Run-time extensibility and librarization of simulation software

Jed Brown
Argonne National Laboratory
University of Colorado Boulder
jedbrown@mcs.anl.gov

Matthew G. Knepley
University of Chicago
knepley@ci.uchicago.edu

Barry F. Smith
Argonne National Laboratory
bsmith@mcs.anl.gov

Abstract—**Build-time configuration and environment assumptions are hampering progress and usability in scientific software. That which would be utterly unacceptable in nonscientific software somehow passes for the norm in scientific packages. The community needs reusable software packages that are easy use and flexible enough to accommodate next-generation simulation and analysis demands.**

Keywords: software library, extensible software, object-oriented, software composability, simulation

I'd like you to use my new web browser, Firetran! It renders HTML 10% faster than Firefox, but only if there is no JavaScript. You can recompile if you want JavaScript, but our performance tests don't show that configuration. The character encoding is compiled in, for efficiency. It has a great plugin community—developers add code directly to the web browser core, guarded by a `#ifdef`. Some developers change things and distribute their own mutually incompatible versions of Firetran. Naturally, users of those packages submit bug reports to me, but I haven't been able to reproduce with my version. Proxy configuration is compiled in so you don't have to worry about run-time configuration dialogs, just edit a makefile and recompile. To keep you secure, the https version of Firetran cannot use http, and vice versa. Although Firetran is open source, our development is done in private; if you submit a bug report or a patch, you'll likely hear "We fixed that in the private repository last year; we'll release when the paper comes out. If you have to view that website, fill out the attached form and fax us a signed copy." Firetran has a parental filter feature: you can list a maximum of 16 websites in a source file, in which case Firetran will refuse to visit any site not on the list. Firetran can be compiled only with last year's version of the ACME Fortran77 compiler. The build system consists of csh, perl, m4, and BSD make. There is no URL entry box in Firetran; to visit a page, you edit a configuration file and run the program. A grad student wrote a Tcl script with a text entry box to automate editing the configuration file and rerunning the Firetran executable. The script is hard to understand, but many in the community believe the way forward is to enhance the script to detect whether the website needs https or http, JavaScript or not, etc., and recompile Firetran on the spot.

Needless to say, Firetran struggles with market share. Yet choices in Firetran represent the status quo in many scientific software packages—often vehemently defended. If it is laughably unacceptable in nonscientific software, why is it tolerated in scientific software? Are scientists suffering from Stockholm Syndrome? Is scientific software so fundamentally different? How could scientific software benefit from adopting the techniques we take for granted in nonscientific software? Let's look at some of the directions that scientific simulation software is heading toward.

I. TRENDS IN SIMULATION-BASED SCIENCE AND ENGINEERING

Modern computational science and engineering is increasingly defined by multiphysics, multiscale simulation [1] while raising the level of abstraction to risk-aware design and decision problems. This evolution unavoidably involves deeper software stacks and the cooperation of distributed teams from multiple disciplines. Meanwhile, each application area continues to innovate and can be characterized as much by the forms of extensibility (e.g., boundary conditions, geometry, subgrid closures, analysis techniques, data sources, and inherent uncertainty/bias) as by the underlying equations. It is no longer the case that the original author can foresee all use cases for their software. We argue that many common approaches to configuration and extensibility create *artificial bottlenecks* that impede science goals, and that the only sustainable approach is to defer these to run-time. Doing this effectively will push applications to minimize the assumptions made about their environment, resulting in more library-like applications better suited to coupling with other models and to performing advanced analysis.

A. Compile-Time Configuration

Many applications, especially those written in Fortran [1], perform configuration in the build system. This approach is motivated by a variety of concerns about efficiency (often ill-founded or fixable by adjusting interface granularity), limitations of software tools (e.g., algorithmic differentiation), poor language support, perceived implementation complexity, and short-term value assessment. Once a package chooses compile-time configuration, the build system becomes a public application programming interface (API), used by scripts that perform higher-level analysis. Ad hoc public APIs inhibit

---

[1]A language that after more than 50 years, has finally started to provide mechanisms for encapsulation in its latest standards (using ISO C bindings) and natively in TS 29113 (scheduled for inclusion in Fortran2015).

software evolution by imposing an unintentionally high cost on change as well as dilution of effort to meet short-term deliverables.

