Current Opinion in Solid State and Materials Science 17 (2013) 271–276

Contents lists available at ScienceDirect

Current Opinion in Solid State and Materials Science

journal homepage: www.elsevier.com/locate/cossms

Developing community codes for materials modeling

Steven J. Plimpton [a,*], Julian D. Gale [b]

[a] Sandia National Laboratories, Albuquerque, NM 87185, USA
[b] Nanochemistry Research Institute, Department of Chemistry, Curtin University, PO Box U1987, Perth, WA 6845, Australia

ARTICLE INFO

Article history:
Available online 10 October 2013

Keywords:
Materials modeling
Open source software
Molecular dynamics
Lattice dynamics
Force fields

ABSTRACT

For this article, we call scientific software a *community* code if it is freely available, written by a team of developers who welcome user input, and has attracted users beyond the developers. There are obviously many such materials modeling codes. The authors have been part of such efforts for many years in the field of atomistic simulation, specifically for two community codes, the LAMMPS and GULP packages for molecular dynamics and lattice dynamics respectively. Here we highlight lessons we have learned about how to create such codes and the pros and cons of being part of a community effort. Many of our experiences are similar, but we also have some differences of opinion (like modeling vs modelling). Our hope is that readers will find these lessons useful as they design, implement, and distribute their own materials modelling software for others to use.

1. Introduction

As described in the abstract, community codes are openly-available software created and maintained by a (small or large) team of developers for a broader user community. The benefits of such efforts for users are obvious; they have free access to software that is continuously improving, resources to turn to for help, as well as input to the development process by requesting bug fixes or suggesting new features. Our experience has been that sponsoring a community code is also a net positive for the developers. Aside from the attendant glory and wealth associated with free software (ok, that's a stretch) the benefits chiefly result from your code having users. Specifically,

- users find and report bugs,
- users suggest improvements and new capabilities for the code,
- users may implement new ideas themselves and give you new or improved code,
- users cite your papers,
- users may become new collaborators and colleagues.

The only downside of having users is the extra work required to support them. This effort comes in the form of documenting your code's capabilities, answering questions, and responding to user feedback. The level of support offered is up to you, so it does not have to be onerous. After all, users get what they pay for, and the code is free. Indeed many such codes come with express statements that support cannot be provided. However, in reality most developers relent and are willing to offer as much assistance as time permits. As a user community enlarges, support can be offered by a growing group of people via web-based forums and mail lists.

As examples of community codes for materials modeling, we highlight our experience with LAMMPS and GULP, since the authors are their lead developers. In order to give context to the subsequent discussion, we start with a short historical perspective on these two codes.

LAMMPS is a classical molecular dynamics (MD) code [15,17], begun in the mid-1990s as a cooperative effort between two US DOE laboratories and several industrial partners to develop a parallel MD code, since parallel machines were then an up-and-coming novelty. For its first 10 years LAMMPS was free but required new users to sign a perfunctory license agreement, which about 100 users did. In hindsight such a license was a significant barrier to attracting users since it often meant a lawyer wanted to read it. In 2004, we re-wrote the code in C++, to make it more flexible for adding new features, and released it openly under the GNU General Public License (GPL) [14]. It was downloaded 1000 times in the first four months, and 150,000 times to date. Since its open release LAMMPS has grown in size from about 50 K to 500 K lines of code, as developers and 100+ users have contributed new code. Some of the new capabilities are features we never imagined being part of LAMMPS or even an atomistic MD code, such as treatment of electrons as variable-radius particles [12], continuum-scale models of fracture dynamics via particles [16], or coupling granular particles to finite-element fluid solvers for two-phase flow modeling [2,13]. Our chief mechanism for supporting LAMMPS users is a mail list where questions or problems can be posted, which we began a year

* Corresponding author.
E-mail addresses: sjplimp@sandia.gov (S.J. Plimpton), J.Gale@curtin.edu.au (J.D. Gale).

1359-0286/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cossms.2013.09.005

after the code's open release. The list now has 1200 subscribers and an archive of 40 K postings.

Similar to LAMMPS, GULP is also based on exploiting a classical force field description of interatomic forces [6,8,9], though there was also a brief flirtation with periodic semi-empirical quantum mechanics [7]. In contrast to LAMMPS, GULP targets the more niche application of lattice dynamics, though it also has a molecular dynamics capability. Because of this focus, our objective has been to provide high levels of analytical derivatives that allow accurate calculation of mechanical and phonon properties for crystalline materials. GULP also started in the early 1990s, at the Royal Institution of Great Britain, as an attempt to automate the fitting of interatomic potential parameters [5], which previously had to be undertaken by hacking source code for each desired fit. At that time disk space was at a premium and was often exhausted by building a different executable for every job!

