Engineering with Computers (2010) 26:205–211
DOI 10.1007/s00366-009-0156-z

ORIGINAL ARTICLE

Software components for parallel multiscale simulation: an example with LAMMPS

Benjamin FrantzDale · Steven J. Plimpton · Mark S. Shephard

Received: 15 December 2008 / Accepted: 21 September 2009 / Published online: 10 December 2009

Abstract Multiscale simulation is a promising approach for addressing a variety of real-world engineering problems. Various mathematical approaches have been proposed to link single-scale models of physics into multiscale models. In order to be effective, new multiscale simulation algorithms must be implemented which use partial results provided by single-scale software. This work considers aspects of software design for interfacing to existing single-scale simulation code to perform multiscale simulations on a parallel machine. As a practical example, we extended the large-scale atomistic/molecular massively parallel simulator (LAMMPS) atomistic simulation software to facilitate its efficient use as a component of parallel multiscale-simulation software. This required new library interface functions to LAMMPS that side-stepped its dependence on files for input and output and provided efficient access to LAMMPS's internal state. As a result, we were able to take advantage of LAMMPS's single-scale performance without adding any multiscale-specific code to LAMMPS itself. We illustrate the use of LAMMPS as a component in three different modes: as a stand-alone application called by a multiscale code, as a parallel library invoked by a serial multiscale code, and as a parallel library invoked by a parallel multiscale code. We conclude that it is possible to efficiently re-use existing single-scale simulation software as a component in multiscale-simulation software.

Keywords Atomistic to continuum coupling · LAMMPS · Software design · Molecular dynamics · Multiscale computation

B. FrantzDale (✉) · M. S. Shephard
Scientific Computing Research Center,
Rensselaer Polytechnic Institute, Troy, NY, USA
e-mail: bfrantz@scorec.rpi.edu; benfrantzdale@gmail.com

M. S. Shephard
e-mail: mark@scorec.rpi.edu

S. J. Plimpton
Sandia National Laboratories, Albuquerque, NM, USA
e-mail: sjplimpton@sandia.gov

1 Introduction

Many real-world engineering problems take place across a range of size scales. While such problems could in principle be solved by describing the problem at the smallest relevant scale, the number of degrees of freedom required quickly makes such approaches intractable. Multiscale modeling shows promise for addressing such challenges by using costly fine-scale models only where they are necessary [1]. As a result, many approaches to multiscale modeling involve simulating a portion of a problem with one single-scale simulation, then transferring the result to another single-scale simulation.

A significant amount of software is available to simulate a variety of single-scale phenomena from quantum mechanics to atomistics to mesoscale to continuum solid and fluid mechanics. This body of software represents a significant collection of building-blocks for multiscale simulation software if it can be effectively reused. We believe an effective and efficient way to accomplish this is to provide interfaces to single-scale software to provide read and write access to its internal state including the configuration of the matter it represents and the stresses, forces, etc. on that matter.

Modern simulation software needs to leverage parallel computing clusters in order to take full advantage of available computing resources. Multiscale modeling

software is no exception. A standard for parallel computing is the message passing interface (MPI). With MPI, the same program runs on many processors which coordinate by passing messages to one another across a network [2]. In order to fully leverage the promise of parallel computing, we will need to allow single-scale software to not only operate in parallel but also to interface in parallel [3].

LAMMPS (large-scale atomistic/molecular massively parallel simulator) is an open-source classical molecular dynamics package for simulating the dynamics of interacting atoms. It also includes coarse-grained particles and interaction potentials, so it can model materials at the mesoscale as well [4]. LAMMPS is written in C++ and can run either on single processors or in parallel. In the latter case, the simulation domain is spatially partitioned across processors which communicate with each other by making calls to MPI. In parallel, LAMMPS can run very large problems (tens of thousands of processors, billions of particles) and achieve high parallel efficiencies of 90% or more [5].

