Streamlining Development of a Multimillion-Line Computational Chemistry Code

Robin M. Betz and Ross C. Walker | San Diego Supercomputer Center

Software engineering methodologies can be helpful in computational science and engineering projects. Here, a continuous integration software engineering strategy is applied to a multimillion-line molecular dynamics code; the implementation both streamlines the development and release process and unifies a team of widely distributed, academic developers.

The needs of computational science and engineering (CSE) projects greatly differ from those of more traditional business enterprise software, especially in terms of code validation and testing. Although software engineers have expended considerable effort to simplify and streamline the development and testing process, such approaches often encounter problems when applied to the scientific software domain. Testing is a notorious example of this difference between the two fields. Although business applications with a well-defined usage are easily testable, scientific applications are quite a different story. When the desired outcome of a program is an unknown subject of research, the traditional measurements of software validity used by software engineers are difficult or even impossible to define or establish. Tools designed to simplify the development process often don't mesh readily with the goals and development timescale of CSE codes, and as a result, test suites are typically written from scratch or not at all.

CSE codes are also typically developed quite differently from other applications. Developers working on scientific code usually have backgrounds that are quite different from those of software engineers, approaching the discipline primarily from a scientific background, and as such they're often unfamiliar with good development practices. This is further complicated by the fact that groups are often distributed among many universities and even countries, making collaboration and group decision making complicated compared to a group of software engineers working in the same office. CSE developers are also researchers first and foremost, and their goal is primarily to generate publication-quality research, rather than develop maintainable, extensible code using the latest development methodologies. This frequently results in a developer culture that's resistant to change, as the time required to implement and understand a new methodology can be prohibitive in a research environment.

Although the tools and methods used in software engineering can be difficult to apply to scientific projects, they provide significant benefits to the development of scientific software. In this case study, we adapt the software engineering practice of continuous integration to assist in the validation and testing of the molecular dynamics code package, Assisted Model Building with Energy Refinement (Amber; http://ambermd.org).[1] Although there were several challenges in applying this methodology to Amber's complex and diverse code base, the introduction of software engineering tools to Amber has proven to be extremely useful in unifying a geographically separated group of developers with different computer and science backgrounds, and ultimately has provided considerable benefits to the project as a whole.

Background

Amber is a package of molecular simulation programs that's widely used within the computational chemistry and computational molecular biology communities. It includes a wide variety of programs that enable the simulation of molecular systems at the atomic level. It includes tools for all stages of the simulation workflow, from setting up the simulation, to running it efficiently, to conducting a comprehensive analysis of the results. Amber also includes a set of molecular mechanics force fields that describe how atoms interact within a variety of settings, and both the software and parameters are under continuous development.

Amber has several unique characteristics that set it apart from traditional software engineering projects. First, the project has been under continuous development since its original publication in 1981 by the late Peter Kollman.[2] The first versions of Amber were written on punch cards, and evidence of this history can still be found in some of the code. For example, some of the input file formats for Amber are still described in terms of cards and namelists, which can be quite confusing for new users.

The code itself reflects Amber's extensive development history. As with many CSE codes, most of the simulation code is in Fortran. Although the language is no longer commonly used in software engineering, it remains popular in the CSE community because of the complexity of transitioning existing code, and the efficiency and ease of optimization and parallelization. Although business applications are frequently translated to more modern languages or rewritten for newer technology, the effort of transitioning Amber to a single, modern language is prohibitive given the funding and time constraints placed on researchers. As such, most new code added to Amber tends to be in more commonly used languages, with C, C++, CUDA, and Python being typical. The integration of multiple languages into a single set of executables adds to the complexity of building Amber.

Amber developers are a geographically and academically diverse group, and are spread across the United States and abroad, with limited opportunities for direct communication. Currently, the principal developers are at Rutgers University, the University of Utah, the University of Florida, SUNY-Stony Brook, and the University of California, San Diego, with many other research institutions also contributing.

The developers have a diverse set of research interests and typically contribute to the code as it relates to their research rather than considering code development as their primary focus. This approach is extremely common in the CSE world, where researchers often have neither the time nor funding to focus solely on software development.

Because each developer works primarily on one program or aspect of a program, the Amber package is extremely modular, and the package is better described as a toolset than a single integrated simulation code. As a result of diverse contributions, Amber has expanded from its original incarnation as a preparation and simulation program and now supports force-field development, trajectory analysis, and a wide variety of simulation types across all major hardware from a single laptop to the most powerful supercomputers.

