Implementing Continuous Integration Software in an Established Computational Chemistry Software Package

Robin M. Betz
San Diego Supercomputer Center
La Jolla, CA 92093
USA
rbetz@ucsd.edu

Ross C. Walker
San Diego Supercomputer Center
and
Department of Chemistry and Biochemistry
UC San Diego
La Jolla, CA 92093
USA
ross@rosswalker.co.uk

***Abstract*—Continuous integration is the software engineering principle of rapid and automated development and testing. We identify several key points of continuous integration and demonstrate how they relate to the needs of computational science projects by discussing the implementation and relevance of these principles to AMBER, a large and widely used molecular dynamics software package. The use of a continuous integration server has both improved collaboration and communication between AMBER developers, who are globally distributed, as well as making failure and benchmark information that would be time consuming for individual developers to obtain by themselves, available in real time. Continuous integration servers currently available are aimed at the software engineering community and can be difficult to adapt to the needs of computational science projects, however as demonstrated in this paper the effort payoff can be rapid since uncommon errors are found and contributions from geographically separated researchers are unified into one easily-accessible web-based interface.**

I. Introduction

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. When the desired outcome of a program is an unknown subject of research, development of unit tests and other traditional measurements of software validity is difficult or even impossible. Tools designed to simplify the development process often do not mesh readily with the goals of CSE codes, and as a result test suites are typically written from scratch and then later abandoned entirely.

In addition, development of CSE codes occur among groups with backgrounds that are very different from that of a software engineer– most CSE researchers approach the discipline primarily from a scientific background, and are 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.

This paper presents an approach to simplifying the development of the molecular dynamics code AMBER [1] [2] by implementing continuous integration software. The challenges of making this work with the complex and diverse code base are discussed, as are the various successes of the implementation. The introduction of software engineering tools to AMBER has ultimately proven to be extremely useful in unifying a geographically separated group of developers with very different computer and computational science backgrounds.

II. Project Background

Assisted Model Building with Energy Refinement, or AMBER, is a package of molecular simulation programs that is 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 a number of unique characteristics that set it apart from traditional software engineering projects. Firstly, the project has been under continuous development since its original publication in 1981 [3] by the late Peter Kollman. The very 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, a fact that can be quite confusing for new users.

The code itself reflects AMBER's extensive development history– as with many CSE codes, the majority 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 due to the complexity of transitioning existing code, and the efficiency and easy ability to optimize and parallelize FORTRAN code. It should be noted that most

978-1-4673-6261-0/13 © 2013 IEEE

SE-CSE 2013, San Francisco, CA, USA

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 an extremely geographically and academically diverse group, and are spread across the US and abroad with limited opportunities for direct communication. Currently the principal developers are at Rutgers University, the University of Utah, University of Florida, SUNY-Stony Brook, and UC 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 to be 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.

Changes to the AMBER development process have historically been introduced by a small number of developers who implement a new idea and present it to the others, who then decide whether to incorporate it into the development process. The project is usually swift to adopt new practices– for example, when several developers started using git and demonstrated the utility of version control, the AMBER project as a whole quickly adopted it for the official development tree.

Since 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 rather 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 AMBER developers are focused on keeping AMBER true to its original ideals of efficiency, accuracy, and functionality under a huge variety of architectures and hardware. 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 Systems, Blue Gene, Cray, and ARM processors. It additionally exploits GPUs to accelerate computation when possible [4] and uses MPI for parallel processing. In its 32 years of existence the AMBER software has undoubtedly been run on every single major supercomputer and every version of micro or vector processor.

The program therefore is compiled from source by the user during the install process. This links with the need to support all major compilers including Intel, GNU, PGI, open64, Cray, IBM, and Solaris, and for support of the OpenMP and MPI toolkits for parallelization. Much of the simulation code now also exploits graphics processing units (GPUs) using CUDA, allowing for extremely cost- and time-efficient simulations but at the expense of even more complex cross compilation.

