Multidimensional Replica Exchange Umbrella Sampling with the LAMMPS Ensembles Interface

version 0.91 (beta release)

Adrian W. Lange

May 13, 2013

1 Introduction

The purpose of this work is to document how to use the Multidimensional Replica Exchange Umbrella Sampling (REUS) part of the LAMMPS Ensemble (LE) interface. LE now also supports one dimensional Hamiltonian exchanges. The LE driver is an external executable program that links to a library build of LAMMPS with the USER-COLVARS package installed. We will assume the user has some familiarity with compiling and running LAMMPS already.

2 System Requirements

The LE is a parallel program written in C, and it should be compiled with a MPI C compiler (e.g., mpicc).

3 Installation

In the future, I would like there to be straightforward scripts to install this code, but until then, the user will just have to follow these instructions in sequence.

3.1 Obtain source code

At the moment, you'll need to obtain the LE code and the modified LAMMPS library code directly from me, Adrian. Feel free to email me about it at alange@alcf.anl.gov.

You will also need to obtain the latest version of LAMMPS, which can be found at `http://lammps.sandia.gov/`. LE has been successfully run with the version from March 9, 2013. For convenience, a tarball of this version is provided with the LE package. In the rest of this manual, we will use path names involving this version of LAMMPS, but you will need to change it appropriately for your set up. If you plan to use this tarball, unpack it with `tar -xzf lammps.tar.gz`.

3.2 Compile LAMMPS

Step 1: Set up your Makefile. You will need to `cd` into `lammps-9Mar13/src/MAKE` and edit your Makefile to make sure that it is set up for your system. There are some examples provided if you need to set up one from scratch.

You *should* compile with OpenMP compiler flags turned on, which tells the compiler to multi-thread loops and sections where OpenMP pragmas exist. How to specify the flag differs depending on which compiler you are using. For example, GCC compilers usually require the flag `-fopenmp` to be included, Intel compilers usually require the flag `-openmp`, and PG compilers usually require the flag `-mp`. Consult your compiler's manual for more details. Compiling with OpenMP turned on is actually not required, but it is highly recommended because it provides the user greater flexibility to make use of a given computer system. And your code will run faster.

Step 2: Install the USER-OMP package. Type `make yes-USER-OMP` in `lammps-9Mar13/src`. This installs the OpenMP threaded routines that will benefit from having the OpenMP flags turned on.

Step 3: Install the USER-COLVARS package. Type `make yes-USER-COLVARS` in `lammps-9Mar13/src`. This installs the interface to the colvars library provided with LAMMPS.

Step 4: Compile the colvars library. Change your directory to `lammps-9Mar13/lib/colvars`. You can modify the makefile Makefil.g++ if you like or copy it over to a new modified one if you need to for your system. It is compiled as a serial library. Once you are satisfied with the Makefile, type `make -f Makefile.g++` (or substitute your modified Makefile if you did so) to make the library.

Step 5: Add the LE files to LAMMPS source. There are only three files that need to be added to the LAMMPS source code in order for LE to work. They are contained in the directory `add_to_lammps`. Copy these over to the LAMMPS source directory, overwriting what is in there. (In the future, we will probably make this a USER package to simply.)

Step 6: Compile LAMMPS as a library. This is documented on the LAMMPS web site, but it follows a simple procedure similar to the usual executable compilation. Change directories into the `lammps-9Mar13/src` directory and type

```
make makelib
make -f Makefile.lib foo
```

where foo is the machine name, corresponding to your Makefile. Then, wait a couple minutes (or longer if optimization is cranked up) for compilation. If all goes well, you will end up with a binary file named `liblmp_foo.a`. It is this file that you will be linking LE against.

3.3 Compile the LAMMPS Ensembles

Step 1: Set up your Makefile. You will need to `cd` into `ens_src` and edit your Makefile to make sure that it is set up for your system, just like for the LAMMPS build. In the Makefile, you will need modify a few variables manually for your system:

- `LAMMPSDIR = dir/foo/src`
  This is the path of your LAMMPS library build from Section 3.2. Set `dir/foo/src` to the appropriate path.

- `LIBCOLVARS` = -L/dir/foo/lib/colvars
  This is the path of the colvars library for LAMMPS that you compiled above. Change to the appropriate path.

- `LIBDIR = foo/fftw/lib/`
  This specifies any external libraries that LAMMPS should be linked with. For example, if you compiled LAMMPS with FFTW, then you need to specify the path of the FFTW library here. Set `dir/foo/src_lammps_lib/` to the appropriate path.

- `LIB = -llmp_foo -lfftw3f`
  These are the binary filenames of the libraries that you are linking with, following the usual convention that a file like `liblmp_foo.a` is included as `-llmp_foo`, removing the "`ib`" and "`.a`". You will need to specify the LAMMPS library binary as well as any other libraries LAMMPS is to be linked with, such as a certain FFTW library. The example, `-lfftw3f`, would be needed if you wanted to link with FFTW3 with single precision.