The range of hardware on which researchers run the Amber software is staggering. It has to run on all major architectures, including less commonly encountered hardware such as IBM Power, Blue Gene, Cray, and even ARM-based systems. It additionally exploits GPUs to accelerate computation when possible[3] and uses a message passing interface (MPI) for parallel processing. In its 32 years of existence, the Amber software has undoubtedly been run on every major supercomputer and every version of micro- or vector processor.

To be so portable, the program must be compiled from source by the user during the install process, and must therefore support all major compilers, including Intel, GNU, PGI, open64, Cray, IBM, and Solaris Studio. The OpenMP[4] and MPI[5] toolkits are used for parallelization, and much of the simulation code also exploits GPUs using CUDA,[6] allowing for extremely cost- and time-efficient simulations, but at the expense of even more complex cross compilation.

Because each developer focuses primarily on coding parts of Amber that relate to his or her research interests, the software's functionality as a whole is often neglected. Amber developers meet once a year to discuss the code base's future, but these meetings aren't sufficient for addressing the frequent problems that can arise in such a large project. Many developers don't have the time or opportunity to extensively test their code for every compiler that Amber can use, and often don't have access to some of the high-end GPUs and other exotic hardware that Amber supports.

The code is also highly complex, and as such, changes in one part of the simulation code might have side effects that aren't caught in an individual's examination of his or her own work. This is especially true for the parallel and GPU-accelerated builds, because errors such as race conditions are often subtle and difficult to replicate or identify. These errors won't result in a program crash or error message, might occur infrequently, and manifest as extremely subtle differences in results.

Overall, these complications result in errors that are reported by users of hardware or compilers that haven't been extensively tested as part of development. As a result, those researchers can lose precious computer time and developers must expend considerable effort in correspondence to identify the exact cause of the error. Debugging the problem can take even more of a developer's time, especially if the error is in code that was written a long time ago.

Thus, the problems of the Amber project clearly differ considerably from those encountered in traditional software engineering environments, where developers work closely on a unified code base with well-defined goals and managerial oversight. However, there remain common points of failure between all computational projects: all code must be compiled and tested in some way before release, no matter the domain. In this case, it's both possible and useful to apply solutions from software engineering to address the problem of identifying errors introduced into the build and test process of Amber and other CSE codes.

Solutions from Software Engineering

The software engineering practice of continuous integration[7,8] was adapted to create a common build-and-test environment in which errors can be seen on a commit-by-commit basis. This saves developers time by running many tests on their own machines, enables verification of more compiler and system combinations, and, most importantly, catches bugs more quickly.

We created a dedicated build-and-test system that verifies that all supported compiler and parallel combinations build and correctly run the tests. The system is available to developers along with a commit history to the central repository so they can identify if their commit broke a test case or if it doesn't work with a certain compiler (that they might not have access to on their personal machines).

Martin Fowler identifies several elements of a proper continuous integration environment.[9] Most of these principles are already present in the Amber project, having been introduced by agreement among the developers regarding good practices. However, the developers were unaware of continuous integration and as such had not yet created a dedicated continuous integration server nor formally identified its components. We therefore identify each of these principles and discuss their applicability and incorporation into Amber development.

Maintain a Single-Source Repository

One important aspect of continuous integration is maintaining a common source repository that all developers can access. This prevents divergence in the code that leads to difficult-to-resolve conflicts close to release, and allows developers to monitor each other's work.

The current Amber development process maintains a common git repository that all developers add their code to once they're comfortable sharing it with others. The two main branches are of the current release with patches, and a development branch containing code that's robust enough to share but might not be fully ready to release. Each developer commits code incrementally to the local branch, which can be hosted on a local machine or on the central git server, which conducts comprehensive backups and allows sharing of such development branches. This repository typically receives upward of 10 commits per day.

Owing to Amber's development process, commits typically occur frequently and without conflict. Merge conflicts are rare, because each developer typically stays focused on a subset of the code, and only edits other subsections to fix small errors, or with permission and collaboration. Programming can also be sporadic on the part of many researchers, because they typically use Amber in a research capacity most of the time and conduct development work only occasionally. To avoid local branches becoming substantially out of date, each developer is encouraged to create a new branch when starting work on a new project and to pull and merge with the master as frequently as possible.

Automate the Build and Tests

As projects get larger, automated build-and-test systems can free developers from having to spend time compiling source files and running tests on their own. An automated build process allows for dependency resolution and easier support for many build options, such as the conditional compilation of CUDA code on systems with GPUs.

