Run-Time Extensibility and Librarization of Simulation Software

Jed Brown | Argonne National Laboratory and University of Colorado Boulder
Matthew G. Knepley | University of Chicago
Barry F. Smith | Argonne National Laboratory

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

I'd like you to use our new Web browser, Firetran! It renders HTML 10 percent faster than Firefox—but only if there's no JavaScript; if you want JavaScript, you can recompile (though our performance tests don't cover that configuration). The character encoding is compiled in, for efficiency. Firetran has a great plug-in community—developers add code directly to the Web browser core, guarded by a `#ifdef`. Some developers change Firetran and distribute their own mutually incompatible versions. (Naturally, users of those packages submit bug reports to us that we haven't been able to reproduce with our version.) Proxy configuration is compiled in so you don't have to worry about run-time configuration dialogs: all you do is edit a makefile and recompile. To keep you secure, Firetran's https version can't use http, and vice versa. And, while the browser is open source, our development is private; if you submit a bug report or a patch, you'll likely receive the following message: *We fixed that in the private repository last year; we'll make a 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 that lets you list a maximum of 16 websites in a source file; the browser will refuse to visit any site not on the list. Also, 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. Firetran has no URL entry box; to visit a page, you edit a configuration file and run the program. A graduate student wrote a Tcl script with a text entry box to automate both configuration file editing 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, and so on, and recompile Firetran on the spot.

Needless to say, Firetran struggles to acquire market share. Yet Firetran's choices represent the status quo in many scientific software packages—which are often vehemently defended. If it's laughably unacceptable in nonscientific software, why is it tolerated in scientific software? Are scientists suffering from Stockholm Syndrome? Is scientific software so fundamentally different from other software? How could scientific software benefit from adopting the techniques we take for granted in nonscientific software?

Here, we'll examine these issues, starting with where scientific simulation software is headed.

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 its forms of extensibility—such as boundary conditions, geometry, subgrid closures, analysis techniques, data sources, and inherent uncertainty/bias—as by the underlying equations. Original authors can no longer foresee all the use cases for their software. Many common configuration and extensibility approaches create *artificial bottlenecks* that impede science goals, and the only sustainable approach is to defer all configuration and extensibility to run-time. Doing this effectively pushes applications to minimize the assumptions made about their environment, resulting in applications that are more like libraries—better suited to coupling with other models and performing advanced analysis.

Compile-Time Configuration

Many applications, especially those written in Fortran, perform configuration in the build system. (Alternatives were limited prior to Fortran 2003's ISO C bindings and now TS29113, which is slated for Fortran2015.) The motivation for configuration in the build system stems from various efficiency concerns (often ill-founded or fixable by adjusting interface granularity), software tool limitations (such as in 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 API used by scripts that perform higher-level analysis. Ad hoc public APIs inhibit software evolution by imposing an unintentionally high cost on change as well as dilution of effort to meet short-term deliverables.

In applications that rely on build-time code generation or pragma-based specialization and optimization, or those written in C++ with heavy template use, the possible combinations must be enumerated at compile-time. Although templates aren't exclusive (you can compile several variants in the same application), it's common to see a combinatorial explosion of variants as well as a direct exposure of templates in public interfaces. Because developers can't compile all combinations into one application, any analysis or testing that explores a large or unpredictable part of the combinations space must include recompilation. Attempting to push the size limits leads to

- error-prone workarounds such as `-mcmodel=large` (a compiler option that affects linking/compatibility);
- processes spanning more than one NUMA node (degrading memory locality); and
- the inability to run the application on low-memory architectures that might otherwise suit it.

Compute nodes often don't have access to compilers, making all build-system and compile-time decisions inaccessible to online analysis. A given application might be unable to run in both configurations on different nodes or on different MPI communicators. This limits analysis capability, requires frequent recompilation, and increases user errors resulting from accidentally using the wrong compiled version. The batch queues' length 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 might 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, increasing the complexity both of application debugging and of reproducing problems encountered on particular architectures.

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

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 parameter influence on the target quantities. Correspondingly, today's models are increasingly used as both forward models and 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 complexity burden to the analysis software and is algorithmically constraining—limiting parallelism, introducing artificial bottlenecks, and preventing some algorithms.

