LAMMPS: A Case Study For Applying Modern Software Engineering to an Established Research Software Package

Axel Kohlmeyer
Institute for Computational Molecular Science
Temple University
Philadelphia, PA 19122, USA
ORCID: 0000-0001-6204-6475

Richard Berger
CAI-1: Applied Computer Science
Los Alamos National Laboratory
Los Alamos, NM 87545, USA
ORCID: 0000-0002-3044-8266

***Abstract*—We review various changes made in recent years to the software development process of the LAMMPS simulation software package and the LAMMPS software itself. We discuss how those changes have impacted the effort and workflow required to develop and maintain a software package that has been in existence for more than 30 years and where a significant part of the code base is contributed by external developers. We also look into how those changes have affected the code quality and ease of modifying and extending the software while at the same time its audience has changed from a cohort with a generally strong software development background to a group containing many researchers with limited software development skills. We explore how this contributes to LAMMPS' significant growth in popularity in that time. We close with an outlook on future steps.**

***Index Terms*—molecular dynamics, software engineering, high performance computing, testing, best practices, modern C++, build system, software packaging, and collaborative software development.**

I. Introduction and Overview

LAMMPS [1] stands for Large-scale Atomic/Molecular Massively Parallel Simulator and is a classical molecular dynamics (MD) simulation code with a focus on materials modeling. It has many features that are not available in other similar software packages to perform a wide variety of particle-based simulations at different time and length scales. LAMMPS is designed to run efficiently on a wide variety of parallel computers using domain decomposition and MPI message passing [2], and be easy to extend and modify. In addition to MPI parallelization throughout, it supports multi-threading, vectorization, and GPU acceleration for varying parts of its functionality [3]. LAMMPS is open source software distributed under the terms of the GNU Public License Version 2 (GPLv2).

Over a period of more than 30 years, LAMMPS has significantly evolved, same as computer hardware, operating systems, programming languages, and build systems have changed with it. In addition, the list of supported features and contributions from developers within and outside the core group has grown [3], and with it the range of research topics that can be addressed. This has been recognized with an R&D 100 award in 2018 [4].

In addition to the original paper describing the message passing parallelization used in LAMMPS [2], a second LAMMPS overview paper was published in 2022 [3]. It summarizes many of the added features and improvements since LAMMPS was converted to C++. The number of citations of both publications that describe the core functionality of LAMMPS (Fig.1) illustrates how LAMMPS has grown in popularity over time. The figure correlates this with milestones in LAMMPS' history, most of which are discussed here.

The growth of LAMMPS has at times exposed limitations of the development process and thus prompted adjustments. For any such large software project, there is always a conflict between conservative developers and progressive developers, and maintainers have to find a balance between those two approaches for the software to remain useful and relevant.

LAMMPS was originally developed at Sandia National Laboratories, a US Department of Energy facility. Currently distributed LAMMPS releases [5] include contributions from many research groups and individuals from many institutions around the world. Most of the funding for LAMMPS has come from the US Department of Energy (DOE).

We discuss various aspects of modernizing the source code and the development process and point out lessons we learned in the process.

II. Source Code Management

From the very beginning, LAMMPS development has followed a continuous delivery paradigm. That is, any released LAMMPS (patch) version is expected to be fully functional, and in case any issues arise, they are addressed as quickly as possible to recover the full functionality.

A. Custom Scripts and Subversion

In the early days, a new LAMMPS version was started by copying the source tree. Then changes were applied, and when a new version was complete custom scripts were run to create and upload a patch file with the changes and a compressed

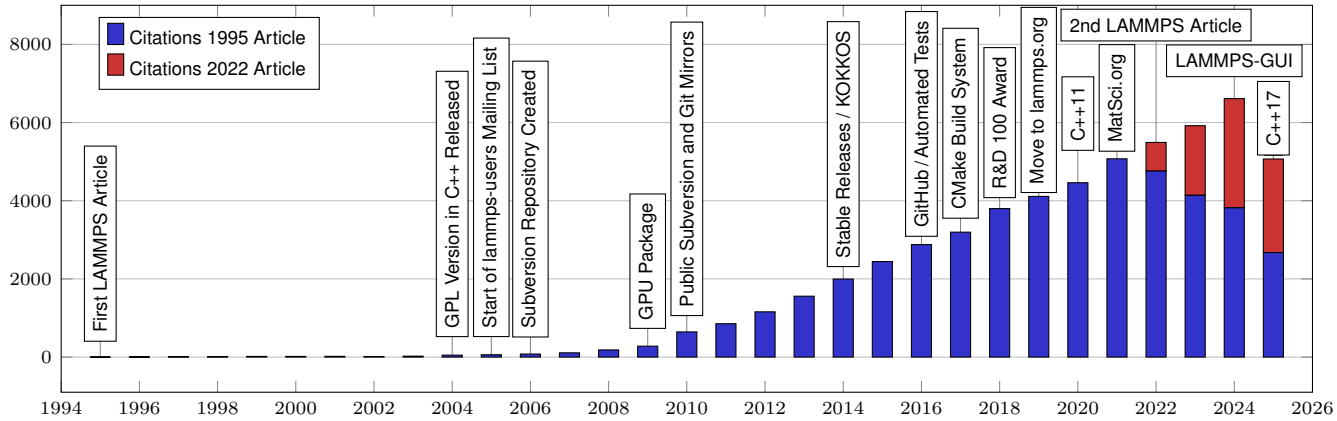


Fig. 1. Citations [6] of the LAMMPS overview articles [2], [3] over time as of the 22 July 2025 stable release, and some milestones in LAMMPS' history

tarball of the sources plus the manual in HTML and PDF formats. Concurrent development of features would require multiple copies and integrating patches as they were released.

At this time, new patches were released frequently, often on subsequent days or only a few days apart. In effect, each patch represents an added or updated feature or a fixed bug. With the growing importance of LAMMPS (inside and outside of Sandia), a subversion repository for LAMMPS was started on a protected server in a Sandia data center.

