SN Computer Science (2020) 1:223
https://doi.org/10.1007/s42979-020-0077-x

ORIGINAL RESEARCH

Distillation of Best Practices from Refactoring FLASH for Exascale

Anshu Dubey[1] · Jared O'Neal[1] · Klaus Weide[2] · Saurabh Chawdhary[1]

Received: 1 July 2019 / Accepted: 20 February 2020 / Published online: 2 July 2020

Abstract
FLASH is a multiphysics software package that was created in 1998 by combining three preexisting packages and has undergone three major revisions. Software design and engineering practices were integrated early in the development and maintenance processes of FLASH, and these processes have evolved strongly at each of the revisions. As high-performance computing enters the age of exascale, challenges along the orthogonal axes of node-level hardware and solver heterogeneity force developers of complex multiphysics software to consider a software architecture overhaul. Because of the nature and scope of necessary changes, an effort to refactor and grow the architecture of the FLASH code has been launched as a separate software project. For this project to succeed, its development team must evaluate, improve, and modernize software processes and policies to meet the unique challenges posed by the exascale era. We describe here our experiences, lessons we have learned, and the methods that we have developed as part of this ongoing project. Within the context of the challenges posed by exascale, we review the FLASH design approach as well as some of the main software engineering processes and tools that have been implemented or updated throughout the lifetime of FLASH. Modernization applied to these processes and tools is also detailed. Reviewing and reevaluating the FLASH experience of establishing and updating software design and engineering practices have been helpful in understanding the needs of the project as it transitions to exascale and in planning the transition. We find that our historical design methodology is still important and relevant. We also believe that using a mixture of plan-based and agile methods is still the best for our project and is in accord with the guidance found in the literature. We present a section on inferences and lessons learned related to software design and engineering practices.

Keywords Exascale · Refactoring · Software design · Software engineering

Introduction

FLASH [1] is a highly configurable scientific software package that has been in development since 1998, when three preexisting code bases were combined to produce general-purpose software for simulating reactive flows often found in astrophysics. The three component codes provided distinct functionalities: Paramesh [2] is an adaptive mesh refinement (AMR) library, Prometheus [3] computes reactive hydrodynamics, and another collection of functions computes equations of state and nuclear burning [4]. The code is written primarily in Fortran, and the current production release contains approximately 1.5 million lines of code. The people responsible for this software are distributed across several academic institutions and national laboratories and function effectively as a "team of small teams." Development and maintenance are therefore carried out by graduate students, postdoctoral researchers, university professors, and professional research staff.

This article is part of the topical collection "Software Challenges to Exascale Computing" guest edited by Amit Majumdar and Ritu Arora.

✉ Anshu Dubey
adubey@anl.gov

Jared O'Neal
joneal@anl.gov

Klaus Weide
kweide@uchicago.edu

Saurabh Chawdhary
schawdhary@anl.gov

1 Argonne National Laboratory, Lemont, IL 60301, USA

2 University of Chicago, Chicago, IL 60637, USA

The importance of good software development and engineering practices was recognized early in the project. Efforts have been made to study such practices in industry and in other computational science, engineering, and mathematics projects and to adapt useful, relevant practices to the needs and challenges of developing FLASH. Where tools and practices were not appropriate for FLASH or were nonexistent, the FLASH team carried out research and development to produce tools and processes to address shortcomings.

FLASH has undergone three major version updates, and the software engineering and auditing process has evolved strongly with each revision [5, 6]. Two of these were refactorings that included deep changes to the software architecture of the code [7]; the third added a significant number of new physics capabilities to enable FLASH's use in high-energy-density physics (HEDP) research. Much of this evolution occurred dynamically as the team reacted to specific challenges posed by the revisions.

As a general rule, when evaluating or designing possible software engineering processes and tools, preference has been given to simplicity and informality in the interest of limiting overhead, which can be important for ensuring productivity in small research teams that are also responsible for professional development of junior team members. This goal is partially responsible for the aforementioned need to adapt processes found, for example, in industry and in library development projects.

As we consider the landscape of exascale and post-exascale supercomputers, we are confronted with the orthogonal axes of both platform heterogeneity and solver heterogeneity. While the onset of platform heterogeneity is a well-known challenge, a concern that is discussed less frequently is that greater scientific understanding has given rise to larger and more complex codebases with diverse solvers that must interoperate. When combined, these two axes of complexity substantially stress the code infrastructure, forcing us to consider a fundamental architectural redesign. Because this architectural update is the most challenging refactoring to date, it also requires the reevaluation and improvement of software processes.