In lieu of tractable deterministic techniques for calibrating empirical phenomenological models, tremendous 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 simulation models. 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 model comparisons.

Model Coupling

Visionary scientists operating in a single domain have produced a large fraction of successful scientific software. Such visionaries predicted many important model configurations and analysis types, and the community has been largely content to explore within their fuzzy scopes. Each package has been king of its own environment and thus choices were often made without concern for interoperability or impact on other packages. However, the gaping holes in our scientific understanding and engineering capability lie increasingly in the gaps not covered by 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, in most cases, effectively nonconvergent.

The most powerful and pragmatic software approach we know of is to formulate models as libraries with a clean interface hierarchy that lets the external client 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 standalone applications, such as compilers (LLVM), Web browsers (KHTML/WebKit), and SQL databases (SQLite). Although process isolation can be useful for security (as in qmail and postfix), reliability (Web browser tabs), and distribution (remote databases), it's 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.

Provenance and Usability

Reproducibility and provenance are perpetual challenges of computational science that become more acute as the software stack deepens and a larger number of models, each 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 if we use each package without modification, compile them in a standard way, and control them entirely via run-time options. This implies that any libraries the application uses (transitively) must be responsible libraries that adhere to the principles discussed here and elsewhere.[2] For both maintenance and provenance reasons, custom components needed for a given computational experiment are better placed in version-controlled plug-ins instead of being implemented by modifying upstream sources. To support a coherent top-level specification 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.

Big Data

Workflows that involve multiple executables usually pass information through the file system. It takes about an hour to read or write the contents of volatile memory to global storage on today's top machines, assuming that peak I/O bandwidth is reached. The largest allocations (as in INCITE or ALCC awards) are on the order of tens of millions of core hours, which means 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 were used in the past, today's scientists can simply run on more cores, up to the entire machine; but, if the entire machine doesn't have enough storage, the allocation simply doesn't have the budget to run an out-of-core algorithm.

If a different application or different application version must be used for the simulation/analysis pipeline's next stage, 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 it's 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 doesn't benefit from the same amount of testing.

Nested Dependencies

Some library dependencies are indirect (transitive) via some intermediate interface that the application actually intends to depend on. A key software engineering principle is that of *encapsulation*, allowing clients to depend only on interfaces that it uses directly, rather than on implementation concerns. Encapsulation isn't possible if a transitive dependency must be reconfigured for each use case, and combining uses into one application can 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 used only indirectly.

A single library can be used by multiple components in the same executable. This might be rare when a library is first being developed, but it's 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 (this is 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 a degraded user experience and increased support workload for library maintainers.

User Modifications

Software project fragmentation 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 to combine the features developed in each fork. Fragmentation is especially toxic for libraries that might be used by multiple higher-level packages combined by the overall experiment.

Packaging and Distribution

Software developers often underestimate the challenge of installing their own packages. From the user experience perspective, 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 OS can break existing package installs if the underlying system libraries change. The most reliable way to distribute packages that will always be in sync with the OS is to have them packaged by many common OSs, such as Debian APT, RedHat RPM, MacPorts, and so on. Configure-time options are package distribution's bane because each variant must be named and conflicts between the variants resolved. Packagers for binary distributions (which are most convenient for users) are justifiably paranoid about the binary interface and hence will be reluctant to package software with fragmented configuration options.

Implementation and Recommendations

To manage these workflow challenges, application developers must think more like library developers[2] and control namespaces; avoid global state; relinquish top-level control; control the parallelism's scope; localize memory allocation; localize complexity so that it doesn't "bubble up" to the top level; and pay attention to the completeness, generality, stability, and extensibility of all public interfaces. Our suggestions are shaped by experience developing and supporting the Portable Extensible Toolkit for Scientific computation (PETSc)[3,4] as well as other packages, from low-level libraries to end-user applications. Developers have implemented similar ideas for extensibility and run-time configuration in applications such as Multiphysics Object-Oriented Simulation Environment (MOOSE; http://mooseframework.org) and PyLith (http://geodynamics.org/cig/software/pylith).

