Journal of
open research software

Dubey, A et al 2014 Software Abstractions and Methodologies for HPC Simulation Codes on Future Architectures. *Journal of Open Research Software,* 2(1): e14, pp. 1-5, DOI: http://dx.doi.org/10.5334/jors.aw

ISSUES IN RESEARCH SOFTWARE

Software Abstractions and Methodologies for HPC Simulation Codes on Future Architectures

Anshu Dubey*, Steve R. Brandt†, Richard Brower‡, Merle Giles§, Paul Hovland‖, Donald Q. Lamb¶, Frank Löffler†, Boyana Norris**, Brian W. O'Shea††, Claudio Rebbi‡, Marc Snir‖, Rajeev Thakur‖ and Petros Tzeferacos¶

Simulations with multi-physics modeling have become crucial to many science and engineering fields, and multi-physics capable scientific software is as important to these fields as instruments and facilities are to experimental sciences. The current generation of mature multi-physics codes would have sustainably served their target communities with modest amount of ongoing investment for enhancing capabilities. However, the revolution occurring in the hardware architecture has made it necessary to tackle the parallelism and performance management in these codes at multiple levels. The requirements of various levels are often at cross-purposes with one another, and therefore hugely complicate the software design. All of these considerations make it essential to approach this challenge cooperatively as a community. We conducted a series of workshops under an NSF-SI2 conceptualization grant to get input from various stakeholders, and to identify broad approaches that might lead to a solution. In this position paper we detail the major concerns articulated by the application code developers, and emerging trends in utilization of programming abstractions that we found through these workshops.

Keywords: programming abstractions

1 Introduction

Large, complex, multi-scale, multi-physics simulation codes, running on high performance computing (HPC) platforms, have become essential to advancing science and engineering in many fields. Progress in computational science, together with the adoption of high-level frameworks and modular approaches, have enabled large code development efforts such as FLASH [1, 8, 11], Cactus [5, 7], Enzo [6, 13, 14] and the Lattice QCD code suite [2, 3, 4]. FLASH was originally designed for simulating astrophysical phenomena dominated by compressible reactive flows that had multiple physical scales and therefore required adaptive mesh refinement (AMR). FLASH has extensible architecture which has allowed its user community to extend to high-energy-density physics, computational fluid dynamics, and fluid-structure interactions through the addition of needed capabilities within the existing framework. Cactus was designed as a general-purpose software framework for high-performance computing with AMR as one of its features. The first set of applications that used the framework were astrophysical simulations of compact objects involving general relativity (GR) such as black holes and neutron stars. While the Cactus framework is generic, its most prominent user today is the Einstein Toolkit [9, 12, 16], a large set of physics modules for relativistic astrophysics simulations. Enzo is a standalone application code that was originally designed to simulate the formation of large-scale cosmological structure, such as clusters of galaxies and the intergalactic medium. The Lattice QCD code suite consists of library modules that can be used by higher level applications in lattice field theory.

These codes simulate multi-scale, multi-physics phenomena with unprecedented fidelity on petascale platforms, and are used by large communities. And yet, the model fidelity achieved so far marks only the beginning. The quest for better understanding demands even higher model fidelity with fewer approximations, which can translate to adding more terms in the equation, using better and therefore more demanding algorithms, or increasing

* Lawrence Berkeley National Laboratory, Berkeley, CA, USA
adubey@lbl.gov

† Louisiana State University, Baton Rouge, LA, USA
sbrandt@cct.lsu.edu, knarf@cct.lsu.edu

‡ Boston University, Boston, MA, USA

§ NCSA, University of Illinois at Urbana-Champaign, Champaign, IL, USA

‖ Argonne National Laboratory, Chicago, IL, USA

¶ University of Chicago, Chicago, IL, USA

** University of Oregon, Eugene, OR, USA

†† Michigan State University, East Lansing, MI, USA

Corresponding author: Anshu Dubey

the resolution of the discretization. Usually an advancement in the simulation technology involves some combination of the three factors mentioned above. Therefore increasing the capabilities of these codes and maintaining their ability to utilize more compute resources of future platforms are as crucial to these communities as continued improvements in instruments and facilities are to experimental scientists.