In this paper, we briefly review some of the main software engineering processes and tools that have been implemented or updated throughout the lifetime of FLASH. The focus of the review is on practices that are either continuing to be useful or have been instrumental in providing insights that are useful now. This review leads into a detailed discussion of our approach to redesigning the software infrastructure and the evolution of our software engineering practices in response to the exascale challenges. Additionally, we present a section on inferences and lessons learned related to software design and engineering practices. The contributions of the paper are twofold: (1) a design methodology that other codes can adopt to prepare for exascale and (2) a crystallization of experience with adapting software engineering practices for very complex scientific research applications developed and maintained by a small team. The former is effectively what we view as best practices for other projects with similar scope and challenges.

Background

Considerable studies have been done regarding refactoring methods and tools. Among them, [8] provides a good survey and [9] considers motivations for refactoring. While the literature for commercial software refactoring is large, our experience has been that commercial techniques can require significant adaptation in order to meet the needs of developing FLASH [1]. Therefore, we focus our attention on publications related to software restructuring and refactoring in the scientific world. Several of these papers, such as [9], are case studies from specific software projects. Articles such as [10–13] discuss the lack of software engineering practices in scientific software development, where refactoring is one of the considered features. Among these [13] is notable in that it attempts to provide a systematic study of scientific software development. Literature on general studies of refactoring in scientific software is sparse, with a few exceptions such as [14, 15], which are restricted to Fortran code refactoring, and [16], which defines a methodology for code development taking into account ongoing changes in specifications that are an inevitable part of scientific software development.

FLASH is a component-based code where different combinations of components constitute different applications. Furthermore, each component can have multiple alternative implementations of its functionality, which adds to the degree of composability of the code. While FLASH is implemented with procedural programming, the organization of files and subroutines and the build system were designed and implemented to take advantage of the benefits of architecting the code with object-oriented programming while still enjoying the benefits of procedural programming.

Specifically, related functionality and data that can be grouped as a single class are organized into a code unit in FLASH. Each unit is isolated in the FLASH source directory as a subdirectory hierarchy containing all related routines. Naming conventions differentiate between routines that implement the public interface of the unit from those that are internal to the unit. All public interfaces have null implementations as well as one or more full implementations. Therefore, by including its null implementation during configuration, this mechanism allows a unit to be excluded functionally without having to change the source code. Also, any of the full implementations can be selected during configuration, thus allowing for multiple alternative implementations of the same API. The design allows for a simple form of inheritance so that common code may be written and either used or overwritten by any particular implementation.

An example of a unit with significantly different implementations is FLASH's Grid unit, which manages the discretized mesh. To expose parallelism, the Grid unit decomposes the spatial domain into non-overlapping blocks, each of which is a logically rectangular collection of cells defined by the mesh. The mesh can be uniformly spaced everywhere, or it can be adaptive, where adaptivity is obtained by decomposing regions of the problem where fine-scale features exist with high-resolution blocks and by using only low-resolution blocks elsewhere. Besides managing the mesh definition and evolution of mesh resolutions in time, the Grid unit stores data defined on the mesh, provides access to the data, provides all mesh-related information for the simulation, and collaborates with other units through blocks and their associated metadata. Because of its extensive support role, we commonly refer to the Grid unit as part of the infrastructure of FLASH and recognize its importance since it is used throughout much of the codebase. For instance, physics units request from the Grid unit a list of blocks on which to operate and proceed by typically looping over blocks to apply their operators (physics solvers) to the data. This mechanism abstracts away the physical location of the block in the domain from the physics operator, which achieves a clean separation of infrastructure from physics.

FLASH Design

FLASH's design was dictated from the outset by its overarching science goals and the composition of the team. Scientific members of the team wanted a code with reusable components that could be combined as needed for their problems of interest. While many features were common among these problems, they were still different enough to warrant composability in configuration. The solution to this challenge has been the most influential design choice to date. A set of configuration files encoding meta-information, which are generally referred to as Config files, and a Setup tool to parse and interpret the Config files' meta-information were established in the first version of the code and are used to configure a unique binary for each application. This approach has persisted throughout the history of the code. However, the syntax of the Config files and features enabled by the setup script have been successively enriched. In fact, the vast majority of changes to the code architecture through version 4 have been enabled by augmenting the Config file syntax and implementing the structure-based inheritance mechanism in the Setup tool (see [7] for more detailed description).

