Contents

Chapter 1

Software Process for Multiphysics Multicomponent Codes

1.1 Introduction

Computational science and engineering (CSE) communities develop complex applications to solve scientific and engineering challenges, but these communities have a mixed record of using software engineering best practices. Many codes developed by CSE communities adopt standard software practices when the size and complexity of an application becomes too unwieldy to continue without them [21]. The driving force behind adoption is usually the realization that without using software engineering practices, the development, the verification and the maintenance of applications can become intractable. As more codes cross the threshold into increasing complexity, software engineering processes are being adopted from practices derived outside of the scientific and engineering domain. State-of-the-art for software engineering practices in CSE codes often lags behind that in the commercial software space [40, 8, 25]. There are many reasons for it: lack of incentives, support and funding; reward system favoring scientific results over software development; limited understanding of how software engineering should be promoted to communities that have their own specific needs and sociology.

Some software engineering practices have been better accepted among the developers of scientific codes than others. The ones that are used quite often include repositories for code version control, licensing process, regular testing, documentation, release and distribution policies, and contribution policies [21, 13, 23]. Less accepted practices include code-review, code-deprecation, and adoption of development methodologies such as Agile [1]. Software best practices that may be very effective in commercial software development environments are not always suited for scientific environments partly because of sociology and partly because of technical challenges. Sociology manifests itself as suspicion of too rigid a process or not seeing the point of adopting a practice. The technical challenges arise from the nature of problems being addressed by these codes. For example, multiphysics and multicomponents codes that run on large High Performance Computing (HPC) platforms put a large premium on performance. It is our experience that good performance

is most often achieved by sacrificing some of the modularity in software architecture (i.e. [19]). Similarly lateral interactions in physics get in the way of encapsulations (See sections 1.3 and 1.4 for more examples and details).

This chapter elaborates on the above challenges and how they were addressed in FLASH and Amanzi, two codes with very different development timeframe and therefore very different development paths. FLASH, whose development began in the late 1990's, is among the first generation of codes that adopted a software process. This was in the era when the advantages of software engineering were almost unknown in the scientific world. Amanzi is from the "enlightened" era (by scientific software standards) where a minimal set of software practices are adopted by most code projects intending long term use. A study of software engineering of two codes from different eras of scientific software development highlight how these practices and the communities have evolved.

FLASH was originally designed for computational astrophysics. It has been almost continuously under production and development since 2000 with three major revisions. It has exploited an extensible framework to expand its reach and is now a community code for over half a dozen scientific communities. The adoption of software engineering practices has grown with each version change and expansion of capabilities. The adopted practices themselves have evolved to meet the needs of the developers at different stages of development. Amanzi, on the other hand, started in 2012 and has developed from the ground up in C++ using relatively modern software engineering practices. It still has one major target community, but is also designed with extensibility as an objective. There are many other similarities and some differences described later in the chapter. In particular, we address the issues related to software architecture and modularization, design of a testing regime, unique documentation needs and challenges, and the tension between intellectual property management and open science.

The next few sections outline the challenges that are either unique to, or are more dominant in scientific software than elsewhere. Section 1.2 describes the typical lifecycle of a scientific code, followed by domain specific technical challenges in section 1.3. This is followed by a description in section 1.4 of technical and sociological challenges posed by the institutions where such codes are typically developed. Section 1.5 is the case study of FLASH and Amanzi developments, and the last two sections provide some general observations and additional considerations for adapting the codes for the more challenging platforms expected in future.

1.2 Lifecycle

Scientific software is designed to model phenomena in the physical world. The term 'physical' used here includes chemical and biological systems since physical processes are underlying building blocks for those systems too. A phenomenon may be at a microscopic level, for example protein folding, or at extremely large scales, for example galaxy cluster mergers. In some applications multiple scales are modeled. The physical characteristics of the systems being studied are translated into mathematical models that are said to describe the essential features of the behavior of those systems. These equations are then discretized, and numerical algorithms are used to solve them. One or more parts of this process may themselves be subjects of active research. Therefore the simulation software development requires diverse expertise and adds many stages in the development and lifecycle described below that may not be encountered elsewhere.