The current generation of mature multi-physics codes exhibit an awareness of the importance of software engineering and most have evolved software maintainability strategies. Many packages have strong enough software process, and under normal circumstances they would have sustainably served their target communities with modest amount of ongoing investment for enhancing capabilities. However, the revolution occurring in the hardware architecture has completely altered the landscape for scientific computing. A code designed for distributed memory bulk synchronous model of parallelism will not only fail to run faster on future platforms, it might actually run slower. This is because the speedup that came automatically with increase in the processor speed has already come to an end. That was followed by the multicore era, which gave higher node performance by exploiting a modest amount of on-node parallelism. Codes could still achieve respectable performance either by using flat MPI (one MPI rank per core) or a hybrid model of distributed and shared memory parallelism, most often MPI with OpenMP. However, the current generation of machines could very well be the last to have homogeneous multi-core parallelism. Even among the current machines, many use some combination of cores and accelerators as their computing units. The trend towards hardware heterogeneity seems likely to accelerate as processor clock speeds are reduced for energy efficiency.

Because of this ongoing fundamental paradigm shift in the hardware, there are many more degrees of freedom in the scientific software design space. The parallelism needs to be tackled at multiple levels and the energy constraints dictate minimization of data movement and increasing data reuse. These requirements are often at cross-purposes with one another, and therefore further complicate the software design. All of these considerations make it essential to approach this challenge cooperatively as a community. We need to develop common abstractions, frameworks, programming models and software development methodologies that can be applied across a broad range of complex simulation codes, and common software infrastructure to support them. We believe that such an infrastructure is critical to the deployment of existing as well as new large, multi-scale, multi-physics codes on future HPC platforms. Furthermore, such an infrastructure should be assembled by a collaborative effort of the teams that develop and maintain such codes – that is, by the people that own the problem, not by people that own solutions looking for a problem.

2 HPC Software Concerns

The needs of the expert programmers who are developing complex, high-performance simulation codes are quite different than the needs of the broader community. Furthermore, the number of such software developers is small; their needs are therefore often ignored. There is a need to foster cooperation among the small number of teams that develop such codes, leading to a more viable ecosystem; and cooperation of these teams with computer science researchers and vendors, leading to a better understanding of the needs of this small community. Also, there is usually a gap between computer science research in areas such as code abstractions and transformations and high-level scientific application software. The gap exists for many reasons, including lack of communication between the communities involved, limits on extensibility and adaptability of scientific software, and differing real and perceived requirements by different application domains. It would be beneficial to the applications communities to overcome these and other barriers to finding and implementing broad, reusable solutions and common software infrastructure.

A similar transformation took place in the 1990s when the groups developing large scientific codes recognized the need for adopting software engineering practices to sustain code reliability in face of rapid capability growth. Each group went its own way and customized some of the prevalent ideas for its own use, many times developing their own tools. However, a close examination of the adopted practices reveals a surprising number of commonalities; component based architecture, object-oriented design, version control, coding standards, unit testing, regression testing, verification frameworks, release policies, contribution policies etc. A shared infrastructure would not only have reduced development costs, but would have facilitated code sharing.

To understand the kind of change software must undergo to adapt to new computing platforms, we first consider the current characteristics of the codes in our target community. They are typically implemented in one or more of C, C++, or Fortran. Parallel computations are implemented primarily using MPI, OpenMP, or a hybrid approach. Underlying many of the simulation implementations are a variety of mesh types (Eulerian or Lagrangian), with or without adaptive refinement support, and a number of implicit and explicit solvers with one or more implementations of each method. Some implementations dispense with meshes completely and take a purely particles based approach, while some others combine both mesh and particles. Application-specific (I/O-intensive) checkpointing schemes are typically used for fault tolerance and to provide restart capabilities. Some simulation codes rely on externally developed numerical libraries, while others are mostly self-contained, with few external dependencies.