III. Problems

Since each developer focuses primarily on their own code, the functionality of AMBER as a whole can often be neglected. AMBER developers meet once a year to discuss the future of the code, but these meetings are not sufficient for addressing the often frequent problems that can arise in such a large project. Many developers do not have the time or the opportunity to extensively test their code for every compiler that AMBER can use, and often do not 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 may have side effects that are not caught in an individual's examination of their own work. This is especially true for the parallel and GPU-accelerated builds, as errors such as race conditions are often subtle and difficult to replicate or identify. These errors will not result in a program crash or error message, may occur infrequently, and manifest as extremely subtle differences in results.

Finally, the modular nature of AMBER requires a complicated build process where correct dependency resolution is critical. Developers who alter the build order or add their code are often unfamiliar with GNU Makefiles, especially at the level of complexity as AMBER's. Dependency problems often are silent unless a full recompilation is made, and errors with "make clean" can result in a build that appears to work on a developer's machine but will fail on a user's first install.

Overall these project characteristics can result in errors that are reported by end users using hardware or compilers that have not 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 had been written a long time ago.

IV. Solution: Continuous Integration

The goal of this project was to solve the problem of maintaining quality and accuracy over a variety of build combinations by developing a comprehensive build and test suite to verify all supported compiler and parallel combinations, build dependencies, and correctness of test cases. The system is available to developers along with a commit history to the central repository so that developers 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 machine). The system allows for easy identification of tests that are failing due to floating point rounding errors versus ones that are legitimately failing so that test tolerances can be adjusted to avoid false positives.

The software engineering practice of continuous integration was adapted to create a common build and test environment that unifies all the developers' code into one test environment so that errors can be seen on a commit by commit basis. Martin Fowler identifies several elements of a proper continuous integration environment [5] that either already exist or have

been adapted into the AMBER development environment as part of this project. This section will identify each of these principles, describe how it has been incorporated into AMBER development, and discuss its applicability to other CSE projects.

A. Maintain a Single Source Repository

One important aspect of continuous integration is to maintain a common source repository that all developers have access to. This prevents divergence in the code that leads to difficult to resolve conflicts close to release, and allows all developers to monitor what each are working on.

The current AMBER development process maintains a common Git repository that all developers add their code to once they are comfortable sharing it with others. The two main branches are of the current release with patches, and a development branch containing code that is robust enough to share but may not be fully ready to release. Each developer commits code incrementally to their local branch, which can be hosted on their local machine or alternatively on the central Git server which conducts comprehensive backups and allows simple sharing of such development branches. When ready they push to the master trunk. This repository typically receives upwards of ten commits per day.

The repository contains everything necessary to build and test a fully-functional version of AMBER as a development or release version. When it comes time for public release of a new version, a build script packages up the contents of the repository trunk, tagged for the release into a tar file that is downloadable by users.

B. Automate the Build

As projects get larger, some sort of automated build process is necessary so that developers do not have to spend time typing in commands to compile each source file on its own. As dependencies grow, the use of automated build software such as Ant or Make is necessary in order to resolve them and maintain efficiency. An automated build process also allows for easier support for many build options, such as conditional compilation of CUDA code on systems with GPUs, etc.

AMBER uses GNU Make to automate its build process. Users run a configure script that generates variables appropriate to their installation, compiler, and desired options, and then make does the rest. There is a root-level Makefile that calls on sub-Makefiles in each folder containing a program. In order to add a new program to AMBER, a developer need only write a sub-Makefile for their program and call it in the right place in the build hierarchy.

The history of AMBER development, complex nature of the code and the variety of systems on which it must be run prevent the project from using automated build systems like CMake, which may not be available on uncommon architectures and can fail at integrating AMBER code written in many languages into a unified set of executables under a variety of build conditions.