B. Move to Git and GitHub

A practical problem of having a subversion server hosted by Sandia is that it is not accessible to LAMMPS contributors outside of Sandia. Thus, a process was established to create a read-only mirror hosted on a publicly accessible server at Temple University. This server also converted the subversion repository to a git repository, which accommodated the growing number of developers who prefer git over subversion.

Since these were read-only mirrors, any changes would still have to be emailed to a developer at Sandia to be added and committed to the subversion server. Those would then be mirrored eventually and show up on the public subversion and git servers within 24 hours. However, with frequent patch releases, this still sometimes led to a situation where new patches would create new issues before bug-fix patches were processed. Additionally, contributed features would not always be included immediately, depending on the availability of a LAMMPS developer with write access to the subversion repository, and thus could be outdated already when released.

To make the development process more transparent and to better accommodate the needs of outside developers, the canonical LAMMPS repository was moved to GitHub [7]. Instead of emailing changes to a LAMMPS developer for inclusion, changes are now submitted as pull requests on GitHub. Crucial for the development process is that pull requests can be easily and quickly updated and tested (Sect.III). Potential merge conflicts are also detected and need to be resolved before merging.

C. Releases and Branches

As mentioned above, LAMMPS patches used to be released frequently, which caused problems for people packaging LAMMPS or installing it on clusters or supercomputers, since it would be impossible to provide packages for each of those releases, and there was no indication whether a specific release had particularly few known issues.

Starting in 2014, about 3–5 of the patch releases in a year would be retroactively labeled as 'stable'. With the move to git and GitHub, releases have been less frequent, since the 'develop' branch into which pull requests are merged has become the equivalent of the (frequent) patch releases from the early days. Developers now simply need to follow that branch to stay up-to-date with development. Instead, regular releases are now called *feature releases* and are done only every 4–8 weeks when a sufficient number of new features and improvements have accumulated. These releases are tracked by the 'release' branch.

Starting in 2021, *stable releases* are made only once per year after a period in which only bug fixes are applied. These are tracked on the 'stable' branch. In addition, bug fixes to the 'develop' branch are also back-ported and added to a 'maintenance' branch. This 'maintenance' branch is occasionally

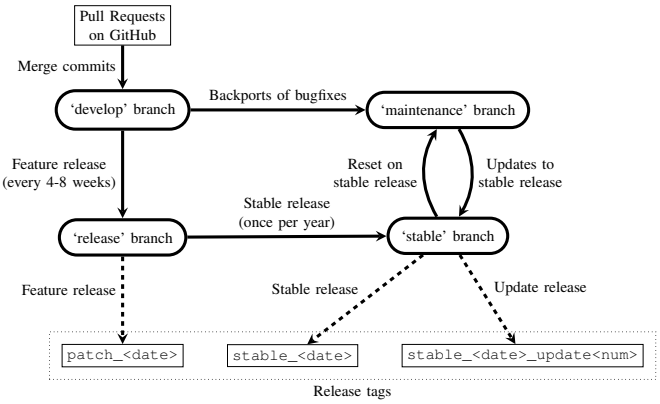
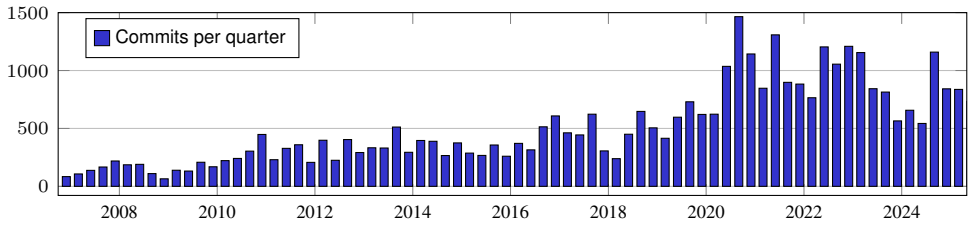


Fig. 2. Flowchart showing relationships between the main branches in the LAMMPS git repository [7] and conventions for applying release tags



Git repository and Github statistics
(approximate, as of 22 July 2025 release):

Total Lines of Code:	1,320,000
Total files:	13,500
Total source files:	6,300
Optional packages:	93
Total commits:	45,000
Contributors:	430
Total pull requests:	3,500
Issues submitted:	1,200

Fig. 3. Graph showing commits per quarter (excluding merge commits) to the main LAMMPS branch and some overall statistics

merged with 'stable' for *stable update releases*; these continue until the next stable release is made. Fig. 2 illustrates the relationship between the four branches. Stable releases and updates are now the primary targets for the various options for LAMMPS packaging. This still maintains the spirit of the original LAMMPS development effort, aiming to have all releases fully functional and differ only in available features.

D. Lessons Learned

Moving the LAMMPS source code management to GitHub not only made the process more transparent, it redistributed effort from the developer performing the integration of contributions to the contributors and the development team.

As shown in Fig. 3 the number of commits has risen from about 100–200 per quarter since the creation of the subversion repository in 2006, to about 300–500 per quarter since the public subversion (and git) mirror was made available in 2010, and to about 600–1200 commits per quarter in recent years. Only the move to git as source code management tool and using the GitHub pull request management have allowed us to scale the development process to this level of activity. Now, contributions are improved and corrected (if needed) *before* they are included in the development branch and the review process distributes the work across the entire LAMMPS team.

To maintain consistency, we use branch protection for the 'develop' and 'stable' branches so that *all* changes must be submitted as pull requests. Accidental commits to a local copy of the 'develop' branch will be refused when pushing the branch. This ensures that all changes go through the automated review and testing process (discussed in Sect. III).

We also found that it is preferable to have a single person performing merges and handling releases. That person may be different for different releases. Individual LAMMPS developers can "chaperone" individual pull requests, and signal this by assigning the pull request to themselves. If they assign the pull request to the developer responsible for merging, it is the signal that the pull request is ready to be merged (from the perspective of the new or improved functionality that is added) and can be subjected to a final round of testing and enforcing various formal and programming style requirements.

In summary, we found that our approach strikes a good balance between integrating new features quickly and maintaining correctness, consistency, and code quality without putting too much burden of code review and maintenance on the code developers.