We chose to use LAMMPS for this work because it has models for a variety of solid-state materials (crystalline and amorphous) and because it already included a simple library interface. Our goal was to use LAMMPS efficiently as a single-scale simulation component within a larger multiscale simulation system while making minimal changes to LAMMPS itself. In particular, we were interested in using it to relax groups of atoms subject to boundary conditions, and to evaluate forces on groups of atoms. These are useful operations in the context of a coupled continuum–atomistic quasistatic relaxation.

This paper details the changes we made to LAMMPS, both to its high-level structure and to its library interface, to allow it to be used effectively as a single-scale component within our multiscale system, or in other coupled software frameworks.

2 Methods

We first describe using LAMMPS as a stand-alone application in a multiscale context. Next, we detail changes made at the top level of the code to enable it to be used more robustly as a library, invoked in this case by a multiscale modeler. We then highlight additions we made to its library interface to provide read/write access to the internal state of LAMMPS and its atoms and illustrate how this is useful in the multiscale context.

2.1 Calling LAMMPS as a stand-alone application

The simplest way to simulate a multiscale model using single-scale components is to invoke them independently as stand-alone applications from a multiscale wrapper program. For example, LAMMPS can be used to evaluate forces on a collection of atoms in the following manner. First the multiscale wrapper writes a LAMMPS input file to disk which contains the coordinates of all the atoms. Next, a request to run a LAMMPS job is submitted to our cluster's job queue. When its turn in the queue arrives, LAMMPS is invoked by the system's `mpirun` command. LAMMPS then reads and parses the input file, evaluates the energy of the system which includes computation of per-atom forces, and writes the results to an output file before terminating. The multiscale wrapper can wait for the LAMMPS job to complete, then read and parse the LAMMPS output file, and continue on with the acquired forces.

This strategy was not difficult to implement, but has drawbacks. First, even if there is no wait in the queue, the time LAMMPS spends evaluating forces is dwarfed by the time to start an MPI process, initialize LAMMPS, and read and write files. All these steps are needed for each successive force evaluation. There are additional problems from a portability perspective. The process for launching an MPI job varies from cluster to cluster. Different systems have different places for storing temporary files. Although this initial approach was crude, it worked and was easily implemented in a few hours. In turned out to be an extremely valuable proof of concept and debugging tool.

2.2 Making the library API more robust

LAMMPS is structured as a large number of C++ classes. At the highest level in the code, a handful of pointers are maintained to single instances of the most fundamental of these classes: an `Atom` class that stores per-atom information, a `Domain` class that stores information about the simulation box, a `Comm` class that stores inter-processor communication details, etc. Prior to this work, these few pointers were declared `static` so that any class in LAMMPS could have easy access to data like atom coordinates (stored as an $N \times 3$ array) via a simple statement like

```
double** x = atom->x;
```

Using even one static variable in a library has one important ramification: it prevents multiple instances of the library in the same address space because all instances would try to write to the same static variables. For example, this prevented a multiscale wrapper from instantiating two or more instances of LAMMPS to handle different subregions of atomistic detail within a continuum domain because each instance of LAMMPS would have tried to use the same `atom` pointer.

One way to avoid the use of static variables is to make them all non-static members of a highest-level parent class, such as `System`, which all other classes inherit from.

However, this introduces its own problems. Each time a class derived from `System` is instantiated, it would create its own copy of the pointers in `System`. Thus if some part of LAMMPS destroyed and created a new version of one of the fundamental classes, the change would not be visible to all the other previously instantiated classes. At the heart of this problem is the fact that inheritance implies an "is-a" relationship but that, e.g., an instance of the `Atom` class is not a `System`. It is, however, implemented *in terms of* the `System` class, suggesting private or protected inheritance could be appropriate [6].