Resource Allocation

To localize configuration, allocating 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 multithreaded scenarios. If necessary, fast implementations like TCMalloc (http://goog-perftools.sourceforge.net/doc/tcmalloc.html) can be recommended, but it's better to contain this complexity in favor of good performance with any `malloc`. Performant allocation can be achieved by associating memory pools or work arrays with algorithm objects, so that `malloc` isn't called in inner loops.

Plug-ins

Source-level dependencies on an implementation (such as directly instantiating a derived class) 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. Plug-ins 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 plug-in architecture, from base linear algebra components to preconditioners, nonlinear solvers, and adaptive controllers for time integration. A plug-in can provide any of these components, which will be indistinguishable from a PETSc native component. Plug-ins consist of a registration function called via `dlopen()`—a creation function called when the plug-in is activated (such as instantiating an object implemented in the plug-in)—and any supporting functions that will be `exposed` via the object's methods. Historically, Fortran's type system and inability to store function pointers have conspired against plug-in implementations, but the new standard provides the necessary tools.

Plug-ins 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's unrelated to `libB`'s public interface, so it doesn't belong there either. We *can* create `libA-plugin` that depends on both `libA` and `libB`, registering itself as a plug-in of `libA` and calling into `libB` in its implementation. Plug-ins can also be used for optional interfaces to third-party libraries. It's best to have plug-in search paths from which plug-ins are loaded by `dlopen`, so that they can be distributed independently from the base system without requiring relinking. Shared libraries should be versioned (`-soname` on most POSIX systems, and `-current_version` and `-compatibility_version` on OSX) to make this distribution more reliable and to assist the layers built on top. (More information on shared library versioning and controlling symbol visibility is available elsewhere.[5])

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

Inversion of Control, Recursive Configuration, and the Options Database

Software libraries' primary purpose 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 aren't known at compile-time (indeed, they might 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 plug-in (implementation unknown to client code) at different locations in the object graph, each with its own configuration. Because the client doesn't know how to configure the object, some inversion of control[6] is necessary. PETSc's approach is similar to service locator,[6] but new projects should consider several variations. In PETSc, multiple objects' instances are distinguished by a *prefix* in the options database, allowing conflict-free, run-time configuration. For example, a multiphysics solver might 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 Turingcomplete 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 the physical regime or other factors. Such 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.

Object-Oriented Design

We turn now to some contentious issues in object-oriented with which we're less than enamored with the oft-repeated recommendations.

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 (that is, matrices with certain properties whose sum equals the original matrix)?

Although matrix entries can be computed in principle, the space and time complexity can be so unaffordable as to render that representation useless. Meanwhile, other operations that are unaffordable for explicitly stored matrices might be fast for matrices with special structure. Different preconditioners (which might reside in plug-ins) can 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.

Changing the run-time implementation. PETSc has found it useful for major objects to change implementations—such as from multigrid to a direct solve—at run-time. One object can have many dependencies/references and be referenced by many other objects. If the implementation can be changed only at object creation, the user ends up holding factory objects (or the equivalent) solely to recreate "similar" objects. Someone must be responsible for keeping track of these factory objects and rewiring the dependencies when replacing an existing object. This turns out to be messy and error-prone; PETSc thus chose 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 that new object's configuration.

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 client code recompilation. PETSc uses a delegator pattern (also known as a "pointer to implementation"[7] or bridge[8] pattern) to keep such implementation concerns out of the binary interface, thus minimizing recompilation and enabling binary distribution of shared library[5] upgrades. This is idiomatic in C, where "objects" are typically 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 classes' private contents. 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 two cycles. An ancillary benefit of the delegator pattern is that there's 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's increasingly popular to expose libraries through more dynamic environments such as Python or Julia. Because different languages have different type systems, it's 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.

Just-in-Time Compilation

With fine-grained composition (such as that in material models and Riemann solvers) and fusion of memory-intensive operations, the number of possible compositions grows combinatorially; in any specific run, however, only a few are 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.