In applications relying on build-time code generation, pragma-based specialization/optimization, or those written in C++ with heavy use of templates, the possible combinations must be enumerated at compile-time. Although templates are not exclusive (you can compile several variants in the same application), it is common to see a combinatorial explosion of variants as well as directly exposing the templates in public interfaces. Since all combinations cannot be compiled into one application, the effect is that any analysis or testing that explores a large or unpredictable part of the space of combinations must include recompilation. Attempting to push the size limits leads to error-prone workarounds like `-mcmodel=large` (a compiler option that affects linking/compatibility), using processes spanning more than one NUMA node (degrading memory locality), and inability to run on low-memory architectures that might otherwise be well-suited to the application.

Compute nodes often do not have access to compilers, making all build-system and compile-time decisions inaccessible to online analysis. A given application may be unable to run in both configurations on different nodes or on different MPI communicators. This limits analysis capability, requires frequent recompilation, and increases the likelihood of user error resulting from accidentally using the wrong compiled version. The length of batch queues exacerbates the issue, sometimes requiring days between compiling an application and actually running it. Every compatibility that must be maintained by hand is another opportunity for mistakes, some of which the user may not realize prior to publication.

Some applications create sophisticated scripts for maintaining consistency through the compilation and batch submission process. These scripts must be ported to each architecture, and they increase the complexity of debugging the application and of reproducing problems encountered on certain architectures.

Integration tests often need to be submitted to batch systems. If different integration tests need dependencies to be compiled differently, those different versions need to be built in advance and kept straight through the test submission and run. When many configurations are needed, the multiple required compilations have a tendency to take a long time and/or burn through disk quota.

B. Advanced Analysis

As models mature in each application area, emphasis shifts from qualitative and subjective interpretation of model output to quantitative analysis of accuracy, reliability, and the influence of parameters on quantities of interest. Correspondingly, today's models are increasingly used not just as forward models but as the target of advanced analysis techniques such as stochastic optimization, risk-aware decisions, and stability analysis. The forward model must then expose an interface for each form of modification that the analysis levels can explore. An interface requiring build-time modification shifts an unacceptable level of complexity to the analysis software and is algorithmically constraining—limiting parallelism, introducing artificial bottlenecks, and preventing some algorithms.

In lieu of tractable deterministic techniques for calibration of empirical phenomenological models, an enormous amount of expert time must be spent tuning parameters. In fields such as climate, earthquakes, and molecular dynamics, this calibration is notoriously sensitive to numerical methods, temporal and/or spatial resolution, and other models used in simulation. And yet when faced with this extreme uncertainty and volatility, these parameters are often hard-coded in the source, thwarting reasonable attempts to automate the calibration or comparison of models.

C. Model Coupling

A large fraction of successful scientific software has been the result of a visionary scientist operating in a single domain. Many important model configurations and analysis types were predicted by that visionary, and the community has been largely content to explore within that fuzzy scope. Each package has been king of its own environment and thus has often made choices without concern for interoperability or impact on other packages. But the gaping holes in our scientific understanding and engineering capability lie increasingly in the gaps not covered by any one of these mature packages. Rarely do multiple models operate on identical spatial and temporal scales with similar model and parameter uncertainties. Thus, coupling often requires grappling with multiscale phenomena and high-variance statistics, each an algorithmic challenge in its own right. When components make excessive assumptions about their environment, attempts to couple are either written off or algorithmic quality falls by the wayside, leading to nominally coupled simulations that are unreliable at best and effectively nonconvergent in most cases. The most powerful and pragmatic software approach we know of is to formulate models as libraries with a clean hierarchy of interfaces, allowing an external client to compose the key capabilities into a coupled model without the higher-level parts that would algorithmically constrain a coupled model. This approach has repeatedly demonstrated its effectiveness outside of scientific computing in areas traditionally dominated by stand-alone applications, such as compilers (LLVM), web browsers (KHTML/WebKit), and SQL databases (SQLite). Although process isolation can be useful for reasons of security (e.g., qmail, postfix), reliability (tabs in a web browser), and distribution (e.g., remote databases), it is easier to add isolation upon library interfaces than to add composition/embedding atop process separation, especially in HPC environments for which oversubscription is usually catastrophic.