III. Automated Testing

As mentioned in Sect. II-B, the move to git and GitHub already helped improve code quality by introducing code reviews *before* merging. Another step that has been *extremely* effective in preventing bugs and compilation issues to be introduced into the code base is that changes in the GitHub hosted LAMMPS repository and submitted pull requests trigger a variety of automated actions, specifically several kinds of tests for different LAMMPS configurations and build environments. Most of these tests are run when pull requests are submitted or updated, thus indicating problems to the contributor to resolve. Failed tests prevent pull requests from being merged until any issues are resolved and all tests pass. The tests are currently run on-premises using equipment operated by the ICMS at Temple University using the Jenkins [8] automation software or in the Azure cloud using GitHub runners. In this section, we discuss the different kinds of test runs and what their benefits are.

A. Integration Testing

LAMMPS is designed to be compatible with many operating systems (mainly Linux variants, Windows, or macOS, but also FreeBSD, Solaris, AIX), using several compilers (for example, GNU, Clang, Intel, Microsoft VC++, IBM), two different build systems, and with a variety of build settings (with a static or shared library, or with 32-bit or 64-bit atom IDs and image flags). Also, there are dependencies between some optional packages and classes where changes to a base class can break compilation of the derived class in a package.

A recurring problem with contributed code in pull requests is that the contribution compiles and runs correctly in the development environment of the contributor with the settings used by that developer, but can fail in one of the other supported environments. Possible causes are incorrect use of some code constructs, lack of testing during development, writing of non-portable code, or limitations of or unavailability of external libraries on all platforms and so on.

To ensure that a pull request does not break compilation of LAMMPS in general, but for all existing features in particular, a wide variety of compilations are triggered after a pull request submission or any update. These run on a dedicated cluster of test machines at Temple University or in the Azure cloud using GitHub runners. For testing Linux systems, a variety of containers are used with old and new Linux distribution installations. These containers are chosen so that the range

of supported Linux systems from the oldest to the very latest ones is checked for compatibility with compiling LAMMPS using the included compilers.

B. Built-in Test Library

Testing whether LAMMPS compiles at all is only one prerequisite to ensure that LAMMPS behaves as expected on all supported platforms. To test its functionality the `unittest` tree was added to LAMMPS in 2020 and populated with a variety of tests. Some of those tests are unit tests in the strict sense, and test utility functions and classes that are used for convenience or portability and for recurring tasks.

A large number of the included tests are more like a hybrid between unit and regression tests. This is necessary since many features of LAMMPS cannot be executed until a viable simulation system that uses physically meaningful settings has been created. Often, this system has to have specific properties enabled or components allocated and populated with reasonable data to be suitable for testing features in LAMMPS.

LAMMPS uses the GoogleTest C++ library as a test framework for C, C++, and Fortran language code, where testing Fortran requires writing suitable Fortran-to-C wrappers. Testing is not enabled by default and is only available by setting `-D ENABLE_TESTING=ON` when compiling LAMMPS using the CMake build system (more on that in Sect. IV-D). This includes the `unittest` tree in the build process and also downloads and compiles a specific version of GoogleTest [9] that is compatible with LAMMPS' build requirements. After compilation is complete, tests can be run with the CTest utility that is part of the CMake software package. Tests for specific features are configured in such a way that they are skipped for LAMMPS compilations that lack the corresponding optional packages or features.

Specifically, most of the test programs in the `unittest/force-styles` folder are tools that generate a variety of inputs on-the-fly from fragments and settings provided in YAML files and compare the energies, stresses and forces for an initial configuration and after a few MD steps against previously recorded reference values. If supported, also the results of the `single()` function is compared to what the `compute()` function produces. Similarly, the writing and reading of binary restart files and (text-mode) data files are tested, and the resulting energies are compared. Finally, different simulation settings that follow different code paths and communication patterns but should produce the same energies and forces are tested, and - if available - accelerated variants of the same style (e.g. for the OPENMP or GPU package).

Those `force-styles` tests are intentionally designed to be very sensitive. Properties such as atomic forces, global energies, global stresses must not have relative errors greater than a global *epsilon* parameter (typically defined in the range of $1 \cdot 10^{-13}$ to $1 \cdot 10^{-8}$) when compared to reference data included in the tests YAML file. The *epsilon* parameter can be adjusted in the YAML file for individual tests to accommodate pair styles with a larger amount of "noise", e.g. due to using interpolation tables. The global *epsilon* parameter is modified by empirical factors for different test settings, including accelerator packages and lower precision force kernels where a correspondingly larger error is expected. The same reference data are used for all test variants, which has led to the detection and correction of a significant number of bugs and incomplete or inconsistent implementations. Still, some tests are more sensitive than others and may fail on different platforms or when compiling LAMMPS at a high optimization level. The YAML files for the tests include keywords for this, so the CTest utility can skip them.

Another set of tests in the `unittest` tree concerns the C-language library interface and its wrappers for Python and Fortran. These tests primarily focus on features that are specific to the interfaces and thus not covered by the tests for force styles and similar.

Finally, there are also custom tests for individual LAMMPS commands or to check the format of files generated with the dump command or of files that are parsed as potential files or molecule files. Where possible, tests are performed not only to get expected results with correct input but also to abort with expected error messages with incorrect input.

Due to the size of the LAMMPS code base, the test coverage is incomplete. At the time of this writing about 80% of the files and classes are covered and about 45% of the lines of source code. Most of the untested code, though, is in packages and features that are not used frequently, so that for most use cases of LAMMPS the test coverage is much better than the percentages indicate. The biggest deficit is for compute styles and fix styles that are not related to force computation or time stepping. This is mostly because of the programming effort required since it is not possible to write a generic test tool like the force-style tests.

C. Regression Testing Using Examples

In regression testing, the result of a given input deck is compared with reference results for the same input created with an older version of the software which are assumed to be correct. Although not originally intended for that purpose, the LAMMPS `examples` tree provides a large pool of resources for that purpose, since examples typically include complete input decks and output logs to run those inputs with 1 and 4 MPI tasks. There are several challenges here: a) the submitted examples are too large or use too many MD simulation steps and therefore may take a long time to complete; b) the examples do not contain a sufficient amount of so-called thermodynamic output; c) the impact of the demonstrated feature is not reflected in the thermodynamic output; d) the simulation contains randomized elements that are not reproducible across platforms; and e) the example will diverge over exponentially growing accumulated noise due to limitations of floating-point math [10].