C. Make the Build Self-testing

All software needs testing to find and eliminate all types of bug. This can include algorithmic and logical bugs as well as more complex race conditions in parallel execution and also bugs in the machine code originating from flaws in the compiler's logic. Testing is so critical to the success of a project that validation needs to be automated and integral to the build process, so that developers do not have to remember to manually test certain aspects of the code. It is much more efficient for developers to write tests before or immediately after writing code than it is to add tests to an already released project. In addition, as the complexity of a project grows, the possibility of an error-producing interaction between two program components grows exponentially.

Testing philosophies popular in software engineering, such as Test Driven Development or Extreme Programming, are often difficult to implement in CSE applications, as the desired behavior of code is often unknown. This arises because the code is often written to solve research problems for which the ultimate answer expected from the code is unknown and the results are highly sensitive to statistical convergence and sampling issues. Nonetheless, rigorous validation is extremely important for CSE codes, as the validity of all research conducted with the code depends on it.

AMBER adopts the philosophy that results should be both valid and reproducible, and developers construct their tests accordingly. Once code is complete and has been verified for correctness by the developer, the correct output (with a set random seed, if the program uses the random number generator) will be saved. Tests will run the program (with the same random seed) and compare the output to the saved one. A tolerance value is set to account for floating point rounding differences that arise from different architectures and environments, and if the difference between the two outputs is within the tolerance, the test passes. Otherwise, it fails.

AMBER's hierarchical build system can be used to run all of the tests that each developer has provided. A "test" target exists in the main AMBER Makefile, and when invoked runs a series of sub-test targets which ultimately run "make test" in each sub-directory of the test tree. When creating the Makefile for a subset of code, the developer will also incorporate the test target to run all of the tests and check the output using a variant of diff maintained by the AMBER developers.

D. Everyone Commits as Often as Possible

Continuous integration practices encourage each developer to add their code to the central repository at least once per day. This approach is beneficial to avoid merge conflicts when two developers have unknowingly been modifying the same part of the code for a long period of time, and allows better communication as to what each person is working on.

AMBER's development process does not mandate commit frequencies, but due to the nature of the development process commits typically occur very frequently and without conflict. Merge conflicts are rare, as each developer typically stays focused on a subset of the code, and will only edit other

subsections in a minor way to fix small errors, or with permission and collaboration. Programming is also sporadic on the part of many researchers, since they typically use AMBER in a research capacity the majority 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.

E. Every Commit Should Result in a Build on an Integration Machine

The most important component of continuous integration involves building and testing the code on a dedicated machine or group of machines that frequently reports its status 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 individual commit are evident nearly immediately, rather than having to unravel which commit is responsible for a particular error.

There are many available implementations of continuous integration servers, including Buildbot [6], Jenkins [7], Cruise Control [8], Automated Build Studio [9], Hudson [10], and many more.

The AMBER project now uses Thoughtworks' Cruise Control continuous integration server to conduct automated building and testing in an easily extensible manner. Although difficulties were encountered in adapting the software to fit the broad needs of the AMBER project (see section V), the introduction of the server has helped simplify debugging and has resulted in numerous improvements to the build and tests.

F. Keep the Build Fast

Continuous integration is designed to allow developers to quickly see the results of their commits, and without near-immediate results the process can quickly fall apart. If the overall compilation and test process takes a long time (Fowler suggests 10 minutes as a cutoff), targets that conduct a quick subset of the tests should be added to give developers immediate feedback on their code.

In order to keep build and test times to a minimum, a dedicated machine was built to serve as a continuous integration server with hardware selected specifically for efficiency. Solid state drives were RAIDed to provide greater than 800 MB/sec 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. A high-end M2090 GPU was selected, and an 8-core processor at a high clock rate of 3.8 GHz was used to minimize the time for executing serial aspects of the build process. This machine cost approximately $3200, with the most expensive component being the $2000 GPU.