Amber uses GNU Make to automate its build-and-test process. Users run a configure script that generates variables appropriate to their desired installation, compiler, and options, and Make does the rest. A root-level Makefile calls sub-Makefiles in each folder containing a program. To add a new program to Amber, a developer need only write a sub-Makefile for the code and call it in the right place in the build hierarchy. A similar Makefile structure is used to run tests on each program.

Software engineering solutions for this problem can become cumbersome when applied to codes like Amber. The history of Amber development, th code's complex nature, and the variety of systems on which it must be run justify the use of GNU Makefiles. Although other projects have had success with alternative build systems such as CMake, the benefits of converting Amber to CMake don't currently offset the effort that would be involved. Testing the resulting software also can be difficult, as the desired behavior of code is often the subject of research, and can be highly sensitive to convergence and sampling issues. Nonetheless, validation is crucial, as the results of research depend on it.

The bugs Amber encounters aren't limited to algorithmic and logical bugs originating from coding errors. Complex race conditions can occur in parallel code, libraries can be problematic, and bugs can be introduced in the machine code due to flaws in the compiler's logic. On multiple occasions in Amber's history, running the code has exposed underlying design flaws in the microprocessor architecture. As a project's complexity grows, the possibility of an error-producing interaction between two program components grows exponentially. Testing is so critical to the success of a project that validation needs to be automated and integral to the build process, so developers don't have to remember to test their code manually.

Software engineering considerations dictate that tests be conducted in the production environment rather than just on developer machines. We define the production environment for our tests as any computer, using any compiler and/or architecture. The requirement of testing in this environment means that results must be reproducible no matter what machines users have or system they're trying to simulate. The Amber tests try to reflect real-life use as much as possible, using real input data rather than contrived examples. As a result, the project contains more than 1 gigabyte of data in the test cases alone.

Amber adopts the philosophy that results should be both valid and reproducible, and developers construct their tests accordingly. Once code is complete and the developer has verified it for correctness, the correct output (with a set random seed, if the program uses the random number generator) is saved. Tests will run the program (with the same random seed) and compare the output to the saved correct output. A tolerance value is set to account for floating-point rounding differences arising from different architectures and environments, and if the difference between the two outputs is within the tolerance, the test passes. Otherwise, it fails. The test suite verifies all programs released as part of Amber, and conducts tests that vary from building and saving a molecule to running short simulations at different levels of machine precision.

Simplify Deployment and Executable Distribution

Deployment refers to the processes by which software becomes usable on a new machine. Efficient, automated deployment is a focus of continuous integration because without a streamlined installation process, automated building and testing is impossible. Software engineering projects often provide a binary installer for supported architectures that sets up binaries, libraries, and data files that the program needs to run. Even open source projects usually provide precompiled binaries in addition to source code to simplify installation for users. However, this isn't always possible for CSE projects such as Amber, which must run on multiple architectures.

Distribution of updates is critically important to CSE projects, as the accuracy of users' results can depend on it. Developers should implement some sort of checking for updates automatically and ensure that users can quickly and easily update to the program's latest version. Software engineering projects have similar concerns involving distribution of updates to patch security holes or runtime errors, and scientific software developers can use similar methods to deliver updates.

Amber's releases are often patched several times, and many of these patches correct problems that might lead to erroneous simulation or offer significant speed improvements over the unpatched version. It's therefore crucial that users have the latest version and can easily patch their software. We ensure that users are aware of new patches by encouraging subscription to a mailing list where patches are announced and users may submit questions regarding use of the software.

The patch process for Amber 12 and AmberTools12 has been greatly simplified with the addition of an automatic patching script. When users prepare for compilation by running the configure script, the patcher is automatically invoked and checks the Amber website for project updates. It then retrieves and applies the latest patches if necessary. The user must then recompile Amber, because, given the heterogeneity of architectures Amber runs on, it's impossible to deliver binary patches for the software that would work for all users.

Deployment of CSE codes can range from trivial to extremely complex depending on the application and system. For Amber, users need only run the configure script, install dependencies if necessary, and then run Make to obtain a fully functional system in the time it takes to make a large cup of coffee. Other CSE codes might need a more involved process to resolve dependencies and make executables accessible to the user.

Every Push Should Result in a Build

The most important component of continuous integration involves building and testing the code on a dedicated machine or group of machines that frequently reports to all the developers. This enables code to be automatically compiled with all combinations of options, and the tests may be monitored so that the effects of each commit are evident almost immediately, rather than having to unravel which commit is responsible for a particular error.

