioned to run a multi-replica simulation with M replicas (or M totally independent simulations). LAMMPS has several options for running multi-replica methods. One is for the enhanced sampling method (ESM) known as replica exchange Monte Carlo or parallel tempering [232]. In this case each replica runs at a different thermostatted temperature. Periodically, pairs of replicas at adjacent temperatures exchange the value of their current potential energy and make a Boltzmann-weighted Monte Carlo decision whether to swap their temperature set points. This improves the sampling efficiency at the lowest temperature state while ensuring that configurations are sampled with the correct probability.

LAMMPS also has implementations of two accelerated MD (AMD) methods which leverage this multi-replica capability, known as parallel replica dynamics (PRD) [233] and temperature accelerated dynamics (TAD) [234]. In contrast to ESMs, AMD methods can produce statistically time-accurate long-timescale trajectories for solid-state systems with a potential energy landscape of well-defined basins with infrequent transition events between basins. PRD runs M replicas of the system starting from the same energy basin until an (escape) event occurs. After a suitable re-equilibration time, all the replicas are restarted from the new basin. The time acceleration can thus be nearly a factor of M. PRD can be used to model small systems for very long timescales with much higher parallel efficiency than could be achieved by using all the processors to run a single small simulation.

The TAD method accelerates a single trajectory by running at a higher temperature so events occur more frequently. Multiple events starting in the same basin are generated. The energy barrier height for each event is calculated via the nudged-elastic band (NEB) method [70]. For each NEB calculation a series of conformational points along the transition pathway (elastic band) are stored as independent replicas of the system. After intra-replica forces are computed by each replica, the NEB algorithm adds inter-replica forces and iteratively relaxes the elastic band to calculate the barrier height at the saddle point as well as the transition pathway. This allows accurate estimation of which event would have occurred at the earliest time at the lower (desired) temperature. That event is performed and the procedure is repeated from the new energy basin. The time acceleration in TAD is a function of the barrier heights and the ratio of the two temperatures, as well as the speed-up provided by the parallelized NEB calculation.

Parallel implementations of two related AMD methods which are not multi-replica methods were also recently added to LAMMPS [235]: the original hyperdynamics (HD) [236] and local hyperdynamics (LHD) [237] algorithms. Both methods apply a bias potential to individual pairs of nearby atoms to effectively increase the likelihood of an event occurring, such as a diffusive hop. The LHD method enables time-acceleration of a single large simulation (up to millions or billions of atoms) by leveraging the fact that many events can be accelerated simultaneously in a large system because they are separated by long enough distances that their transition probabilities are independent of each other

Support for multi-walker (replica) metadynamics in LAMMPS is provided by the Colvars [238] and PLUMED [239] libraries, each of which LAMMPS wraps with a *fix* style. In parallel this means each processor creates an instance of the library. Typically each *walker* is an independent calculation for a specific configuration which can access common data shared by all processors. Each walker periodically updates its accumulated bias from the shared data and adds its results to the shared data so it can be read by others. Walkers can also be added or removed at any time.

6.7. Unit definitions

LAMMPS has an input script *units* command which selects one of several unit styles. Each style defines physical units for quantities such as mass, length, time, energy, pressure, etc. These are then used for all inputs and outputs to and from the code. When simulating solid materials at the atomic scale, *metal* units (Å, ps, eV) are commonly used for (length, time, energy). *Real* units (Å, fs, kcal/mol) are more common in biomolecular and chemistry codes. For systems that include electrons as particles, *electron* units (Bohr, fs, Hartrees) may be more convenient. However, for simulations at the meso or continuum scales, none of these are convenient. *Micro* units (μm, μs, picogram-$\mu m^2/\mu s^2$) or *cgs* units (cm, s, ergs) can be used instead.

LAMMPS implements each of its unit styles using a dozen or so numeric conversion factors which are hard-coded internally and used throughout the code in calculations whenever physical quantities are combined to produce a new physical quantity. For example, the factor *ftm2v* is used in time integration to increment the velocities by an expression of the form *force*time/mass*. The factor *qqr2e* is used in *pair* styles involving electrostatic charges to convert from *charge*charge/distance* to energy units. If the unit style is consistent, like *cgs* or reduced (dimensionless) Lennard-Jones *lj* units, these factors are typically unity. But for inconsistent styles, like *metal* or *real*, they are not.

If an existing style is not convenient for parameterizing a new model, a user can extend LAMMPS with a new unit style by simply calculating appropriate values for the dozen conversion factors and adding them to the code. LAMMPS will then read inputs and produce outputs in the new units, and perform all its internal calculations in a manner consistent with the new units.

6.8. Additional features

Finally, here is a brief summary of additional features in LAMMPS, some of which are non-traditional in the MD context.

- Non-equilibrium MD
  In addition to equilibrium Green-Kubo methods for measuring material properties like viscosity and heat conductance, LAMMPS has non-equilibrium MD (NEMD) methods [240] which can be used for the same purpose. These include reverse perturbation methods from Müller-Plathe and collaborators [241,242], as well as methods that impose continuous deformations or temperature gradients on the system. For shear viscosity, a triclinic simulation box like the one illustrated at the lower left of Fig. 1 can be deformed as a simulation runs

until its skew angle reaches a threshold, then "flipped" to the corresponding negative skew angle without changing the force on any particle. By iterating this process, shear can be applied continuously for as long as desired. Extensional viscosity can also be measured for uniaxial or biaxial flows using the method of Dobson [243], where both the box shape and its alignment with respect to the flow field change continuously, again for as long as desired.[18] For either shear or extensional flow, the SLLOD equations of motion [244] can be used in the NVT or NPT ensembles. The methods have been used to model flow in simple fluids, solvated systems of large particles or rigid bodies, and polymer melts [245,246].