We thus opted to have no one class inherit `System` directly; instead we have each class receive a reference in their constructors to the one shared `System` instance. Doing just that, however, would require all references to pointers stored in the `System` class to use explicitly scoped code; e.g.,

```
double** x = system.atom->x;
```

This would require changing literally thousands of lines of code within LAMMPS, in all of its classes. To avoid this, all of `System`'s symbols need to be brought into the scope of each class that previously inherited it. This was done by creating a class called `UsesSystem` which consists of a reference to `System` as well as a reference to the handful of member variables of `System`.

```
class UsesSystem {
protected: // only those who inherit this class have access
  System& system;
  Memory*& memory;
  Error*& error;
  Atom*& atom;
  Domain*& domain;
  Comm*& comm;
  ...
public:
  // The constructor sets all references
  // to the members of the System it is given.
  explicit UsesSystem(System& sys)
    : system(sys),
      memory(sys.memory),
      error(sys.error),
      atom(sys.atom)
      domain(sys.domain)
      comm(sys.comm)
      ...
     {}
};
```

All classes in LAMMPS now inherit `UsesSystem` protectedly rather then publicly inheriting `System`.

For example a low-level class for performing file output can be declared as

```
class FileOutput : protected UsesSystem {
  FileOutput(int narg, char** arg, System& sys)
    : UsesSystem(sys) {
    ...
  }
  ...
};
```

and any of its methods can access atom coordinates in the desired way as `double** x = atom->x`, where the symbol `atom` now refers to `UsesSystem::atom` which is a reference to `sys.atom`. Thus `sys.atom` is now effectively a single pointer to per-atom quantities stored by this one instance of the LAMMPS library which can be used transparently by any class in LAMMPS.

A second, more-minor change we made to the LAMMPS library instantiation procedure was to allow the caller (the multiscale wrapper in our case) to assign a subset of processors (an MPI "communicator") to a particular LAMMPS instantiation to run on. Underneath, LAMMPS uses this communicator to partition the physical problem across processors, or can further sub-divide the communicator if it is asked to run multiple independent simulations simultaneously.

As a result of these changes, LAMMPS's `main()` function, used when LAMMPS runs as a stand-alone program, became simpler. It now simply initializes MPI, instantiates LAMMPS with `MPI_COMM_WORLD`, and directs standard input to the instance. That is, the `main()` function simply invokes one instance of LAMMPS as a library, similar to what the multiscale wrapper can now do in a more general fashion.

2.3 Enhancing the library interface to avoid files

LAMMPS's library API initially included three simple functions: one to initialize LAMMPS, one to send LAMMPS a single command in the form of a text string, and one to close LAMMPS. By invoking the second call repeatedly, this API allowed the LAMMPS library to be driven by a calling program similarly to how a run of stand-alone LAMMPS would be driven by reading lines from an input script. From a multiscale wrapper, this allows LAMMPS to be called repeatedly to compute forces on new atom coordinates without the need to launch and terminate an MPI process at each iteration. For example, the following command strings can be passed one-at-a-time to the LAMMPS library by the multiscale wrapper, for as many iterations as desired:

```
dump        1 all custom 1 force.values tag fx fy fz
read_data   atom.coords
run         0
clear
dump        1 all custom 1 force.values tag fx fy fz
read_data   atom.coords
run         0
clear
...
```

The dump command requests that the run command (for 0 timesteps) write a force.values file containing the ID (tag) of each atom and its force components (f_x, f_y, f_z). The read_data command reads in the initial coordinates of all atoms from the atom.coords file. The clear command can be used between successive runs to wipe out all previous data and start afresh.

This methodology still depends on files to pass atomic-level data between LAMMPS and various operations performed by the multiscale wrapper. To remove this I/O bottleneck, we added API functions to request the forces and positions of the atoms and to set new atom positions. This allows the multiscale wrapper to initialize LAMMPS once and then repeatedly have LAMMPS set new atomic positions, evaluate forces (via the run command), and request the forces. Effectively, the read_data and dump commands in the above script are replaced by library calls that write/read directly to/from the internal LAMMPS data structures used to store per-atom quantities.