The second far-reaching design choice was engendered by the unusual (for its time) composition of the team, which consisted of astrophysicists, applied mathematicians, and computer scientists. The team differed from those working on other contemporary code development efforts in that although science concerns were driving the development, the scientists were visionary enough to understand that investing in well-designed and well-engineered code would yield long-term efficiency and success. An enduring design choice that came out of this interdisciplinary team of developers was separation of concerns, achieved through the establishment of an API for functional units of the code.

Design Approach

A methodology for code design is depicted in Fig. 1. It has been implicit in our design process since the transition from the second to third version, and insight from this methodology is now used to inform most of our design process. The premise of this methodology is that there is a fundamental difference between the two types of components within the code: those that are relatively stable and those that change on a much smaller timescale. The code units that pertain to the infrastructure and the framework are the stable backbone of the code and generally are not the subjects of ongoing research. This type of code does, however, tend to interact with all other components, making it more difficult to shield other code units from infrastructure modifications. Therefore, their design space exploration should be deeper. On the other hand, the solver units are subjects of science and algorithmic research and are therefore liable to change frequently. Since these units are clients of the infrastructure, they have a more contained sphere of influence, and thus, their design and development are best done in an agile manner.

Both types of code are encapsulated within functional units encoded within a framework. The framework takes care of the data layout, owns the global state data, arbitrates among various units for ownership of other data, and manages interaction among code units. Following object-oriented programming principles, units differentiate between protected and public functions, and there is explicit scoping of local unit data. Interfaces are defined for each individual unit, whether it is infrastructure or a physics solver, and are the contracts that specify how units must interact.

The development and maintenance of the infrastructure are done largely by a collocated group of developers at Argonne National Laboratory and the University of Chicago. Long-term researchers on this team help ensure long-term quality and coherence across the stable portions of the code and across all contributions made by shorter-term contributors.

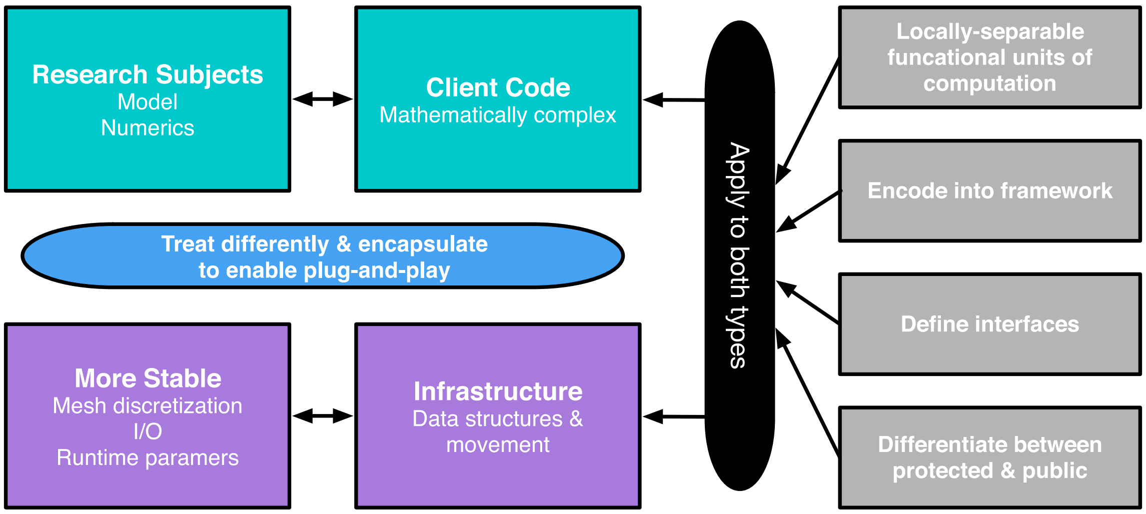

Fig. 1 Software design with separation of concerns by treating infrastructure and research functional units differently. Infrastructure, such as the mesh and I/O units, is more stable and needs a longer upfront design cycle compared with physics and numerical units, which evolve more rapidly. Although they are to be treated differently for design, all components become part of a framework through defined interfaces

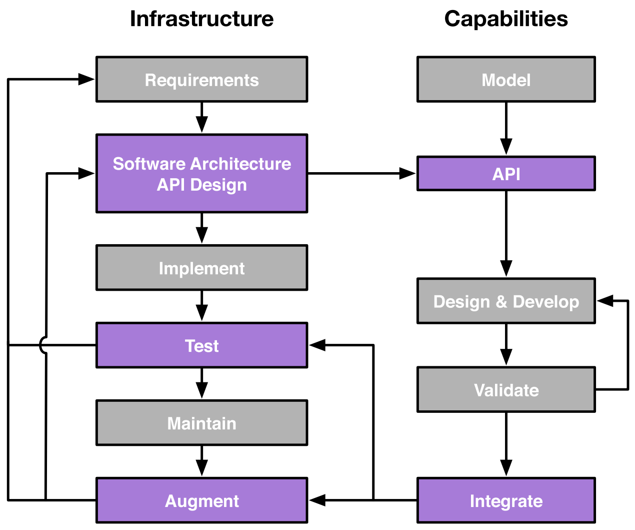

Fig. 2 Methodology for developing infrastructure and research functional units differently. Purple boxes indicate the points of interaction between the two lines of development (color figure online)