- Monte Carlo methods
  LAMMPS supports a variety of Monte Carlo (MC) techniques for randomly sampling configurations according to different thermodynamic ensembles [247]. By evolving a system via non-physical MC rules, the methods can sample regions of configuration space that are difficult to reach using MD alone. MC moves are randomly generated and then accepted or rejected with a probability that depends on the potential energy change, according to the Metropolis acceptance criterion for a particular ensemble. It is often convenient to interleave these methods with short MD runs, either sequentially or simultaneously. In the latter case, the sampling no longer satisfies detailed balance (microscopic reversibility), so that correct sampling of a particular thermodynamic ensemble can not be guaranteed. But through careful testing it is still possible to achieve sampling that approximates the correct ensemble [248].
  These methods are implemented as *fix* styles in the *MC* package. The *gcmc* style enables MC sampling of both the canonical and grand canonical ensembles. The available moves are insertion or deletion of an atom or molecule, as well as translation and rotation. The *atom/swap* style allows the chemical identity of pairs of atoms to be randomly altered. The change can be a simple swap, or it can be a random selection of new identities according to the specified semi-grand ensemble chemical potentials [249]. The *charge/regulation* style allows composite moves that change the charge on a coarse-grained particle while also adding or removing a neutralizing counter-ion. This allows regulation of pH and dissociation in solutions containing acids, bases, and electrolytes [250].
  The *widom* fix style uses the same approach to estimate the chemical potential of an atomic or molecular species by randomly inserting a "ghost" molecule and measuring the change in potential energy. The *bond/swap* style can be used to more quickly equilibrate coarse-grained (bead-spring) polymer systems by swapping a bond between two adjacent chains so that two new chains are created: one from the first part of one chain and second part of the other, and vice versa [251].
  All of these *fix* styles can be used in parallel. In the case of the *bond/swap*, multiple MC moves are attempted simultaneously by different processors within their spatial subdomains. However for the more rigorous *gcmc*, *atom/swap*, and *charge/regulation* styles, each move is performed by a single processor, and the outcome must be communicated to all processors before proceeding. This is due to the fundamentally sequential nature of the acceptance criteria for Metropolis Monte Carlo moves, required for preserving detailed balance. Because the *widom* style only makes test insertions and does not actually modify the system, insertions can be performed simultaneously on all processors.
- Heuristic bond breaking and formation
  Separate from the reactive force fields and reactive DPD models mentioned above, LAMMPS has a heuristic *fix bond/react* command for performing chemical reactions which change the topology of covalent bonds in molecules, e.g. for cross-linking or scission reactions in polymer systems [252,253]. It uses probabilistic distance-based rules for triggering reactions and can be used with all-atom or coarse-grained molecular models. The user specifies a kinetic rate for the reaction and provides two molecule *template* files that encode the bond topology around the reaction site both before and after the reaction. The files include chemical identifiers for all bond and associated angle/dihedral/improper interactions. A series of rates and template files can be specified to represent a complex network of competing chemical reactions.
- Shock physics
  LAMMPS has a SHOCK package which contains styles that enable shock physics simulations where a high-velocity collision (km/s) with a wall is modeled, inducing a shock wave in the material. The package includes a *fix wall/piston* command to treat the simulation box boundary as a piston which rapidly thermalizes and changes the velocity of atoms near the wall and a *fix append/atoms* command to continuously add new atoms at the other end of the simulation box as the shock wave propagates outward. It also includes fixes which perform time integration via the constant-pressure Hugoniostat equations [254] or multi-scale shock technique (MSST) [255] which are important for achieving steady-state shock conditions.
- Polarization effects
  There are several pair styles in LAMMPS which enable simulation of polarization effects. For solid-state systems this includes the adiabatic core/shell model [256][19] and COMB many-body potentials [97,98] which have an induced-dipole term. For fluids and molecular systems it includes the simple Drude model [257][20] Work is also nearly complete[21] to add the AMOEBA [258,259] and HIPPO [260] polarized force fields developed by Jay Ponder's group at Washington University in St. Louis, based on the Fortran implementation in their Tinker MD code [261]. These have parameterizations for water and various biomolecular systems including proteins. The DIELECTRIC package enables polarization effects at the interface between two media with dielectric mismatch (e.g. an oil-water interface) to be incorporated into mesoscale simulations using boundary-element solvers [262].
- Particle motion on 2d surfaces
  LAMMPS includes a MANIFOLD package which enables MD simulation of particles constrained to a 2d manifold (generalized surface) via the RATTLE constraint algorithm [263]. The package has been used to model self-assembly of a coarse-grained virus capsid protein model [264]. It includes fixes to compute the constraint force and perform the appropriate time integration in either the NVE or NVT ensemble. It also includes a virtual parent *manifold* class and child classes (styles) for more than a dozen surfaces defined by user-specified parameters. These include an ellipsoid, dumbbell, torus, and thylakoid membrane. New surfaces can be easily added by coding two functions, one that determines whether a particle coordinate is on the surface or not, and a second that returns the normal direction at a point on the surface.

[18] David Nicholson (MIT) implemented continuous extensional flow options for LAMMPS in the UEF package.

[19] Hendrik Heenen (Technical University of Munich) implemented the CORESHELL package in LAMMPS.

[20] Alain Dequidt (U Clermont Auvergne), Julien Devemy (CNRS), and Agilio Padua (ENS de Lyon) implemented the DRUDE package in LAMMPS.

[21] Josh Rackers (Sandia National Laboratories) is helping implement the AMOEBA and HIPPO force fields to LAMMPS.

- Solid state structural analysis
  LAMMPS has several *compute* styles that calculate a per-atom property based on the geometry of the atom's local neighborhood. They are useful in solid-state physics for identifying point or extended defects or which kind of local crystalline lattice the atom belongs to. They include compute styles for an atom's coordination number, centrosymmetry parameter [265], common neighbor analysis (CNA) metric [266,267], polyhedral template matching (PTM) metric [268], and Voronoi tesselation volume surrounding each atom.[22] As mentioned in Section 4.2.4, the latter is calculated by use of the Voro++ library [51,52].
  The DIFFRACTION package has two commands, *compute saed* and *compute xrd*, which calculate electron and x-ray diffraction intensity values respectively on a mesh of reciprocal lattice points [269]. For electron diffraction the values can be time averaged and written to a VTK-formatted file for post-processing visualization as a diffraction image of the system.
  The PHONON package calculates a dynamical matrix for the system via an FFT, which can then be used to calculate phonon dispersion relations [270]. It also has commands to calculate the dynamical matrix and third-order force-constant tensor via finite difference methods.[23]
- Codes that build systems for LAMMPS
  There are many third-party software packages, both open-source and commercial, that can build molecular systems in a format compatible with input to LAMMPS.[24] Here we mention two which are specifically designed for use with LAMMPS and which are powerful design tools for complex models. Both allow assembly of large multi-component systems using scripts that define and replicate individual components. The first is the Enhanced Monte Carlo (EMC) tool [271,272] which can generate either all-atom or coarse-grained models, compatible with a variety of biomolecular or organic molecule force fields. Monte Carlo moves can be invoked on the initial geometry to relax the system energetically (unoverlap atoms), so that it is suitable for immediately running MD. The second is the Moltemplate tool [273] which is aimed at building novel coarse-grained biomolecular systems. It can be used to generate large multi-component membranes or whole-cell models [274].

7. Conclusion

In this paper we have described the current state of the LAMMPS molecular dynamics package, focusing on aspects that have made it both a powerful and popular tool. These include design features that enable flexibility for users or developers to (a) customize models they simulate, (b) extend the code with new features they need and are willing to implement, or (c) couple to other codes for multiphysics or multiscale applications.

Two increasingly popular trends for MD simulation in materials science, whether at the atomistic, coarse-grained, meso, or continuum scales, are pipelined workflow computations on big data and applications of machine learning techniques. As described in Sections 4.3 and 6.3, we have made recent efforts to enhance LAMMPS as a useful tool in these contexts. In particular, with the rapid growth of machine learning (ML) interatomic potentials and the data associated with training and quantifying their errors, there are opportunities for new algorithms, new descriptors (ML inputs), and other novel approaches. There are also challenges for comparing the accuracy and transferability of ML potentials, and in a software sense, for making ML potentials easier to implement, test, and use.

Another challenge LAMMPS faces, which we assume is common to other long-lived, open-source materials and biomolecular modeling codes, MD or otherwise, is how to deal with longevity in a software sense. Core developers move to new jobs or grow old and retire. People who have contributed code may move to other institutions or leave the field or simply disappear (in an email sense). Complex code, whether it be low-level kernels or entire features like the style files or packages described in Section 4.2, may only be well enough understood by the original authors to easily debug, enhance with new capabilities, or adapt when the rest of LAMMPS changes. Facing these realities, how can a large code, which may have hundreds of contributors, be efficiently maintained over the timescale of decades?

A related issue is that complex models, like interatomic potentials with tens of thousands of lines of code, are not static entities. The original developer(s) of a model, who are often not the same person(s) who implement it in a code like LAMMPS, continue to fix bugs and enhance their potential with new capabilities. These upgrades are typically implemented in their private code and described in new papers. How does a user of LAMMPS know which version of the model they are using, or whether bugs in it have been fixed? Should LAMMPS include multiple versions of a long-lived potential? We note that the OpenKIM project [275] is working to address questions like these for interatomic potentials, by providing a software archive which can include multiple time-stamped versions of a potential, and which many MD codes can use and cite correctly (to enable others to reproduce results).

Moving forward, LAMMPS itself is also attempting to address these issues, not only for potentials it implements natively, but for other complex features which similarly evolve over time. We hope to do this while continuing to rapidly enhance LAMMPS with new models and algorithms contributed by a community of creative domain experts and computational scientists. Our goal is for the code to thus continue to be a vibrant and productive tool for its users.

CRediT authorship contribution statement

APT and SJP Conceptualization, Methodology, Software, Supervision, Writing. **All Authors:** Methodology, Software, Writing.

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.

Acknowledgements

We thank the many people who have contributed code and expertise to LAMMPS to help make it a broad and powerful tool. We have mentioned some of them in the text or footnotes or cited their papers, but unfortunately many other contributions could not be explicitly recognized, due to space limitations. See the Authors page https://www.lammps.org/authors.html on the LAMMPS website for a list of significant contributors.