1.2.1 Development Cycle

For scientific simulations, modeling begins with equations that describe the general class of behavior to be studied. For example, the Navier-Stokes equations describe the flow of compressible and incompressible fluids, and van der Waals equations describe interactions among molecules in a material. There may be more than one set of equations if all behaviors of interest are not adequately captured by one set. In translating the model from mathematical representation to computational representation two processes go on simultaneously, discretization and approximation. One can argue that discretization is, by definition, an approximation because it is in effect sampling continuous behavior where information is lost in the sampling interval. This loss manifests itself as error terms in the discretized equations, but error terms are not the only approximations. Depending upon the level of understanding of specific sub-phenomena, and available compute resources, scientists also use their judgement to make other approximations. Sometimes, to focus on a particular behavior, a term in an equation may be simplified or may be even completely dropped. At other times some physical details may be dropped from the model because they are not understood well enough by the scientists. Or the model itself may be an approximation.

The next stage in developing the code is finding appropriate numerical methods for each of the models. Sometimes existing methods can be used without modification, but more often, customization or new method development is needed. A method's applicability to the model may need to be validated if the method is new or significantly modified. Unless a reliable implementation is already available as a third party software (stand-alone or in a library), the method has to be implemented and verified. It is at this stage

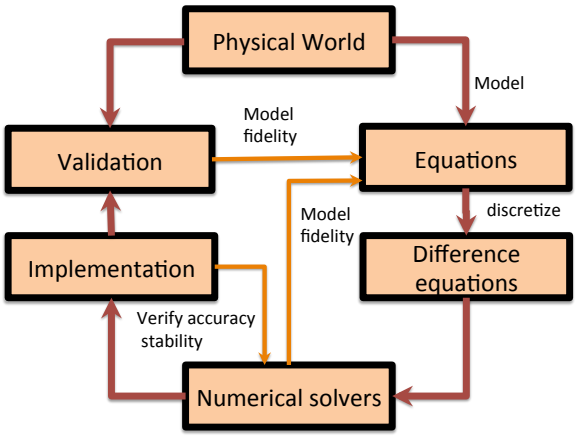

FIGURE 1.1: Development cycle of modeling with partial differential equations

that the development of a CSE code begins to resemble that of general software. The numerical algorithms are specified, the semantics are understood, and they need to be translated into executable code. Even then there are differences because scientific code developers work iteratively [39], requirement specifications evolve through the development cycle. Figure 1.1 gives an example of the development cycle of a multiphysics application modeled using partial differential equations.

1.2.2 Verification and Validation

The terms verification and validation are often used interchangeably, but to many scientific domains they have specific meaning. In their narrow definition, validation ensures that the mathematical model correctly defines the physical phenomena, while verification makes sure that the implementation of the model is correct. In other words, a model is validated against observations or experiments from the physical world, whereas a model is verified by other forms of testing [33]. Other definitions give broader scope to verification and validation (i.e. [37]). For example, validation of a numerical method may be constructed through code-to-code comparisons, and its order can be validated through convergence studies. Similarly, the implementation of a solver can be validated against an analytically obtained solution for some model if the same solver can be applied and the analytical solution is also known, though this is not always possible [32]. Irrespective of specific definitions, what is

true is that correctness must be assured at all the stages, from modeling to implementation.

There are many degrees of freedom in the process of deriving a model as discussed in the previous section, therefore, the validation of the model must be carefully calibrated by scientific experts. Similarly, verification of a numerical method requires applied math expertise, because, in addition to correctness the method needs verification of its stability, accuracy and order of convergence. Many numerical methods are themselves objects of ongoing research, so their implementation may need modifications from time to time. Whenever this happens, the entire gamut of verification and validation needs to be applied again. This is an instance of a particular challenge in CSE software where no amount of specification is enough to hand the implementation over to software engineers or developers who do not have domain or method knowledge. A close collaboration with applied mathematicians and method developers is necessary because the process has to be iterative with scientific judgement applied at every iteration.

One other unique verification challenge in CSE software is caused by finite machine precision of floating point numbers. Any change in compilers, optimization levels, and even order of operations can cause numerical drift in the solutions [29]. Especially in applications that have a large range of scales, it can be extremely difficult to differentiate between a legitimate bug and a numerical drift [6]. Therefore, relying upon bitwise reproducibility of the solution is rarely a sufficient method for verifying the continued correctness of an application behavior. Robust diagnostics (such as statistics or conservation of physical quantities) need to be built into the verification process. This issue is discussed in greater detail in the chapter *Testing in Scientific Software* .

