Recent Changes in LAMMPS Development

Dr. Axel Kohlmeyer

Assistant Dean, College of Science and Technology
Associate Director, Institute for Computational Science
High-Performance Computing Team Lead

Temple University
Philadelphia PA, USA

a.kohlmeyer@temple.edu

ICMS
Institute for Computational Molecular Science

TEMPLE UNIVERSITY®

Preparing LAMMPS for the Future

Since the last LAMMPS workshop there have been changes to the development process, coding style, conventions, and requirements:

- Infrastructure and organizational changes
- Source code refactoring, coding style updates
- Build system updates, Package reorganization
- Documentation processing and reorganization
- Revised GitHub procedures and workflows
- Automated testing and code analysis

Infrastructure and Organization

- The LAMMPS homepage and supporting sites have moved to the lammps.org domain and to servers run by developers at Temple U
- Automated testing and website, download, and documentation deployment run @ Temple U
- Updates to the Google search index initiated but not yet completed
- New forum on Materials Science Discourse https://matsci.org/lammps which includes the mailing list archive. We encourage you to join

Overview of Code Changes

- C++-11 standard now minimum requirement
- Relaxed the “C with classes” style to include more C++ features where beneficial and helpful to simplify the code and make it more readable. std::string, std::vector, std::map etc. may be used (but use const references for arguments!)
- Custom classes and `utils` namespace with convenience functions for common tasks
- Integrated fmtlib (to replace printf()) (→ C++20)
- **Still** using stdio library, not iostreams for I/O

String Handling

- Replaced `char *` function arguments with `const std::string &` in many places
- Added convenience functions like `utils::split_words()`, `utils::count_words()`, `utils::split_lines()`, `utils::trim()`, `utils::trim_comment()`, `utils::utf8_subst()`
- Added `utils::strmatch()` for more flexible comparisons based on simplified regular expressions (e.g. “^rigid” will match **all** fix rigid variant styles, including accelerated versions)

String/File/Argument Parsing

- New `Tokenizer` class to replace strtok()
- New `ValueTokenizer` class to read numbers
- New `TextFileReader` and `PotentialFileReader` classes for processing files
- New `ArgInfo` class for processing command arguments like `f_name[dim], c_name[dim1][dim2], v_name, d_name, i_name`
- Functions like numeric(), inumeric(), tnumeric(), expand_args(), bounds() have been moved from class members to the `utils` namespace

Convenience Functions

- New platform neutral functions in `utils` for path manipulations: `utils::path_basename()`, `utils::path_dirname()`, `utils::path_join()`
- “Safe” file read functions `utils::sfgets()`, `utils::sfread()` that check for read errors
- New function `utils::logmesg()` to output the same text to screen and logfile
- New `utils::getsyserror()` function get error strings from failed C library operations
- `open_potential()` function moved to `utils::`

fmtlib Integration

- LAMMPS includes a copy of fmtlib (accepted as part of C++20) to replace printf()-like functions
  - Type safe for 32- and 64-bit integers (no more need for TAGINT_FORMAT, or BIGINT_FORMAT)
  - fmt::format(<format>, args…) returns a std::string
  - Uses “{}” as generic placeholder (for strings **or** numbers), “{:10}” pads to at least 10 chars, “{:.2f}” is equivalent to “%.2f” and much more
  - Flexible formatting options not available in printf()
  - Mechanism to add format string to custom functions

Code Simplification

- Printing adjustable error and warning messages used to require allocating a buffer, using sprintf(), output the message, free buffer
  → utils::logmesg(), Error::all(), Error::warning() now accept variable number of arguments, if more than one, the first is used as fmtlib format
- Adding/replacing fixes/computes/groups used to require to build an argv-style argument list
  → convenience overload accepts string (often created via fmt::format()) and will split words into argv list via utils::split_words()