Much of the recent work on LAMMPS described in this paper was supported by the the EXAALT and CoPA projects within the Exascale Computing Project (No. 17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

GPU benchmarking described in Section 5.1 was performed using the Lassen machine at Lawrence Livermore National Laboratory.

[22] Daniel Schwen (Los Alamos National Laboratory) implemented the compute voronoi/atom command in LAMMPS.

[23] Charlie Sievers (UC Davis) added the finite-difference options to the PHONON package in LAMMPS.

[24] See a list of such tools at https://www.lammps.org/prepost.html.

The load-balancing results in Table 2 used resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office of Science User Facility operated under contract DE-AC02-06CH11357.

This work was performed, in part, at the Center for Integrated Nanotechnologies, an Office of Science User Facility operated for the U.S. Department of Energy (DOE) Office of Science.

This work was supported in part by the Office of Fusion Energy Sciences program "Scientific Machine Learning and Artificial Intelligence."

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Appendix A. User support

This appendix describes several ways to support users of an open-source code which we have had success with, especially for those who wish to add code to LAMMPS, either for their own use or public release.

- **Documentation** No contributed feature is released in LAMMPS unless it is also documented. The manual has a Programmer's Guide section which describes the various ways to understand and extend the source code. To make authoring documentation quick and easy for non-experts, while still producing high quality web-pages and documents, we use plain-text files with the lightweight reStructuredText (RST) markup language, then convert them to formats like HTML or PDF using Sphinx. Selected source code is also documented with docstrings (Python) or Doxygen-style comments (C++) and the resulting documentation extracted and imported into Sphinx and included in the Programmer's guide.
- **Mailing list and Forum** Users can submit queries by email to a mailing list: lammps-users@lists.sourceforge.net. Over many years, this has been our most effective way to address questions, either from users or developers. It is also where many bugs have been reported and new features requested. The mailing list currently has a searchable archive of around 27K (thousand) threads with 90K messages, which is a rich (though unorganized) resource of information for users. Recently, LAMMPS joined the Materials Science Community Discourse forum [276] as an alternative means for users to post questions as well as have fruitful cross-interactions with the user communities of other materials modeling codes and tools.
- **Make and CMake** Building LAMMPS can be challenging for some users and on some platforms, due to its many build options and packages, some of which use external libraries. To help address this, we have worked to streamline the traditional GNU *make* build system, adding scripts and tools that require fewer (if any) modifications and customizations. We have also recently added support for building via *CMake*, which automates detection of system software and support libraries, as well as handling many external dependencies and optional features. This simplifies customization of the LAMMPS compilation.[25] *CMake* also adds support for parallel build tools like Ninja, can generate project files suitable for use with integrated development environments (IDEs), and provides the facility for building and executing unit tests.
- **Pre-compiled binary packages** The addition of a *CMake* build capability simplified the integration of LAMMPS into various infrastructures for distributing pre-compiled LAMMPS binaries that integrate cleanly into the host operating system. This includes CentOS, Conda, Fedora, FreeBSD, Homebrew, Spack, Ubuntu, and Windows.[26] For better integration into desktop environments and to support exploring and learning to work with LAMMPS, a *LAMMPS Shell* program is included, that adds enhancements for interactive use (line editing, TAB-completion, command history, non-fatal errors, compatibility with drag-n-drop) to the LAMMPS executable. Docker container images with a ready-to-use LAMMPS installation are also available. For developers there are pre-configured Singularity container images that have all prerequisites installed and configured to build LAMMPS. Those images are used by the LAMMPS developers for automated testing.
- **GitHub** For the last several years all LAMMPS development has been managed on GitHub [5]. For users making code contributions this has several advantages. Each code contribution or issue reported has its own URL for on-topic discussions and comments to the contributed or modified source code. The LAMMPS developers can request changes. Contributors can ask questions, receive answers, and iterate on their contributed code. GitHub tracks the history of the contribution and allows developers to inspect or download it. This information is not lost when a pull request is merged or an issue closed, but can be searched and reviewed at a later time, if needed.
- **Attribution of code contributions** The following are ways we try to insure that people who contribute code to LAMMPS receive appropriate credit. Contributors of a new feature to LAMMPS, either as new style files or a significant enhancement to an existing file, have their name added to a contribution header at the top of the source file, as well as to the Authors page of the LAMMPS website https://www.lammps.org/authors.html. New packages include README files which can give more details about the author(s), including contact information. If there is a journal publication associated with the new feature, the source code can include citation information, and LAMMPS will print it to the screen and a logfile whenever a simulation is run that uses the feature. GitHub also provides a direct means of assigning credit for code contributions via its history tracking of code commits.
- **Testing of code contributions** Using GitHub has also facilitated three kinds of automated testing which benefits users generally and contributors of new code specifically. First, there is integration testing, where new or modified code is compiled on different Linux platforms and with different compilers, compile-time settings, and inclusion of different packages. Second, there is unit testing, where separate test programs are compiled and linked against the LAMMPS library. These test selected classes of functionality or individual functions without running a full LAMMPS simulation. Some of the tests consist of tools generating abbreviated simulations with a variety of settings (including the use of accelerated styles) that are tested against reference data. Third, there is regression testing where a suite of example input decks is run on one or more processors and relevant parts of their output is compared to previous results that are considered correct. The

[25] Christoph Junghans (Los Alamos National Laboratory) worked closely with the LAMMPS developers to enable the use of *CMake* for LAMMPS.

[26] Several of the Linux distributions are made available through the efforts of Christoph Junghans (Los Alamos National Laboratory) and Anton Gladky (Debian).

Codecov toolset is used to track which parts of the LAMMPS source code are covered by the automated testing.

- **External packages or other software tools** As explained in Section 4.2 the LAMMPS source code has sub-directories for *packages* which are collections of style files with a common theme. Any package can be optionally included in a LAMMPS build. Package styles can also use libraries which can be included in the LAMMPS distribution in the *lib* folder, or can be an external library which the *make* or *CMake* build system may download and build. However, some contributors prefer to distribute their add-on package themselves, including any related libraries they have written. Their code may be usable independent of LAMMPS, they may have their own website and distribution mechanisms, or they may wish to change and enhance their code often without interacting with the LAMMPS GitHub procedures. We fully support this mode of creating code which works within LAMMPS. As long as no changes to the core LAMMPS code are required, a user who downloads LAMMPS and also the external add-on package can copy the source files of the package into the source directory of LAMMPS and then configure and compile the code. LAMMPS also supports a plugin mechanism, where optional styles and commands may be compiled independently as shared objects and then loaded into LAMMPS at runtime. Code that is maintained outside the LAMMPS distribution requires the package author to keep their code up-to-date with changes made to the core LAMMPS code. With few exceptions, those changes are minimal. Both external packages as well as stand-alone third-party software tools (both open source and commercial) which work in tandem with LAMMPS are listed on the LAMMPS website.
- **Slack work space** This is a new communication mechanism we are experimenting with, aimed specifically at allowing contributors to interact in real-time with the LAMMPS development team and monitor the progress of activities on GitHub or the associated testing tools. LAMMPS contributors can request an invitation to join by sending an email to slack at lammps.org and describing the LAMMPS contribution they are working on.
- **Continuous release philosophy** New features are effectively released to the public as soon as they are merged into the GitHub main branch (“develop”). In principle that branch is always fully functional, but the process is not perfect; induced bugs are sometimes detected later. To shield users or computing centers who don’t wish to stay current with the development branch, we also create “patch” (∼monthly) and “stable” (1–2 times a year) releases (branches within GitHub). Stable releases undergo additional reviews and manual testing.

References

[1] M. Griebel, S. Knapek, G. Zumbusch, Numerical Simulation in Molecular Dynamics, Springer Verlag, Heidelberg Germany, 2007.

[2] J.B. Gibson, A.N. Goland, M. Milgram, G.H. Vineyard, Phys. Rev. 120 (4) (1960) 1229–1253.

[3] A. Rahman, Phys. Rev. 136 (2A) (1964) 405–411.

[4] L. Verlet, Phys. Rev. 159 (1967) 98–103.