1.2.2.1 Testing

Testing of CSE software needs to reflect the layered complexity of the codes. The first line of attack is to develop unit tests that isolate testing of individual components. However, in CSE codes, often there are dependencies between different components of the code that can not be meaningfully isolated, making unit testing more difficult. In these cases, testing should be performed with a minimal possible combination of components. In effect, these minimally combined tests behave like unit tests because they focus on possible defects in a very narrow section of the code. In addition, multicomponent CSE software should test various permutations and combination of components in different ways. Configuring tests in this manner can help verify that the configurations of interest are within the accuracy and stability constraints (see section 1.5.1.2 for an example of test-suite configuration for FLASH).

1.2.3 Maintenance and Extensions

In a simplified view of software lifecycle, there is a design and development phase, followed by production and maintenance phase. Even in well engineered codes this simplified view typically applies only to the infrastructure and APIs which have a distinct development phase with limited spill into the remainder of the lifecycle. The numerical algorithms and solvers can be in a continually evolving state reflecting the advances in their respective fields. The development of CSE software is usually responding to an immediate scientific need, so the codes get employed in production as soon as a minimal set of computational modules necessary for even one scientific project are built. Similarly, the development of computational modules almost never stops through the code lifecycle because new findings in science and math almost continuously place new demands on them. The additions are mostly incremental when they incorporate new findings into an existing feature, they can also be substantial when new capabilities are added. The need for new capabilities may arise from greater model fidelity, or from trying to simulate a more complex model. Sometimes a code designed for one scientific field may have enough in common with another field that capabilities may be added to enable it for the new field [2].

Irrespective of the cause, co-existence of development and production/maintenance phases is a constant challenge to the code teams. It becomes acute when the code needs to undergo major version changes. The former can be managed with some repository discipline in the team coupled with a robust testing regime. The latter is a much bigger challenge where the plan has to concern itself with questions such as how much backward compatibility is suitable, how much code can go offline, and how to reconcile ongoing development in code sections that are substantially different between versions. FLASH's example in section 1.5.1.3 describes a couple of strategies that met the conflicting needs of developers and production users in both scenarios. Both required co-operation and buy-in from all the stakeholders to be successful.

1.2.4 Performance Portability

Performance portability is an important requirement of multiphysics CSE software. HPC machines are expensive and rare resources and in order to achieve high application performance, codes need to be optimized for the unique HPC architectures. However, typical lifecycle of a multiphysics application spans many generations of HPC systems which have a typical lifespan of about 4-5 years. Depending upon the size of the code, optimization for a specific target platform can take a significant fraction of the platform lifecycle, time when the code may not be available for science runs. Without careful planning and coordination, a large fraction of scientists' time could be lost in porting and optimizing a code for a new system. Developers have the choice

of adding machine specific optimizations or creating more general optimization that will work for a broader class of systems. CSE codes must consider the trade-offs and advantages of a highly optimized code for a single platform versus designing their software using constructs that perform modestly well across a range of platforms.

1.2.5 Using CSE Software

There is a fundamental requirement from the users of scientific software that rarely comes into play for users of other kinds of software. For obtaining valid scientific results the users must have a basic understanding of the phenomena being modeled and the approximations, and they must know which questions can be validly addressed by these models. For example, Eulerian equations of hydrodynamics do not account for viscosity. If one is modeling a fluid where viscosity is important, one should not use a code that uses Eulerian equations to model the fluid. Similarly, the users must know and understand the valid regimes for applicability of numerical algorithms, as well as their accuracy and stability behavior. For example a numerical method that can resolve a smooth fluid flow only will fail if there are discontinuities. Similarly, in order to use the Fast Fourier Transform (FFT) method, users must ensure that their sampling interval resolves all modes. Failure to do so may filter out important information and lead to wrong results. Additionally, sometimes equations have mathematically valid but physically invalid solutions. An inappropriately applied numerical scheme may converge to such a non-physical solution.

At best any of the above situations may lead to a waste of computing resources if the defect in the solution is detected. At worst they may lead to wrong scientific conclusions being drawn. Some in the scientific community even argue that those who have not written at least some basic version of their own code for their problem should not be using other's public or community code. Though that argument goes too far, it embodies a general belief in the scientific community that users of scientific codes have a great responsibility to know their tools and understand their capabilities and limitations. These are some of the reasons that also play a role in tendency of scientific codes to do strict gatekeeping for contributions.

