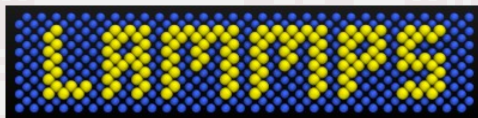
LAMMPS Developer Meeting 2024

LAMMPS Refactoring and Testing

Dr. Axel Kohlmeyer

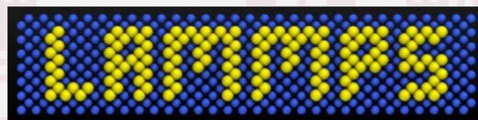
LAMMPS Core Developer

a.kohlmeyer@temple.edu

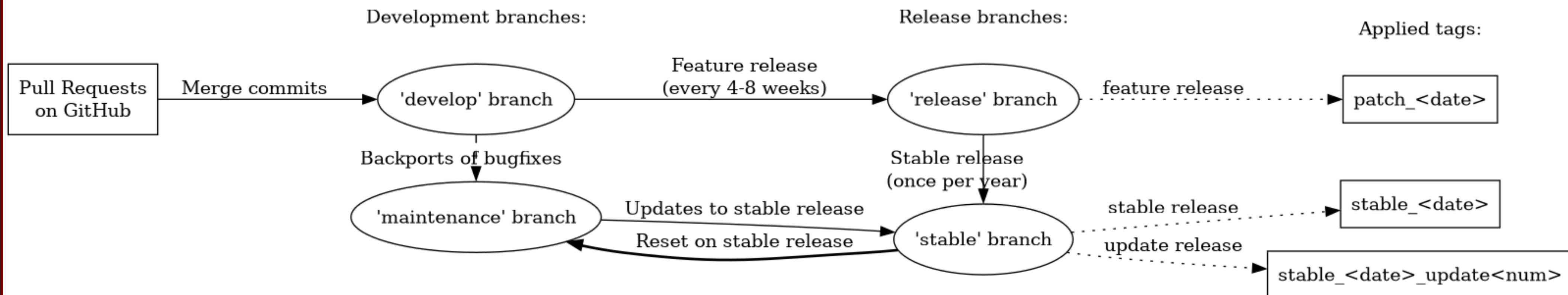


LAMMPS Programming Language History

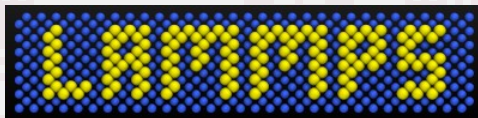
- LAMMPS started as a Fortran 77 program
- It was later updated to Fortran 90/95
- Rewritten in C++ (98 standard, or rather C with classes). For portability, advanced C++ features and STL were restricted to optional code
- Since 2020 LAMMPS requires C++11 and makes more use of STL and templates.
  → refactoring to modernize code (C → C++)
- Ongoing discussions on how closely to follow the C++ standard evolution



LAMMPS Development Process

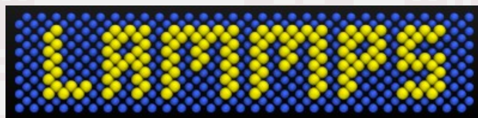


- Main branches in Git repository on GitHub:
  → *develop*, *release*, *stable, maintenance*
  → changes merged as pull requests to *develop*
  → *releases* with new features every 2-3 months
  → *stable* release about once per year
  → bugfixes ported to *stable* in *maintenance*



Suggested Git/GitHub Workflow (I)

- Create a GitHub account, upload ssh key
- Go to https://github.com/lammps/lammps in your web browser and create a fork
- On your desktop create a clone of your fork:
```
git clone git@github.com:<UserID>/lammps.git
```
- Set up access to the upstream repository
```
git remote add upstream git@github.com:lammps/lammps.git
git fetch upstream
```
- Connect “develop” branch to upstream
```
git checkout develop
git branch -u upstream/develop
```



Suggested Git/GitHub Workflow (II)

- Create a feature branch for your work

```
git checkout -b improve-errors develop
```

- Work on changes, commit frequently when a related group of changes is done, push to fork

```
git push origin -u improve-errors
```

- When you have a sufficient set of changes, submit your branch as a pull request
- You may be asked to make additional changes, LAMMPS developers may add changes, too.

```
git checkout improve-errors; git pull
[…]
git commit; git push
```

Suggested Git/GitHub Workflow (III)

- You may work on multiple feature branches concurrently and switch between them
- It may be needed to update "your" *develop* branch or your feature branches to include changes from upstream:

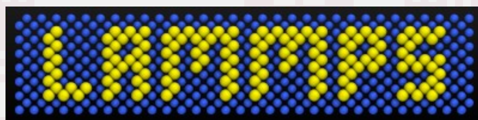
```
git checkout develop; git pull
git checkout improve-errors
git merge develop
```

- **<u>Never</u>** commit any changes to develop! Undo:

```
git checkout develop
git reset upstream/develop
git checkout improve-errors
```

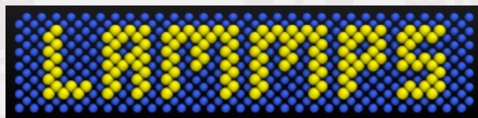
Suggested Git/GitHub Workflow (IV)

- Before submitting a pull request, test that your code compiles and passes all checks (the ci.lammps.org server will also run these checks and some more and block merging on failures):
  - `cd src; make check`
  - `cd doc; make pdf; make html; make spelling`
  - `cd build; cmake --build . ; ctest`
- Running tests with ctest requires compilation with CMake and -DENABLE_TESTING=on
- https://docs.lammps.org/Howto_cmake.html
  https://docs.lammps.org/Howto_github.html



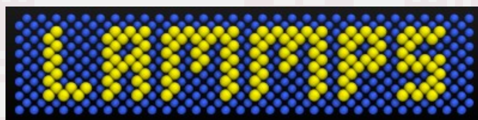
GitHub Command Line Interface

- Available at https://cli.github.com New command: `gh`
- Simplifies testing / reviewing process.
- Authenticate with `gh auth login`
- List pull requests: `gh pr list`
- Show pull request info: `gh pr view 4304`
- Show pull request checks: `gh pr checks 4304`
- **Check out pull request branch: `gh co 4304`**
- List issues: `gh issue list`
- View issue /w comments: `gh issue 4303 --comments`



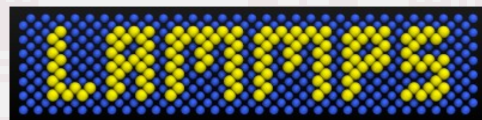
Ongoing LAMMPS Refactoring Projects

- Improve error messages to be more specific
- Replace C style strings with C++ strings
- Use tokenizer classes instead of strtok()
- Specialized classes for (potential) file reading
- Make better use of utility functions in utils:: namespace and platform:: namespace
  → more compact, consistent, and portable code
- Use improved accessor functions for accessing style instances (fix, compute, group, region, ...)



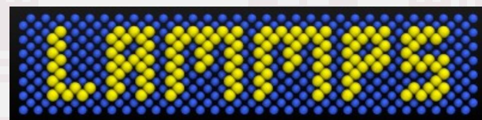
Project: Improve Error Messages (I)

- Error messages in LAMMPS were terse and generic (Ex: Illegal XXX command) thus not user friendly.
- The error message would show the source file and line number. The cause of the error could be figured out from studying the source code and the documentation. This is not an option for non-programmer users
- There are additional explanations in the manual for errors and warnings, but those are often generic, too.
- Error messages should remain brief (1-2 lines of text) but add the extra bit of information needed to figure out what the error cause is from the manual alone.



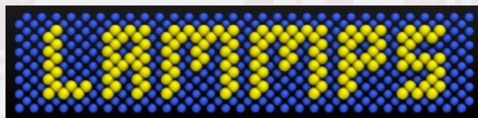
Project: Improve Error Messages (II)

- Error messages should be more specific as to which keyword failed and how; also flag unknown keywords.
- When an error or warning refers to a missing or invalid group, molecule, variable, compute, fix, or dump ID augment the error message to include this ID or name.
- Errors with complex explanations or multiple causes should have an explanatory paragraph in the manual https://docs.lammps.org/Errors_details.html and just call the utils::errorurl() function pointing there.
- There is a utils::missing_cmd_args() convenience function for the common case of missing arguments
- The integration of the {fmt} lib and overloaded error class functions simplify creating more specific errors



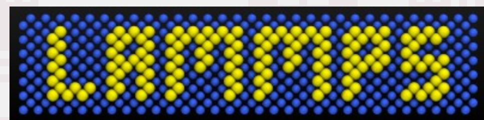
Project: Favor C++ String Handling

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



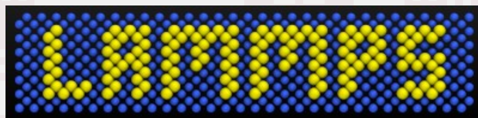
Project: String/File/Argument Parsing

- `Tokenizer` class to replace strtok()
- `ValueTokenizer` class to read numbers
- `TextFileReader` and `PotentialFileReader` classes for processing files
- `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