- `DEBUG = -g`
  This is for optional debugging flags. It can be left blank. Including `-g` is useful for compiling with line references for stack traces or using the `gdb` debugger. You may also choose to specify the preprocessor macro `-DCOORDX_DEBUG` for extra printed output that shows what the random numbers and energies are being used for the Metroplis acceptance criteria during attempted exchanges.

- `CCFLAGS = -O3 -fopenmp $(DEBUG) -I$(LAMMPSDIR)`
  This is for any additional compiler flags for the C compiler. The optimization level is not very important since not much time is actually spent doing computations in the REUS driver, as most of it takes place in LAMMPS. However, you should leave the `$(DEBUG) -I$(LAMMPSDIR)` part in tact. You will need to include OpenMP flags also, where the above example is for gcc style compiler.

Step 2: Compile LE. As you can see from the Makefile, there are really only two commands you will need for making the REUS driver. To compile from scratch, simply type

```
make
```

If all goes well, you wll end up with an executable named `ens_driver`, which is short for "Ensembles Driver." That's it! You can now run LE.

The only other command you may need is for deleting your object files, which is done by typing (don't do this unless you want to re-compile, though)

```
make clean
```

4 Using LAMMPS Ensembles for Multidimensional REUS

4.1 Modified LAMMPS input file

The idea behind LE is to split the MPI universal communicator into separate subcommunicators, each of which launches its own LAMMPS run. Each subcommunicator corresponds to a replica in the REUS algorithm, and each subcommunicator reads a specified LAMMPS input file. This means

we need to set up a seperate input file for each replica we want to run. For the REUS algorithm to be useful, each replica should have a different umbrella bias potential not too far from at least one other replica's bias potential. This is something the user will need to tinker with to find the best acceptance ratio. The umbrella sampling bias potential is controlled via the colvars library, for which there is extensive documentation online at the LAMMPS website.

Assuming the user has properly set up each replica's input for a usual non-LE run with the colvars library, we can now modify it to run with LE.

First and very importantly, comment out or delete the `run #` command in the LAMMPS input script. If this is not done, the LE driver will not work properly. Secondly, there are a number of LE tags to place at the top of the input file (not required to be at the top, but it is a good place for it). There are 4 relative tag types, discussed in turn below. The syntax is somewhat strict, so follow as closely as possible and *do not* add extra spaces.

The COORDX tag. This is the main tag that LE looks for in deciding what to do. COORDX is short for "Coordinate Exchange" because this code swaps the coordinates of each replica, making the code very generalizable for any sort of exchanges. The syntax is as follows:

```
#COORDX: fix [F], seed [N]
```

The variables in the square brackets are described here:

- `[F]` = A string corresponding to the ID name of the colvar fix in this LAMMPS input script.
- `[N]` = An integer. Seeds the random number generator for the Metropolis acceptance critera. If `[N]` = 0, then the direction of swapping alternates as up/down. Otherwise, the direction of swaps is random. This is meant to be used for debugging purposes.

The REPLICA tag. After the COORDX tag, LE will search for the REPLICA tag to gather information specific to the given replica. It's syntax is:

```
#REPLICA: id [I], ndim [Ndim], temp [T], tdim [TD], sdim [SD]
```

The variables in the square brackets are described here:

- `[I]` = An integer greater than or equal to zero. The lowest replica index must be zero and should increase by unit increments. That is, the range is 0 to (n-1), where n is the number of replicas. This is the index of this replica.
- `[Ndim]` = An integer corresponding to the number of dimensions in the multidimensional REUS run. Must be 1 or greater.
- `[T]` = A positive floating point number. The temperature of this replica. It should be the same as the temperature used later in the LAMMPS input script for the NVT ensemble, etc.
- `[TD]` = An integer. This is the index of the dimension on which to perform parallel tempering (i.e., temperature exchanges). This is optional, and it can only be for one dimension. If you don't want any parallel tempering, set this variable to -1, which informs LE to ignore it.
- `[SD]` = An integer. This is the index of the dimension on which to perform Hamiltonian exchanges, also called the "scaling dimension", since it is intended for use as. If you don't want any scaling dimension, set this to -1. Or, you can just omit this part altogether, ending the line at `tdim [TD]`.

The DIMENSION tag. The next tag(s) to create describe how each exchange dimension should be set up. This is controlled via the DIMENSION tag:

```
#DIMENSION: [D] num [NUM] run [RUN] swaps [SWAPS]
```

You can have an arbitrary number of DIMENSION tags, each on their own line, as long as that number corresponds to [Ndim] for the REPLICA tag. The variables in the square brackets are described here:

- `[D]` = An integer greater than or equal to zero. The lowest replica index must be zero and should increase by unit increments. That is, the range is 0 to (n-1), where n is the number of replicas. This is the index of this dimension.
- `[NUM]` = An integer corresponding to the coordinate of this replica along this dimension. In range 0 to (x-1), where x is the number of replicas in this dimension.
- `[RUN]` = A positive integer. How many MD steps in total to take along this dimension.
- `[SWAPS]` = A positive integer. How many replica exchanges to attempt along this dimension throughout the simulation.