With the intention to not "pollute" the `examples` tree with the modifications needed for regression tests, a separate tree with copies of the examples is used where inputs are adjusted as needed and examples not suitable for regression

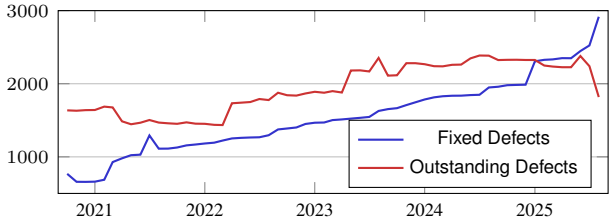


Fig. 4. Graph of defects reported by Coverity Scan as of the 22 July 2025 stable release of LAMMPS

testing are removed or only tested for completion of the run without checking the output. This approach provided us with a large library of regression tests quickly, but it proved to be a significant effort to maintain it, considering the pace at which LAMMPS development advances. As a result, the regression test set of examples has diverged from the examples in the LAMMPS source distribution and is missing most recent contributions. Thus, a new effort is underway to use more advanced scripts for testing that can use the examples from the LAMMPS source distribution directly and require only some non-invasive modifications to a few examples to make them compatible.

Testing the complete collection of example inputs for regressions is still time consuming, and thus only performed *after* the merge of a pull request or after adding a specific label to the pull request. The new regression test tool in development speeds up the processing by distributing the tests across multiple runners. It also has a "quick mode" suitable for testing pull requests; that mode first checks which commands and styles are changed by the pull request, and then only runs regression tests on examples containing those commands or styles. In "quick mode" also the maximum number of tests is curbed; if the list is larger than a given threshold, only a randomly chosen subset is tested.

D. Static Code Analysis

Part of the testing infrastructure for LAMMPS to eliminate bugs and improve code quality is the use of multiple tools using static code analysis. The first such step is the option to compile LAMMPS with verbose warnings enabled (e.g., using `-Wall -Wextra` with the GCC or Clang compilers). During the parsing process of the compiler, often deeper insights into the code structure may be obtained, and certain kinds of logic errors or typos can be identified. For that to work well, all easily avoidable warnings have to be eliminated (e.g., about unused variables) so that any remaining warnings stand out more. It is also important to use different compilers, since those tend to detect different issues differently well.

A more straightforward approach to static code analysis is provided by tools such as *Coverity Scan* [11] or *CodeQL* [12]. These perform a more thorough pattern-based and heuristic analysis of the source code, and thus take significant time to process and analyze it. For that reason, the *CodeQL* analysis is performed only after merging, and the (most time-consuming) processing with *Coverity Scan* only once per week (unless explicitly requested). Given the programming conventions used in LAMMPS, there will be a significant number of false positives, especially about class members not initialized in the constructor.

Using *Coverity Scan* in particular has been immensely useful in detecting hard-to-find issues such as data type conversion errors, copy-and-paste errors, logic errors, dead code, and much more. Fig. 4 lists the number of detected and resolved problems in the last five years as identified by *Coverity Scan*. Static code analysis is applied not only to C++ code but also to Python code. Using the *CodeQL* tool has been especially beneficial in this regard.

E. Tools for Code Quality and Programming Style

The CMake build system for LAMMPS has provisions to enable a number of additional tools to analyze and improve the LAMMPS source code. For example, there is support for *clang-tidy* and *include-what-you-use*, which are based on the LLVM compiler infrastructure and are capable of detecting incorrect or outdated C++ code constructs and inconsistent use of include statements, respectively.

The GCC and Clang compilers both also support the creation of instrumented executables using the `-fsanitize` flag. A selection of sanitizers are available, for instance, for undefined behavior, memory leaks, or invalid memory access. These are less detailed and accurate than the Valgrind memory checker, but execution is much faster.

We also provide custom suppression files for the Valgrind memory checker tool, so that (ideally) only errors within LAMMPS are flagged. Especially the MPI and OpenMP runtime libraries trigger a large number of false positives, and the provided suppressions hide most of them. Valgrind has been particularly effective in detecting memory management issues when used in combination with built-in unit testing via CTest. For this purpose, the so-called "death tests", which would also lead to many false positives, can be excluded from the test runs. A similar option to use Valgrind [13] or instrumented executables is available for the regression checking tool that is currently in development.

Finally, there are some custom scripts that check source files if they conform to various documented conventions. The same applies to configuration files for the clang-format tool. Both promote a homogeneous and consistent format and programming style. These conventions are explained in the "Programmer's Guide" part of the LAMMPS manual (Sect. V-C).

F. Lessons Learned

The extensive testing and application of tools aimed at improving code quality has resulted in a quite noticeable improvement of the LAMMPS source code and increased the productivity of the LAMMPS core developer team. Many tasks are now performed automatically (and therefore more consistently than if executed manually by humans), and feedback about any submitted pull request is given to contributors and LAMMPS developers automatically and as soon as available.

Programming errors are much easier to locate and correct when they are flagged during the development process, rather than (much) later when they are reported by users. Some bugs in rarely used features have remained undetected in the source code for long times until they were detected and reported by users or a static analysis tool. This transfers effort required for integration of additions or modifications of LAMMPS from the core developers to the contributors, unless they explicitly ask for assistance (e.g. because of limited familiarity with LAMMPS or C++ programming).

Static code analysis tools can help detect uncommon and otherwise hard-to-find issues, although the design of LAMMPS and some of its programming conventions result in many false positives. However, this was much more of a problem at the beginning, when there were many reports of problems. Over time, the number of newly reported issues from new or changed source code has become relatively small, and it is not difficult to separate actionable items from false positives. These can then be resolved (or ignored) rather quickly.