1.3 Domain Challenges

Multiphysics codes, by their definition, solve more than one mathematical model. A typical code combines 3-4 diverse models, in extreme cases codes may employ many more. Rarely, all models work with the same discretization using similar algorithmic approach (for instance stencil computations in

explicit PDE solvers). Models with diverse discretizations and algorithms are more common. In such cases, each operator has its own preferred data layout and movement that may differ from those needed by other operators. Normally these challenges can be mitigated through encapsulation and well defined API's. The outer wrapper layers of the operators can carry out data transformations as needed. There are two factors against taking this approach in CSE codes: (1) physics is not always friendly to encapsulation, and (2) the codes are performance sensitive and wholesale data movement significantly degrades performance.

The CSE simulation codes model the physical world which does not have neat modularization. Various phenomena have tightly coupled dependencies that are hard to break. These dependencies and tight couplings are transferred into the mathematical models and hinder the elimination of lateral interactions among code modules. An attempt to force encapsulations by hoisting up the lateral interactions to the API level can explode the size of the API. And if not done carefully this can also lead to extra data movement. The module designs, therefore, have to be cognizant of potential lateral interactions and make allowances for them. Similarly, the data structures have to take into account the diverse demands placed on them by different operators and carefully consider the trade-offs during software design. Considerations such as these are not common in software outside of CSE.

When designing CSE software, it is especially important to create modular components wherever possible. This is because different expertise may be required to understand each component, and thus a modular design allows application developers to focus on the areas they know best. For example, the numerical algorithms associated with physics operators require mathematical expertise, which is different from the code architecture which requires software engineering expertise. In addition, modular code allows various components to interface with each other in a clearer way. Another challenge for CSE codes is that many of them run on large HPC systems and require code parallelization. Parallel codes typically must implement a parallel domain decomposition and manage synchronization and load balancing between tasks or threads. With appropriate code modularization that enables separation of concerns these different aspects of the code do not need to interfere with one another and can help make application development more tractable.

Multiphysics multiscale codes often require tight integration with third party software, which comes in the form of numerical libraries. Because multiphysics codes combine expertise from many domains, the numerical solvers they use also require diverse applied mathematics expertise. It can be challenging for any one team to assemble all the necessary expertise to develop their own software and so many turn to third party math libraries for highly optimized routines. However, as mentioned in section 1.2.5, the use of third party software does not absolve them from understanding its appropriate use. Additionally, information about appropriate use of third party software within

the context of a larger code must also be communicated to the users of the code.

1.4 Institutional Challenges

Many adaptations in software engineering for CSE applications described in the previous section pertained to software design and testing. A number of challenges also arise because of the kind of organizations and the research communities where these codes are developed. The most crippling and pervasive challenge faced by CSE codes in general, and multiphysics codes in particular, is that funding for software development and maintenance is difficult to attain. There is evidence that when software is designed well it pays huge dividends in scientific productivity from the small number of projects that secured such funding for software infrastructure design. Examples include community codes such as NAMD [36], Amber [14] and Enzo [41] which are used by significant number of users in their respective communities. More persuasive case can be made by a handful of codes such as FLASH [17, 22], Cactus [11] and Uintah [35, 10] that were built for one community, but have since expanded their capabilities to serve several other communities using a common infrastructure. Even with this evidence it remains difficult to obtain funding for investment in software engineering best practices. Available funding is most often carved out of scientific goal oriented projects that have their own priorities and time-line. This model often ends up short-changing the software engineering.

The scientific output of applications is measured in terms of publications, which in turn depend upon the data produced by the simulations. Therefore, in a project driven purely by scientific objectives, the short-term science goals can lead to situations where quick-and-dirty triumphs over long term planning and design. The cost of future lost productivity may not be appreciated until much later when code base has grown too large to remove the deficiencies in any easy way. Software engineering is forcibly imposed on the code, which is at best a band-aid solution. This is another reason why many of the software practices are not embraced by the CSE community.