Many implementations of continuous integration servers exist, including Buildbot, Jenkins, CruiseControl, Automated Build Studio, and Hudson. The choice of server depends on the compile language, the tests being run, the user interface desired, and numerous other factors—including cost and choice of open source or proprietary license.

This was the most challenging element of continuous integration to introduce to the Amber project, because continuous integration software is designed for use in the software engineering community and there was no out-of-the-box server that offered all of the features we needed for our scientific code. We set up a continuous integration server using Thoughtworks' CruiseControl (http://cruisecontrol.sourceforge.net) to conduct automated building and testing in an easily extensible manner. Although we encountered difficulties in adapting the software to fit the broad needs of the Amber project (see the "Implementation Details" section), the introduction of the server has helped simplify debugging and has resulted in numerous improvements to the build and tests.

To keep build and test times to a minimum, we built a dedicated machine to act as a server. Solid state drives were RAIDed to provide greater than 800 megabits per second (Mbit/s) of I/O and to allow for a number of targets to be run simultaneously without the I/O throttling that can occur with spinning disks. We selected a high-end Nvidia M2090 GPU to run the GPU-accelerated tests, and used an 8-core processor at a high clock rate of 3.8 GHz to minimize the time for executing serial aspects of the build process. This machine cost approximately $3,200, with the most expensive component being the $2,000 GPU.

The compilation time for one of the Amber build targets is approximately 8 to 10 minutes on this machine, but the timings for the tests vary from less than 10 minutes to more than half an hour, depending on whether the tests are for serial, parallel, or GPU code. Because the purpose of our continuous integration server is to test as many combinations of serial, parallel, and GPU code with various compilers as possible, the server is set up to allow multiple targets to be run at once to hasten overall completion. This incremental testing is often referred to as a staged build, build pipeline, or deployment pipeline in the software engineering community.

Development Should Be Communicative and Collaborative

Continuous integration aims to simplify the development process by increasing transparency of results and simplifying developers' access to information regarding the state of the code. Using a server to conduct automated builds allows developers to see who is committing and whose commits cause errors or significantly impact program speed.

Good communication is important to any software development project, and especially to scientific endeavors in which contributors are distributed among various institutions. Continuous integration practices allow geographically dispersed teams to work more closely with each other, and constant feedback on which parts of the code colleagues are developing allows researchers to more clearly see opportunities for contribution.

The addition of the Java-based continuous integration server, CruiseControl, has been extremely beneficial to Amber development in this respect, as previously developers couldn't see the immediate effects of their commits on every compiler and option. This often led to confusion over what broke the build, and overall the development process was far more opaque.

Implementation Details

The continuous integration principle exists mainly within the mainstream software engineering community, and as such can be difficult to apply to CSE projects. The implementation of an automated build-and-test server is perhaps the most important step of the process, and unfortunately the most challenging when it comes to scientific applications.

Of the 28 most common available continuous integration servers, 60 percent were under a propri etary license, making them difficult to modify an potentially unaffordable on a limited research budget. Only half of the remaining servers supported building from the command line, and even fewer allowed success or failure notifications more complex than an email. Amber uses the CruiseControl continuous integration server, which is primarily aimed at Java applications but offers an elegant Web dashboard that displays useful information about each build target. The choice of CruiseControl was made with the Amber build and test environment in mind, although it wasn't the only logical choice. For example, the molecular dynamics project Gromacs[10] has had considerable success with the Jenkins continuous integration environment (http://jenkins-ci.org).

CruiseControl assumes that all code tested with it will also be in Java and built using Apache Ant, which uses XML-based files to define how to compile each target. Although Ant is primarily aimed at compiling Java bytecode, its build definitions let users execute arbitrary commands. Our build scripts each run the configure script with the appropriate flags for the target, then executes "make install" to begin compilation. Any errors in this process result in the build being marked as failing. All of the compilation targets are triggered whenever the code in the main git repository is modified.

Build targets were created for all possible combinations of serial, parallel, CUDA serial, and CUDA parallel with the GNU and Intel compilers for Amber and AmberTools. Several additional targets were created to test building in parallel with different make job counts, executing "make –jX," where X is typically 1 to 16. This checks that the build order dependencies are defined correctly in the Makefiles.

Test cases were defined as separate targets that are triggered to run following a successful build of the source code corresponding to the options and compiler of the test target. Test targets aren't run or shown if the compilation of the build target fails, and are considered to pass if every test passes Amber's comparison process.