[5] S. Plimpton, J. Comp. Phys. 117 (1995) 1–19; LAMMPS website: https://www.lammps.org, 2021; LAMMPS GitHub repository: https://github.com/lammps/lammps, 2021.

[6] D. Perez, B.P. Uberuaga, Y. Shim, J.G. Amar, A.F. Voter, Annu. Rep. Comput. Chem. 5 (2009) 79.

[7] B.E. Husic, V.S. Pande, J. Am. Chem. Soc. 140 (2018) 2386–2396.

[8] Y. Yang, Q. Shao, J. Zhang, L. Yang, Y.Q. Gao, J. Chem. Phys. 151 (2019) 070902.

[9] G. Bussi, A. Laio, Nat. Rev. Phys. 2 (2020) 200–212.

[10] S.J. Plimpton, A.P. Thompson, Mater. Res. Soc. Bull. 37 (2012) 513.

[11] D.A. Case, I.T.E. Cheatham, T. Darden, H. Gohlke, R. Luo, J.K.M. Merz, A. Onufriev, C. Simmerling, B. Wang, R. Woods, J. Comput. Chem. 26 (2005) 1668–1688; Amber website: https://ambermd.org, 2021.

[12] B.R. Brooks, I.C.L. Brooks, J.A.D. MacKerell, L. Nilsson, R.J. Petrella, B. Roux, Y. Won, G. Archontis, C. Bartels, S. Boresch, A. Caflisch, L. Caves, Q. Cui, A.R. Dinner, M. Feig, S. Fischer, J. Gao, M. Hodoscek, W. Im, K. Kuczera, T. Lazaridis, J. Ma, V. Ovchinnikov, E. Paci, R. Pastor, C.B. Post, J.Z. Pu, M. Schaefer, B. Tidor, R.M. Venable, H.L. Woodcock, X. Wu, W. Yang, D.M. York, M. Karplus, J. Comput. Chem. 30 (2009) 1545–1614; CHARMM website: https://www.charmm.org, 2021.

[13] M.J. Abraham, T. Murtola, R. Schulz, S. Páll, J.C. Smith, B. Hess, E. Lindahl, SoftwareX 1–2 (2015) 19–25; GROMACS website: http://www.gromacs.org, 2021.

[14] 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 (2020) 044130; NAMD website: https://www.ks.uiuc.edu/Research/namd, 2021.

[15] I.T. Todorov, W. Smith, K. Trachenko, M.T. Dove, J. Mater. Chem. 16 (2006) 1911–1918; DL_POLY website: https://www.scd.stfc.ac.uk/Pages/DL_POLY.aspx, 2021.

[16] J.A. Anderson, J. Glaser, S.C. Glotzer, Comput. Mater. Sci. 173 (2002) 109363; HOOMD website: http://glotzerlab.engin.umich.edu/hoomd-blue, 2021.

[17] DCS Computing website, https://www.aspherix-dem.com, 2021.

[18] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandie, J. Madsen, N. Al Awar, M. Gligoric, G. Shipman, G. Womeldorff, Comput. Sci. Eng. 23 (5) (2021) 10–18.

[19] C. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D.S. Hollman, D.A. Ibanez, N. Liber, J. Madsen, J.S. Miles, D.Z. Poliakoff, A.J. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, J. Wilke, IEEE Trans. Parallel Distrib. Syst. (2021) 1-1.

[20] S. Meloni, M. Rosati, L. Colombo, J. Chem. Phys. 126 (2007) 121102.

[21] M. Tuckerman, B.J. Berne, G.J. Martyna, J. Chem. Phys. 97 (1992) 1990–2001.

[22] P.J. in ’t Veld, S.J. Plimpton, G.S. Grest, Comput. Phys. Commun. 179 (2008) 320–329.

[23] T. Shire, K.J. Hanley, K. Stratford, Comput. Part. Mech. 8 (2021) 653–663.

[24] R.W. Hockney, J.W. Eastwood, Computer Simulation Using Particles, Adam Hilger, New York, NY, 1988.

[25] E.L. Pollock, J. Glosli, Comput. Phys. Commun. 95 (1996) 93–110.

[26] T. Darden, D. York, L. Pedersen, J. Chem. Phys. 98 (1993) 10089–10092.

[27] M. Deserno, C. Holm, J. Chem. Phys. 100 (1998) 7678.

[28] G. Sutmann, in: A. Bajaj, P. Zavattieri, M. Koslowski, T. Siegmund (Eds.), Proceedings of the Society of Engineering Science 51st Annual Technical Meeting, Purdue University Libraries Scholarly Publishing Services, 2014.

[29] ScaFaCoS website, http://www.scafacos.de, 2021.

[30] S.G. Moore, P.S. Crozier, J. Chem. Phys. 140 (2014) 234112.

[31] U. Essmann, L. Perera, M.L. Berkowitz, T. Darden, H. Lee, L.G. Pedersen, J. Chem. Phys. 103 (1995) 8577–8593.

[32] S.J. Plimpton, R. Pollock, M. Stevens, in: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, 1997.

[33] M. Frigo, S.G. Johnson, Proc. IEEE 93 (2) (2005) 216–231; FFTW website: http://www.fftw.org, 2021.

[34] fftMPI website, https://fftmpi.sandia.gov, 2021.

[35] J.J. Cerda, V. Ballenegger, O. Lenz, C. Holm, J. Chem. Phys. 129 (2008) 234104.

[36] R.E. Isele-Holder, W. Mitchell, A.E. Ismail, J. Chem. Phys. 137 (2012) 174107.

[37] D.M. Duffy, A.M. Rutherford, J. Phys. Condens. Matter 19 (2007) 016207.

[38] A.M. Rutherford, D.M. Duffy, J. Phys. Condens. Matter 20 (2007) 496201–496210.

[39] C.L. Phillips, R.J. Magyar, P.S. Crozier, J. Chem. Phys. 133 (2010) 144711.

[40] S.J. Plimpton, J.D. Gale, Curr. Opin. Solid State Mater. Sci. 17 (2013) 271–276.

[41] mpi4py website, https://mpi4py.readthedocs.io, 2021.

[42] A. Jaramillo-Botero, J. Su, A. Qi, W.A. Goddard III, J. Comput. Chem. 32 (2011) 497–512.

[43] A.P. Thompson, S.J. Plimpton, W. Mattson, J. Chem. Phys. 131 (2009) 154107.

[44] D. Surblys, H. Matsubara, G. Kikugawa, T. Ohara, Phys. Rev. E 99 (2019) 051301.

[45] F.H. Streitz, J.W. Mintmire, Phys. Rev. B 50 (1994) 11996–12003.

[46] B.V. Derjaguin, L. Landau, Acta Physicochim. USSR 14 (1941) 633–662.

[47] E.J.W. Verwey, J.T.G. Overbeek, Theory of the Stability of Lyophobic Colloids, Dover, 1999.

[48] J. Finkelstein, G. Fiorin, B. Seibold, Mol. Phys. 118 (6) (2020) e1649493.

[49] R.E. Jones, J. Templeton, J. Zimmerman, in: Multiscale Materials Modeling for Nanomechanics, Springer, 2016, pp. 223–259.

[50] N. Bock, M.J. Cawkwell, J.D. Coe, A. Krishnapriyan, M.P. Kroonblawd, A. Lang, C. Liu, E.M. Saez, S.M. Mniszewski, C.F.A. Negre, A.M.N. Niklasson, E. Sanville, M.A. Wood, P. Yang, LATTE, https://github.com/lanl/LATTE, 2021.

[51] C.H. Rycroft, Chaos 19 (2009) 041111.

[52] Voro++ website, http://math.lbl.gov/voro++, 2021.

[53] B. Frantzdale, S.J. Plimpton, M.S. Shephard, Eng. Comput. 26 (2010) 205–211.

[54] D.M. Beazley, Future Gener. Comput. Syst. 19 (2003) 599–609; SWIG website: http://www.swig.org, 2021.