Figure 2 demonstrates how the design methodology can map onto a development process that allows parallel development and coexistence of the stable and research code units. If the interfaces of the infrastructure are mature and are sufficiently general, matching the interface of a new physics capability to the infrastructure should be natural and easy. In order to include a new physics capability in simulations, the physics code must expose its high-level interface to the unit that drives the evolution of the simulation so that it is invoked. Occasionally, a new capability needs to make its presence known to an existing capability (e.g., a new method for elliptical solve that can be used by the Gravity unit), and in rare instances, a new capability may need some augmentation of the infrastructure (e.g., mesh replication for radiation in HEDP simulations). In such cases, new tests may need to be developed for verification of interoperability.

The design methodology exists in the middle of the spectrum of possibilities between plan-based and agile methodologies [17–19]. Specifically, informal plan-based techniques are used for the development of the framework. We prefer expending a reasonable amount of upfront resources to understand the requirements of the more stable portion of the code in the hope of increasing the likelihood that early in the development process we will achieve clean, mature interfaces that are sufficiently general to be compatible with the interface needs of the different physics codes. If we were to use agile methods for this development, we perceive a higher risk of having to undertake frequent, expensive refactoring efforts. This is related to the possibility that delaying important design decisions for as long as possible can itself be an important design decision with serious consequences [17].

We also believe that another result of this informal style of requirements engineering (see "Design Process" section) is having a multidisciplinary team arrive at a minimally sufficient, upfront understanding of its goals, the constraints that are imposed on the design, and the constraints that we choose to impose on the design. This understanding is important because changes in the infrastructure and its interface affect every other part of our large codebase. While this does not imply that changes are not possible or not required once the infrastructure has been implemented and accepted, it does limit the number and size of changes.

Design Process

Beginning with version 3, FLASH's design process has involved a combination of brainstorming sessions to discover our requirements, subsequent cost–benefit trade-off analysis, and readjustment of the requirements. The results of these discussions are captured as snapshots of white boards and are archived for future reference so that they may serve as guidelines for developing prototypes. The prototypes themselves are lightweight and are meant to explore the design space rapidly. It is understood that none of these prototypes are to be pulled into the code without going through the auditing steps of testing, verification of coding standards, and documentation.

The snapshots serve as a proxy for formally documented requirements and design documents. Since the development teams are small, simplifying formal processes such as documentation of requirements gathering and analysis is reasonable and desirable. In mathematically complex software, relying on new members of the team to become productive through reading of such documents alone is unrealistic. Following agile principles, we therefore deem such documents as low value and prefer a more effective process of veteran team members mentoring new members, training new members, and serving as a source of requirements knowledge. Those members who stay with the team long enough become veterans by acquiring a comprehensive mental model of the software and storing the history of requirements in their head and in private notes. We adopt such simplifications where possible, with the understanding that at any time there must be sufficient overlap between experienced and inexperienced developers for this mentorship model to remain viable. Note that this can allow junior staff to avoid engineering processes that could be perceived to interfere with their immediate career goals. Conversely, they are not exposed to potentially important parts of software design and maintenance.

Preparing for Exascale

Prior refactoring experience has shown that a good first step in restructuring complex software is to consider and plan the degree and scope of change. For exascale, we identified several major functionalities that we need to add to the infrastructure. Specifically, we need to expose and capitalize on parallelism at arbitrary levels of granularity, to have more sophisticated load distribution and balancing in order to achieve necessary scaling performance, and to introduce asynchronicity within physics operators and between such operators.

Some of these functionalities can be obtained by expanding the capabilities of the Grid unit. To this end, this refactoring effort includes adding a new Grid unit implementation based on the third-party adaptive mesh refinement library AMReX that is being developed at Lawrence Berkeley National Laboratory specifically for exascale applications. In doing so, our software leverages AMReX developments such as the inclusion of tiling for finer granularity in spatial decomposition, better regridding for scaling, use of fork-join parallelism for some degree of overlap between operators, and asynchronicity between different levels of refinement [20].