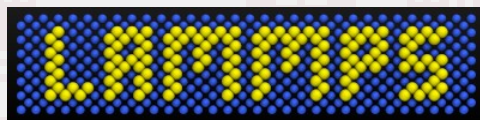
Project: Use Convenience Functions

- Platform neutral functions in `platform` for path manipulations: `platform::path_is_directory()`, `platform::path_basename()`, `platform::path_dirname()`, `platform::path_join()`
- "Safe" file read functions `utils::sfgets()`, `utils::sfread()` that check for read errors
- Function `utils::logmesg()` to output the same text to screen and logfile, supports {fmt}
- Function `utils::getsyserror()` to get error strings from failed C library operations



Project: 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

```
// 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
```

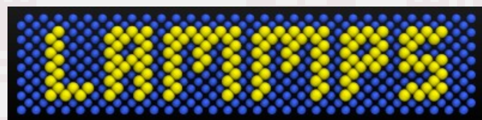
```
id_temp = utils::strdup(std::string(id) + "_temp");
modify->add_compute(fmt::format("{} all temp",id_temp));
tcomputeflag = 1;

// 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
```

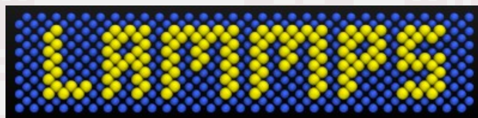
Porting External LAMMPS Code

- LAMMPS was designed to be easily modified
- Not all externally developed features were submitted for inclusion into then LAMMPS distribution.
- External code may be “abandoned”, i.e. no longer updated to be compilable with current LAMMPS.
- Information about known incompatibilities and how to address them are in the LAMMPS manual at: https://docs.lammps.org/Developer_updating.html
- Add LAMMPS plugin loader and the styles can be added to LAMMPS at runtime and code may be included in https://github.com/lammps/lammps-plugins. Also for legacy styles removed from LAMMPS?



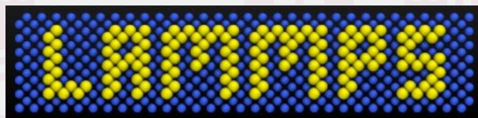
The lammps-plugin repository

- Includes external packages or styles ported to current LAMMPS and converted to plugin
- Are offered for Windows as additional packages
- Less constraints to be added than for LAMMPS
  - No requirement to include documentation
  - May have known bugs waiting to be resolved
  - May not be contributed by the original author
- A place for staging and testing before the code is added to the LAMMPS distribution



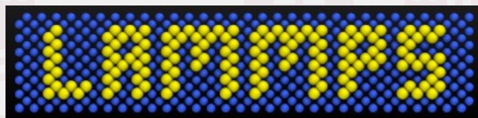
Integrated Testing in LAMMPS

- Tests exist at multiple levels:
  - Tests of individual functions and standalone classes
  - Tests of individual LAMMPS commands
  - Tests for the C++, C, and Fortran Library interfaces
  - Tests for the LAMMPS Python module (in Python)
  - Tests for complex LAMMPS operations on force styles (pair, bond, etc.) using generic executables and input files with customization and reference data in YAML format
- Tests use googletest or Python unittest
- Enable code coverage to detect untested code paths
- Integration testing is handled by external scripts running on https://ci.lammps.org or GitHub actions running on Azure



Why so Much Testing?

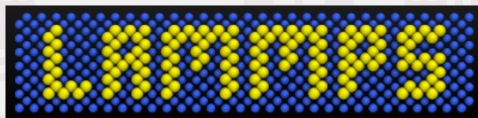
- Early testing limits complexity of bugs:
  → bugs are eliminated early in the development
  → saves time and money
- Testing confirms that added functionality is in compliance with the specified requirements
- Unit testing encourages modular programming
  → easier to add new functionality
- Tests demonstrate correct **and** incorrect usage
- Testing is easy and can be automated;
  debugging is complex and requires humans



TEMPLE UNIVERSITY®

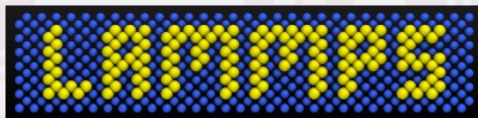
Automated Testing

- Uses Jenkins server (hosted at Temple) or GitHub Actions (hosted by Azure)
- 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
  - Run and regression tests
  - Coding style checks
  - Static code analysis tests
  - All tests must pass to merge pull request