D. Provenance and Usability

Reproducibility and provenance are perpetual challenges of computational science that become more acute as the software stack becomes deeper and more models of greater complexity are coupled. How can we capture the state of all configuration

knobs so that a computational experiment can be reproduced? Compare the complexity of a single configuration file to be read at run-time with that of a heterogeneous configuration consisting of multiple build systems, files passed from earlier stages of computation, and run-time configuration. Provenance is simplified by using each package without modification, compiled in a standard way, and controlled entirely via run-time options. This implies that any libraries used (transitively) by the application must be responsible libraries that adhere to the principles discussed here and in [2]. For both maintenance and provenance reasons, custom components needed for a given computational experiment are better placed in version-controlled plugins rather than by modifying upstream sources. If a coherent top-level specification is to be supported in a system with build-time or source-level choices, those configuration options must be plumbed through all the intermediate levels, often resulting in another layer of "workflow" scripts and bloated, brittle high-level interfaces.

E. "Big" Data

Workflows that involve multiple executables usually pass information through the file system. It takes about one hour to read or write the contents of volatile memory to global storage on today's top machines, assuming peak I/O bandwidth is reached. The largest allocations are on the order of tens of millions of core hours (e.g., INCITE), meaning that the entire annual compute budget can be burned in a few reads and writes. Global storage as an *algorithmic* mechanism is dead: where out-of-core algorithms have been used in the past, today's scientists can simply run on more cores, up to the entire machine; but if the entire machine does not have enough storage, the allocation simply does not have the budget to run an out-of-core algorithm.

If a different application or different version of an application must be used for the next stage in the simulation/analysis pipeline, data must be dumped to the file system. In situ analysis provides an excellent opportunity to increase efficiency by reducing dependence on the file system, but is viable only if the more varied analysis workflow can be performed in the same application. Interfaces for exchanging data in-memory between different software components could be the same as those used to describe data sets for parallel IO.

Some of today's simulations support a large and diverse community that analyzes the output. Transitioning to in situ analysis will require dynamic and extensive analysis interfaces to support varied analysis demands. Unlike most parts of mature simulation software, the analysis code often changes with each question a scientist asks and thus is highly volatile and does not benefit from the same amount of testing.

F. Nested Dependencies

Some library dependencies are indirect (transitive), via some intermediate interface that the application actually intends to depend on. One of the most important software engineering principles is that of encapsulation, allowing clients to depend only on interfaces that it uses directly, rather than implementation concerns. There is no encapsulation if a transitive dependency must be reconfigured for each use case, and combining uses into one application may cause conflicts. The build system for any "library" that requires use-specific configuration effectively becomes a public API that top-level components must interact with even when the library is only used indirectly.

A single library can be used by multiple components in the same executable. This may be rare when a library is first being developed, but it is common among popular and versatile libraries. If a library has mutually incompatible configurations, the entire executable can use only one version unless the library developer has taken great care (often impractical, especially when linking statically—an unfortunate necessity on many HPC architectures). Even in the best case, needing to use multiple versions complicates the installation and debugging process, invariably leading to degraded user experience and increased support workload for library maintainers.

G. User Modifications

Fragmentation of software projects is notoriously expensive and should be avoided when possible. Maintaining local modifications with no plan for upstreaming is a recipe for divergent design—technical debt that must be paid off in order to combine the features developed in each fork. Fragmentation is especially toxic for libraries that may be used by multiple higher-level packages that are combined in the overall experiment (see subsection I-F)

H. Packaging and Distribution

Software developers often underestimate the challenge of installing their own packages. From the perspective of user experience, it hardly matters if an installation failure was caused by a user's broken environment (a circumstance all-too familiar to maintainers of popular packages). Upgrading an operating system can break existing installs of a package if the underlying system libraries change. The most reliable way to distribute packages that will always be in sync with the operating system is to have them be packaged by many common operating systems (Debian APT, RedHat RPM, MacPorts, etc.). Configure-time options are the bane of package distribution due to the need to name each variant and to resolve conflicts between the variants. Packagers for binary distributions (most convenient for users) are justifiably paranoid about the binary interface and hence will be reluctant to package software with fragmented configuration options.