The DIMENSION tags must be consistent across all LAMMPS inputs for each replica. Otherwise, the run will fail.

The NEIGHBORS tag. Finally, you will need to make a NEIGHBORS tag for each DIMENSION tag, specifying which subcommunicators are to act as neighbors in the attempted replica exchanges. Neighbors are arbitrary, but if input incorrectly, LE will deadlock in communication. (Efforts are underway to automatically detect and report such errors.) The syntax for this tag is:

```
#NEIGHBORS: [D] [M] [P]
```

The variables in the square brackets are described here:

- `[D]` = An integer greater than or equal to zero. The lowest replica index must be zero and should increase by unit increments. That is, the range is 0 to (n-1), where n is the number of replicas. This is the index of the dimension being described.
- `[M]` = An integer. The minus direction neighbor subcommunicator index for this dimension. If it is negative, this means that this replica has no neighbor in the minus direction for this dimension. Otherwise, it must be 0 or greater in the range of subcommunicator indexes.
- `[P]` = An integer. The plus direction neighbor subcommunicator index for this dimension. If it is negative, this means that this replica has no neighbor in the plus direction for this dimension. Otherwise, it must be 0 or greater in the range of subcommunicator indexes.

It is possible to define circular dimensions by making appropriate NEIGHBOR tags. For instance, for a set of three replicas (0, 1, 2, 3), we might have the following NEIGHBOR tags in each appropriate LAMMPS input script:

```
#NEIGHBORS: 0 3 1
#NEIGHBORS: 0 0 2
#NEIGHBORS: 0 1 3
#NEIGHBORS: 0 2 0
```

In order from replica 0 to replica 2, the above would create a circular dimension for four replicas.

Example of tags. Below is a possible example of the LE header for a LAMMPS input script for a one dimensional parallel tempering run with the same collective variable in each temperature:

```
#COORDX: fix cv, seed 0
#REPLICA: id 2, ndim 1, temp 275.0, tdim 0, sdim -1
#DIMENSION: 0 num 2 run 500 swaps 5
#NEIGHBORS: 0 1 3
```

4.2 Calling the REUS driver executable

The call to the binary executable `ens_driver` has a few command line options. The general call looks like this:

```
ens_driver P [-suffix omp] [-log] -readinput (infile)
```

The parts in square brackets are optional, but the parts in parenthese are mandatory variables. The variable descriptions are here:

- `ens_driver` = The binary executable filename.
- `P` = A positive integer. The number of replicas (*i.e.*, the number of input files).
- `[-suffix omp]` = Optional flag for running LAMMPS with OpenMP multithreading. Recommended to turn on.
- `[-log]` = Flag to turn on writing output to log files for each replica. Otherwise, no log files are written. Turning this will produce lots of text and may become a disk space issue if not careful.
- `(infile)` = The file containing the list of LAMMPS input script files and how many processors to use for each replica.

The `infile` is a text file and has the following syntax:

```
[Subcommunicator index] [LAMMPS input script filename] [Number of MPI ranks for this subcomm]
```

There should not be anything else in this text file. The number of MPI ranks for each subcommunicator must sum to the number of MPI ranks in MPI_COMM_WORLD. You may need to specify the full path for each LAMMPS input script.

The above executable, of course, will need to be launched as part of an MPI run. This means that the user will need to use a command like

```
mpiexec -np 8 ens_driver 2 -suffix omp -readinput foo.bar
```

to launch 8 MPI processes with the LE driver split into 2 replicas, with input files described in file foo.bar. Finally, note that you will want to set the Unix/Linux shell environment variable `OMP_NUM_THREADS` to the appropriate number of CPU cores prior to your run in order to take advantage of the multithreading parallelism.

4.3 Output

By default, no LAMMPS log files nor LAMMPS screen files are created during a REUS run to cut down on disk space usage. However, writing to the LAMMPS log file can be turned on (see Section 4.2), which can be useful in debugging.

The colvars library will write output to various files, and you should make sure that each is labelled correspondingly to the LAMMPS inputs for each subcommunicator.

4.4 Examples

There are three examples provided in the directory `examples`. Each uses the same molecular system, a small peptide in water with a CHARMM force field. README files are provided for each example. There is an example of 1-D REUS, 1-D parallel tempering for a single umbrella sampling restraint, and a 2-D REUS/parallel tempering example with 16 replicas. Give these a try, and see how it goes. They should work and run in a short amount of time.

There is also now an example of 1-D Hamiltonian exchange. This works by having each replica use different parameters in the data files.