The report of the 2011 Workshop on Exascale Programming Challenges [10] discusses in depth many of the challenges that scientific software is facing in the near future, ranging from programming models to runtime systems. Many of these challenges are not limited to extreme scales and are present even now in every scale. We briefly overview these software challenges.

- **Numerical methods and frameworks.** Certain disruptive architecture changes may require rethink-

ing of numerical approaches. Identifying the right methods for our target communities will be critical in assessing where the applied mathematics research and development will be required to effectively use future platforms. A number of numerical *frameworks* provide some means of integrating the new solutions and hiding the code and underlying data structure changes from applications. Therefore, adapting the framework, where possible, to new platforms is one way in which multiple applications can successfully migrate to new paradigms without significant reimplementation of application code.

- **Programming Models.** The programming model (e.g., MPI) used in a particular application and its supporting infrastructure is a fundamental, pervasive aspect of the implementation. Switching between programming models is labor-intensive and may require significant redesign of key algorithms, as well as massive code rewrites. At the same time, it is difficult to estimate a priori the benefits of moving to a different programming model, for example, when switching from a pure distributed memory MPI-based implementation to a model that supports a global shared memory view.
- **Programming Languages.** The high level programming languages that currently prevail in scientific computing were adequate when there was reasonable homogeneity in the basic abstract machine model across platforms, and the codes themselves used homogeneous models. With the advent of more capable machines the modeling is likely to get more demanding with diverse solvers that will need to interoperate. Heterogeneity in the abstract machine models will make the languages such as C/C++/FORTRAN too low level to meet the needs of performance portability and interoperability. While languages like Python address the latter, they do not provide adequate solution for the former. There is need for moving the programming abstractions a level higher with either richer instrinsics or embedded domain specific languages that expose the semantics, which when combined with code transformation and auto-tuning back-ends for different platform architectures can provide a good solution. A significant challenge moving forward is the mixing of abstractions in a single application in a type-safe, high-performance manner; in a large, multi-component application, there is no single "perfect" high-level abstraction.
- **Performance Portability.** Effective utilization of HPC resources is historically a balancing act between portability and performance. On one hand, extensive performance optimizations of certain key computations are crucial for achieving good performance on a certain platform. On the other hand, making such changes permanent negatively impacts code readability and performance on other platforms. it is important to identify and eliminate barriers to enabling greater flexibility in choosing among different optimized implementations. Another critical aspect of future performance portability is the ability to support different levels and types of parallelism.
- **Resilience.** Current codes rely on checkpointing for error recovery. This solution, however, will be prohibitively expensive on large-scale systems; hence, new ways of ensuring resilience are required which may involve changes in programming models, runtimes, and application design and implementation.
- **Productivity and Maintainability.** Alternative technical approaches to managing increased levels of parallelism, heterogeneity, and other architectural features have different impacts on programming effort and software maintainability. Furthermore, complex codes do not allow easy testing of new concepts and thus slow down advances in both the scientific and numerical approaches. We must understand the current and desired mode of development in our target communities to guide the approach to design decisions.

We note that the same set of challenges largely apply to both physics-rich simulation codes and to the software that will analyze and/or visualize the massive amount of data produced. As a result, innovative analysis and visualization strategies (pioneered by, e.g., the *yt* toolkit [15]), may require that these tools are co-designed with each other and with the hardware platform, and careful thought must be given to how the different data access patterns of simulation vs. analysis/visualization tools will affect performance on many-core, heterogeneous computers. Furthermore, it may be that most simulation analysis must occur during simulation runtime rather than after the calculation is complete in order to maximize scientific yield and manage scarce storage resources.