II. IMPLEMENTATION AND RECOMMENDATIONS

To manage these workflow challenges, application developers will need to think more like library developers [2]: controlling namespaces; avoiding global state; relinquishing top-level control; controlling the scope of parallelism; localizing memory allocation; localizing complexity so that it does not "bubble up" to the top level; and paying attention to the completeness, generality, stability, and extensibility of all

public interfaces. Our suggestions are shaped by experience developing and supporting (PETSc) [3], [4], as well as other packages from low-level libraries to end-user applications. Similar ideas for extensibility and run-time configuration have been implemented in applications such as MOOSE [5] and PyLith [6].

A. Resource Allocation

To localize configuration, allocation of resources such as memory should be done locally, with reference counting when appropriate. Contrary to urban legend, static memory allocation offers no tangible performance advantage (so long as dynamic allocations are amortized) and unavoidably ties the workflow into the build system, while committing the sin of needless global variables. Different `malloc` implementations have varying performance, especially in multi-threaded scenarios. If necessary, fast implementations like TCMalloc [7] can be recommended, but it is better to contain this complexity in order to perform well with any `malloc`. This can involve having memory pools or work arrays associated with algorithm objects, so that `malloc` is not called in inner loops.

B. Plugins

Source-level dependencies on an implementation (e.g., direct instantiation of a derived class or a template parameter) rather than a generic interface cause choices from deep in the stack to "bubble up" via brittle interfaces that plumb the user's configuration to the appropriate component. Plugins provide a strong way to identify interfaces that can be extended by users and distributed separately from the core package. For example, every class in PETSc has a plugin architecture, from base linear algebra components to preconditioners, nonlinear solvers, and adaptive controllers for time integration. Any of these components can be provided by a plugin and will be indistinguishable from a native component of PETSc. Plugins consist of a registration function that is called via `dlopen()`, a creation function that is called when the plugin is activated (e.g., instantiation of an object implemented in the plugin), and any supporting functions that will be exposed via methods of the object. Historically, Fortran's type system and inability to store function pointers have conspired against plugin implementations, but the new standard provides the necessary tools.

Plugins also provide a mechanism to invert dependencies without creating dependency loops. For example, suppose `libB` depends on `libA`, but we would like to provide an optional implementation of an interface in `libA` that depends on `libB`. We can't put it in `libA` because this would make a cyclic dependency, but it is unrelated to `libB`'s public interface so doesn't belong there either. We *can* create `libA-plugin` that depends on both libA and libB, registering itself as a plugin of `libA` and calling into `libB` in its implementation. Note that plugins can also be used for optional interfaces to third-party libraries. It is best to have plugin search paths from which plugins are loaded by `dlopen`, so that they can be distributed independently from

the base system and no relinking is required. Shared libraries should be versioned (e.g., `-soname` on most POSIX systems, `-current_version` and `-compatibility_version` on OSX) to make this distribution more reliable and to assist the layers built on top. See [8] for more on shared library versioning and controlling symbol visibility.

While distribution via shared libraries is convenient for users and packagers, some important HPC execution environments do not support shared libraries. If such anti-productive environments must be used, the plugin structure can be preserved, but the build system ultimately needs to be able to link everything statically. For an application, this typically means that plugin source trees are placed in a location that the build system picks up, then code to call the registration function is generated, and everything is linked together. For a library, plugins either need to be compiled into a single static archive or the user needs to explicitly link the plugins (in the correct order). The linking interface is a public interface, so changing it should not be taken lightly. The library can either distribute a tool that determines which plugins are available and generates a suitable link line or it can create a static archive containing all plugins. Unfortunately, the `pkg-config` tool is lacking in management of multiple configurations and optional dependencies, so many libraries will need their own executable. Wrapper compilers are exclusive (only one library can use a wrapper compiler) and thus should be avoided.