The compilation time for one of the AMBER build targets is approximately eight to ten minutes on this machine, but the timings for the tests vary from less than ten minutes to over half an hour, depending on whether the tests are for serial, parallel, or GPU code. Since the purpose of our continuous integration server is to test as many combinations of serial, parallel, and GPU code with various compilers, the server is set up to allow multiple targets to be run at once to hasten the completion of all of them. This is only possible due to the use of solid-state disks.

CSE codes with long build times can create multiple continuous integration targets with varying build times, so that developers can get immediate feedback following a commit at a basic level, followed by more detailed information later. More in-depth tests can then be run pending the success of the basic target to create a tiered system of targets that take increasing time to run but provide greater coverage of the application's correctness. This is often referred to as a staged build, build pipeline, or deployment pipeline in the software engineering community. AMBER incorporates this idea by building and conducting tests separately for each combination of options– one target builds the parallel CUDA executables, for example, instead of trying to build every combination of options at once.

G. Test in a Clone of the Production Environment

Software engineering projects often feature a production environment where software is run by the end user that can be very different from the development or test environment. The continuous integration principle of testing in a clone of this production environment aids developers in finding errors that may not be present on their machines.

This is especially true with applications that will run in conjunction with a large database. Testing with smaller data sets may not produce any errors, but in the final environment the program may suffer catastrophic crashes. However, whether or not this constraint is present for CSE codes is completely dependent on the application. For AMBER and many other simulation codes, the data is generated by the application and then passed to others for analysis, rendering this principle inapplicable. However, for other projects, this may be a crucial part of the testing process.

The AMBER tests try to reflect real-life usage as much as possible, using real input data rather than contrived examples. As a result, the project contains over 1 GB of data in the test cases alone. Developers also test different compilers, compiler versions, vector math and system libraries to mimic the different Linux environments on user machines, and our ultimate goal is to use virtualization to test all major operating systems.

H. Simplify Executable Distribution

Often during the development process there are a variety of builds of the current executable– development, debug, stable release, etc. Finding the most recent product for a demonstration or release should be simple to both users and developers. It should also be simple for users to identify if their software is out of date and, if desired, easily update it to the latest version.

AMBER's releases are often patched several times, and many of these patches correct problems that may lead to

erroneous simulation or offer significant speed improvements over the unpatched version. It is 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.

Distribution of updates is of critical importance 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 latest version of the program.

The patch process for AMBER 12 and AmberTools12 has been greatly simplified since previous years with the addition of an automatic patching script. When users prepare for compilation by running the configure script, the patcher is automatically invoked and will check the AMBER website for project updates then retrieve and apply the latest patches if necessary.

I. 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 the speed of the program.

Good communication is very important to CSE projects, especially so for those that are distributed over a variety of 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 Cruise Control server has been extremely beneficial to AMBER development in this aspect as previously developers were unable to see the immediate effects of their commits on every compiler and option. This would often lead to confusion over what broke the build, and overall the development process was much more opaque.

J. Automate Deployment

Deployment refers to the processes by which software becomes usable on a new machine, be it compilation, installation, or updating. Efficient, automated deployment is a focus of continuous integration as without an efficient installation process, automated building and testing is impossible.

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 may need a more involved process to resolve dependencies and make executables accessible to the user.

Rolling deployments offer the opportunity for complicated installations to be completed one step at a time while checking for errors. The concept is used frequently for websites with a large user base, where a feature is rolled out to a subset of users at a time, or on supercomputers where a sysadmin will deploy software to one node at a time.

In general, the installation process should require as little input from the user as possible– shell variables should export themselves, dependencies should be automatically checked for, etc. Often users of a CSE code may be unfamiliar with Linux and are unprepared for an involved series of deployment steps, and may be confounded by cryptic errors. Complicated installation processes are also difficult to automate on a continuous integration server, and complicate testing by developers themselves.

V. Implementation Details

Continuous integration is a principle that 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.

Out of the 28 most common available continuous integration servers, 60% were under a proprietary license, making the server difficult to modify and 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 e-mail. AMBER uses the Cruise Control 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 Cruise Control was made with the AMBER build and test environment in mind although it is not the only logical choice but one of several possibilities. For example, the molecular dynamics project Gromacs [11] has had considerable success with the Jenkins [7] continuous integration environment.