While this is a significant advancement, many facets of exploiting parallelism still remain to be addressed. These include arbitration of what operators run on which device, mapping data structures to different available memory types, and accompanying cost–benefit analysis for determining these mappings. Clearly, tackling all these enhancements in the infrastructure simultaneously would make the process too complex, with a high probability of failure. Efforts such as [21] have demonstrated the feasibility of managing some degree of platform heterogeneity within an existing code architecture, which gave us confidence to rearchitect in two stages.

Stages of Rearchitecting

In the first stage, we interfaced the code with AMReX in a way that would allow us to use the advanced parallelism features being added to it; in the second stage, we are optimizing for remaining facets of parallelism by minimizing the penalties of data movement through judicious orchestration of computation on different devices. To manage this, it is necessary to design the first stage with awareness of potential needs in the second stage so that a reasonable design in the second stage is not precluded by the design in the first stage. As the second stage development is in a very early phase, we focus primarily on the first stage in this article.

A conceptual realization for the architecture of AMR-based codes was presented in [22], which became our starting point for the first stage. It has since been modified to incorporate new information that has become available about forthcoming platforms as well as the insights we have gained during implementation. Our current working conceptualization of high-level code architecture is shown in Fig. 3, where the scope of the first stage of the process is in the top right-hand quadrant above the dashed lines. During this stage, our focus is on the virtualization of decomposed domain sections so that the mesh manager can decompose or coalesce blocks as needed. A beneficial side effect of this virtualization is that it enables the mesh to operate asynchronously without the necessity of any further changes to the mesh interface.

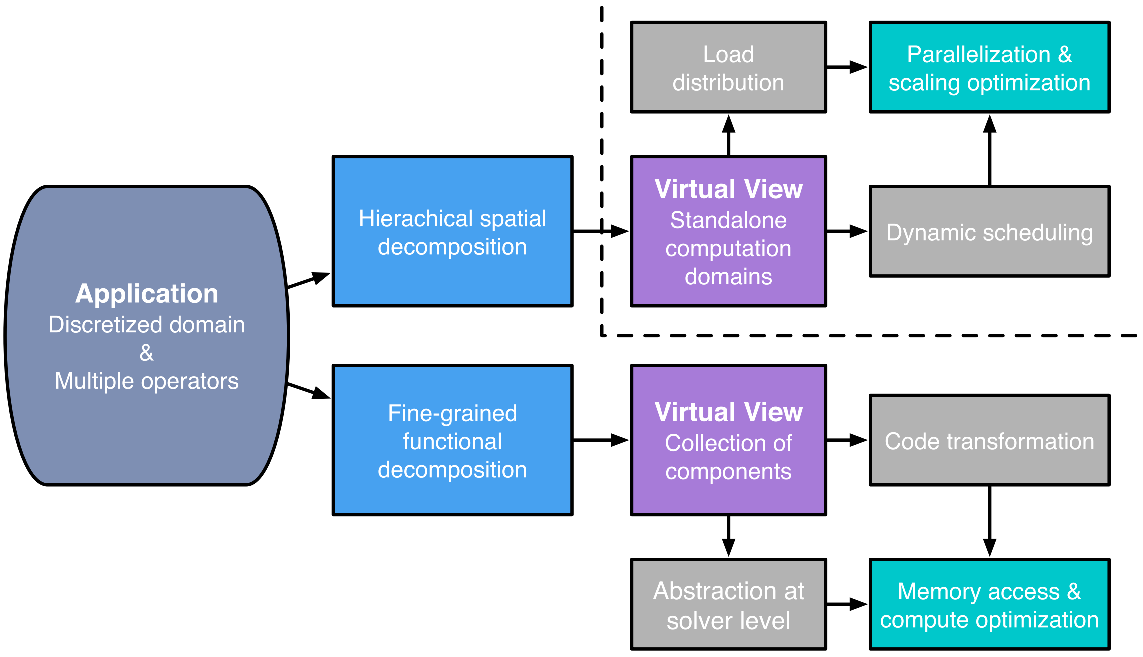
Application
Discretized domain
&
Multiple operators
Hierachical spatial decomposition
Fine-grained functional decomposition
Load distribution
Parallelization & scaling optimization
Virtual View
Standalone computation domains
Dynamic scheduling
Virtual View
Collection of components
Code transformation
Abstraction at solver level
Memory access & compute optimization