C. Inversion of Control, Recursive Configuration, and the Options Database

The primary purpose of software libraries is to contain complexity. Public interfaces should be as simple as possible (but no simpler), meaning that transitive complexity *must not* be a mandatory part of the public interface. Furthermore, extensible components are not known at compile-time (indeed, they may not have been written yet) and thus would be rendered useless if implementation complexity leaked into the public interface. It should be possible to instantiate the same plugin (unknown to client code) with different configurations at different locations in the object graph, each with its own configuration. Since the client does not know how to configure the object, some inversion of control [9] is necessary. PETSc's approach is similar to "service locator" in [9], but several variations should be considered by new projects. In PETSc, multiple instances of objects are distinguished by a *prefix* in the options database, allowing conflict-free run-time configuration. For example, a multiphysics solver may use a block decomposition and geometric-algebraic multigrid with choices and diagnostics for each block and at each level of one or more multigrid solves, each instance of which we distinguish by prefix. The basic principle is to choose good defaults and defer precise configuration to the run-time interface. Some packages take dynamic extensibility further by embedding a Turing-complete programming language such as Lua, JavaScript, or Scheme.

PETSc also acknowledges that some users take *active* control over method configuration, adapting it in response to

physical regime or other factors. This active control is more naturally implemented and debugged with an object-based run-time interface; thus any run-time configuration exposed via the options database is also exposed via the object-oriented interface. The most challenging compromise in this scenario occurs when an algorithm adaptively configures recursive levels, but the client wants to actively configure portions. Solutions include fine-grained interfaces for "forcing" (in the lazy functional programming sense) certain parts of the setup and callbacks to configure portions when reached. Neither is completely satisfactory.

D. Object-Oriented Design

We turn now to some contentious issues in object-oriented design, for which we are less than enamored with oft-repeated recommendations.

1) Partial implementation: Some people believe that all errors should be compile-time errors; thus, any incompatibility must be visible to the compiler. Unfortunately, this approach leads to extremely complicated and fragile type hierarchies. For example, a `Matrix` is a linear transformation on finite-dimensional vector spaces. Should a `Matrix` have computable entries? Should the diagonal be extractable? Can the transpose be applied? Are "Neumann" subproblems available (matrices with certain properties whose sum equals the original matrix)? While the entries of matrices can be computed in principle, the space and time complexity may be so unaffordable as to render that representation useless. Meanwhile, other operations that are unaffordable for explicitly stored matrices may be fast for matrices with special structure. Different preconditioners (which may reside in plugins) may require different functionality from the `Matrix`. Any type system that can guarantee full implementation of a given `Matrix` interface will end up conflating the desired generic interface with implementation-specific semantics, especially when the *Matrix* type is also extensible, leading to undesirable dependencies and leakage of transitive complexity. Moreover, the "not implemented" run-time error is likely to be more understandable than a type mismatch error.

2) Run-Time Implementation Changing: PETSc has found it useful for major objects to be able to change implementations at run-time (e.g., from multigrid to a direct solve). One object can have many dependencies/references and be referenced by many other objects. If the implementation can only be changed at object creation, the user ends up holding factory objects (or the equivalent) for the sole purpose of recreating "similar" objects. Someone has to be responsible for keeping track of these factory objects and for rewiring the dependencies when replacing an existing object. This turns out to be messy and error-prone, so PETSc has chosen to absorb the "factory" functionality into the object itself, allowing reconfiguration of any sort at any time. This also removes the need for special interfaces to pass a factory object around to all components that should have a say in how the new object will be configured.

3) Controlling the Binary Interface: Time spent recompiling code is nothing but wasted productivity. Implementation concerns such as private variables and new (virtual) methods should never require recompilation of client code. PETSc uses a delegator (aka. "pointer to implementation" [10] or "bridge" [11]) pattern to keep such implementation concerns out of the binary interface, thus minimizing recompilation and enabling binary distribution of shared library [8] upgrades. This is idiomatic in C where "objects" are usually implemented via opaque pointers, but often under-utilized in C++ because it entails a bit more boilerplate than the native object model that reveals the private contents of classes. Delegator incurs an additional static function call, but tests with classic virtual methods and delegator indicate that the main function call overhead (several cycles) comes from the indirect call (virtual function) rather than the static call to the delegator, thus the incremental cost of using the delegator pattern is usually less than 2 cycles. An ancillary benefit of the delegator pattern is that there is a unique place to set a debugging breakpoint for each function (rather than having to choose the correct virtual function) and a common place for input validation.