Before and After

```
char *ptr;
for (int i = 0; i < size_one; i++) {
  if (i == 0) ptr = strtok(format," \0");
  else ptr = strtok(NULL," \0");
  if (ptr == NULL) error->all(FLERR,"Dump_modify format line is too short");
  delete [] vformat[i];

  if (format_column_user[i]) {
    vformat[i] = new char[strlen(format_column_user[i]) + 2];
    strcpy(vformat[i],format_column_user[i]);
  } else if (vtype[i] == Dump::INT && format_int_user) {
```

```
auto words = utils::split_words(format);
if ((int) words.size() < nfield)
  error->all(FLERR,"Dump_modify format line is too short");

int i=0;
for (auto word : words) {
  delete[] vformat[i];

  if (format_column_user[i])
    vformat[i] = utils::strdup(std::string(format_column_user[i]) + " ");
  else if (vtype[i] == Dump::INT && format_int_user)
```

Before and After

```
MPI_Allreduce(&count,&allcount,1,MPI_INT,MPI_SUM,world);

if (comm->me == 0) {

  if (screen) {
    if (strcmp(arg[origarg],"cc") == 0)
      fprintf(screen,"  %d settings made for %s index %s\n",
              allcount,arg[origarg],arg[origarg+1]);
    else
      fprintf(screen,"  %d settings made for %s\n",
              allcount,arg[origarg]);
  }
  if (logfile) {
    if (strcmp(arg[origarg],"cc") == 0)
      fprintf(logfile,"  %d settings made for %s index %s\n",
              allcount,arg[origarg],arg[origarg+1]);
    else
      fprintf(logfile,"  %d settings made for %s\n",
              allcount,arg[origarg]);
  }
}
```

```
MPI_Allreduce(&count,&allcount,1,MPI_INT,MPI_SUM,world);

if (comm->me == 0) {
  if (strcmp(arg[origarg],"cc") == 0)
    utils::logmesg(lmp,"  {} settings made for {} index {}\n",
                   allcount,arg[origarg],arg[origarg+1]);
  else
    utils::logmesg(lmp,"  {} settings made for {}\n",
                   allcount,arg[origarg]);
}
```

Before and After

```
// create a new compute temp style
// id = fix-ID + temp
// compute group = all since pressure is always global (group all)
// and thus its KE/temperature contribution should use group all

int n = strlen(id) + 6;
id_temp = new char[n];
strcpy(id_temp,id);
strcat(id_temp,"_temp");

char **newarg = new char*[3];
newarg[0] = id_temp;
newarg[1] = (char *) "all";
newarg[2] = (char *) "temp";

modify->add_compute(3,newarg);
delete [] newarg;
tcomputeflag = 1;

// create a new compute pressure style
// id = fix-ID + press, compute group = all
// pass id_temp as 4th arg to pressure constructor

n = strlen(id) + 7;
id_press = new char[n];
strcpy(id_press,id);
strcat(id_press,"_press");

newarg = new char*[4];
newarg[0] = id_press;
newarg[1] = (char *) "all";
newarg[2] = (char *) "pressure";
newarg[3] = id_temp;
modify->add_compute(4,newarg);
delete [] newarg;
                                                        61,1
```

```
// create a new compute temp style
// id = fix-ID + temp
// compute group = all since pressure is always global (group all)
// and thus its KE/temperature contribution should use group all

id_temp = utils::strdup(std::string(id) + "_temp");
modify->add_compute(fmt::format("{} all temp",id_temp));
tcomputeflag = 1;

// create a new compute pressure style
// id = fix-ID + press, compute group = all
// pass id_temp as 4th arg to pressure constructor

id_press = utils::strdup(std::string(id) + "_press");
modify->add_compute(fmt::format("{} all pressure {}",id_press, id_temp));
pcomputeflag = 1;
}
                                                        49,1
```