Although a dynamic interface is far more maintainable, the performance overhead is unacceptable for certain applications. When the interface granularity can't be increased to amortize the overhead of dynamicism, just-in-time (JIT) compilation is an attractive approach to preserve strong encapsulation and debuggability. We expect technologies such as LLVM and OpenCL to become ubiquitous, allowing judicious use of JIT for dynamic kernel fusion and plug-in-style packaging of fine-grained components without sacrificing performance. This might involve tighter integration with languages like Julia and the Numba package for Python, or language extensions to support JIT within traditionally compiled languages.

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, while recognizing 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 is so costly as to 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's 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,[9] developers should provide versatile extension points so that contributions can be made without compromising existing functionality and without degrading package maintainability. Developers should see this 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's 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. In contrast, conditional compilation (#ifdef) is not checked, 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).

Configuration and environment design decisions made by developers of 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. ■

Acknowledgments

Jed Brown and Barry F. Smith were supported by the US Department of Energy, Office of Science, Advanced Scientific Computing Research (contract DE-AC02-06CH11357). Matthew G. Knepley was partially support by the US DOE (contract DE-AC02-06CH11357) and the US National Science Foundation (grant OCI-1147680).

This article was created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a US DOE Office of Science laboratory, is operated under contract DE-AC02-06CH11357. The US Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

References

1. D.E. Keyes et al., "Multiphysics Simulations: Challenges and Opportunities," *Int'l J. High Performance Computing Applications*, vol. 27, no. 1, 2013, pp. 4–83.
2. W.D. Gropp, "Exploiting Existing Software in Libraries: Successes, Failures, and Reasons Why," *Proc. SIAM Workshop Object-Oriented Methods for Interoperable Scientific and Eng. Computing*, 1999, pp. 21–29.

3. S. Balay et al., *PETSc Users Manual*, tech. report ANL-95/11, revision 3.5, Argonne Nat'l Lab., 2014.
4. S. Balay et al., *PETSc Developers Manual*, tech. report, Argonne Nat'l Lab., 2011.
5. U. Drepper, "How to Write Shared Libraries, 2002–2011," Dec. 2011; www.akkadia.org/drepper/dsohowto.pdf.
6. M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern," 23 Jan. 2004; http://martinfowler.com/articles/injection.html.
7. H. Sutter, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*, Addison-Wesley, 2000.
8. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education, 1994.
9. M.J. Turk, "How to Scale a Code in the Human Dimension," *arXiv preprint arXiv:1301.7064*, 2013.

Jed Brown is an assistant computational mathematician at Argonne National Laboratory and an assistant professor adjoint at the University of Colorado Boulder. His research interests include scalable solvers for implicit multiphysics, high-order partial differential equation (PDE) discretization in complex geometry, compatible discretizations for heterogeneous flows, and PDE-constrained optimization. Brown has a PhD in glaciology from ETH Zürich. He's a member of SIAM, the American Geophysical Union (AGU), ACM Special Interest Group on High-Performance Computing (SIGHPC), and Consortium for Mathematics in the Geosciences (CMG++). He received the 2014 SIAM Activity Group on Supercomputing Junior Scientist Prize and the 2014 IEEE Technical Committee on Scalable Computing Young Achievers Award. Contact him at jedbrown@mcs.anl.gov.

Matthew G. Knepley is a senior research associate at the Computation Institute at the University of Chicago. His research interests focus on scientific computation, including fast methods, parallel computing, software development, numerical analysis, and multicore architectures. He's an author of Argonne National Laboratory's Portable Extensible Toolkit for Scientific computation (PETSc) library for scientific computing and principal designer of the PyLith library for solving dynamic and quasi-static tectonic deformation problems. Knepley has a PhD in computer science from Purdue University, and won the R&D 100 Award in 2009 as part of the PETSc team. Contact him at knepley@ci.uchicago.edu.

Barry F. Smith is a senior computational mathematician at Argonne National Laboratory's Mathematics and Computer Science Division. His research interests include numerical algorithms for linear algebra and partial differential equations, and software for high-performance computing. He's a developer of PETSc. Smith has a PhD in applied mathematics from New York University's Courant Institute of Mathematical Sciences, and is a fellow of SIAM and a member of ACM. He was co-recipient (with Lois Curfman McInnes) of the US Department of Energy's 2011 Ernest Lawrence Award for outstanding contributions in research and development. Contact him at bsmith@mcs.anl.gov.