It is difficult to quantify the impact, but the general impression is that the bugs reported in stable releases are mostly bugs in recently added features. Bug reports for code that has been in use for a long time have become very rare and are usually for obscure and rarely used functionality. Instead, the number of bug reports that are due to misunderstandings or errors in the documentation is growing.

IV. Code Refactoring

While adoption of git (Sect. II-B) and extensive automated testing (Sect. III) are primarily aimed at making the process of integrating new contributions into LAMMPS more efficient without breaking existing functionality, there are also changes to the existing LAMMPS code to refactor it (i.e. rewrite without changing functionality) to modernize it and make it easier to maintain. This applies to different components of the LAMMPS source code and in this section we are trying to categorize them. The refactoring process goes hand in hand with unit testing (Sect. III-B), as it is advisable to first write tests to establish the status quo to which any refactoring changes are compared. Ideally, any refactoring changes will reproduce the same behavior as before.

The LAMMPS sources are organized into basic functionality and optional "packages". Originally, these packages were grouped into "core" packages (assuming that they were well maintained by the LAMMPS core developers), "user" packages (maintained by external contributors), and a "USER-MISC" package for all kinds of one-off contributions. Over time, the quality of the code and the level of maintenance for different packages diverged substantially. After observing that the LAMMPS core developers were often blamed for issues with "user" packages (which were supposed to be maintained externally) and after some of them had been orphaned, a concerted effort was started to eliminate this distinction. Many of the changes were motivated by reports of defects from static code analysis (Sect. III-D). This effort culminated in a reorganization and renaming of packages in Summer 2021: a) all "USER-" prefixes were eliminated; b) the "USER-MISC" package that had become very large was split into several "EXTRA-" packages and some core styles were moved there as well; c) packages with similar applications would have the same prefix (e.g. "DPD-" for dissipative particle dynamics variants, "CG-" for coarse-grain models, or "ML-" for machine learning potentials).

A. Adoption of Modern C++ Features

As the C++ standard evolves over time, the features of the C++ standard library become more reliable and powerful. The LAMMPS developers have thus decided to gradually abandon the strict "C with classes" programming styles. This applies specifically to the `std::string` class and STL container classes, particularly `std::vector`. Eventually, the minimum required C++ standard was raised to C++11 (in 2020), which represents a major upgrade over the previous standard (C++98) and contains a lot of useful and convenient new functionality. After completion of the 22 July 2025 stable LAMMPS release, the following LAMMPS releases will require the C++17 standard. This will provide further opportunities to modernize and improve LAMMPS.

Using standardized C++ features eliminates the need to have custom implementations or platform abstractions in LAMMPS (Sect.IV-B3) for compilers and platforms that are fully standard compliant. With the "C with classes" programming style, LAMMPS depends on the equivalent functionality of the standard C library. However, that is not as portable and consistent since only a part of it is standardized in the C-language standards, other parts conform to standards like POSIX which are not available or use different conventions on different platforms.

There are a few exceptions where the original code is retained for simplicity or efficiency reasons. For example, LAMMPS continues to use the "stdio" library for file I/O instead of the more complex and less efficient "iostreams" of C++. Furthermore, for large memory allocations (for example, per-atom vectors or arrays, or neighbor lists), LAMMPS employs wrapper classes calling `malloc()` and `free()` to reduce overhead from memory management and memory fragmentation, lower the memory use, improve CPU cache efficiency, and also to simplify MPI communication.

B. General Code

The modernization changes are too many to discuss them all in detail here. Instead, we will focus on a few exemplary changes. The overarching goal is to retain or – where possible – improve on a key property of the LAMMPS source code: it is easy to extend and modify in order to add new models, system manipulations, and on-the-fly analysis computations. The more complex and sophisticated source code is restricted to parts that usually do not require changes for adding new functionality.

1) String Handling: C-style string handling is rather complex, error-prone, and difficult to implement in a safe manner because of having to adjust dynamically allocated memory all the time and handling pointers. Thus, the `std::string` class is used in many places, especially for local, temporary strings, but also when passing strings to functions. This is aided by automatic conversions from `const char *` to `const std::string &`. This and the `c_std()` member function of C++ strings to return a C-style string pointer have allowed to transform the code to modern string handling incrementally. However, some key parts will require a larger refactoring effort to be transformed and therefore have been skipped so far. The preference for C++ strings is also reflected in the collection of functions in the `utils` namespace that was added, as discussed below.

2) Using the {fmt} library: The preference for C++ strings and the adoption of C++11 is further supported by integrating the {fmt} formatting library [14]. It improves on the C-style `printf()` functions in being type safe with a generic placeholder ({}) for any type of (supported) argument. The {fmt} library is predominantly used three ways: a) as the `fmt::format` function directly to produce formatted strings similar to `sprintf()`, b) through utility functions like `utils::print()` which can be used like `fprintf()` only that it uses a {fmt}-style format string, and c) as overload to several functions that would previously only take a single string as argument, but now may use a format string and a variable number of arguments. For example, this has massively simplified the output of customized error messages. Previously, this required allocating a suitable-sized string buffer, using `snprintf()` to create the custom string, output of the buffer, and freeing the temporary storage. Now, this operation has become a single statement that differs only from the static string version by having a variable number of additional arguments.

Many scientists are familiar with the Python programming language, so the similarity of {fmt} formatting to Python string formatting is welcome. Since the {fmt} formatting functions have been adopted into the C++20 standard, we will eventually be able to eliminate the explicit copy of the implementation in LAMMPS and switch from `fmt::format` to `std::format` and related functions of the standard C++ library.

3) `utils` and `platform` namespaces: As with any large software package, there are repetitive tasks that can be abstracted by implementing custom functions that perform those tasks. Not only does this reduce the amount of repeated code, it improves consistency and avoids having to fix the same bug in multiple locations. Some of these functions were originally implemented as members of the foundational LAMMPS classes. During the modernization these functions were moved to a `utils` namespace, and then many more utility functions were added. These provide tasks like transforming strings (to upper or lower case, from UTF-8 to ASCII), removing comments, trimming or compressing whitespace, splitting or joining strings following LAMMPS-specific rules, or conversion of strings to numbers and many more. These functions have extensive unit tests (see Sect. III-B).