Before and After

```
  // open file on proc 0

  FILE *fp;
  if (comm->me == 0) {
    fp = force->open_potential(file);
    if (fp == NULL) {
      char str[128];
      snprintf(str,128,"Cannot open Stillinger-Weber potential file %s",file);
      error->one(FLERR,str);
    }
  }

  // read each set of params from potential file
  // one set of params can span multiple lines
  // store params if all 3 element tags are in element list

  int n,nwords,ielement,jelement,kelement;
  char line[MAXLINE],*ptr;
  int eof = 0;

  while (1) {
    if (comm->me == 0) {
      ptr = fgets(line,MAXLINE,fp);
      if (ptr == NULL) {
        eof = 1;
        fclose(fp);
      } else n = strlen(line) + 1;
    }
    MPI_Bcast(&eof,1,MPI_INT,0,world);
    if (eof) break;
    MPI_Bcast(&n,1,MPI_INT,0,world);
    MPI_Bcast(line,n,MPI_CHAR,0,world);

    // strip comment, skip line if blank

    if ((ptr = strchr(line,'#'))) *ptr = '\0';
    nwords = atom->count_words(line);
    if (nwords == 0) continue;

    // concatenate additional lines until have params_per_line words

    while (nwords < params_per_line) {
                                                                368,1
```

```
  // open file on proc 0

  if (comm->me == 0) {
    PotentialFileReader reader(lmp, file, "sw", unit_convert_flag);
    char * line;

    while ((line = reader.next_line(NPARAMS_PER_LINE))) {
      try {
        ValueTokenizer values(line);

        std::string iname = values.next_string();
        std::string jname = values.next_string();
        std::string kname = values.next_string();

        // ielement,jelement,kelement = 1st args
        // if all 3 args are in element list, then parse this line
        // else skip to next entry in file
        int ielement, jelement, kelement;

        for (ielement = 0; ielement < nelements; ielement++)
          if (iname == elements[ielement]) break;
        if (ielement == nelements) continue;
        for (jelement = 0; jelement < nelements; jelement++)
          if (jname == elements[jelement]) break;
        if (jelement == nelements) continue;
        for (kelement = 0; kelement < nelements; kelement++)
          if (kname == elements[kelement]) break;
        if (kelement == nelements) continue;

        // load up parameter settings and error check their values

        if (nparams == maxparam) {
          maxparam += DELTA;
          params = (Param *) memory->srealloc(params,maxparam*sizeof(Param),
                                              "pair:params");

          // make certain all addional allocated storage is initialized
          // to avoid false positives when checking with valgrind

          memset(params + nparams, 0, DELTA*sizeof(Param));
        }

                                                                300,1
```

C-Library Interface Refactoring

- C-style library interface is base for Python module, Fortran interfaces, unit tests, and more
- Now more consistent and complete API
- library.h does not require mpi.h by default
- Many introspection functions added (available packages, compilation settings etc.)
- Functions documented via Doxygen and integrated into manual with examples
- Unit tests added (work in progress)

Doxygen Documentation Comment

```
/** Variant of ``lammps_open()`` using a Fortran MPI communicator.
 *
\verbatim embed:rst

This function is a version of :cpp:func:`lammps_open`, that uses an
integer for the MPI communicator as the MPI Fortran interface does.  It
is used in the :f:func:`lammps` constructor of the LAMMPS Fortran
module.  Internally it converts the *f_comm* argument into a C-style MPI
communicator with ``MPI_Comm_f2c()`` and then calls
:cpp:func:`lammps_open`.

If for some reason the creation or initialization of the LAMMPS instance
fails a null pointer is returned.

.. versionadded:: 18Sep2020

*See also*
   :cpp:func:`lammps_open_fortran`, :cpp:func:`lammps_open_no_mpi`

\endverbatim
 *
 * \param  argc   number of command line arguments
 * \param  argv   list of command line argument strings
 * \param  f_comm Fortran style MPI communicator for this LAMMPS instance
 * \return        pointer to new LAMMPS instance cast to ``void *`` */

void *lammps_open_fortran(int argc, char **argv, int f_comm)
{
  lammps_mpi_init();
  MPI_Comm c_comm = MPI_Comm_f2c((MPI_Fint)f_comm);
  return lammps_open(argc, argv, c_comm, nullptr);
}
```

Python Module Refactoring

- Now folder with multiple files like other modules
- Co-developed with C-library refactoring
- Benefits from introspection and query functions
- More “pythonic” behavior (query for data type) and thus allow “duck typing” of return values
- NumPy wrappers to return NumPy arrays or use them as arguments
- Benefits from -DLAMMPS_EXCEPTIONS to recover from failures (particular in serial)