[55] B. Hourahine, B. Aradi, V. Blum, F. Bonafé, A. Buccheri, C. Camacho, C. Cevallos, M.Y. Deshaye, T. Dumitrică, A. Dominguez, S. Ehlert, M. Elstner, T. van der Heide, J. Hermann, S. Irle, J.J. Kranz, C. Köhler, T. Kowalczyk, T. Kubař, I.S. Lee, V. Lutsker, R.J. Maurer, S.K. Min, I. Mitchell, C. Negre, T.A. Niehaus, A.M.N. Niklasson, A.J. Page, A. Pecchia, G. Penazzi, M.P. Persson, J. Řezáč, C.G. Sánchez, M. Sternberg, M. Stöhr, F. Stuckenberg, A. Tkatchenko, V.W. z. Yu, T. Frauenheim, J. Chem. Phys. 152 (2020) 124101.

[56] DFTB+ website, https://www.dftbplus.org, 2021.
[57] N. Goldman, I. Tamblyn, J. Phys. Chem. A 117 (2013) 5124.
[58] A.Z. Panagiotopoulos, Mol. Phys. 61 (1987) 813–826.
[59] E. Martinez, A. Caro, Phys. Rev. B 86 (2012) 214109.
[60] A.H. Larsen, J.J. Mortensen, J. Blomqvist, I.E. Castelli, R. Christensen, M. Dulak, J. Friis, M.N. Groves, B. Hammer, C. Hargus, E.D. Hermes, P.C. Jennings, P.B. Jensen, J. Kermode, J.R. Kitchin, E.L. Kolsbjerg, J. Kubal, K. Kaasbjerg, S. Lysgaard, J.B. Maronsson, T. Maxson, T. Olsen, L. Pastewka, A. Peterson, C. Rostgaard, J. Schiotz, O. Schutt, M. Strange, K.S. Thygesen, T. Vegge, L. Vilhelmsen, M. Walter, Z.H. Zeng, K.W. Jacobsen, J. Phys. Condens. Matter 29 (2017) 273002.
[61] ASE website, https://wiki.fysik.dtu.dk/ase, 2021. A detailed description of how ASE Python scripts can use the LAMMPS library interface is described here: https://wiki.fysik.dtu.dk/ase/ase/calculators/lammps.html.
[62] FitSNAP website, https://github.com/FitSNAP/FitSNAP, 2021.
[63] D. Perez, E.D. Cubuk, A. Waterland, E. Kaxiras, A.F. Voter, J. Chem. Theory Comput. 12 (2016) 8–28.
[64] R. Huang, W. Yuhua, A.F. Voter, D. Perez, Phys. Rev. Mater. 2 (2018) 126002.
[65] R. Perriot, B.P. Uberuaga, R.J. Zamora, D. Perez, A.F. Voter, Nat. Commun. 8 (2017) 618.
[66] CSlib website, https://cslib.sandia.gov, 2021.
[67] T.A. Barnes, E. Marin-Rimoldi, S. Ellis, T.D. Crawford, Comput. Phys. Commun. 261 (2021) 107688.
[68] MolSSI MDI website, https://molssi-mdi.github.io/MDI_Library, 2021.
[69] D. Smith, D. Altarawy, L. Burns, M. Welborn, L. Naden, L. Ward, S. Ellis, T. Crawford, Wiley Interdiscip. Rev.: Comput. Mol. Sci. 11 (2020) e1491.
[70] G. Henkelman, B.P. Uberuaga, H. Jonsson, J. Chem. Phys. 113 (2000) 9901–9904.
[71] W.M. Brown, J.-M.Y. Carrillo, N. Gavhane, F.M. Thakkar, S.J. Plimpton, Comput. Phys. Commun. 195 (2015) 95–101.
[72] W.M. Brown, P. Wang, S.J. Plimpton, A.N. Tharrington, Comput. Phys. Commun. 182 (4) (2011) 898–911.
[73] W.M. Brown, A. Kohlmeyer, S.J. Plimpton, A.N. Tharrington, Comput. Phys. Commun. 183 (2012) 449–459.
[74] W.M. Brown, M. Yamada, Comput. Phys. Commun. 184 (12) (2013) 2785–2793.
[75] N. Kondratyuk, V. Nikolskiy, D. Pavlov, V. Stegailov, Int. J. HPC Appl. 35 (2021) 312–324.
[76] Kokkos website, https://github.com/kokkos/kokkos, 2021.
[77] G.S. Grest, B. Dünweg, K. Kremer, Comput. Phys. Commun. 55 (1989) 269–285.
[78] R. Berardi, C. Fava, C. Zannoni, Chem. Phys. Lett. 297 (1998) 8–14.
[79] T.D. Nguyen, J.-M.Y. Carrillo, M.A. Matheson, W.M. Brown, Nanoscale 6 (6) (2014) 3083–3096.
[80] L.A. Zepeda-Ruiz, A. Stukowski, T. Oppelstrup, V.V. Bulatov, Nature 550 (2017) 492.
[81] M.A. Wood, D.E. Kittell, C.D. Yarrington, A.P. Thompson, Phys. Rev. B 97 (2018) 014109.
[82] M. Cusentino, M. Wood, A. Thompson, Nucl. Fusion 61 (4) (2021) 046049.
[83] K.N. Cong, J. Willman, S. Moore, A. Belonoshko, R. Gayatri, E. Weinberg, M.A. Wood, A.P. Thompson, I. Oleynik, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC21, IEEE Press, 2021.
[84] S.J. Plimpton, C. Knight, J. Parallel Distrib. Comput. 147 (2021) 184–195.
[85] M.J. Berger, S.H. Bokhari, IEEE Trans. Comput. C-36 (5) (1987) 570–580.
[86] C. Begau, G. Sutmann, Comput. Phys. Commun. 190 (2015) 51–61.
[87] J.L. Fattebert, D.F. Richards, J.N. Glosli, Comput. Phys. Commun. 183 (2012) 2608–2615.
[88] S.J. Plimpton, S.G. Moore, A. Borner, A.K. Stagg, T.P. Koehler, J.R. Torczynski, M.A. Gallis, Phys. Fluids 31 (2019) 086101.
[89] J.A. Bollinger, M.J. Stevens, Soft Matter 14 (10) (2018) 1748–1752.
[90] A. Stukowski, Model. Simul. Mater. Sci. Eng. 18 (2010) 015012; OVITO website: https://www.ovito.org, 2021.
[91] W. Humphrey, A. Dalke, K. Schulten, J. Mol. Graph. 14 (1996) 33–38.
[92] M.A. Gallis, T.P. Koehler, J.R. Torczynski, S.J. Plimpton, Phys. Fluids 27 (2015) 084105.
[93] K. Kadau, T.C. Germann, P.S. Lomdahl, Int. J. Mod. Phys. C 15 (2004) 193–201.
[94] J. Tersoff, Phys. Rev. B 37 (1988) 6991.
[95] D.W. Brenner, O.A. Shenderova, J.A. Harrison, S.J. Stuart, B. Ni, S.B. Sinnott, J. Phys. Condens. Matter 14 (2002) 783–802.
[96] S.J. Stuart, A.B. Tutein, J.A. Harrison, J. Chem. Phys. 112 (2000) 6472–6486.
[97] T.-R. Shan, B.D. Devine, T.W. Kemper, S.B. Sinnott, S.R. Phillpot, Phys. Rev. B 81 (2010) 125328.
[98] T. Liang, T.-R. Shan, Y.-T. Cheng, B.D. Devine, M. Noordhoek, Y. Li, Z. Lu, S.R. Phillpot, S.B. Sinnott, Mat. Sci. Eng. 74 (2013) 255–279.
[99] K.K. Chenoweth, A.C.T. van Duin, W.A. Goddard, J. Phys. Chem. A 112 (2008) 1040–1053.
[100] M.S. Daw, M.I. Baskes, Phys. Rev. B 29 (1984) 6443–6453.