Continuous integration servers must make build results easily available to all developers. Most servers feature a Web interface or regular email updates, although numerous options are available, including desktop notifications, RSS, or even a Twitter feed. The Amber project used a Web interface. CruiseControl's Web interface, or dashboard, is designed for software engineers who need an at-a-glance display of whether the application is ready to deploy, and features colored squares that indicate whether a project is building, succeeding, or failed.

This sort of display is insufficient for the needs of a scientific software project, where failure context is

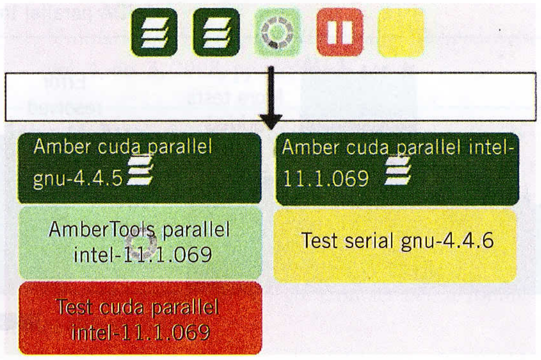
Amber cuda parallel gnu-4.4.5
Amber cuda parallel intel-11.1.069
AmberTools parallel intel-11.1.069
Test serial gnu-4.4.6
Test cuda parallel intel-11.1.069

Figure 1. Modified CruiseControl Web interface. Many improvements were made to the interface, including making project names easily viewable in the main build window. Targets are represented with colored squares by default.

extremely important. For example, if a new compiler version is introduced and all of the tests suddenly start failing, the problem might be a bug introduced in the compiler rather than in the code. This scenario is surprisingly plausible—certain compiler versions won't compile Amber correctly due to bugs within the compiler itself.

Because CruiseControl is open source, we were able to modify the Web dashboard to provide more useful information about each build and test. As Figure 1 illustrates, instead of showing colored squares, each build is now clearly labeled with its name, so developers can find the compiler and build options resulting in failure with only a quick glance at the dashboard.

This dashboard setup provides basic success or failure information for many different ways of compiling Amber and running the tests. However, to make the dashboard useful to developers, more information is needed regarding test results. The nature of each test failure is important. Frequently, the tolerance on a test will be too stringent, and rounding differences will cause the system to report a failure. Each test target therefore runs a bash script that collects all of the diff files created for a test run, and CruiseControl is configured to make them available via the dashboard for easy inspection. If a build fails, the error is shown on the dashboard, and log files from both the build and test targets are made available for download if developers wish to have more information about tests that crashed before completion.

Testing for program correctness isn't sufficient for the Amber project, however, because performance is of paramount importance in our simulation code. Someone could quite plausibly introduce

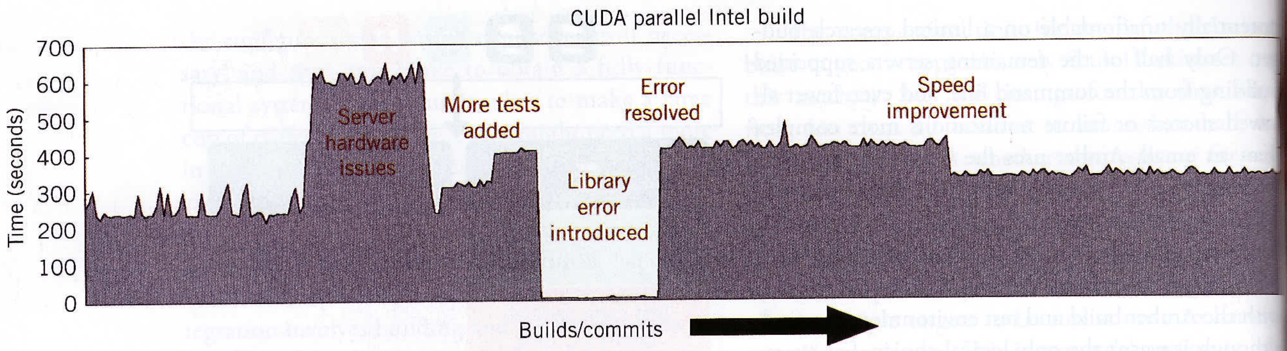

Figure 2. Graphical display of time taken to complete the test targets for the CUDA parallel build of Amber using the Intel compiler. Machine load affects the timing information somewhat, although the baseline level stays the same. Sharp changes in build time have occurred from a variety of causes, and are labeled accordingly.

a useful commit that introduces no errors but significantly increases runtime. Such commits are nearly impossible to identify before release, so finding them right after they're written is very helpful. We define an approximate measure of code efficiency as the time taken to complete the test targets.