Net Impact of Code Refactoring

- Code reduction (up to 5x for some cases)
- Increase of code reuse
- Improved readability of the source
- Added code comes with extensive unit tests and embedded documentation via doxygen
- Consistent behavior between C and Python
- Complete replication of C API to Python API
- SWIG interface file for wrapping LAMMPS
- New Fortran (95 style) module in progress

Coding Style Updates

- Checks (whitespace, permissions, old URLs)
- Clang-format configuration file (not mandatory),
- Updated include file conventions to follow best practices and expose hidden dependencies → follow IWYU principle (can use iwyu tool)
- Header file more strictly required to follow conventions (no “using”, PIMPL, forward decls, namespace) to avoid clashes between styles
- Example: recent refactor of REAXFF package

Before and After

```
#ifndef LMP_PAIR_REAXC_H
#define LMP_PAIR_REAXC_H

#include "pair.h"
#include "reaxc_types.h"

namespace LAMMPS_NS {

class PairReaxC : public Pair {
 public:
  PairReaxC(class LAMMPS *);
  ~PairReaxC();
  void compute(int, int);
  void settings(int, char **);
  void coeff(int, char **);
  virtual void init_style();
  double init_one(int, int);
  void *extract(const char *, int &);
  int fixbond_flag, fixspecies_flag;
  int **tmpid;
  double **tmpbo,**tmpr;

  control_params *control;
  reax_system *system;
  output_controls *out_control;
  simulation_data *data;
  storage *workspace;
  reax_list *lists;
```

```
#ifndef LMP_PAIR_REAXFF_H
#define LMP_PAIR_REAXFF_H

#include "pair.h"

namespace ReaxFF {
  struct API;
  struct far_neighbor_data;
}

namespace LAMMPS_NS {

class PairReaxFF : public Pair {
 public:
  PairReaxFF(class LAMMPS *);
  ~PairReaxFF();
  void compute(int, int);
  void settings(int, char **);
  void coeff(int, char **);
  virtual void init_style();
  double init_one(int, int);
  void *extract(const char *, int &);
  int fixbond_flag, fixspecies_flag;
  int **tmpid;
  double **tmpbo,**tmpr;

  ReaxFF::API *api;
  typedef double rvec[3];
```

Build System Updates

- Make traditional make and CMake build behave more consistent:
  - Always build library and link executable to it
  - Package collections (basic, most, all, nolib) as yes/no-<arg> or CMake preset
- Packages reorganized: no more “USER”, a few new packages, USER-MISC styles distributed
- Automated download for many external libs (not recommended for OpenKIM and PLUMED)
- CMake builds DLL on Windows, include Python

Documentation Updates

- Directly built from .rst files via sphinx
- Custom Python3 virtual environment in doc
- Using multiple sphinx extensions: spelling, math typesetting, breathe (doxygen docs embedding)
- Python scripts to check for missing or duplicate links, packages, styles etc.
- New programmer guide: includes content of Developer.pdf, but also documentation of utility functions and library interface, Python support

called once. See `lammps_mpi_finalize()` for an alternative to invoking `MPI_Finalize()` explicitly from the calling program.

void *lammps_open(int argc, char **argv, MPI_Comm comm, void **ptr)

Create instance of the LAMMPS class and return pointer to it.

The `lammps_open()` function creates a new `LAMMPS` class instance while passing in a list of strings as if they were command-line arguments for the LAMMPS executable, and an MPI communicator for LAMMPS to run under. Since the list of arguments is **exactly** as when called from the command line, the first argument would be the name of the executable and thus is otherwise ignored. However `argc` may be set to 0 and then `argv` may be `NULL`. If MPI is not yet initialized, `MPI_Init()` will be called during creation of the LAMMPS class instance.

If for some reason the creation or initialization of the LAMMPS instance fails a null pointer is returned.

Changed in version 18Sep2020: This function now has the pointer to the created LAMMPS class instance as return value. For backward compatibility it is still possible to provide the address of a pointer variable as final argument *ptr*.

Deprecated since version 18Sep2020: The *ptr* argument will be removed in a future release of LAMMPS. It should be set to `NULL` instead.