Another institutional challenge in developing good software engineering practices for CSE codes is training students and staff to use the application properly. Multiphysics codes require a broad range of expertise in domain science from their developers, and software engineering skills is an added requirement. Often experts in a domain science who develop CSE codes are not trained in software engineering and many learn skills on the job through reading, or talking to colleagues. Practices are applied as they understand them, usually picking only what is of most importance for their own development. This can be both good and bad. Good because it sifts out the unnecessary aspects of SE practice, and bad because it is not always true that the sifted

out aspects were really not necessary. It just means that the person adopting the practice did not understand the usefulness and impact of those aspects.

Institutional challenges also arise from scarcity and stability of resources. The domain and numerical algorithmic expertise is rarely replicated in a team developing the multiphysics CSE application. Even otherwise, deep expertise in the domain may be needed to model the phenomenon right, and that kind of expertise is relatively rare. Then there is the challenge of communicating the model to the software engineer, if there is one on the team, or to team members with some other domain expertise. It requires at least a few developers in the team who can act as interpreters for various domain expertise and are able to integrate them. Such abilities take a lot of time and effort to develop, neither of which are easy in academic institutions where these codes are typically organically grown. The available human resources in these institutions are post-docs and students who move on, so there is little retention of institutional knowledge about the code. A few projects that do see the need for software professionals struggle to find ways of funding them or providing a path for their professional growth.

The above institutional challenges are among the reasons why it is hard and often even undesirable to adopt any set software development methodology in CSE application projects. For example, the principles behind the agile manifesto apply, but not all the formalized processes do. Agile software methods [1] are lightweight evolutionary development methods with focus on adaptability and flexibility, as opposed to waterfall methods which are sequential development processes where progress is perceived as a downward flow [4]. Agile methods aim to deliver working software as early as possible within the lifecycle and improve it based upon user feedback and changing needs. These aims fit well with the objectives of scientific software development as well. These codes are developed by interdisciplinary teams where interactions and collaborations are preferred over regimented process. The code is simultaneously developed and used for science, so that when requirements change there is quick feedback. For the same reason, the code needs to be in working condition almost all the time. However, scarcity of resources does not allow the professional roles in the agile process to be played out efficiently. There is no clear separation between the developer and the client, many developers of the code are also scientists who use it for their research. Because software development goes hand-in-hand with research and exploration of algorithms, it is impossible to do either within fixed time-frames. This constraint effectively eliminates using agile methods such as *sprints* [1]. Similarly, extreme programming, another component of Agile, is impossible to implement in an environment which has to incorporate research into the process. The waterfall model is even less useful because it is not cost-effective or even possible to have a full specification ahead of time. The code has to grow and alter organically as the scientific understanding grows, the effect of using technologies are digested and requirements change. A reasonable solution is to adopt those elements of the methodologies that match the needs and objectives of

the team, adjust them where needed, and develop their own processes and methodologies where none of the available options apply.

Due to the need for deep expertise, and the fact that the developer of a complex physics module is almost definitely going to leave with possibly no replacement, documentation of various kind takes on a crucial role. It becomes necessary to document the algorithm, the implementation choices, and the range of operation. The generally preferred practice of writing self explanatory code helps, but does not suffice. To an expert in the field, who has comprehensive understanding of the underlying math, such a code might be accessible without inline documentation. But not to non-experts (i.e. from another field or a software engineer in the team if there is one) who may have reasons to look at the code. For longevity and extensibility, a scientific code must have inline documentation explaining the implementation logic, and reasons behind the choices made.

1.5 Case Studies

1.5.1 FLASH

The FLASH code [5, 17] has been under development for nearly two decades at the Flash Center at the University of Chicago. The code was originally developed to simulate thermonuclear runaways in astrophysics such as novae and supernova. It was created out of an amalgamation of three legacy codes, Prometheus for shock hydrodynamics, PARAMESH for adaptive mesh refinement (AMR) and locally developed equation of state and nuclear burn code. It has slowly evolved into a well-architected extensible software with a user base in over half a dozen scientific communities. FLASH has been applied to a variety of problems including supernovae, X-ray bursts, galaxy clusters, and stellar structure, fluid instabilities, turbulence, laser-experiments design and analysis and nuclear reactor rods. It supports an Eulerian mesh combined with a Lagrangian framework to cover a large class of applications. Physics capabilities include compressible hydrodynamics and magnetohydrodynamics solvers, nuclear burning, various forms of equations of state, radiation, laser drive, fluid-structure interactions and many more.