Fig. 3 Schematic of software architecture with added abstraction layers for tackling heterogeneity. The main abstraction is achieved by looking at decomposition through a virtualization process. Spatial virtualization is obtained by surrounding each block with halo of ghost cells and mapping its physical location in the domain to an integer index space so that the operator views each block as a stand-alone domain. Similarly, the functional virtual view treats operators as collections of functions with their dependencies articulated. The section of the figure above and to the right of dashed lines show the scope of this work

Evaluation of Historic Processes and Updates

In any rearchitecting effort, it is not just the code and its development that need to be considered. It is equally important to evaluate software processes and plan for necessary changes in policies and practices. This effort also provides an opportunity to incorporate new tools and methodologies developed by research in software engineering, productivity, and sustainability. An evaluation of the development team's current processes and toolkit indicated that the launching of the new project should include improvements to the version control management of the code, licensing of the code, and the release mechanism.

While previous instances of the FLASH code have been managed via Subversion, an important upfront decision was to manage the new project via the distributed version control system Git and with the repository hosted in GitHub. A side effect of this decision is the adoption of the software development practice of social coding, which leads to enhanced communication and collaboration. An example of this is clearly laying out on the repository's landing page coding policies as well as rules that govern how integration of parallel development of code is managed through the version control system. Other examples are triggering code reviews via pull requests, using GitHub Issues to track work, and augmenting our requirement engineering scheme by gathering requirements using Issues. This new layer of communication and the fact that all team members use this layer and follow our standards and policies allow individual developers to follow easily the state of the overall project. The result is a broader, inclusive understanding of the development across the codebase so that each developer has a partial understanding of the work being done by others and the potential impacts of that work on their tasks. Notably, new members of the team and especially those new to large-scale development projects report that this practice allows for learning the development process easily, in turn leading to quicker integration in the team.

The switch from a centralized version control system to a distributed version control system inevitably led to reconsidering the way in which many developers working in parallel must collaborate by integrating their distinct and possibly conflicting changes through the version control system. The new code has adopted a Git workflow that relies more heavily on branches and rigorous integrated testing. Specifically, we have policies in place so that code contributions from a larger base of contributors can be handled efficiently and without sacrificing the correctness and quality of the code. As shown in Fig. 4, we have adopted a workflow that has two persistent branches, *staged* and *development*, in addition to the customary *master* branch.

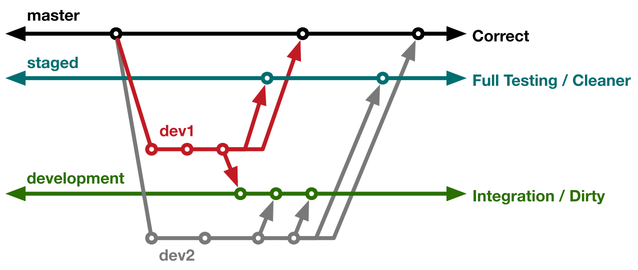

Fig. 4 Conflict resolution in the current Git workflow. The dev1 branch is able to merge cleanly, but conflicting work in parallel dev2 branch finds a merge conflict. The resolution is done in dev2 and reintegrated in development before moving to stage

Each new development or maintenance task is carried out on its own separate, dedicated feature branch, created strictly from *master*. In the spirit of continuous integration, these branches are usually expected to be short-lived. When a task is ready for formal inclusion in the software, the feature branch is first merged into *development*, where integration with other lines of new development occurs and where all merge conflicts are expected to be resolved. After developers manually run the entire code test suite, or a subset of it, to ensure that to the best of their knowledge there are no remaining conflicts in this branch, a pull request from the feature branch into *staged* is created. The pull request is accepted after code review and if all workflow policies have been followed. A full test suite run can be triggered manually or occurs nightly through a Jenkins-based interface on both the *staged* and *master* branches. Merging of the feature branch into *master* is allowed to proceed only if all tests on *staged* pass. The insistence that feature branches must be merged into *development* and *staged* before *master* ensures that the contents of these three branches cannot diverge significantly. These rules also effectively isolate *master* from the persistent branches on which integration is done and where abandoned work can be found. We intend to install automatic continuous integration testing with a small subset of tests to serve as immediate smoke testing.

The creation of test suites for different platforms and the management of test suite execution are done with a custom test framework known as FlashTest [23], which was developed into its current form with version 3 of the FLASH code. It evolved from a much simpler automated system developed at version 2 out of necessity due to the lack of Fortran-based testing frameworks at the time. While FlashTest lacks some features that are commonly found in modern test frameworks, we find that the tool is still good enough because of its ability to grow test suites easily, view detailed results of each test run, and easily group baseline results along with its context. Therefore, we choose to dedicate resources to other tasks rather than modernizing the FLASH testing infrastructure.