See also

`lammps_open_no_mpi()`, `lammps_open_fortran()`

Note

This function is **only** declared when the code using the LAMMPS `library.h` include file is compiled with `-DLAMMPS_LIB_MPI`, or contains a `#define LAMMPS_LIB_MPI 1` statement before `#include "library.h"`. Otherwise you can only use the `lammps_open_no_mpi()` or `lammps_open_fortran()` functions.

Parameters:
- **argc** – number of command line arguments
- **argv** – list of command line argument strings
- **comm** – MPI communicator for this LAMMPS instance
- **ptr** – pointer to a void pointer variable which serves as a handle; may be `NULL`

Returns: pointer to new LAMMPS instance cast to `void *`

void *lammps_open_no_mpi(int argc, char **argv, void **ptr)

Variant of `lammps_open()` that implicitly uses `MPI_COMM_WORLD`.

This function is a version of `lammps_open()`, that is missing the MPI communicator argument. It will use `MPI_COMM_WORLD` instead. The type and

Numerical representation of the LAMMPS version this module was taken from. Has the same format as the result of lammps.version().

2.4.1. The `lammps` class API

The `lammps` class is the core of the LAMMPS Python interfaces. It is a wrapper around the LAMMPS C library API using the Python ctypes module and a shared library compiled from the LAMMPS sources code. The individual methods in this class try to closely follow the corresponding C functions. The handle argument that needs to be passed to the C functions is stored internally in the class and automatically added when calling the C library functions. Below is a detailed documentation of the API.

class lammps.lammps(*name=''*, *cmdargs=None*, *ptr=None*, *comm=None*)

Create an instance of the LAMMPS Python class.

This is a Python wrapper class that exposes the LAMMPS C-library interface to Python. It either requires that LAMMPS has been compiled as shared library which is then dynamically loaded via the ctypes Python module or that this module called from a Python function that is called from a Python interpreter embedded into a LAMMPS executable, for example through the python invoke command. When the class is instantiated it calls the `lammps_open()` function of the LAMMPS C-library interface, which in turn will create an instance of the `LAMMPS` C++ class. The handle to this C++ class is stored internally and automatically passed to the calls to the C library interface.

Parameters:

- **name** (*string*) – "machine" name of the shared LAMMPS library ("mpi" loads `liblammps_mpi.so`, "" loads `liblammps.so`)
- **cmdargs** (*list*) – list of command line arguments to be passed to the `lammps_open()` function. The executable name is automatically added.
- **ptr** (*pointer*) – pointer to a LAMMPS C++ class instance when called from an embedded Python interpreter. None means load symbols from shared library.
- **comm** (*MPI_Comm*) – MPI communicator (as provided by mpi4py). `None` means use `MPI_COMM_WORLD` implicitly.

property numpy

Return object to access numpy versions of API

It provides alternative implementations of API functions that return numpy arrays instead of ctypes pointers. If numpy is not installed, accessing this property will lead to an ImportError.

Returns: instance of numpy wrapper object

Return type: numpy_wrapper

close()

Automated Testing

- Uses Jenkins server (hosted at Temple) or GitHub Actions (for CodeQL, macOS Test)
- Pushes to GitHub or merges trigger test runs
  - Integration Testing: compilation using both build systems and different compilation settings
  - Unit tests via CMake and CTest (see next slide)
  - Run and regression tests
  - Coding style checks
  - Static code analysis tests
  - Tests must pass to merge pull requests

Unit Tests?

- The LAMMPS unittest source tree has a variety of tests using the CTest software from CMake
- The C/C++ tests use the googletest library (automatically downloaded and compiled)
- These include unit tests in the strict sense (e.g. for utility functions and classes) but also tests that require a partial of full setup of a simulation
- Tests for force styles (pair, bond, etc.) are more like regression tests using a YAML file per test with reference data to compare to