[101] M.I. Baskes, Phys. Rev. B 46 (1992) 2727–2742.
[102] X.W. Zhou, F.P. Doty, Phys. Rev. B 78 (2008) 224307.
[103] D.G. Pettifor, I.I. Oleinik, Phys. Rev. B 59 (1999) 8487–8499.
[104] J. Murdick, X.W. Zhou, H.N.G. Wadley, D. Nguyen-Manh, R. Drautz, D.G. Pettifor, Phys. Rev. B 73 (2006) 045206.
[105] D.K. Ward, X.W. Zhou, B.M. Wong, F.P. Doty, J.A. Zimmerman, Phys. Rev. B 85 (2012) 115206.
[106] A. Jaramillo-Botero, J. Su, A. Qi, W.A. Goddard, J. Comput. Chem. 32 (2011) 497–512.
[107] Y. Zuo, C. Chen, X. Li, Z. Deng, Y. Chen, J. Behler, G. Csányi, A.V. Shapeev, A.P. Thompson, M.A. W̃ood, S.P. Ong, J. Phys. Chem. A 124 (4) (2020) 731–745.
[108] E.B. Tadmor, R.S. Elliott, J.P. Sethna, R.E. Miller, C.A. Becker, JOM 63 (2011) 17.
[109] OpenKIM website, https://openkim.org, 2021.
[110] M. Bishop, M.H. Kalos, H.L. Frisch, J. Chem. Phys. 70 (1979) 1299–1304.
[111] K. Kremer, G.S. Grest, J. Chem. Phys. 92 (8) (1990) 5057–5086.
[112] R. Goetz, R. Lipowsky, J. Chem. Phys. 108 (1998) 7397–7409.
[113] M.J. Stevens, J. Chem. Phys. 121 (23) (2004) 11942–11948.
[114] W. Shinoda, R. DeVane, M.L. Klein, Mol. Simul. 33 (2007) 27–36.
[115] M.L. Klein, W. Shinoda, Science 321 (5890) (2008) 798–800.
[116] T.E. Ouldridge, A.A. Louis, J.P.K. Doye, J. Chem. Phys. 134 (2011) 085101.
[117] P. Šulc, F. Romano, T.E. Ouldridge, L. Rovigatti, J.P.K. Doye, A.A. Louis, J. Chem. Phys. 137 (2012) 135101.
[118] O. Henrich, Y.A.G. Fosado, T. Curk, T.E. Ouldridge, Eur. Phys. J. E 41 (2018) 57.
[119] K. Kremer, G.S. Grest, J. Chem. Phys. 92 (1990) 5057–5086.
[120] T. Ge, F. Pierce, D. Perahia, G.S. Grest, M.O. Robbins, Phys. Rev. Lett. 110 (2013) 098301.
[121] M.J. Stevens, J. Chem. Phys. 149 (17) (2018) 174905.
[122] F. Milde, G. Tauriello, H. Haberkern, P. Koumoutsakos, Comput. Part. Mech. 1 (2014) 211–227.
[123] K. Lykov, Y. Nematbakhsh, M. Shang, C.T. Lim, I.V. Pivkin, PLOS Comput. Biol. 13 (2017) e1005726.
[124] G. Voth, Coarse-Graining of Condensed Phase and Biomolecular Systems, CRC Press/Taylor and Francis Group, Boca Raton, FL, 2009.
[125] W. Noid, J. Chem. Phys. 139 (2013) 090901.
[126] M.S. Shell, Adv. Chem. Phys. 161 (2016) 395–441.
[127] Z. Li, X. Bian, X. Yang, G. Karniadakis, J. Chem. Phys. 145 (2016) 044102.
[128] G. Milano, F. Müller-Plathe, J. Phys. Chem. B 109 (39) (2005) 18609–18619.
[129] V. Rühle, C. Junghans, A. Lukyanov, K. Kremer, D. Andrienko, J. Chem. Theory Comput. 5 (2009) 3211.
[130] S.Y. Mashayak, M.N. Jochum, K. Koschke, N.R. Aluru, V. Rühle, C. Junghans, PLOS ONE 10 (2015) e131754.
[131] Votca website, https://www.votca.org, 2021.
[132] W.G. Noid, J.-W. Chu, G.S. Ayton, V. Krishna, S. Izvekov, G.A. Voth, A. Das, H.C. Andersen, J. Chem. Phys. 128 (2008) 244114.
[133] W.G. Noid, P. Liu, Y. Wang, J.-W. Chu, G.S. Ayton, S. Izvekov, H.C. Andersen, G.A. Voth, J. Chem. Phys. 128 (2008) 244115.
[134] MSCG website, https://github.com/uchicago-voth/MSCG-release, 2021.
[135] A. Yu, A.J. Pak, P. He, V. Monje-Galvan, L. Casalino, Z. Gaieb, A.C. Dommer, R.E. Amaro, G.A. Voth, Biophys. J. 120 (2021) 1097–1104.
[136] P.J. in 't Veld, S.J. Plimpton, G.S. Grest, Comput. Phys. Commun. 179 (2008) 320–329.
[137] R. Everaers, M.R. Ejtehadi, Phys. Rev. E 67 (2003) 041710.
[138] A. Malevanets, R. Kapral, J. Phys. Chem. 110 (1999) 8605–8613.
[139] J.T. Padding, A.A. Louis, Phys. Rev. E 74 (2006) 402.
[140] A. Kumar, J.J.L. Higdon, Phys. Rev. E 82 (2010) 051401.
[141] F.E. Mackay, S.T.T. Ollila, C. Denniston, Comput. Phys. Commun. 184 (2013) 2021–2031.
[142] D.S. Bolintineanu, G.S. Grest, J.B. Lechman, F. Pierce, S.J. Plimpton, P.R. Schunk, Comput. Part. Mech. 1 (2014) 321–356.
[143] P. Hoogerbrugge, J. Koelman, Europhys. Lett. 19 (1992) 155.
[144] Z. Li, Y.-H. Tang, H. Lei, B. Caswell, G.E. Karniadakis, J. Comp. Phys. 265 (2014) 113–127.
[145] Z. Li, G.H. Hu, Z.L. Wang, Y.B. Ma, Z.W. Zhou, Phys. Fluids 25 (2013) 072103.
[146] Z. Li, A. Yazdani, A. Tartakovsky, G.E. Karniadakis, J. Chem. Phys. 143 (2015) 014101.
[147] T.I. Mattox, J.P. Larentzos, S.G. Moore, C.P. Stone, D.A. Ibanez, A.P. Thompson, M. Lísal, J.K. Brennan, S.J. Plimpton, Mol. Phys. 116 (2018) 2061–2069.
[148] T.D. Nguyen, S.J. Plimpton, Comput. Phys. Commun. 243 (2019) 12–24.
[149] A.P. Santos, D.S. Bolintineanu, G.S. Grest, J.B. Lechman, S.J. Plimpton, I. Srivastava, L.E. Silbert, Phys. Rev. E (2020) 032903102.
[150] K.M. Salerno, D.S. Bolintineanu, G.S. Grest, J.B. Lechman, S.J. Plimpton, I. Srivastava, L.E. Silbert, Phys. Rev. E 98 (2018) 050901.
[151] F.Y. Fraige, P.A. Langston, A.J. Matchett, J. Dodds, Particuology 6 (2008) 455–466.
[152] J. Wang, H.S. Yu, P. Langston, F. Fraige, Granul. Matter 13 (2011) 1–12.
[153] C. Kloss, C. Goniva, A. Hager, S. Amberger, S. Pirker, Prog. Comput. Fluid Dyn. 12 (2012) 140–152.
[154] J.J. Monaghan, Annu. Rev. Fluid Mech. 44 (2012) 323–346.
[155] P. Espanol, M. Revenga, Phys. Rev. E 67 (2003) 026705.
[156] S. Leroch, M. Varga, S.J. Eder, A. Vernes, M.R. Ripoll, G. Ganzenmüller, Int. J. Solids Struct. 81 (2016) 188–202.
[157] F. Bobaru, J.T. Foster, P.H. Geubelle, S.A. Silling, Handbook of Peridynamic Modeling, CRC Press, 2016.