1.5.1.1 Code Design

From the outset FLASH was required to have composability because the simulations of interest needed capabilities in different permutations and combinations. For example, most simulations needed compressible hydrodynamics, but with different equations of state. Some needed to include self-gravity while others did not. An obvious solution was to use object-oriented programming model with common APIs and specializations to account for the different mod-

els. However, the physics capabilities were mostly legacy with F77 implementations. Rewriting the code in an object oriented language was not an option. A compromise was found by exploiting the unix directory structure for inheritance, where, for a code unit the top level directory defined the API and the subdirectories contained the multiple alternative implementations of the API. Meta-information about the role of a particular directory level in the object oriented framework was encoded in a very limited domain-specific language (configuration DSL). The meta-information also included state and runtime variables requirements, dependences on other code units etc. A "setup tool" parsed this information to configure a consistent "application". The setup tool also interpreted the configuration DSL to implement inheritance using the directory structure. For more details about FLASH's object oriented framework see [17].

FLASH design is aware of the need for separation of concerns and achieves it by separating the infrastructural components from physics. The abstraction that permits this approach is very well known in CSE, that of decomposing a physical domain into rectangular blocks surrounded by halo cells copied over from the surrounding neighboring blocks. To a physics operator, the whole domain is not distinguishable from a box. Another necessary aspect of the abstraction is not to let any of the physics modules own the state variables. They are owned by the infrastructure that decomposes the domain into blocks. A further separation of concern takes place within the units handling the infrastructure, that of isolating parallelism from the bulk of the code. Parallel operations such as ghost cell fill, refluxing or regridding have minimal interleaving with state update in the blocks from application of physics operators. To distance the solvers from their parallel constructs, the required parallel operations provide an API with corresponding functions implemented as a subunit. The implementation of numerical algorithms for physics operators is sequential, interspersed with access to the parallel API as needed.

Minimization of data movement is achieved by letting the state be completely owned by the infrastructure modules. The dominant infrastructure module is the *Eulerian* mesh, owned and managed by the *Grid* unit. The physics modules query the *Grid* unit for the bounds and extent of the block they are operating on, and get a pointer to the physical data. This arrangement works in most cases, but gets tricky where the data access pattern does not conform to the underlying mesh. An example is any physics dealing with Lagrangian entities (LEs). They need a different data structure, and the data movement is dissimilar from that of the mesh. Additionally, the LEs interact with the mesh, so maintaining physical proximity of the corresponding mesh cell is important in their distribution. This is an example of unavoidable lateral interaction between modules. In order to advance, LEs need to get field quantities from the mesh and then determine their new locations internally. They may need to apply near- and far-field forces, or pass some information along to the mesh, or redistributed after advancing in time. FLASH solves this conundrum through keeping the LE data structure extremely simple, and

using argument passing by reference in the APIs. The LEs are attached to the block in the mesh that has the overlapping cell, an LE leaves its block when its location no longer overlaps with the block. Migration to a new block is an independent operation from everything else that happens to the LEs. In FLASH parlance this is the Lagrangian framework (see [20] for more details). The combination of *Eulerian* and *Lagrangian* frameworks that interoperate well with one another has succeeded in largely meeting the performance critical data management needs of the code.

1.5.1.2 Verification & Validation

FLASH instituted a rigorous verification program early in its lifecycle. The earliest versions of FLASH were subjected to a battery of standard hydrodynamics verification tests [24]. These verification tests were then used to set up an automated regression test suite run on a few local workstations. Since then the test suite has evolved into a combination of variety of tests that aim to provide comprehensive coverage for verifying correct operation of the code [18, 12]. Because FLASH is in a constant state of production and development, verification of its correctness on a regular basis is a critical requirement. The testing is complicated both by the variety of environments in which FLASH is run, and the sheer number of ways in which it can be configured.