The new library functions were written in two different flavors, described here and in the next sub-section. If LAMMPS is running in parallel, then each processor owns some subset of atoms, and per-atom quantities like positions and forces are distributed across processors. If the multiscale wrapper needs to know the attributes of all atoms, so that it can perform a serial operation on them, then the following library functions are used:

```
int  lammps_get_global_natoms(void* instance);
void lammps_get_global_coords(void* instance,
                              double* positions);
```

The first function returns the total number of atoms in the LAMMPS model. This is computed by performing an MPI_Allreduce() to sum the local atom count stored by each processor. Then the multiscale wrapper can allocate a vector, positions of length $3 \times n_{\text{atoms}}$ and pass it to lammps_get_global_coords(). This library function collects atom coords from all processors into a contiguous vector via an MPI_Allgather() call.

Clearly, the second function requires an $O(n_{\text{atoms}})$ operation, where n_{atoms} is the total number of atoms in the LAMMPS system, and is thus not scalable in a parallel sense. However, by accessing LAMMPS data structures directly, this interface is significantly faster than the communicate-by-file approach.

Note that each library function call includes an instance argument, which is the pointer returned when an instance of the LAMMPS library is instantiated. The library functions cast this void pointer into a LAMMPS pointer and use it to access class data or methods within LAMMPS via C++ syntax, e.g., the body of the first function above is simply

```
return ((LAMMPS*) instance)->atom->natoms;
```

2.4 Enhancing the library interface to enable parallelism

The previous sub-section described library functions suitable for having a single-processor multiscale wrapper invoke LAMMPS in parallel. It assumes that each processor will have sufficient memory to hold copy of all atomic position or force data. For small problems with a few million atoms, this may be a reasonable assumption, but not for large problems. Furthermore, on large numbers of processors, the cost of accumulating all data to a single-processor may dominate the computation. In addition, if the wrapper or its other single-scale components are also running in parallel (e.g., a continuum finite-element solver), then a true parallel version of the library functions is preferable, one whose cost scales as $O(n_{\text{atoms}}/P)$, where P is the number of processors.

These functions are actually simpler to implement than the global versions of the previous section, since they require no communication. Atom data within LAMMPS is stored in arrays in contiguous chunks of memory. For example, atom positions on each processor are stored in a double** x array which is dimensioned as n_{local} by 3, for the x, y, and z components of each of the n_{local} atoms owned by that processor. The individual values are stored contiguously as x_1, y_1, z_1, x_2, y_2, z_2, ... , $x_{n_{\text{local}}}$, $y_{n_{\text{local}}}$, $z_{n_{\text{local}}}$. Thus, these small functions suffice to enable the multiscale wrapper to directly access atom data within LAMMPS:

```
int lammps_get_local_natoms(void* instance) {
  return ((LAMMPS*) instance)->atom->nlocal;
}

double const* lammps_get_local_coord(void* instance,
                                     int i) {
  return &((LAMMPS*) instance)->atom->x[i][0];
}

void lammps_put_local_coord(void* instance,
                            int i,
                            double* pos) {
  double* x =
    &((LAMMPS*) instance)->atom->x[i][0];
  x[0] = pos[0];
  x[1] = pos[1];
  x[2] = pos[2];
}
```

This interface can be used to perform various operations in the multiscale wrapper in $O(n_{\text{atoms}}/P)$ time, without making any changes to LAMMPS itself. We note that this approach is also flexible because it removes the need for the multiscale wrapper to know how LAMMPS stores data internally. As an example, consider the problem of resetting the positions of atoms within a volume according to a displacement field. The volume and field are defined on all processors and are known to the multiscale wrapper.

```
void MoveAtoms(void* instance,
               Volume& v,
               DisplacementField& d) {
  for (size_t i = 0;
       i != lammps_get_local_natoms(instance);
       ++i) {
    double* pos = lammps_get_local_coord(instance, i);
    if (v.contains(pos)) {
      double displacement[3];
      d.displacement_at(pos, displacement);
      for (size_t j = 0; j != 3; ++j) {
        pos[j] += displacement[j];
      }
    }
  }
}
```