Similarly, platform-specific functionality was collected and abstracted into the `platform` namespace. As a consequence, the LAMMPS code has very few places where C preprocessing is required for conditional compilation on different host platforms. One such abstracted feature is the traversal of paths and folders and the manipulation of filenames. Another abstracted platform-specific feature is the dynamic loading of object files as part of the "plugin" command.

4) Tokenizer and FileReader classes: Some of the repetitive tasks mentioned in the previous section are better implemented as specialized classes. Most prominent are the `Tokenizer` and `FileReader` classes and its variants. Many of LAMMPS' pair styles require reading and parsing of files with the potential parameters. Historically, this was done using C-style string buffers and the `strtok()` function. This function is not reentrant and overall its use led to rather complex and hard to read (and debug!) code. Using the new tokenizer and file reader classes reduced the number of lines of code required to read and parse parameter files by a factor of 3–5, sometimes even more, and made the resulting code more readable.

5) Use of Exceptions: Using C++ exceptions improves the flexibility of error handling and simplifies code in many places. This has been introduced locally in places, e.g., when using the tokenizer classes (see Sect. IV-B4). But this has also been applied to the general error handling in the `LAMMPS_NS::Error` class. Rather than aborting the LAMMPS simulation directly, an exception is thrown and caught by an exception handler in the `main()` function. This exception handler also processes exceptions thrown by C++ standard library functions and classes or the {fmt} library and thus makes it easier to determine the cause of an error.

An important benefit exists for the C-language library interface (Sect. IV-C) where all functions that may potentially throw exceptions are wrapped into an exception handler, and functions were added to detect if an error has occurred and to retrieve the error message. This prevents programs using the library interface from crashing immediately; instead, they can check the status and then choose how to proceed, that is, either ignore the error or recreate the LAMMPS instance and repeat the simulation with different settings.

6) Improved Error Messages: Another refactoring effort concerns the content of error messages that LAMMPS would issue. Historically, many error messages were of the kind "there was an error" with no details about the cause and how to remedy it. Removing the error would require a careful study of the manual and often also the source code to determine the exact cause. The latter part was made easier by adding a pre-processor macro so that the error message would contain the name of the source file and the line number where the error message was dispatched. However, this became a growing problem as the audience of LAMMPS shifted from scientists familiar with programming and able to read source code to ones that use pre-compiled executables and have little

programming knowledge, and in some cases no direct access to the specific source code that was used to compile the LAMMPS executable they are using.

We use the following three steps to improve the error message: a) we expand error messages to state which keyword was failing, and thanks to including the {fmt} library (Sect. IV-B2) error messages can include more details about the faulty data; b) for errors requiring a more detailed explanation, we add a mechanism to output a coded URL that refers to a specific section in the LAMMPS online manual with a discussion of that particular error; c) where possible, we use a mechanism that repeats the line of input where the error happened (if needed, before and after variable substitution) as part of the error message and use ASCII graphics to highlight the specific word in the last line of input that was causing the error.

C. Library Interface

An unusual feature of LAMMPS is that it is designed so that an instance of the `LAMMPS_NS::LAMMPS` class holds the complete state of a simulation. The `main()` function of the LAMMPS executable is quite minimal and mostly creates such a class instance while passing the command-line arguments and then sequentially processes each line of input until it reaches the end. This enables, for example, the use of LAMMPS as a component in multiscale applications [15] or in task farming workflows. Especially the LAMMPS Python interface (Sect.IV-C2) has become very popular for these types of applications. This is not limited to a single LAMMPS instance, but custom applications can be written that create multiple class instances, either sequentially or concurrently.

1) C-language Interface: To make the encapsulation of a simulation more accessible, a C-language library interface was added, where only a single header `library.h` would be needed to have access to functions that create, query, and manipulate a LAMMPS class instance. Since the interface uses C and not C++, it is rather straightforward to call it from other programming languages, either directly or by providing an interface file for the SWIG tool [16].

Historically, the LAMMPS library interface was created in a rather *ad hoc* fashion, that is, new functions were added when someone needed them, and the specific use case would determine the type and number of arguments of those functions. This led to a rather inconsistent design and some degree of redundancy. Since removing inconsistencies and redundancies would result in some incompatible changes, the decision was made to have a major overhaul of the library interface that would make future breaking changes unlikely.

Additional motivation for a refactoring was the inconsistent documentation (Sect. V-B) and the need for a cleaner and more consistent Python interface (Sect. IV-C2). The library interface is also used in most of the unit test tools (Sect. III-B) and, in combination with the Python interface, a significant expansion of the introspection abilities was needed.

2) Python Interface: The LAMMPS Python interface is a Python package that provides access to the C-language library interface (Sect. IV-C1) from Python using the "ctypes" package to interface with dynamically loaded libraries; this requires building LAMMPS as a shared library instead of the default static library. Initially, it followed the same *ad hoc* development model as the C-language interface and, as a consequence, was neither consistent nor complete. Thus, it was refactored and made more consistent alongside the refactoring of the C-language interface. However, it also required extensions to the C-language interface, especially introspection functions, which would then be used to make the behavior more Python-like. That includes the use of exceptions (Sect. IV-B5) which are monitored by the Python interface and then re-thrown as Python exceptions, if needed.

Unlike the C-language interface, which due to the restrictions of the C programming language, consists of a collection of global functions that use a common `lammps_` prefix, and where the first argument is an opaque pointer to the instance of the `LAMMPS_NS::LAMMPS` class, the Python interface uses a `lammps` Python class, with the functions of the library interface becoming methods of the class.

In addition, new functionality was added to the LAMMPS Python interface, such as accessing arrays via NumPy, a `cmd` property that makes code look more "pythonic" (example: `lmp.command("units lj")` becomes `lmp.cmd.units("lj")`), an `ipython` property that integrates the Python `lammps` class with Jupyter notebooks (e.g. to embed snapshot images created by the `dump image` command).