Regression Test Tool from Trung

- Python script to run LAMMPS with inputs in the “examples” tree and compares results to logs
- YAML style config file to control how to run
- Can run on entire tree or subsets
- Single step or two steps with multiple workers; two step creates list of inputs for workers
- Quick mode searches inputs with commands changed in the current branch
- Output summaries in YAML files



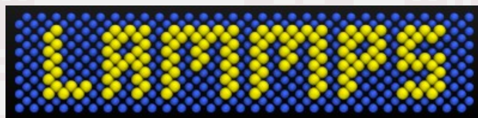
Example: Single Step for Folders

```
export LAMMPS_REG=$LAMMPS_DIR/tools/regression-tests
python3 $LAMMPS_REG/run_tests.py --lmp-bin=$LAMMPS_DIR/build/lmp \
  --config-file=$LAMMPS_REG/config_serial.yaml \
  --example-folders="$LAMMPS_DIR/examples/flow;$LAMMPS_DIR/examples/melt"
```

Creates the output:

```
Entering /home/akohlmey/compile/lammps/examples/flow
2 input script(s) to be tested: ['in.flow.couette', 'in.flow.pois']
   + in.flow.couette (1/2)
     all 6 checks passed.
   + in.flow.pois (2/2)
     all 6 checks passed.
--------------------------------------------------------------------------------
Entering /home/akohlmey/compile/lammps/examples/melt
1 input script(s) to be tested: ['in.melt']
   + in.melt (1/1)
     all 6 checks passed.

Summary:
  Total number of input scripts: 3
  - Skipped  : 0
  - Failed   : 0
  - Completed: 3
    - numerical tests passed: 3
```



Example: Quick Regression (1)

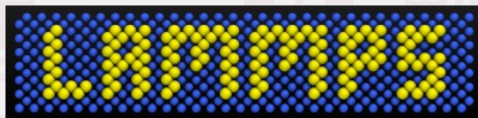
```
export LAMMPS_REG=$LAMMPS_DIR/tools/regression-tests
python3 $LAMMPS_REG/run_tests.py --lmp-bin=$LAMMPS_DIR/build/lmp \
  --config-file=$LAMMPS_REG/config_quick.yaml \
  --examples-top-level=$LAMMPS_DIR/examples \
  --quick --quick-reference=$LAMMPS_REG/reference.yaml \
  --quick-branch=origin/develop --quick-max=100 --num-workers=2
```

Creates an output like this:

```
There are 184 input scripts with changed styles relative to branch github/develop.
Changed styles: {'command': ['ndx2group', 'angle_write', 'dihedral_write', 'info'],
'atom': [], 'compute': ['reduce'], 'fix': ['charge/regulation', 'gcmc', 'widom',
'efield'], 'pair': ['lj/sf/dipole/sf', 'granular', 'aip/water/2dm', 'ilp/graphene/hbn',
'lepton/coul', 'lepton/sphere', 'bop', 'meam/spline', 'meam/sw/spline', 'rebomos',
'tersoff/mod/c', 'pace', 'quip'], 'body': [], 'bond': [], 'angle': ['harmonic'],
'dihedral': ['multi/harmonic', 'opls'], 'improper': [], 'kspace': [], 'dump': [],
'region': [], 'integrate': [], 'minimize': []}
Trimming inputs using reference data from 745 previous runs: trimmed list has 104
entries
Testing 100 randomly selected inputs
```

And two lists with input files:

```
input-list-0.txt input-list-1.txt
```



TEMPLE UNIVERSITY®

Example: Quick Regression (2)

```
Now run the tests concurrently on two runners (or the same node) with:

export LAMMPS_REG=$LAMMPS_DIR/tools/regression-tests
python3 $LAMMPS_REG/run_tests.py --lmp-bin=$LAMMPS_DIR/build/lmp \
  --config-file=$LAMMPS_REG/config_quick.yaml \
  --list-input=input-list-0.txt --output-file=output-0.xml \
  --progress-file=progress-0.yaml --log-file=run-0.log

And:

export LAMMPS_REG=$LAMMPS_DIR/tools/regression-tests
python3 $LAMMPS_REG/run_tests.py --lmp-bin=$LAMMPS_DIR/build/lmp \
  --config-file=$LAMMPS_REG/config_quick.yaml \
  --list-input=input-list-1.txt --output-file=output-1.xml \
  --progress-file=progress-1.yaml --log-file=run-1.log

These will each loops over the 50 input files selected in the previous step:

Input scripts to test as listed in the file:
  input-list-0.txt

There are 50 input scripts listed in input-list-0.txt.

Regression test configuration file:
  /home/akohlmey/compile/lammps/tools/regression-tests/config_quick.yaml
```