3 Community Input

In order to facilitate exchange of information among various stakeholders and to find generally applicable solutions to the software problems described above we have been conducting a series of workshops under the aegis of an NSF SI2 conceptualization grant. The first of these workshops brought together domain experts in several scientific fields: software developers who are involved in implementing and optimizing many of the large codes for these fields and sectors, researchers in applied computer science, and hardware and software vendors. The domain experts and/or the code developers for the domain provided information about the current models and algorithms used in simulations with emphasis on the data structures, memory characteristics and communication patterns. They also spoke about the direction their research was going to take in near and far future, the limitations in their models and resources that they are faced with, and what they would like to achieve. They further spoke about the preparations they are making for the future, where their greatest challenges lie in making those preparations, and where they could be helped by a community based effort. The programming abstractions experts informed the attendees about the state-of-the-art in their respective fields and the future trends. They were asked to speak about the degree of penetration that

various tools developed by their communities have had among the scientific code developers, and the barriers to their adoption. Domain experts commented on the pros and cons of various models and abstractions discussed.

The second workshop focused on HPC in manufacturing. The adoption by the manufacturers of software developed in academic institutions is very limited even though these packages offer many capabilities desired by the manufacturers. The workshop, therefore, set out to find the barriers to adoption and what, if anything, would help.

The third workshop focused on major classes of numerical algorithms that are critical for scientific codes, and the codes used by the industry.

The common theme running through all the workshops was gathering a variety of perspectives and wish-lists and trying to understand how they could be used in developing possible approaches toward common abstractions and frameworks.

4 Emerging Themes and Recommendations

In the course of the year and a half during which the workshops were conducted a few themes emerged among the applications. Almost all applications have greater concern about heterogeneity than the scale of the machines. In general strong scaling challenges are receiving greater attention than weak scaling because they will be faced by everyone, not just those groups who are specialized to using HPC. All the applications groups expect their complexity to rise both for algorithms and for interaction between code components. All of the above issues have led to a growing acceptance among the scientists that disruptive changes to their software bases are inevitable, and that there is great uncertainty about what, if any, actions can be taken in the near future to mitigate the problem. Even the manufacturers, who tend to prefer not to operate at the bleeding edge, have recognized a cause for concern regarding the sustainability of their software.

Two dominant themes are also emerging with regard to programming abstractions for taking the codes to the next generation; code transformation and asynchronous runtime management. While the individual tools and compilers for providing these functionalities will be different, the applications will have to provide footholds for the related abstractions. It is known that fine-grain parallelism could impose data and housekeeping overheads, therefore constraints on the location of data and operations on it have to be relaxed so that auto-tuning tools can rearrange them as needed. Similarly, it is known that bulk synchronous processing makes the worst use of the network, imposes the harshest performance penalties on algorithms, and may not scale. Therefore in addition to relaxing control on where data resides and who executes it, constraints have to be relaxed also on when a task executes. The applications can do so by taking the separation of concerns a step further than they already do, that of separating numerical and parallel complexity. They have to leave data-staging and assembly to the infrastructure and expose minimum computation units, both spatial and temporal, that can be exploited by the code transformation tools. The applications also have to explicitly articulate the dependencies within the code to plug in dynamic task scheduling.

Because of the above considerations, the re-factorization and transformations which are needed are at the fundamental implementation design level in the codes and the libraries. The data layout, the wrapper layers, and the communication channels between different code components have to be designed with an awareness of the semantics of the programming abstractions using asynchronous task management and code transformations. Such refactoring is also likely to pay dividends in other ways such as reliability and resiliency of the code. It is imperative that support is provided for refactoring of the mature codes that are already serving their communities well in this manner because the changes to the architecture of most codes will be highly disruptive and therefore labor intensive. The alternative, code development from scratch, might succeed in a few instances, but is unlikely to meet with broad success. The reasons are: (1) an unconstrained design space has a potential to not converge, as happened to many high level frameworks 12–15 years ago, (2) code verification during refactoring remains tractable when solutions can be compared against a known set of solvers, and (3) to build a robust multiphysics code is long and arduous process, most mature codes have taken 5–8 years to arrive at the level of confidence that they now enjoy. We believe a judicious combination of disruptive and incremental changes are the optimal way to continue to serve the cause of science.