Initial distribution of GULP began by sharing copies with other groups in the UK, but eventually progressed to the point where it was distributed to any academic group by emailing tar files. In the late 1990s, this was placed on a more formal footing by Imperial College. Instead of opting for an open-source approach, a different licensing arrangement was made. While the right to free access for academics was enshrined in the agreement, commercial distribution was assigned to the company that is now Accelrys Inc. In this respect, and the fact it remains a staunchly proud Fortran code, the pathway for GULP has forked considerably from that taken by LAMMPS. However, there are also many similarities, including the size of the current version which also runs to nearly 500 K lines of code. Today, academic distribution is handled by an automatic web-based registration system; access for anyone with a University email address is thus almost instantaneous, similar to a GPL code. Because of the different access mechanism, we tally registered users rather than downloads; this currently runs to more than 6000 people.

The next section distills six lessons we have learned about what helps a materials modeling code attract a community of users. Many of the ideas reflect the current state of LAMMPS and GULP, but we learned them by trial-and-error and implemented them incrementally. We argue that turning your home-grown research code into a community code is not a decision with a large energy barrier, but is more a philosophical approach to software design, development, and release. Like most software tasks, adopting a community-oriented strategy is least difficult if it is part of the up-front design of your software and maintained incrementally over time. But of course existing legacy software can also be released openly and become a community code; hopefully these ideas will benefit that process as well.

2. Lessons learned

Here are six rules-of-thumb for creating and maintaining a successful community materials modeling code:

1. make something people want,
2. the perfect is the enemy of the good,
3. make it easy for others to understand, modify, and extend your code,
4. choose an appropriate license,
5. support your users,
6. choose an appropriate name for your software.

2.1. Make something people want

This is a mantra of the internet start-up culture [10] when a handful of friends create a company to turn their software idea into money. In a commercial setting, it seems obvious that customers will only visit your web site, use your software, and give you money if they get some value from it. But it also applies to freely-available research software. Knowing how to write code to perform some computational task is a necessary first step, but is not sufficient to attract users to your software.

Asking yourself questions such as these is a useful exercise: What problem do people want to solve? How can our software make it easier for them to do so? In what ways will our code be different or better than others that already exist? It also helps to examine your software from a new user's perspective. Every programmer knows it is easy and fun for an expert (you) to write code with only yourself in mind, adding features or obscure and tricky options that confuse mere mortals. If your software is not easy for a new user to quickly do something useful with, they probably will not continue to use it. If your code has a mailing list where new users post questions, you will be surprised by what issues they stumble over which you never thought would be important.

Our initial goal with LAMMPS was to create an MD code which exploited spatial parallelism, in hopes it would scale well to new machines with hundreds or thousands of processors. At the time GULP was conceived, there had been a series of codes going back several decades that already fulfilled the same basic need to optimize the structure of solids from interatomic potentials and compute their properties. This is nicely captured in a tribute to the pioneering contribution of Michael Norgett [21]. So why create yet another program in this field? Largely, this came from frustration, as a user of the programs of the day, that the input file format was too rigid and less friendly than it could be. Furthermore, for those not of a C persuasion, the timing coincided with the arrival of Fortran 90 and dynamic memory allocation. By removing the need to regularly recompile for each problem, this allowed distribution of executables for the benefit of those not inclined to worry about the finer details of computers. Thus experience as a user, combined with changes in technology, created the opportunity for a new code.

2.2. The perfect is the enemy of the good

This aphorism is attributed to Voltaire, who apparently was a savvy software developer. When you first contemplate releasing your code to the unwashed masses, you imagine users will pore over its innards, test every option, and mock you whenever it breaks. Thus the natural tendency is to wait to release until you are confident the code is near-perfect. Aside from the improbability of ever reaching that state, the problem with this strategy is you delay having users, with all the benefits listed in the introduction.

A better mantra for community codes is release early and often. For example if your current code has 5 simple bullet-proof features and 3 bleeding-edge brittle ones that are only suitable for experts, you are better off removing the 3 features and making an initial release of the simpler version of the code. You may get feedback that a capability you had not thought of is more useful than the 3 you are working on. Or when you do release the bleeding-edge features (one at a time), you will hopefully have users eager to try them out and give you feedback about what works and what does not and what would make the new feature easier to use. Having such beta testers has a synergistic effect on the development process, speeding the rate at which a bleeding-edge feature is converted into a bullet-proof one. It is also more satisfying to a developer to release something immediately and get feedback than it is to wait for perfection.

With LAMMPS, we initially released a new version of the code a few times a year. The releases became artificial deadlines which developers stressed over getting code ready for. Instead, we now release every bug fix or new feature as soon as we finish it, posting a patch and new tarball on our web site, often 100 s of times per