Testing is an area where the standard practices do not adequately meet the needs of the code. Many multiphysics codes have legacy components in them that are written in early versions of Fortran. Contrary to popular belief, a great deal of new development continues in Fortran because it still is the best HPC language in which to express mathematical algorithms. All of solver code in FLASH is written in F90, so popular unit test harnesses aren't available for use. Small scale unit tests can only be devised for infrastructural code because all the physics has to interact with the mesh. Also, because regular testing became a part of FLASH development process long before formal incorporation of software engineering practices in the development process, FLASH's designation of tests only loosely follows the standard definitions. So a unit test in FLASH can rely on other parts of the code, as long as the feature being tested is isolated. For example testing for correct filling of halo cells uses a lot of AMR code that has little to do with the halo filling, but it is termed unit test in FLASH parlance because it exclusively tests a single limited functionality, we refer to them as *FLASH-unit-test* from here on. The dominant form of regular testing is integration testing, where more than one code capability is combined to configure an executable. The results of the run are compared against pre-approved results to verify that changes are within a specified acceptable range. Because of a large space of possible valid and useful combinations, selection of tests is challenging. FLASH's methodology for test design and selection is described below, for more details see [6].

FLASH's testsuite consists of about a dozen FLASH-unit-tests and over 80 multi-purpose composite tests. A composite test runs in two stages, and

has a comparison *benchmark* for each stage. A *benchmark* in FLASH testsuite is a full state checkpoint. Composite test benchmarks are checkpoints at two distinct timesteps M and N where $M < N$. Both are analyzed and approved as producing expected results by a domain expert, if there is a legitimate cause for change in the results (i.e. and improved method) benchmarks have to be re-approved. The first stage of a composite test verifies that no errors have been introduced into the covered code units and their interactions covered by comparing against the benchmark at time M. The second stage restarts the execution from checkpoint at M and runs up to time N. This stage of the test verifies that the covered code can correctly start from a checkpoint without any loss of state information. FLASH testsuite configures and builds each test every time it is run, therefore, build and configuration tests are built-in. Progress of a test is reported at every stage, the final stage being the outcome of the comparison. Because FLASH has many available application configuration whose union provides very good code coverage, the task of building a test suite is simplified to selection among existing applications. In many applications the real challenge is picking parameters that exercise the targeted features without running for too long. As mentioned earlier a matrix is used to ensure maximum coverage. For each selected test all the covered features are marked off in the matrix. Following order is used for filling the matrix:

- unit tests
- setups used in science production runs
- setups known to be sensitive to perturbations
- simplest and fastest setups that provide the remaining coverage.

FLASH's testing can be broadly classified into three categories: the daily testing, described above, to verify ongoing correctness of the code, more targeted testing related to science production runs, and finally porting to and testing on new platforms. Daily testing is performed on multiple combinations of platforms and software stacks. In preparing for a production schedule, testing is a combination of scaling tests, cost estimation tests, and looking for potential trouble spots. Scientists and developers work closely to devise meaningful weak scaling tests (which can be difficult because of non-linearity and adaptive mesh refinement), and tests that can exercise the vulnerable code sections without overwhelming the test suite resources. Sample smaller scale production runs are also performed on the target platform to help make informed estimates of cpu hours and disk space needed to complete the simulation. For more details on simulation planning see [18]. For porting the code to a new platform a successful production run from the past is used as a benchmark for exercising the code on a new platform, along with using a subset of the standard test suite.

FLASH has had some opportunities for validation against experiments.

For example FLASH could model a variety of laboratory experiments involving fluid instabilities [16, 27]. These efforts allowed researchers to probe the validity of models and code modules, and also served to bolster the experimental efforts by creating realistic simulation capabilities for use in experimental design. The newer high-energy density physics (HEDP) initiative involving FLASH is directed at simulation-based validation and design of experiments at the major laser facilities in the US and Europe. Other forms of validation have been convergence tests for the flame model that is used for supernova simulations, and validation of various numerical algorithms against analytical solutions of some known problems. For example, the Sedov [38] problem, which is seeded by a pressure spike in the center that sends out a spherical shock-wave into the domain has a known analytical solution. It is used to validate hydrodynamics in the code. There are several other similar examples where a simple problem can help to validate a code capability through known analytical solutions.

1.5.1.3 Software Process

The software process of FLASH has evolved organically with the growth of the code. For instance, in the first version there was no clear design document, the second version had a loosely implied design guidance, whereas the third version documented the whole design process. The third version also published the developer's guide which is a straight adaptation from the design document. Because of multiple developers with different production targets, versioning repository was introduced early in the code life cycle. The repository used has been SVN since 2003, though its branching system has been used in some very unorthodox ways to meet peculiar needs of the Flash Center. Unlike most software projects where branches are kept for somewhat isolated development purposes, FLASH uses branches also to manage multiple ongoing projects. This particular need arose when there were four different streams of physics capabilities being added to the code. All projects needed some code from the trunk, but the code being added was mostly exclusive to the individual project. It was important that the branches stay more or less in sync with the trunk and that the new code be tested regularly. This was accomplished by turning the trunk into essentially a merge area, with a schedule of merge from individual branches, and an intermediate branch for forward merge. The path was tagged-trunk => forward-branch => projects => merge into trunk => tag trunk when stabilized. Note that the forward branch was never allowed a backward merge to avoid the possible inadvertent breaking of code for one project by another one. For the same reason the project branches never did a forward merge directly from the trunk.