While being a good choice, considerable customization and modification of Cruise Control was required to match the needs of the AMBER developers. Targets for the AMBER builds and tests were defined using Apache Ant scripts, which are an XML format that is primarily aimed at compilation of Java bytecode but also provides the ability to execute arbitrary commands. Each build script runs configure with the appropriate flags for the target, then executes "make install". Any errors in this process cause the build to be reported as failing.

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. In addition, targets were created for various builds to test building in parallel with different job counts, executing "make -jX" where X is typically 1 to 16. This checks for correct dependencies in the Makefile and provides a measure of the performance improvements that building in parallel can bring.

Separate test targets were created for each of the Amber builds, since the Amber tests include those for AmberTools. The target is triggered on successful completion of its parent

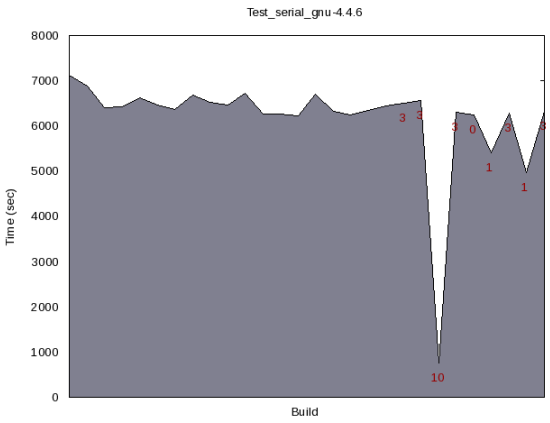

Fig. 1. Graphical display of time taken to complete the test targets aided developers in finding a segmentation fault error in one of the main simulation programs. The sharp drop in time taken in near the right of the graph corresponds to a build where many of the tests suddenly crashed. The red numbers indicate the number of failed tests during each attempt to run the target.

build target, and runs make test in the source directory. A script is then run that collects all of the diff files from failed tests and lists tests that crashed to provide an easy review of results in the web interface.

The build and test targets check the latest AMBER code for compilation and correctness, but efficiency is also of paramount importance to the developers. Although collection and display of timing information is not supported by Cruise Control, it was extended to provide this functionality.

Time stamps are created 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 graphing program gnuplot [12] is invoked to create a graph of these values, and this is made available on the web interface so that developers can have a visual representation of how the latest commits have affected program speed.

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. For example, the sharp decrease in time shown in the middle of the graph in Fig. 1 corresponds to a commit that caused sander (one of the main AMBER simulation programs) to crash, and therefore many of the tests instantly failed as well.

Finally, "artifacts," or files generated by each build, are saved and made available via the dashboard so developers can access logs and test failure information without having to replicate the work done by the testing machine.

The Cruise Control web interface required considerable modifications to meet the needs of this CSE project. The interface by default depicts small colored squares to indicate whether or not a target is succeeding or failing and it is necessary to click on each square to identify which target it is. To simplify easy identification of problems, the squares were turned into rectangles with the project name written inside

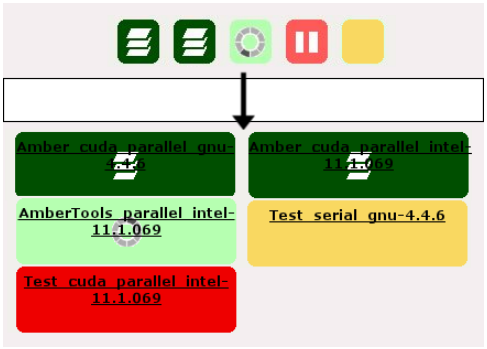

Fig. 2. Many improvements were made to the Cruise Control web interface, including making project names easily viewable in the main build window. The top shows how targets are represented with colored squares by default, and the bottom shows the modified interface used in the AMBER Cruise Control dashboard.

as shown in Fig. 2. Showing targets as boxes is useful for software engineering applications of continuous integration, where all build targets need to be quickly identified as passing or failing and the specifics of the targets are less important. However for CSE applications the specific context in which failure occurs is of critical importance, as the difference between a serial and a parallel target can be a considerable amount of code.