Although CruiseControl doesn't support the collection and display of timing information, we extended it to provide this functionality. Time stamps are written before and after running the tests, and these values are passed to a script that subtracts them and appends this value to a data file. The gnuplot graphing program is invoked to create a graph of these values for the most recent 50 builds, and this is made available on the Web interface so developers can have a visual representation of how the latest commits affected program speed. Each data point on the graph is marked by the number of tests that failed with that build.

This graphing functionality has been extremely useful to the Amber developers, who in addition to gaining timing information now have data about the number of failed tests in an easily accessible format. Figure 2 gives an overall picture of the changes in Amber's tests for one build target. Changes in measured timing data can occur from changes in code efficiency, addition or removal of tests, dynamic linking errors that prevent program launch, and/or machine latency.

Although the amount of effort required to set up and modify a continuous integration server to fit the needs of scientific software might deter CSE projects from adopting these techniques, once the server was established, convincing developers of its utility was straightforward, as the gains from having immediate validation of each commit quickly became evident.

Future goals for Amber's continuous integration environment include the creation of a benchmarking server to track the performance of serial, parallel, GPU serial, and GPU parallel Amber targets on standard molecular dynamics benchmarks and the implementation of virtual machines to more thoroughly test different build environments. We might investigate other continuous integration servers to provide this functionality, or continue to extend CruiseControl if at all possible.

Overall, using continuous integration software in the Amber project has been worth the effort, and although we continue to look to improve the usefulness and robustness of the testing system, it has already been of significant utility in accomplishing development goals and fostering unity among a diverse team of developers. ■

Acknowledgments

Ross Walker received funding for this work from the US National Science Foundation through the Scientific Software Innovations Institutes Program—NSF SI2-SSE (grants NSF1047875 and NSF1148276), a University of California grant (UC Lab 09-LR-06-117792), and a CUDA fellowship from Nvidia.

References

1. R. Salomon-Ferrer, D. Case, and R. Walker, "An Overview of the Amber Biomolecular Simulation Package," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 3, no. 2, 2013, pp. 198–210.
2. P.K. Weiner and P.A. Kollman, "Amber: Assisted Model Building with Energy Refinement. A General Program for Modeling Molecules and Their Interactions,

J. Computational Chemistry, vol. 2, no. 3, 1981, pp. 287–303.

3. R. Salomon-Ferrer et al., "Routine Microsecond Molecular Dynamics Simulations with AMBER on GPUs. 2. Explicit Solvent Particle Mesh Ewald," *J. Chemical Theory and Computation*, vol. 9, no. 9, 2013, pp. 3878–3888.
4. L. Dagum and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming," *Computational Science & Eng.*, vol. 5, no. 1, 1998, pp. 46–55.
5. M. Snir et al., *MPI: The Complete Reference*, MIT Press, 1995.
6. Nvidia, *Compute Unified Device Architecture Programming Guide*, 2007; http://docs.nvidia.com/cuda/cuda-c-programming-guide.
7. P. M. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley Professional, 2007.
8. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Addison-Wesley Professional, 2010.
9. M. Fowler and M. Foemmel, "Continuous Integration," *Thought-Works*, 2006; http://martinfowler.com/articles/continuousIntegration.html.
10. B. Hess et al., "GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation," *J. Chemical Theory and Computation*, vol. 4, no. 3, 2008, pp. 435–447.

Robin M. Betz is a member of the Walker Molecular Dynamics Lab at the San Diego Supercomputer Center. Her research focuses on developing methods for improving and assessing the accuracy of molecular dynamics simulations. Betz has a BS in bioinformatics from the University of California, San Diego. Contact her at robin@robinbetz.com.

Ross C. Walker is an associate research professor at the San Diego Supercomputer Center, an adjunct associate professor in the Department of Chemistry and Biochemistry at the University of California, San Diego, CEO of Verizyme, and an Nvidia Fellow. He also runs the Walker Molecular Dynamics Lab in San Diego. His research interests include developing classical and quantum mechanics/molecular dynamics (QM/MM) techniques. Walker has a PhD in computational chemistry from Imperial College. Contact him at ross@rosswalker.co.uk.