References

1. **FLASH user's guide** Available at: http://flash.uchicago.edu/site/flashcode/user_support/flash4_ug
2. **The chroma library for lattice field theory** Available at: http://usqcd.jlab.org/usqcd-docs/chroma
3. **Quda: A library for qcd on gpus** Available at: http://lattice.github.com/quda
4. **Usqcd sofware releases** Available at: http://usqcd.jlab.org/usqcd-software
5. **Allen, G, Benger, W, Goodale, T, Hege, H-C, Lanfermann, G, Merzky, A, Radke, T, Seidel, E,** and **Shalf, J** 2000 The cactus code: a problem solving environment for the grid. In: The Ninth International Symposium on High-Performance Distributed Computing, 2000. Proceedings, Pittsburgh, PA in August 2000, pp. 253–260.
6. **Bryan, G L, Norman, M L, O'Shea, B W,** et al 2014 ENZO: An Adaptive Mesh Refinement Code for Astrophysics. *The Astrophysical Journal Supplement*, 211(2): 19.
7. **Cactus developers** 2013 Cactus Computational Toolkit. Available at: http://www.cactuscode.org/
8. **Dubey, A, Antypas, K, Ganapathy, M K, Reid, L B, Riley, K, Sheeler, D, Siegel, A** and **Weide, K** 2009 Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10–11): 512–522.
9. **EinsteinToolkit maintainers** 2013 Einstein Toolkit: Open software for relativistic astrophysics. Available at: http://einsteintoolkit.org/

10. **Exascale Programming Challenges Report** 2011 Ascr programming challenges for exascale computing. Available at: http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/ProgrammingChallengesWorkshopReport.pdf
11. **Fryxell, B, Olson, K, Ricker, P, Timmes, F X, Zingale, M, Lamb, D Q, MacNeice, P, Rosner, R, Truran, J W** and **Tufo, H** 2000 FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement*, 131: 273–334. DOI: http://dx.doi.org/10.1086/317361
12. **Löffler, F, Faber, J, Bentivegna, E, Bode, T, Diener, P, Haas, R, Hinder, I, Mundim, B C, Ott, C D, Schnetter, E, Allen, G, Campanelli, M** and **Laguna, P** 2012 The Einstein Toolkit: A Community Computational Infrastructure for Relativistic Astrophysics. *Classical and Quantum Gravity*, 29(11): 115001. DOI: http://dx.doi.org/10.1088/0264-9381/29/11/115001
13. **Norman, M L, Bryan, G L, Harkness, R, Bordner, J, Reynolds, D, O'Shea, B** and **Wagner, R** 2008 Simulating Cosmological Evolution with Enzo In: *Petascale Computing: Algorithms and Applications*. Chapman & Hall/CRC, pp. 83–102. Arxiv preprint arXiv: 0705.1556.
14. **O'Shea, B W, Bryan, G, Bordner, J, Norman, M L, Abel, T, Harkness, R** and **Kritsuk, A** 2005 Introducing Enzo, an AMR cosmology application. In: Plewa, T, Timur, L and Weirs, V G (eds.) *Adaptive Mesh Refinement – Theory and Applications*, *volume 41 of Lecture Notes in Computational Science and Engineering*. Springer.
15. **Turk, M J, Smith, B D, Oishi, J S, Skory, S, Skillman, S W, Abel, T** and **Norman, M L** 2011 yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *The Astrophysical Journal Supplement*, 192: 9.
16. **Zilhão, M** and **Löffler, F** 2013 An Introduction to the Einstein Toolkit. *submitted to IJMPA*. (arXiv:1305.5299).

How to cite this article: Dubey, A, Brandt, S R, Brower, R, Giles, M, Hovland, P, Lamb, D Q, Löffler, F, Norris, B, O'Shea, B W, Rebbi, C, Snir, M, Thakur, R and Tzeferacos, P 2014 Software Abstractions and Methodologies for HPC Simulation Codes on Future Architectures. *Journal of Open Research Software,* 2(1): e14, pp. 1-5, DOI: http://dx.doi.org/10.5334/jors.aw

Published: 9 July 2014