VI. Conclusion

Implementation of continuous integration principles can be extremely beneficial to CSE projects. They offer considerable opportunities to foster unity and collaboration among diverse groups of developers, and the automated build and testing environment greatly simplifies the debugging process.

The cost of implementing these techniques is fairly low, with the most expensive component being the dedicated build machine at approximately $3000. Training costs may be kept to a minimum by presenting the system setup as a project for a student or intern.

The majority of continuous integration software is designed for more traditional software engineering applications, and are difficult to adapt to the significantly different needs of CSE projects. Extensive modifications need to be made to these systems to provide necessary information to developers, and the amount of effort needed to implement a continuous integration server may deter CSE projects from adopting the techniques.

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 benchmarks, and the implementation of virtual machines to more thoroughly test different build environments.

Overall, using continuous integration software in the AMBER project has proved to be worth the initial efforts, and while 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.

ACKNOWLEDGMENTS

This work was funded in part by the National Science Foundation through the Scientific Software Innovations Institutes Program - NSF SI2-SSE (NSF1047875 and NSF1148276) grants to R.C.W and also by the University of California (UC Lab 09-LR-06-117792) grant to R.C.W. The work was also supported by a CUDA fellowship to R.C.W from NVIDIA Inc.

REFERENCES

[1] R. Salomon-Ferrer, D. Case, and R. Walker, "An overview of the Amber biomolecular simulation package," *WIREs Comput. Mol. Sci.*, 2012. [Online]. Available: http://dx.doi.org/10.1002/wcms.1121

[2] D. Case, T. Darden, T. Cheatham, C. Simmerling, J. Wang, R. Duke, R. Luo, R. Walker, W. Zhang, K. Merz, B. Roberts, S. Hayik, A. Roitberg, G. Seabra, J. Swails, A. Goetz, I. Kolossváry, K. Wong, F. Paesani, J. Vanicek, R. Wolf, J. Liu, X. Wu, S. Brozell, T. Steinbrecher, H. Gohlke, Q. Cai, X. Ye, J. Wang, M.-J. Hsieh, G. Cui, D. Roe, D. Mathews, M. Seetin, R. Salomon-Ferrer, C. Sagui, V. Babin, T. Luchko, S. Gusarov, A. Kovalenko, and P. Kollman, "AMBER 12," 2012.

[3] P. K. Weiner and P. A. Kollman, "Amber: Assisted model building with energy refinement. a general program for modeling molecules and their interactions," *Journal of Computational Chemistry*, vol. 2, no. 3, pp. 287–303, 1981.

[4] A. W. Götz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, and R. C. Walker, "Routine microsecond molecular dynamics simulations with AMBER on GPUs," *Journal of Chemical Theory and Computation*, vol. 8, no. 5, pp. 1542–1555, 2012. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ct200909j

[5] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works*, 2006. [Online]. Available: http://www.thoughtworks.com/Continuous Integration.pdf

[6] Buildbot. [Online]. Available: trac.buildbot.net

[7] Jenkins continuous integration. [Online]. Available: http://jenkins-ci.org

[8] Cruise Control. [Online]. Available: http://cruisecontrol.sourceforge.net

[9] Automated Build Studio. [Online]. Available: http://smartbear.com/products/development-tools/build-management

[10] Hudson. [Online]. Available: http://hudson-ci.org

[11] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "GROMACS 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," *Journal of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, 2008.

[12] J. Racine, "Gnuplot 4.0: a portable interactive plotting utility," *Journal of Applied Econometrics*, vol. 21, no. 1, pp. 133–141, 2006.