Force Style Tests

- Test programs are like input file generators that create LAMMPS instance and then run many variants of short runs and compare forces and energies to reference data: newton on/off, single() vs. compute(), using data/restart file, using different suffixes (if available).
- All variants are compared to the same reference → found inconsistencies between accelerator styles and base or single / compute
- Uses C-library API and its introspection support

```
140: Environment variables:
140:  LAMMPS_POTENTIALS=/home/akohlmey/compile/lammps/potentials
140:  PYTHONPATH=/home/akohlmey/compile/lammps/unittest/force-styles/tests:
140: Test timeout computed to be: 1500
140: [==========] Running 7 tests from 1 test suite.
140: [----------] Global test environment set-up.
140: [----------] 7 tests from PairStyle
140: [ RUN      ] PairStyle.plain
140: [       OK ] PairStyle.plain (31 ms)
140: [ RUN      ] PairStyle.omp
140: [       OK ] PairStyle.omp (11 ms)
140: [ RUN      ] PairStyle.gpu
140: /home/akohlmey/compile/lammps/unittest/force-styles/test_pair_style.cpp:851:
Skipped
140:
140: [  SKIPPED ] PairStyle.gpu (0 ms)
140: [ RUN      ] PairStyle.intel
140: [       OK ] PairStyle.intel (6 ms)
140: [ RUN      ] PairStyle.opt
140: [       OK ] PairStyle.opt (7 ms)
140: [ RUN      ] PairStyle.single
140: [       OK ] PairStyle.single (8 ms)
140: [ RUN      ] PairStyle.extract
140: [       OK ] PairStyle.extract (4 ms)
140: [----------] 7 tests from PairStyle (70 ms total)
140:
140: [----------] Global test environment tear-down
140: [==========] 7 tests from 1 test suite ran. (70 ms total)
140: [  PASSED  ] 6 tests.
140: [  SKIPPED ] 1 test, listed below:
140: [  SKIPPED ] PairStyle.gpu
1/1 Test #140: MolPairStyle:lj_cut .............   Passed    0.14 sec

The following tests passed:
        MolPairStyle:lj_cut
```

More on Tests

- Tests also include validation of the library interfaces: C-library, Python module, Fortran
- Tests for behavior of various input commands (including “death tests” of required errors)
- Tests for tools like binary2txt or lammps-shell
- Unit tests have been crucial for the refactoring:
  → tests were added for original code
  → refactored code has to reproduce it
- Code coverage data is collected to guide where more tests are needed. Current coverage: 33%

GitHub Procedures

- **All** code added to LAMMPS via pull requests
- Typically one core developer manages merging for a development cycle, other core developers review contributions. One approval required
- Reviewer may block PR by requesting changes
- Failed automatic tests also block a merge
- Assigned LAMMPS developer usually assists with making the code compliant and consistent with either comments or by pushing changes

Impact on Releases, Branches

- Three branches: master, unstable, stable
- Releases are much less frequent now and interval between releases is (still) growing
- master branch similar to patch releases before switch to using git and GitHub,
  → automated testing reduces breakage
- Patch releases typically after 300-500 commits
- Stable releases 1-2 times per year with extra stabilization period and added manual tests

Other GitHub Features

- Contributors should receive “Collaborator” invite
- Templates for Issues and Pull requests to guide submitters what information to provide
- Filling out the pull request template is **required** as it confirms the agreement with releasing the contribution under the LAMMPS licensing terms (GPLv2 or LGPLv2.1 on request)
- CODEOWNERS file maps repository files to GitHub user ids → automatic review requests

Frequent Issues with Contributions

- Formatting, whitespace (tabs, CR-LF, trailing whitespace) or permissions, old URLs
- Use of #include for non-LAMMPS or non-system headers in *.h files defining styles
- Missing or incomplete documentation
- Does not compile with -DLAMMPS_BIGBIG
- Does not compile for Windows (with MinGW)
- Missing updates to src/.gitignore, src/.Purge.list, src/<pkg>/Install.sh, lib/<pkg>Install.py

Frequent Issues with Contributions

- Memory leaks, accessing unitialized data
- Pointers in classes not initialized to `nullptr`
- Mismatched new/delete vs malloc()/free()
- Unused variables, dead code
- Commented out debug statements, commented out segments from original code
- Inconsistent file / style / class names
- Variable length arrays (not a C++ feature, extension in GNU, Clang, PGI, but not MSVC)