[158] M.L. Parks, R.B. Lehoucq, S.J. Plimpton, S.A. Silling, Comput. Phys. Commun. 179 (2008) 777–783.
[159] J.T. Foster, S.A. Silling, W.W. Chen, Int. J. Numer. Methods Eng. 81 (2010) 1242–1258.
[160] Y.D. Ha, F. Bobaru, Eng. Fract. Mech. 78 (6) (2011) 1156–1168.
[161] S.A. Silling, M.L. Parks, J.R. Kamm, O. Weckner, M. Rassaian, Int. J. Impact Eng. 107 (2017) 47–57.
[162] S.A. Silling, M. Fermen-Coker, Theor. Appl. Fract. Mech. 113 (2021) 102947.
[163] R. Gayatri, S. Moore, E. Weinberg, N. Lubbers, S. Anderson, J. Deslippe, D. Perez, A.P. Thompson, arXiv e-prints, arXiv:2011.12875, 2020.
[164] A. Singraber, T. Morawietz, J. Behler, C. Dellago, J. Chem. Theory Comput. 15 (5) (2019) 3075–3092.
[165] J. Behler, M. Parrinello, Phys. Rev. Lett. 98 (14) (2007) 146401.
[166] N2P2 website, https://compphysvienna.github.io/n2p2, 2021.
[167] A. Bartók, M.C. Payne, K. Risi, G. Csányi, Phys. Rev. Lett. 104 (2010) 136403.
[168] QUIP website, https://github.com/libAtoms/QUIP, 2021.
[169] A.P. Thompson, L.P. Swiler, C.R. Trott, S.M. Foiles, G.J. Tucker, J. Comput. Phys. 285 (2015) 316–330.
[170] M.A. Wood, A.P. Thompson, J. Chem. Phys. 148 (24) (2018) 241721.
[171] Nicholas Lubbers, 2020, private communication.
[172] H. Yanxon, D. Zagaceta, B.C. Wood, Q. Zhu, J. Chem. Phys. 153 (5) (2020) 054118.
[173] M.A. Cusentino, M.A. Wood, A.P. Thompson, J. Phys. Chem. A 124 (26) (2020) 5456–5464.
[174] mlearn website, https://github.com/materialsvirtuallab/mlearn, 2021.
[175] A.V. Shapeev, Multiscale Model. Simul. 14 (2016) 1153.
[176] MTP website. http://gitlab.skoltech.ru/shapeev/mlip-dev, 2021.
[177] R. Drautz, Phys. Rev. B 102 (2020) 024104.
[178] Y. Lysogorskiy, C. van der Oord, A. Bochkarev, S. Menon, M. Rinaldi, T. Hammerschmidt, M. Mrovec, A. Thompson, G. Csányi, C. Ortner, R. Drautz, npj Comput. Mater. 7 (1) (2021) 97.
[179] PACE website, https://github.com/ICAMS/lammps-user-pace, 2021.
[180] H. Wang, L. Zhang, J. Han, W. E, Comput. Phys. Commun. 228 (2018) 178–184.
[181] DeepMD website, https://github.com/deepmodeling/deepmd-kit, 2021.
[182] M. Wen, E.B. Tadmor, npj Comput. Mater. 6 (1) (2020) 124.
[183] M. Wen, E.B. Tadmor, Phys. Rev. B 100 (2019) 195419.
[184] R. Lot, F. Pellegrini, Y. Shaidu, E. Küçükbenli, Comput. Phys. Commun. 256 (2020) 107402.
[185] J.S. Smith, B. Nebgen, N. Mathew, J. Chen, N. Lubbers, L. Burakovsky, S. Tretiak, H.A. Nam, T. Germann, S. Fensin, K. Barros, Nat. Commun. 12 (1) (2021) 1–13.
[186] V. Botu, R. Ramprasad, Int. J. Quant. Chem. 115 (16) (2015) 1074–1083.
[187] H. Mori, T. Ozaki, Phys. Rev. Mater. 4 (4) (2020) 040601.
[188] D. Dickel, M. Nitol, C. Barrett, Comput. Mater. Sci. 196 (2021) 110481.
[189] G.P. Pun, R. Batra, R. Ramprasad, Y. Mishin, Nat. Commun. 10 (1) (2019) 1–10.
[190] A. Seko, A. Togo, I. Tanaka, Phys. Rev. B 99 (21) (2019) 214108.
[191] W.J. Mortier, K. Vangenechten, J. Gasteiger, J. Am. Chem. Soc. 107 (4) (1985) 829–835.
[192] W.J. Mortier, S.K. Ghosh, S. Shankar, J. Am. Chem. Soc. 108 (15) (1986) 4315–4320.
[193] A.K. Rappe, W.A. Goddard, J. Phys. Chem. 95 (8) (1991) 3358–3363.
[194] R.G. Parr, R.A. Donnelly, M. Levy, W.E. Palke, J. Chem. Phys. 68 (8) (1978) 3801–3807.
[195] R.T. Sanderson, J. Am. Chem. Soc. 105 (8) (1983) 2259–2261.
[196] S.W. Rick, S.J. Stuart, B.J. Berne, J. Chem. Phys. 101 (7) (1994) 6141–6156.
[197] J. Mintmire, D. Robertson, C. White, Phys. Rev. B 49 (21) (1994) 14859–14864.
[198] M.S. Daw, M.I. Baskes, Phys. Rev. B 29 (12) (1984) 6443–6453.
[199] X.W. Zhou, H.N.G. Wadley, J.S. Filhol, M.N. Neurock, Phys. Rev. B 69 (3) (2004).
[200] M.I. Baskes, J.S. Nelson, A.F. Wright, Phys. Rev. B 40 (9) (1989) 6085–6100.
[201] V. Swamy, J. Muscat, J.D. Gale, N.M. Harrison, Surf. Sci. 504 (1–3) (2002) 115–124.
[202] B.S. Thomas, N.A. Marks, B.D. Begg, Phys. Rev. B 69 (14) (2004).
[203] A. Hallil, R. Tetot, F. Berthier, I. Braems, J. Creuze, Phys. Rev. B 73 (16) (2006).
[204] J. Yu, S.B. Sinnott, S.R. Phillpot, Phys. Rev. B 75 (8) (2007).
[205] T.-R. Shan, B.D. Devine, J.M. Hawkins, A. Asthagiri, S.R. Phillpot, S.B. Sinnott, Phys. Rev. B 82 (23) (2010).
[206] T. Liang, T.-R. Shan, Y.-T. Cheng, B.D. Devine, M. Noordhoek, Y. Li, Z. Lu, S.R. Phillpot, S.B. Sinnott, Mater. Sci. Eng. R 74 (9) (2013) 255–279.
[207] A.C.T. van Duin, S. Dasgupta, F. Lorant, W. Goddard, J. Phys. Chem. A 105 (41) (2001) 9396–9409.
[208] A.C.T. van Duin, A. Strachan, S. Stewman, Q.S. Zhang, X. Xu, W. Goddard, J. Phys. Chem. A 107 (19) (2003) 3803.
[209] H.M. Aktulga, J.C. Fogarty, S.A. Pandit, A.Y. Grama, Parallel Comput. 38 (4–5) (2012) 245–259.
[210] R. Car, M. Parrinello, Phys. Rev. Lett. 55 (1985) 2471–2474.
[211] E. Bitzek, P. Koskinen, F. Gaehler, M. Moseler, P. Gumbsch, Phys. Rev. Lett. 97 (17) (2006).
[212] D. Dragoni, T.D. Daff, G. Csányi, N. Marzari, Phys. Rev. Mater. 2 (1) (2018) 013808.
[213] Y. Zhou, J. Tranchida, Y. Ge, J. Murthy, T.S. Fisher, Phys. Rev. B 101 (22) (2020) 224303.
[214] G. Dos Santos, R. Aparicio, D. Linares, E. Miranda, J. Tranchida, G. Pastor, E. Bringa, Phys. Rev. B 102 (18) (2020) 184426.
[215] M. Jaime, A. Saul, M. Salamon, V. Zapf, N. Harrison, T. Durakiewicz, J. Lashley, D. Andersson, C. Stanek, J. Smith, K. Gofryk, Nat. Commun. 8 (1) (2017) 1–7.
[216] P. Nieves, J. Tranchida, S. Arapan, D. Legut, Phys. Rev. B 103 (9) (2021) 094437.