It is increasingly popular to expose libraries through more dynamic environments such as Python or Julia. Since different languages have different type systems, it is easier and more reliable to develop language bindings with a simple type system and stable binary interface. Naturally, static methods and opaque pointers are simpler than struct definitions and template-based systems.

E. Just-in-Time Compilation

With fine-grained composition such as occurs in material models and Riemann solvers, as well as fusion of memory-intensive operations, the number of possible compositions grows combinatorially, but in any specific run, only a small number is important. Precompiling and dispatching (via C++ templates or other inlining techniques) every combination leads to large compile times, bloated executables, confusing debugging, and compromises about which combinations will be made available. While a dynamic interface is far more maintainable, the performance overhead is unacceptable for certain applications. When the interface granularity cannot be increased to amortize the overhead of dynamicism, just-in-time (JIT) compilation is an attractive approach to preserve strong encapsulation and debuggability. We anticipate technologies such as LLVM and OpenCL becoming ubiquitous, allowing judicious use of JIT for dynamic kernel fusion and plugin-style packaging of fine-grained components without sacrificing performance. This may involve tighter integration with languages like Julia and the Numba package for Python, or language extensions to support JIT within traditionally compiled languages.

F. Upstreaming, Distribution, and Community Building

To provide attractive alternatives to forking, maintainers must be diligent in creating a welcoming environment for

upstream contributions. The maintainers should nurture a community that can review contributions, advise about new development approaches, and test new features, with recognition for all forms of contribution. In a transparent community, paper reviewers can easily determine who did the work to implement a new feature; thus any attempt to "scoop" a result based on new capability is easily spotted. We believe that scooping is a purely social problem and that the secrecy inherent in any technical solution bear a cost so expensive that it can rarely be justified. Several major tech companies have famously underestimated this cost when forking open source packages such as the Linux kernel for internal use, later repaying the technical debt to reintegrate with upstream. In science, it is exceedingly difficult to obtain funding to pay off the technical debt incurred by forking, leading to a wasteland of abandoned forks. This is contrary to the interests of stakeholders ranging from the program managers and taxpayers to other scientists in the field.

In addition to community building [12], developers should provide versatile extension points so that contributions can be made without compromising existing functionality and without degrading package maintainability. This should be thought of as a technical prerequisite for maintainable extension rather than private forking. Such extensions must be accompanied by tests lest they break as interfaces evolve. It is far easier to write tests for dynamic configuration sets than to add new build-time configurations. Additionally, compilers and static analysis tools can check combinations that are not actively used, but conditional compilation is unchecked, invariably leading to more frequent breakage by other developers (in the test suite if covered, otherwise the breakage will be found by users and other developers).

III. THINKING LIKE LIBRARY DEVELOPERS

Configuration and environment design decisions made by today's scientific libraries and applications are often disproportionately harmful to usability, productivity, and capability. In such cases, the most effective way to increase scientific or engineering value is to design and refactor software using best practices for extensible library development.

REFERENCES

[1] David E. Keyes, Lois Curfman McInnes, Carol Woodward, William Gropp, Eric Myra, Michael Pernice, John Bell, Jed Brown, Alain Clo, Jeffrey Connors, Emil Constantinescu, Don Estep, Kate Evans, Charbel Farhat, Ammar Hakim, Glenn Hammond, Glen Hansen, Judith Hill, Tobin Isaac, Xiangmin Jiao, Kirk Jordan, Dinesh Kaushik, Efthimios Kaxiras, Alice Koniges, Kihwan Lee, Aaron Lott, Qiming Lu, John Magerlein, Reed Maxwell, Michael McCourt, Miriam Mehl, Roger Pawlowski, Amanda Peters Randles, Daniel Reynolds, Beatrice Rivière, Ulrich Rüde, Tim Scheibe, John Shadid, Brendan Sheehan, Mark Shephard, Andrew Siegel, Barry Smith, Xianzhu Tang, Cian Wilson, and Barbara Wohlmuth. Multiphysics simulations: Challenges and opportunities. *International Journal of High Performance Computing Applications*, 27(1):4–83, Feb 2013. Special issue.
[2] William D. Gropp. Exploiting existing software in libraries: Successes, failures, and reasons why. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing*, pages 21–29. SIAM, 1999.
[3] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.5, Argonne National Laboratory, 2014.
[4] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, and Hong Zhang. PETSc developers manual. Technical report, Argonne National Laboratory, 2011.
[5] David Andrs, Cody Permann, Andrew Slaughter, Richard Martineau, Derek Gaston, John Peterson, and Jason Miller. Multiphysics Object-Oriented Simulation Environment (MOOSE). http://mooseframework. org.
[6] Brad Aagaard, Matthew G. Knepley, and Charles Williams. PyLith. http://geodynamics.org/cig/software/pylith/, 2012.
[7] Sanjay Ghemawat and Paul Menage. TCMalloc: Thread-caching malloc, 2009. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.
[8] Ulrich Drepper. How to write shared libraries, 2002–2011. http://www. akkadia.org/drepper/dsohowto.pdf.
[9] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, 2004.
[10] Herb Sutter. *Exceptional C++: 47 engineering puzzles, programming problems, and solutions*. Addison-Wesley Professional, 2000.
[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
[12] Matthew J. Turk. How to scale a code in the human dimension. *arXiv preprint arXiv:1301.7064*, 2013.