Consider a second example of a multiscale wrapper function, which makes use of MPI to count the number of atoms within a given volume, where `world` is the communicator for this invocation of the LAMMPS library.

```
int atoms_in_volume(void* instance,
                    Volume &v,
                    MPI_Comm world) {
  int count = 0;
  for (size_t i = 0;
       i != lammps_get_local_natoms(instance);
       ++i) {
    double* pos = lammps_get_local_coord(instance, i);
    if (v.contains(pos)) ++count;
  }
  int total_count;
  MPI_Allreduce(&count, &total_count, 1,
                MPI_INT, MPI_SUM, world);
  return total_count;
};
```

3 Generic persistent parallel data storage

Now that per-atom data can be extracted from and inserted into LAMMPS, it is useful to be able to store that data for later use. For example, computing the displacement of each atom from a previous time to the current time, requires knowing its previous position. One might think this can be calculated straightforwardly by storing a copy of each atom's previous position in the multiscale wrapper, and then extracting current positions. However, because atoms may migrate from one processor to another as they move, position data for a particular atom may not be on the same processor at a later time. LAMMPS handles this issue by allowing the user to invoke a "fix" command in the input script which creates a particular `Fix` class that stores needed per-atom quantities inside the class and has methods that are invoked to move that data with the atom when it migrates to a new processor. For example the `FixCoordOriginal` class in LAMMPS stores the original coordinates of an atom (at the time the fix command is invoked) with the atom so they can be used to compute a displacement at any future time.

To enable this functionality to be used more generally by a multiscale wrapper we added a `FixGeneralAtomVector` class to LAMMPS. It enables the multiscale wrapper to define an arbitrary number of new quantities it wants to store with each atom, which it does by instantiating the `Fix`, and then read/write these quantities through the library interface.

For example, the following lines of code in the multiscale wrapper compute displacements for each atom:

```
// setup FixGeneralAtomVector and store atom coords in it

lammps_atom_storage(instance, 3); // 3 quantities per atom
int nlocal = lammps_get_local_natoms(instance);
for (int i = 0; i < nlocal; i++) {
  double const* pos =
    lammps_get_local_coord(instance, i);
  lammps_put_atom_storage(instance, i, 0, pos[0]);
  lammps_put_atom_storage(instance, i, 1, pos[1]);
  lammps_put_atom_storage(instance, i, 2, pos[2]);
}

// run LAMMPS for 1000 timesteps

lammmps_command(instance, ''run 1000'');

// compute displacements at end of run
// nlocal may have changed

nlocal = lammps_get_local_natoms(instance);
std::vector<double> displace(nlocal);
for (int i = 0; i < nlocal; i++) {
  double const* pos = lammps_get_local_coord(instance, i);
  double dx = pos[0] -
    lammps_get_atom_storage(instance, i, 0);
  double dy = pos[1] -
    lammps_get_atom_storage(instance, i, 1);
  double dz = pos[2] -
    lammps_get_atom_storage(instance, i, 2);
  displace[i] = sqrt(dx*dx + dy*dy + dz*dz);
}
```

3.1 Example: multiscale energy minimization

The library functions of the previous section, which allow a multiscale wrapper to store and access per-atom quantities within LAMMPS, provide a building block for implementing more complex multiscale algorithms. For example, optimization algorithms seek to minimize an objective function in M-dimensional space. For an all-atom system, typically $M = 3 \times n_{\text{atoms}}$. This is the space the nonlinear conjugate-gradient algorithm used within LAMMPS normally operates on. To perform an energy minimization as a function of atomic coordinates, it iteratively evaluates the energy of the system and forces on each atom (gradient of the energy function), performs a line search in a direction in M-dimensional space which is a function of current (and past) gradient directions, and updates the position of each atom.

Now consider performing an energy minimization in a multiscale setting, where $M = 3N + C$. Here C represents additional degrees of freedom due, for example, to boundary conditions imposed on the atoms, and displacements of nodal points in a finite element mesh that surrounds the atomic region(s). Now it is desirable to write the nonlinear optimization algorithm in the multiscale wrapper itself, using calls to the single-scale components (LAMMPS, finite-element solver) to provide contributions to the overall objective function and gradient vector. This can be done straightforwardly for LAMMPS using the library calls previously described. The energy of the current configuration and per-atoms forces can be accessed by functions similar to `lammps_get_global_natoms()` and `lammps_get_local_coord()`, e.g., `lammps_get_global_energy()` and `lammps_get_local_force()`. Per-atom gradient information for previous iterations (needed for nonlinear conjugate-gradient and other optimization algorithms) can be stored and accessed through the interface to the`FixGeneralAtomVector` class. When atoms need to move in a gradient direction, their new positions are computed by the multiscale wrapper and can be passed to LAMMPS via `lammps_put_local_coord()`.