[217] M.P. Surh, L.X. Benedict, B. Sadigh, Phys. Rev. Lett. 117 (8) (2016) 085701.
[218] K.A. Gschneidner Jr, Y. Mudryk, V.K. Pecharsky, Scr. Mater. 67 (6) (2012) 572–577.
[219] P.-W. Ma, S. Dudarev, Phys. Rev. B 90 (2) (2014) 024425.
[220] J. Tranchida, S. Plimpton, P. Thibaudeau, A.P. Thompson, J. Comput. Phys. 372 (2018) 406–425.
[221] V. Antropov, M. Katsnelson, B. Harmon, M. Van Schilfgaarde, D. Kusnezov, Phys. Rev. B 54 (2) (1996) 1019.
[222] O. Eriksson, A. Bergman, L. Bergqvist, J. Hellsvik, Atomistic Spin Dynamics: Foundations and Applications, Oxford University Press, 2017.
[223] P.-W. Ma, C. Woo, S. Dudarev, Phys. Rev. B 78 (2) (2008) 024434.
[224] P.-W. Ma, S. Dudarev, in: Handbook of Materials Modeling: Methods: Theory and Modeling, 2020, pp. 1017–1035.
[225] I. Omelyan, I. Mryglod, R. Folk, Phys. Rev. Lett. 86 (5) (2001) 898.
[226] R.F. Evans, in: Handbook of Materials Modeling: Applications: Current and Emerging Materials, 2020, pp. 427–448.
[227] A.V. Ivanov, V.M. Uzdin, H. Jónsson, Comput. Phys. Commun. 260 (2021) 107749.
[228] A.V. Ivanov, D. Dagbjartsson, J. Tranchida, V.M. Uzdin, H. Jónsson, J. Phys. Condens. Matter 32 (2020) 345901.
[229] J.-Y. Chauleau, T. Chirac, S. Fusil, V. Garcia, W. Akhtar, J. Tranchida, P. Thibaudeau, I. Gross, C. Blouzon, A.e. Finco, M. Bibes, B. Dkhil, D.D. Khalyavin, P. Manuel, V. Jacques, N. Jaouen, M. Viret, Nat. Mater. 19 (4) (2020) 386–390.
[230] P.F. Bessarab, V.M. Uzdin, H. Jónsson, Comput. Phys. Commun. 196 (2015) 335–347.
[231] V.M. Uzdin, M.N. Potkina, I.S. Lobanov, P.F. Bessarab, H. Jónsson, J. Magn. Magn. Mater. 459 (2018) 236–240.
[232] Y. Sugita, Y. Okamoto, Chem. Phys. Lett. 314 (1999) 141–151.
[233] A.F. Voter, Phys. Rev. B 57 (1988) R13985.
[234] M.R. Sørenson, A.F. Voter, J. Chem. Phys. 112 (2000) 9599.
[235] S.J. Plimpton, D. Perez, A.F. Voter, J. Chem. Phys. (2020).
[236] A.F. Voter, J. Chem. Phys. 106 (1997) 4665.
[237] S.Y. Kim, D. Perez, A.F. Voter, J. Chem. Phys. 139 (2013) 144110.
[238] G. Fiorin, M.L. Klein, J. Henin, Mol. Phys. 111 (2013) 3345–3362.
[239] G.A. Tribello, M. Bonomi, D. Branduardi, C. Camilloni, G. Bussi, Comput. Phys. Commun. 185 (2014) 604–613.
[240] D.J. Evans, G.P. Morriss, Statistical Mechanics of Nonequilibrium Liquids, Academic Press, 1990.
[241] F. Muller-Plathe, Phys. Rev. E 59 (1999) 4894–4898.
[242] F. Muller-Plathe, J. Chem. Phys. 106 (1997) 6082–6085.
[243] M. Dobson, J. Chem. Phys. 141 (2014) 184103.
[244] D.J. Evans, G.P. Morriss, Phys. Rev. A 30 (1984) 1528.
[245] D.A. Nicholson, G.C. Rutledge, J. Chem. Phys. 145 (24) (2016) 244903.
[246] T.C. O'Connor, N.J. Alvarez, M.O. Robbins, Phys. Rev. Lett. 121 (2018) 047801.
[247] D. Frenkel, B. Smit, Understanding Molecular Simulation: From Algorithms to Applications, 2nd edition, Elsevier, 2001.
[248] G.S. Heffelfinger, F.v. Swol, J. Chem. Phys. 100 (10) (1994) 7548–7552.
[249] D.A. Kofke, E.D. Glandt, Mol. Phys. 64 (6) (1988) 1105–1131.
[250] T. Curk, E. Luijten, Phys. Rev. Lett. 126 (13) (2021) 138003.
[251] R. Auhl, R. Everaers, G.S. Grest, K. Kremer, S.J. Plimpton, J. Chem. Phys. 119 (2003) 12728.
[252] J.R. Gissinger, B.D. Jensen, K.E. Wise, Polymer 128 (2017) 211–217.
[253] J.R. Gissinger, B.D. Jensen, K.E. Wise, Macromolecules 53 (2020) 9953–9961; Reacter website: https://www.reacter.org, 2021.
[254] R. Ravelo, B.L. Holian, T.C. Germann, P.S. Lomdahl, Phys. Rev. B 70 (2004) 014103.
[255] E.J. Reed, L.E. Fried, J.D. Joannopoulos, Phys. Rev. Lett. 90 (2003) 235503.
[256] P.J. Mitchell, D. Fincham, J. Phys. Condens. Matter 5 (1993) 1031–1038.
[257] G. Lamoureux, B. Roux, J. Chem. Phys. 119 (2003) 3025–3039.
[258] P. Ren, J.W. Ponder, J. Phys. Chem. B 107 (2003) 5933–5947.
[259] Y. Shi, Z. Xia, J. Zhang, R. Best, C. Wu, J.W. Ponder, P. Ren, J. Chem. Theory Comput. 9 (2013) 4046–4063.
[260] J. Rackers, J.W. Ponder, J. Chem. Phys. 150 (2019) 084104;
J.A. Rackers, R.R. Silva, Z. Wang, J.W. Ponder, arXiv preprint arXiv:2106.13116, 2021.
[261] Tinker website, https://dasher.wustl.edu/tinker, 2021.
[262] T.D. Nguyen, H. Li, D. Bagchi, F.J. Solis, M.O. de la Cruz, Comput. Phys. Commun. 241 (2019) 80–91.
[263] H.C. Anderson, J. Comp. Phys. 52 (1983) 24–34.
[264] S. Paquay, R. Kusters, Biophys. J. 110 (2016) 1226–1233.
[265] C.L. Kelchner, S.J. Plimpton, J.C. Hamilton, Phys. Rev. B 58 (1998) 11085–11088.
[266] D. Faken, H. Jónsson, Comput. Mater. Sci. 2 (1994) 279–286.
[267] H. Tsuzuki, P.S. Branicio, J.P. Rino, Comput. Phys. Commun. 177 (2007) 518–523.
[268] P.M. Larsen, S. Schmidt, J. Schiotz, Model. Simul. Mater. Sci. Eng. 24 (2016) 0550073.

[269] S.P. Coleman, M.M. Sichani, D.E. Spearot, JOM 66 (2014) 408–416.
[270] L.T. Kong, Comput. Phys. Commun. 182 (2011) 2201–2207.
[271] P.J. in 't Veld, G.C. Rutledge, Macromolecules 36 (2003) 7358–7365.
[272] EMC website, http://montecarlo.sourceforge.net/emc/Welcome.html, 2021.
[273] moltemplate website: https://www.moltemplate.org, 2021.
[274] A.I. Jewett, D. Stelter, J. Lambert, S.M. Saladi, O.M. Roscioni, M. Ricci, L. Autin, M. Maritan, S.M. Bashusqeh, T. Keyes, R.T. Dame, J.-E. Shea, G.J. Jensen, D.S. Goodsell, J. Mol. Biol. 433 (2021) 166841.
[275] E.B. Tadmor, R.S. Elliott, J.P. Sethna, R.E. Miller, C.A. Becker, JOM 63 (2011) 17.
[276] MatSci Community Discourse forum website, https://matsci.org, 2021.