AUTHOR BIOS

Jed Brown

Jed grew up in Alaska, earning BS degrees in Mathematics and Physics and an MS in Mathematics at the University of Alaska Fairbanks, during which he was the principle author of the Parallel Ice Sheet Model (PISM). In 2011, he completed a Dr.Sc. at ETH Zrich, where he investigated computational methods for ice sheet and glacier dynamics. He became an active PETSc developer during this time, leading naturally to a postdoc at Argonne National Lab. He is now an Assistant Computational Mathematician at Argonne and holds a joint appointment at the University of Colorado Boulder. He received the 2014 SIAG/SC Junior Scientist Prize and 2014 IEEE TCSC Young Achievers Award. Brown is a member of SIAM, AGU, SIGHPC, and CMG++.

Matthew G. Knepley

Matthew G. Knepley received his B.S. in Physics from Case Western Reserve University in 1994, an M.S. in Computer Science from the University of Minnesota in 1996, and a Ph.D. in Computer Science from Purdue University in 2000. He was a Research Scientist at Akamai Technologies in 2000 and 2001. Afterwards, he joined the Mathematics and Computer Science department at Argonne National Laboratory (ANL), where he was an Assistant Computational Mathematician, and a Fellow in the Computation Institute at University of Chicago. In 2009, he joined the Computation Institute as a Senior

Research Associate. His research focuses on scientific computation, including fast methods, parallel computing, software development, numerical analysis, and multicore architectures. He is an author of the widely used PETSc library for scientific computing from ANL, and is a principal designer of the PyLith library for the solution of dynamic and quasi-static tectonic deformation problems. He developed the PETSc scalable unstructured mesh support based upon ideas from combinatorial topology. He was a J. T. Oden Faculty Research Fellow at the Institute for Computation Engineering and Sciences, UT Austin, in 2008, and won the R&D 100 Award in 2009 as part of the PETSc team.

Barry F. Smith

Barry Smith was born in Townsville, Australia, in 1964. He received the B.S. degree in mathematics from Yale University in 1986, and the Ph.D. degree in applied mathematics from the Courant Institute of Mathematical Sciences (NYU) in 1990. In 1990, he joined the Mathematics and Computer Science Division at Argonne National Laboratory as the Wilkinson Post-doctorial Fellow. From 1992 to 1994 he was a Computational and Applied Mathematics Post-doctorial fellow in the Mathematics Department at the University of California, Los Angeles. Since June 1994, he has been with the Mathematics and Computer Science Division at Argonne National Laboratory where he was an Assistant Scientist, became a Computational Mathematician in 1995, and a Senior Computational Mathematician in 2008. His current research interests include numerical algorithms for linear algebra and partial differential equations and software for high performance computing. He is one of the developers of the Portable Extensible Toolkit for Scientific computation (PETSc). Dr. Smith is a Fellow of the Society of Industrial and Applied Mathematics (SIAM) and a member of the Association for Computing Machinery (ACM). He, along with his colleague Lois Curfman McInnes, was the recipient of the 2011 Ernest Lawrence award for outstanding contributions in research and development supporting the Department of Energy and its missions.