Finally, the initial single-file Python "wrapper" module was refactored into a regular Python package with a corresponding directory structure. The installation support based on the deprecated "distutils" was updated to use "setuptools" instead. The build process was structured such that an "install-python" target was added to the build system which executes the build of the LAMMPS Python package with "pip", "build", "wheel", and "setuptools" into a so-called wheel archive which contains not only the Python code, but also the LAMMPS shared library. This makes the "wheel" self-contained and directly installable through the `pip` command. In fact, the "install-python" build target will attempt to install the created wheel of the `lammps` Python package.

3) Fortran Interface: Several attempts at providing a Fortran interface to LAMMPS have been made over the years, and their source code has been included into the LAMMPS source code distribution. But they were incomplete and most relied on some additional library of Fortran functions to support the interface.

In 2020, the development of a new Fortran interface was started, reusing as much of the existing interface code as possible, but with the following design decisions: a) the entire interface is contained in a single source file providing a `LIBLAMMPS` Fortran module; b) the module provides a `LAMMPS` derived type that can be used in ways very similar to the `lammps` class of the LAMMPS Python interface; c) a compiler compliant with Fortran 03 is required since the ISO_C_BINDINGS module is used for calling the functions in the C-language library interface; d) the mod-

ule itself does not contain any MPI library calls and the optional communicator argument of the `lammps` class uses Fortran 77 style MPI communicators that are represented by integers. Internally, those can be inter-converted using the `MPI_Comm_f2c()` and `MPI_Comm_c2f()` functions. To make d) work, a `lammps_open_fortran()` "constructor" function was added to the C-language library interface to support passing an MPI communicator through its integer representation.

Thanks to the help of contributors of previous Fortran interfaces, the new LAMMPS Fortran now interfaces with the complete C-language API and is integrated into the automated testing procedure (Sect. III-B). All previous interfaces have been declared obsolete and removed from the distribution. Thanks to the single source file approach, adding this new interface to a Fortran code requires to add this one file to the build process so it gets compiled before any files that make use of the module and link to the LAMMPS library, by preference with the shared library so that all dependencies to external libraries of LAMMPS itself are automatically included.

D. CMake-based Build System

The traditional build system of LAMMPS was designed around the GNU Make program in combination with some Bourne shell and Python scripts to handle package dependencies and specific build requirements. The build process consists of three steps: 1) building bundled libraries in the `lib` sub-folders or interfacing to externally built libraries, and 2) enabling or disabling packages (by copying files from package folders to the main source folder or by deleting them from the main source folder) and 3) compiling the sources in the main source folder into a LAMMPS library and executable based on so-called "machine makefiles" that were customized for specific compilers, operating systems, features and libraries, and computer hardware. Over the years, this process has been optimized and sources of errors reduced, but it intrinsically requires a good understanding of using compilers and libraries, and adjustments have to be made manually. During this time, asking for help compiling LAMMPS was a recurring topic of discussion on the LAMMPS mailing list.

After intense discussions, a concerted effort was started in August 2017 to implement an alternative build system based on the CMake tool. Original design goals were that the behavior of the build process, specifically how to include optional packages, should be similar and that both build systems should co-exist. CMake offers several advantages: it can largely automate the detection and configuration of features based on available tools and libraries and the operating systems; it supports out-of-source builds (in fact, LAMMPS' CMake scripting enforces that) and thus separates binaries and sources and allows to build different configurations from the exact same source code in the same tree; it provides a platform neutral and platform aware script language that simplifies porting of LAMMPS to different operating systems and compiler environments; it provides a build and installation process similar to many other software packages, and thus simplifies creation of pre-built binary packages for Linux distributions and automated build environments like homebrew or spack. Over the years, the CMake build process has been continuously improved and modernized and includes features that are not available to the legacy build process, such as unit testing (Sect. III-B).

Following the 22 July 2025 stable release, the legacy build process will no longer support packages that require the building of or interfacing with libraries in the `lib` folder, which is the most error-prone step of the legacy build system. The expectation is that eventually only the CMake build system remains. Having only the CMake build system will facilitate further improvements in the build process that are currently prevented by having to support the legacy build process as well.

V. Improved Documentation

In step with the modernization of the LAMMPS source code, we are modernizing the documentation. Historically, LAMMPS used a simple, home-grown markup language that would be translated into HTML format on a file-by-file basis. This was simple and very fast, but had a major drawback: mathematical expressions had to be rendered into images first. The resulting HTML looked rather plain and reflected the typical style of web pages in the 1990s and early 2000s. In addition, navigation was minimal. The PDF version of the manual, including its table of contents, was created from the HTML pages with a tool called HTMLDOC [17].

Due to the large number of features and packages that were added over the years, the LAMMPS documentation has become large: currently (22 July 2025 version) it consists of over 1000 source files (not counting documentation that is extracted from C++ or Python source code) and has nearly 3000 pages when translated into PDF format.

A. Use ReStructuredText and Sphinx

Initially, the simple markup was retained and then converted on the fly with a custom tool to reStructuredText [18]. Those `rst` files would be translated into HTML or PDF with the Sphinx [19] tool. This required some "massaging" of some of the documentation sources to obtain a readable translation, and some minor flaws were unavoidable. However, the output was much more pleasant to look at.

In a second step, the translated `rst` files became the canonical source files for the documentation, and any new documentation files would be added only in reStructuredText format. With the available translation tools, it would still be possible for veteran LAMMPS developers to write a first draft in the old markup, translate that into `rst`, and correct and polish the `rst` file for submission. Over time, the number of cross-references to other commands or specific sections of the manual has been growing, thus providing readers of the manual with more context.

Once the `rst` files became the canonical source, further improvements could be added, including some provided by

Sphinx extensions and others from custom additions. Notable is, for example, the use of MathJAX for typesetting mathematical expressions in LaTeX so that equations are no longer imported as images, which makes adding and correcting them much easier. Or, the use of syntax highlighting for code examples, including LAMMPS input examples.

Custom scripts were added to ensure that all included commands are documented, included the index, listed in the command summary tables and lists, and that the markers for accelerated versions are correct. In addition, references are verified to be available and unique.