Tests run with FlashTest can be classified either as regression tests or as tests that return to FlashTest a simple pass or fail status. Generally, the regression tests consist of full simulations whose runtime parameters have been judiciously chosen so as to test a specific functionality included in the simulation or to increase the sensitivity of the test to errors. The second class of tests often confirm correct functionality at the level of a FLASH unit. However, while integrating the library AMReX into FLASH as a new Grid unit implementation, we found that simulation-level regression tests and the comprehensive Grid unit test were too high level and coarse grained. This shortcoming exists because the amount of functionality provided by the Grid unit is high and some functionality is sufficiently complex that high-level tests are not useful for this type of integration. Therefore, we have expanded our development practices to include finer-grained testing at the level of routines within a unit, in the hope that this leads to more productive development. In addition, these tests, which have short runtimes, can be used in CI-based smoke testing and can also be used to more efficiently identify the location of bugs.

Unlike earlier versions of FLASH, which have been and continue to be released through a custom license with restrictions on distribution, the new code is released under the open source Apache 2.0 license. The new code is developed with continuous release with master branch tags. At this writing, its releases are in the alpha stage and are largely meant to give the community a preview of what to expect and to become familiar with the changes.

Inferences and Future Work

Based on our experience during this first stage of refactoring, as well as from the earlier refactoring efforts, we draw the following inferences regarding likely best practices for rearchitecting scientific software while preparing it for next-generation platforms.

- It is critical to think through the motivation, scope, and extent of changes to be made in the code before starting the refactoring process. One should clearly have a map between the starting stage and where the refactoring will bring the code.
- Similarly, it is important to draft a decomposition of a large refactoring task into smaller refactoring steps that can each be executed independently and each verified by preexisting tests. Our experience indicates that it is sometimes necessary to abandon a refactoring effort, redo the refactoring decomposition into steps, and start anew.
- Resource estimation in terms of both developer time and disruption in production schedule is necessary, but it is extremely difficult to do accurately. One should err on

the side of too much rather than too little. All the auxiliary activities such as test development and code cleanup should be part of the estimate.
- There is no substitute for investment in software architecture design. It may take some effort and non-trivial amounts of resources in the beginning, but it pays off by saving effort later [24].
- Use of an automated testing framework saves developer time throughout the refactoring process. It can be as simple as a script that runs the tests and produces results that can be easily and correctly interpreted by new members and team members with insufficient domain knowledge.
- Checking for code coverage and developing useful tests where there are gaps can prove to be the biggest saving throughout the process. By useful tests, we mean tests that can quickly help pinpoint errors when they occur. It is also important to verify that the tests fail as expected when there is an error.
- The refactoring process should allow for quick and dirty prototyping to explore the design space, with appropriate provisions for either rewriting or cleaning up the code once there is convergence on a design. Applying a full auditing process during design exploration is inefficient, and allowing quick and dirty code to remain in the repository increases technical debt.
- Establishing a policy for development workflow and branch management in the repository helps minimize the technical debt without undue strain on team resources. Branch structures can act as a filter for bad code, with minimal investment needed for code review.
- The same applies to coding standards agreed upon at the beginning of the refactoring project.

With an experienced though small team (4 developers at 20–60% time), infrastructure refactoring has taken roughly 2 years from the time we started to the time we had an alpha release of the code. Details of the implementation of this stage of work are given in [24]. We found that the early investment in designing for separation of concern considerably reduced the amount of code that needed to be modified for this project. A large fraction of the original architecture design during the revision to version 3 remains useful and will continue to exist in the code.

In the second stage of refactoring the architecture, which involves orchestration of data movement and work distribution among devices, we intend to leverage the configuration system. The philosophy of the configuration system is to distribute the knowledge of code unit developers as meta-information encoded in the form of application-specific text-based syntax, in appropriate places throughout the source code. The aforementioned Setup tool can parse this information and has sufficient intelligence built into it to configure an application by selecting only the necessary pieces from diverse components and their subcomponents. By dividing the intelligence between both the meta-information provided by the unit developer and the tool that reads it, we ensure that entity needs to be neither too intelligent nor too complex. We intend to use the same philosophy of dividing the intelligence between meta-information encoding and the tool that uses the information for configuration in the next stage of architecture development to build an orchestration system to manage data movement between various devices within a compute node, whether these devices are accelerators or different flavors of memory.

Acknowledgements This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security. The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up non-exclusive, 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. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan, http://energy.gov/downloads/doe-public-access-plan.