4 Performance results

To measure the performance of our LAMMPS library API, we consider its use in different modes of operation by a multiscale wrapper. For all of these tests, the code was built with the GNU C++ compiler with optimization, and run on an MPI cluster with 2.2 GHz AMD Opteron processors and switched gigabit Ethernet running Linux. The atomistic input was a rectangular block of atoms on a face-centered cubic (FCC) lattice, modeled with a simple (and computationally cheap) Lennard-Jones (LJ) interatomic potential. The reduced density of the system was 1.0 in LJ units (typical of a solid), with a pairwise cutoff distance of 2.5 σ, yielding about 60 pairwise neighbors per atom.

First, we examine the cost for the multiscale wrapper to run LAMMPS as a standalone code, exchanging atomic data with LAMMPS via files, to perform a single force evaluation. LAMMPS input and output file typically contain one line of data per atom with five numbers: a unique atomic ID, the atom's type, and the atom's xyz position. LAMMPS reads or writes these files (in text mode) at a rate of about 400,000 atoms/s. For LAMMPS to evaluate energies and forces for a configuration of atoms requires it to build neighbor lists and evaluate the LJ potential. For this cutoff distance, the neighbor list build is performed at a rate of roughly 160,000 atoms/s. The potential is evaluated at a rate of roughly 200,000 atoms/s.

The I/O operations are not parallel in LAMMPS (actually the output can be written to multiple files simultaneously by multiple processors, but an input configuration is not read in parallel). Thus the rate of 400,000 atoms/s is independent of the number of processors used. The neighbor-list and potential-evaluation rates are parallel, and will scale as roughly n_{atoms}/P so long as there are many atoms per processor. For example, running on 16 processors, the neighbor rate would be 2.5 million atoms/s and the potential-evaluation rate would be 3.2 million atoms/s. Thus, as more processors are used, the I/O rate will quickly dominate the run-time if LAMMPS is invoked by the multiscale wrapper in this way.

Next, we measure the cost for a multiscale wrapper running on a single-processor to invoke LAMMPS in parallel. Instead of using files to exchange atomic data, the global "put" and "get" functions of Section 3.1 are used. Recall that these use operations like `MPI_Allgather()`, which scale as $O(n_{\text{atoms}})$, n_{atoms} being the total number of atoms in the system, independent of the number of processors.

In this mode, the multiscale wrapper can retrieve per-atom coordinates or set per-atom forces in LAMMPS at a rate of roughly 2 million atoms/s, when running on between 2 and 64 processors. Since this faster I/O rate is still independent of the number of processors (to first order, although an `MPI_Allgather()` operation can have an additional cost term which is $O(\log(P))$ in the number of processors), it will again become a bottleneck as LAMMPS runs on more processors due to speed-ups in the neighbor-list and potential-evaluation rates. Additionally, we ran out of memory to store a copy of all atom coordinates on each processor when running a 16-million-atom problem on eight or more processors, showing the limitations of this approach. Note that the memory used by the multiscale

wrapper to store atom coordinates is in addition to the memory LAMMPS requires to store per-atom quantities (several per atom) and neighbor lists (many quantities per atom).

Finally, we tested the cost of a parallel multiscale wrapper accessing LAMMPS atomic data in parallel via the local "put" and "get" methods of Section 3.1 in the two ways. First, we had the multiscale wrapper compute the center-of-mass position of all the LAMMPS atoms by simply summing the atom positions (locally on processor, then across processor via an `MPI_Allreduce`), and dividing by the number of atoms. This operation performed at a rate of 9.6 million atoms/s per processor. That is, 154 million atoms/s on 16 processors. Second, we had the multiscale wrapper retrieve atom positions from LAMMPS via the `lammps_get_local_coord()` library function and make a local copy on each processor (which could then be used for further processing). This operation is $O(n_{\text{atoms}}/P)$ in both time and memory. It performed at a rate of 5.3 million atoms/s per processor.