B. Integrate Python Docstrings and Doxygen Comments

Switching to Sphinx and reStructuredText enabled a deeper integration between source code and documentation by including Python docstrings and Doxygen-style [20] comments into relevant parts of the manual. Specifically for the library interface this was a major improvement, since previously there were multiple versions of descriptions: comments in the source code, a README file, some documentation in the manual about the C-language version, and the Python wrapper that sits on top of that. Sadly, those were usually not in agreement with each other and with what was implemented in the source code. Having the canonical documentation embedded as comments or strings into the source code resulted in just one location (with cross-references) with the detailed description focused on the C-language interface and others mostly describing what is different in their specific language bindings compared to the C-language version.

C. Adding a Programmer's Guide

The largest addition to the LAMMPS manual in recent years has been the "Programmer's Guide" part. As we were modernizing the source code, we felt the need to better document the various aspects of the LAMMPS program design and its conventions and available abstract programming patterns. This is done with the understanding that most contributors to a scientific software package are *not* formally trained programmers, but scientists with the need to extend or modify simulation software to realize their research projects. Some parts were previously scattered across the manual, but having a dedicated "Programmer's Guide" section separates the documentation for users (which is predominantly a command reference) from the documentation for programmers that want to program LAMMPS or write programs on top of LAMMPS.

VI. Summary

In this paper, we analyze how various modern software development methods and tools have helped to improve the LAMMPS molecular dynamics software package and its development process. It would not be possible today to develop and maintain LAMMPS with the same efficiency and the same level of code quality had we continued with the historical development process.

Many of the improvements are "under the hood", that is, in the implementation of the functionality without changing the interface (much) or in the development process itself so that developers can better cope with the growing popularity and the correspondingly growing volume of contributions to LAMMPS. Other changes are aimed at the perceived change in the kind of users from people with experience in programming to people who lack that experience and use precompiled executables. The new "Programmer's Guide" targets developers instead and tries to improve their understanding of development strategies and provide documentation of added utilities and portability abstractions.

VII. Future Plans

Moving forward, the current plan is to continue the refactoring process and gradually replace outdated and error-prone code constructs with simpler and modern ones, as long as the changes do not negatively affect the serial and parallel performance of LAMMPS. In that process also the minimum required C++ standard version and CMake version will have to be reconsidered on the basis of which versions are still widely in use, specifically on major supercomputers. Eventually, the legacy build system may be removed entirely, and then the CMake system made more modular and better aligned with best practices and common conventions.

The expansion of the library of tests and the "Programmer's Guide" part of the LAMMPS manual are also ongoing projects. Those will benefit from a further expansion of the core LAMMPS team and thus an increase of the available workforce. Automated testing will also benefit from a new expanded test cluster that will be implemented on top of the buildbot software [21] instead of the current Jenkins-based [8] facility. Selected tests will remain hosted by GitHub, e.g. where the corresponding hardware is not available in the Temple testing cluster.

We also need to spend more effort training junior developers in the core team to take over more maintenance responsibilities and handle more management tasks, and thus prepare the LAMMPS project for an inevitable transition of leadership as senior developers are retired (though still contributing) or close to retirement age.

Acknowledgments

The authors thank Steve Plimpton and Aidan Thompson for providing detailed feedback and encouragement during the writing of this article and for first-hand information on the early days of LAMMPS development.

Financial support provided by Sandia National Laboratories under POs 2149742 and 2407526 is gratefully acknowledged.

This work was supported by the U.S. Department of Energy through the Los Alamos National Laboratory. Los Alamos National Laboratory is operated by Triad National Security, LLC, for the National Nuclear Security Administration of the US Department of Energy (Contract No. 89233218CNA000001).

References

[1] LAMMPS Homepage, www.lammps.org, Accessed: 2025-04-24.

[2] S. Plimpton, “Fast Parallel Algorithms For Short-Range Molecular-Dynamics,” J. Comput. Phys, vol.117 (1), pp. 1–19, 1996, doi: 10.1006/jcph.1995.1039.
[3] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in’t Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, et al, “LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales,” Comp. Phys. Comm. 2022; 271:108171, doi: 10.1016/j.cpc.2021.108171.
[4] R&D 100 Archive of Winners www.rdworldonline.com/rd-100-archive/?YEAR=2018, Accessed: 2025-05-04.
[5] LAMMPS Releases on GitHub, github.com/lammps/lammps/releases, doi: 10.5281/zenodo.3726416, Accessed: 2025-04-24.
[6] Source: Google Scholar, scholar.google.com, Accessed: 2025-05-02.
[7] LAMMPS Project on GitHub, github.com/lammps/lammps/, Accessed: 2025-04-24.
[8] Jenkins Home, jenkins.io, Accessed: 2025-05-03.
[9] GoogleTest Project on GitHub, github.com/google/googletest/, Accessed: 2025-04-25.
[10] D. Goldberg, “What every computer scientist should know about floating-point arithmetic,” ACM Computing Surveys, vol. 23 (1), pp. 5-48, 1991, doi: 10.1145/103162.103163.
[11] Coverity Scan Home, scan.coverity.com, Accessed: 2025-04-27.
[12] CodeQL Home, codeql.github.com, Accessed: 2025-04-27.
[13] Valgrind Home, valgrind.org, Accessed: 2025-04-27.
[14] {fmt} – A modern formatting library, fmt.dev, Accessed: 2025-04-28.
[15] B. FrantzDale, S. J. Plimpton, M. S. Shephard, “Software components for parallel multiscale simulation: an example with LAMMPS,” Eng. Comput., vol. 26, pp. 205-211, 2010
[16] SWIG Home, swig.org, Accessed: 2025-05-01.
[17] HTMLDOC Home, msweet.org/htmldoc, Accessed: 2025-05-01.
[18] reStructuredText Home, docutils.sourceforge.io/rst.html, Accessed: 2025-05-01
[19] Sphinx Home, sphinx-doc.org, Accessed: 2025-05-01.
[20] Doxygen Home, doxygen.nl, Accessed: 2025-05-01.
[21] Buildbot Home, buildbot.net, Accessed: 2025-05-03.