References

1. Dubey A, Antypas K, Coon E, Riley K. Software process for multiphysics multicomponent codes. In: Carver J, Hong NC, Thiruvathukal G, editors. Software engineering for science. London: Taylor & Francis Group; 2016.
2. MacNeice P, Olson K, Mobarry C, de Fainchtein R, Packer C. PARAMESH: a parallel adaptive mesh refinement community toolkit. Comput Phys Commun. 2000;126(3):330–54.
3. Fryxell B, Müller E, Arnett D. Numerical methods in astrophysics. New York: Academic; 1989. p. 100.
4. Timmes F. Integration of nuclear reaction networks. Astrophys J Suppl Ser. 1999;124:241–63.
5. Dubey A, Antypas K, Calder A, Fryxell B, Lamb D, Ricker P, Reid L, Riley K, Rosner R, Siegel A, Timmes F, Vladimirova N, Weide K. The software development process of FLASH, a multiphysics simulation code. In: Proceedings of the 5th international

workshop on software engineering for computational science and engineering, San Francisco; 2013.
6. Dubey A, Antypas K, Calder A, Daley C, Fryxell B, Gallagher J, Lamb D, Lee D, Olson K, Reid L, Rich P, Ricker P, Riley K, Rosner R, Siegel A, Taylor N, Timmes F, Vladimirova N, Weide K, ZuHone J. Evolution of FLASH, a multiphysics scientific simulation code for high performance computing. Int J High Perform Comput Appl. 2013;28(2):225–37.
7. Dubey A, Antypas K, Ganapathy M, Reid L, Riley K, Sheeler D, Siegel A, Weide K. Extensible component based architecture for FLASH, a massively parallel, multiphysics simulation code. Parallel Comput. 2009;35:512–22.
8. Tourwe T, Mens T. A survey of software refactoring. IEEE Trans Softw Eng. 2004;30(2):126–39.
9. Mäntylä M, Lassenius C. Drivers for software refactoring decisions. In: ACM/IEEE international symposium on empirical software engineering, ISESE'06, New York; 2006.
10. Hannay JE, MacLeod C, Singer J, Langtangen HP, Pfahl D, Wilson G. How do scientists develop and use scientific software? In: 2009 ICSE workshop on software engineering for computational science and engineering; 2009.
11. Sletholt M, Hannay JE, Pfahl D, Langtangen HP. What do we know about scientific software development's agile practices? Comput Sci Eng. 2012;14(2):24–37.
12. Storer T. Bridging the chasm: a survey of software engineering practice in scientific programming. ACM Comput Surv. 2017;50(4):47:1–32.
13. Farhoodi R, Garousi V, Pfahl D, Sillito J. Development of scientific software: a systematic mapping, a bibliometrics study, and a paper repository. Int J Softw Eng Knowl Eng. 2013;23(4):463–506.
14. Overbey JL, Negara S, Johnson RE. Refactoring and the evolution of Fortran. In: ICSE workshop on software engineering for computational science and engineering; 2009.
15. Overbey J, Xanthos S, Johnson R, Foote B. Refactorings for Fortran and high-performance computing. In: The second international workshop on software engineering for high performance computing system applications; 2005.
16. Tinetti M, Méndez FG. Change-driven development for scientific software. J Supercomput. 2017;73(5):2229–57.
17. Sommerville I. Software engineering. 10th ed. Boston: Pearson; 2016.
18. Sillitt A, Succi G. Requirements engineering for agile methods. In: Engineering and managing software requirements, Berlin, Heidelberg, Springer; 2005.
19. Ruparelia NB. Software development lifecycle models. SIGSOFT Softw Eng Notes. 2010;35(3):8–13.
20. AMReX [Online]. https://amrex-codes.github.io/.
21. Messer O, Harris J, Parete-Koon S, Chertkow MA. Multicore and accelerator development for a leadership-class stellar astrophysics code. In: Proceedings of the 11th international conference on applied parallel and scientific computing, Helsinki, Finland; 2013.
22. Dubey A, Graves D: A design proposal for a next generation scientific software framework. In: HeteroPar, Vienna; 2015.
23. Dubey A, Weide K, Lee D, Bachan J, Daley C, Olofin S, Taylor N, Rich P, Reid LB. Ongoing verification of a multiphysics community code: FLASH. Softw Pract Exp. 2015;45(2):233–44.
24. O'Neal J, Weide K, Dubey A. Experience report: refactoring the mesh interface in FLASH, a multiphysics software. In: 2018 IEEE 14th international conference on e-Science (e-Science), Amsterdam; 2018.