5 Conclusions and future work

This work demonstrates that it is possible to efficiently reuse existing single-scale simulation software as a component in multiscale-simulation software. We showed that the software-development process for doing this can be made in incremental steps, and can be done without exposing internals of the single-scale software.

In particular, we illustrated the use of the LAMMPS molecular dynamics package in three different modes by a multiscale wrapper code: as a stand-alone application called by the multiscale wrapper, as a parallel library invoked by a serial multiscale wrapper, and as a parallel library invoked by a parallel multiscale wrapper. By adding different functions to the LAMMPS library interface, each of these three modes can be supported, and offer increasingly scalable performance.

For the last mode, the performance data of the previous section illustrate that the cost for the multiscale wrapper to directly access per-atom data in LAMMPS scales as $O(n_{\text{atoms}}/P)$ in the number of atoms, n_{atoms}, and number of processors, P, just as do the costs within LAMMPS to build neighbor lists and evaluate the interatomic potential. The timing numbers provided indicate that a multiscale wrapper operating in this mode can exchange per-atom data with LAMMPS without any significant bottleneck compared to the computations LAMMPS or the multiscale code would typically need to perform.

We also showed how it is possible to implement a multiscale algorithm for energy minimization outside of a single-scale code like LAMMPS, while using the single-scale code to provide portions of the data needed for optimization. This motivates a possible approach for minimizing the energy or other optimizations of coupled systems.

We note that the software-design approaches we advocate are straightforward to implement and could be applied to other single-scale software packages for use in a multiscale framework.

Finally, this paper focused on the interaction between a multiscale wrapper with one single-scale component. We have not addressed issues which might arise if two or more single-scale components were invoked by a multiscale wrapper and the single-scale components both operate in parallel and need to share information. For example, if a finite-element package uses one decomposition of the overall simulation domain, and LAMMPS uses another or decomposes only atomistic portions of the overall domain, then efficient exchange of data between the finite-element solver and LAMMPS would be complicated. It would potentially involve irregular point-to-point communication. We note previous work by one of the authors on this topic [7], but have not yet implemented a general solution to this issue in our multiscale framework.

Acknowledgments The enhancements to LAMMPS described in this paper are part of the general open-source release, available for download from http://lammps.sandia.gov/. This work is supported by in part by a SBIR project from the DOD (ARMY) to Simmetrix Inc. and Rensselaer Polytechnic Institute entitled "The Multiscale Application Suite" (project number W911QX-06-6-0048).

References

1. Shephard MS, Seol SE, FrantzDale B (2007) Toward a multimodel hierarchy to support multiscale simulation. In: Fishwick PA (ed) CRC handbook of dynamic system modeling. CRC Press, OH
2. Pacheco PS (1997) Parallel Programming with MPI. Morgan Kaufmann, CA
3. Snir M, Otto SW, Walker DW, Dongarra J, Huss-Lederman S (1995) MPI: the complete reference. MIT Press Cambridge, MA, USA
4. Parks ML, Lehoucq RB, Plimpton SJ, Silling SA (2008) Implementing peridynamics within a molecular dynamics code. Comput Phys Commun 179(11):777–783
5. Plimpton SJ (2008) LAMMPS benchmarks. http://lammps.sandia.gov/bench.html. Accessed March 2008
6. Sutter H (2008) Guru of the week: exception-safe class design, Part 2: inheritance. http://www.gotw.ca/gotw/060.htm. Accessed April 2008
7. Plimpton S, Hendrickson B, Stewart J (1998) A parallel rendezvous algorithm for interpolation between multiple grids. In: Proceedings of the 1998 ACM/IEEE conference on supercomputing. IEEE Computer Society Washington, DC, USA, pp 1–8