Run-time extensibility: anything less is unsustainable

Jed Brown[*], Matthew G. Knepley[†], Barry F. Smith[‡]

Modern computational science and engineering is increasingly defined by multiphysics, multiscale simulation [5] while raising the level of abstraction to risk-aware design and decision problems. This evolution unavoidably involves deeper software stacks and the cooperation of more distributed teams from multiple disciplines. Meanwhile, each application area continues to innovate and can often be characterized as much by the forms of required extensibility (e.g., boundary conditions, geometry, subgrid closures, analysis techniques, data sources, and inherent uncertainty/bias) as by the underlying equations. Sanitary workflow is paramount for smooth interaction in this environment, but it is too often compromised so long as the original author's use case is deemed acceptable. 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. We present recommendations for implementing such an approach.

Build system and compile-time configuration. The status quo for many applications, especially those written in legacy Fortran, is to perform configuration in the build system. From the perspective of higher-level analysis, the build system must then be thought of as the public application programming interface (API). In other applications, especially those written in C++ or with heavy use of conditional compilation, the choices must be made at compile time. Compute nodes often do not have access to compilers, making all build-system and compile-time decisions inaccessible to online analysis. It may be impossible for the same application to run in both configurations on different nodes or on different MPI communicators.

Advanced analysis. Today's physics 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.

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. 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

[*]Argonne National Laboratory, jedbrown@mcs.anl.gov
[†]University of Chicago, knepley@ci.uchicago.edu
[‡]Argonne National Laboratory, bsmith@mcs.anl.gov

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.

"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 for open science are on the order of tens of millions of core hours per year (e.g., INCITE), meaning that the entire annual compute budget can be burned in a few reads and writes from memory. 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. So, while global storage is sometimes convenient for inspection, and perhaps necessary if a human is in the loop (window selection in seismic tomography is currently semi-manual; other applications have especially expensive "preprocessing" steps that are reused in a sequence of simulations, where a human needs the results of the last simulation to decide what to run next), its presence in a workflow is an artificial bottleneck that must be eradicated.

Upstreaming process. 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, a form of technical debt that must be paid off 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. Meanwhile, creating a welcoming environment for contributions requires diligence from maintainers, both social and technical. The maintainers should nurture a community that can review contributions, advise about new development approaches, and test new features, as well as a tangible way to recognize contributions. Communities that become introverted in fear of "scooping" pay a heavy price in duplicate effort, barriers to refactoring, limited advising, and poor distribution of maintenance, review, and testing. In a transparent community, it is immediately clear to reviewers who did the work to implement a new feature; thus any attempt to "scoop" a result based on new capability is easily spotted. In addition to community building [6], which is a topic in its own right, developers should strive to provide versatile extension points so that contributions can be made without compromising existing functionality and without degrading package maintainability.

Implementation and recommendations

To manage the workflow challenges described above, application developers will need to think more like library developers [4], controlling their 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, and extensibility of all public interfaces. Our suggestions below are shaped by experience developing the Portable Extensible Toolkit for Scientific computing (PETSc) [2, 1].

Resource allocation. In order 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 and unavoidably ties the workflow into the build system, while committing the sin of needless global variables. Resource pools can be used to amortize allocation cost without sacrificing modularity.

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" to the top level, via typically brittle interfaces to plumb the user's configuration to the appropriate component. Plugins provide a strong way to identify interfaces that can be extended by users without modification of client code. Since a plugin implementation can reside completely outside the software package providing the interface, distribution is no longer coupled to upstream, and the plugin can be submitted at any time. 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 explicitly or via `dlopen()`, a creation function that is called when the plugin is activated, and any supporting functions that will be exposed when the plugin is instantiated. PETSc uses a delegator pattern and a dynamic per-object vtable, but other choices are also workable. Historically, Fortran's type system and inability to store function pointers have conspired against plugin implementations, but the new standard provides the necessary tools.

Inversion of control, recursive configuration, and the options database. Effective use of plugins requires that they be configurable after loading. Additionally, it should be possible to instantiate the same plugin with different configurations at different locations in the object graph. Since the client does not know how to configure the object, some inversion of control [3] is necessary (the approach taken by PETSc is similar to the "service locator" described in [3], though others also work). Multiple instances of objects are distinguished by a *prefix* in the options database, allowing arbitrary run-time configuration.

Just-in-time compilation. Occasionally, a component would be beneficial as a plugin but must be called in a context where the performance overhead of a dynamic solution is unacceptable. When the interface granularity cannot be increased to amortize the overhead of dynamicism, just-in-time (JIT) compilation is an attractive approach that can preserve strong encapsulation. We hope that the ubiquity of technologies such as LLVM and OpenCL will allow judicious use of JIT for dynamic kernel fusion and plugin-style packaging of fine-grained components without sacrificing performance.

Acknowledgments. JB and BFS were supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract DE-AC02-06CH11357. MGK acknowledges partial support from DOE Contract DE-AC02-06CH11357 and NSF Grant OCI-1147680.

References

[1] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, *PETSc developers manual*, tech. rep., Argonne National Laboratory, 2011.

[2] ——, *PETSc users manual*, Tech. Rep. ANL-95/11 - Revision 3.4, Argonne National Laboratory, 2013.

[3] M. Fowler, *Inversion of control containers and the dependency injection pattern*, 2004.

[4] W. 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, SIAM, 1999, pp. 21–29.

[5] D. E. Keyes, L. C. McInnes, C. Woodward, W. Gropp, E. Myra, M. Pernice, J. Bell, J. Brown, A. Clo, J. Connors, E. Constantinescu, D. Estep, K. Evans, C. Farhat, A. Hakim, G. Hammond, G. Hansen, J. Hill, T. Isaac, X. Jiao, K. Jordan, D. Kaushik, E. Kaxiras, A. Koniges, K. Lee, A. Lott, Q. Lu, J. Magerlein, R. Maxwell, M. McCourt, M. Mehl, R. Pawlowski, A. P. Randles, D. Reynolds, B. Rivière, U. Rüde, T. Scheibe, J. Shadid, B. Sheehan, M. Shephard, A. Siegel, B. Smith, X. Tang, C. Wilson, and B. Wohlmuth, *Multiphysics simulations: Challenges and opportunities*, International Journal of High Performance Computing Applications, 27 (2013), pp. 4–83. Special issue.

[6] M. J. Turk, *How to scale a code in the human dimension*, arXiv preprint arXiv:1301.7064, (2013).