One of the biggest challenges in managing a code like FLASH occurs during major version changes, when the infrastructure of the code undergoes deep changes. FLASH has undergone two such changes where the first transition took the approach of keeping the development branch synchronized with the

main branch at all times. An effort was made to keep new version backward compatible with the old version. During and after the process the team realized many shortcomings of this approach. One was that the code needed to have deeper structural changes than were possible under this approach. Also, the attempt to keep the development and production versions in sync placed undue demands on the developers of the new version, leading to inefficient use of their time. The adoption of the new version was delayed because keeping up with the ongoing modifications to the older version (needed by the scientists to do their work) turned the completion of the transition into a moving target.

Because of these lessons learned the second transition took a completely different approach and was much more successful. The infrastructural backbone/framework for the new version was built in isolation from the old version in a new repository. The framework design leveraged the knowledge gained by the developers about the idiosyncracies of the solvers in earlier versions and focussed on the needs of the future version. There was no attempt at backward compatibility with the framework of the previous version. Once the framework was thoroughly tested, physics modules were transitioned. Here the emphasis was on transitioning all the capabilities needed for one project at the same time, starting with the most stable modules. Once a module was moved to the new version it was effectively frozen in the old version (the reason for selecting the most stable and mature code sections). Any modification after that point had to be made simultaneously in the new version as well. Though it sounds like a lot of duplicate effort, in reality such instances were rare. This version transition was adopted by the scientists very quickly.

FLASH's documentation takes a comprehensive approach with a user's guide, a developer's guide, robodoc API, inline documentation, and online resources. Each type of documentation serves a different purpose and is indispensable to the developers and users of the code. There are scripts in place that look for violations of coding standards and documentation requirements. The user's guide documents the mathematical formulation, algorithms used and instructions on using various code components. It also includes examples of relevant applications explaining the use of each code module. The developer's guide specifies the design principles and coding standards with an extensive example of the module architecture. Each function in the API is required to have a robodoc header explaining the input/output, function and special features of the function. Except for the third party software, every non-trivial function in the code is required to have sufficient inline documentation so that a non-expert can understand and maintain the code.

FLASH effectively has two versions of release - internal, which is close to the agile model, and general, which is no more than twice a year. The internal release amounts to tagging a stable version in the repository for the internal users of the code. This signals to the users that a forward merge into their production branch is not going to break the code. The general releases have a more rigorous process which makes them infrequent. The general releases undergo some amount of code pruning, get checked for compliance with coding and

documentation standards and meet stringent requirements from the testing process. They are expensive in terms of developers resources. The dual model ensures that the quality of code and documentation are maintained without unduly straining the team resources, while near continuous code improvement is still possible for ongoing projects.

1.5.1.4 Policies

In any project, policies regarding attributions, contributions and licensing matter. In CSE arena intellectual property rights, and interdisciplinary interactions are additional policy areas that are equally important. Some of these policy requirements are a direct consequence of the strong gatekeeping regimes that majority of CSE publicly distributed software follow. Many arguments are forwarded for dominance of this model in the domain, the most compelling one relates to maintaining the quality of software. Recollect that the developers in this domain are typically not trained in software engineering, and software quality control varies greatly between individuals and/or groups of developers. Because of tight, and sometimes lateral, coupling between functionalities of code modules, a lower quality component introduced into the code base can have disproportionate impact on the overall reliability of output produced by the code. Strong gate-keeping is desirable, and that implies having policies in place for accepting contributions. FLASH again differentiates between internal and external contributors in this regard. The internal contributors are required to meet the quality requirements such as coding standards, documentation, and code verification in all of their development. Internal audit processes minimize the possibility of poorly written and tested code from getting into a release. The internal audit also goes through a periodic pruning to ensure that bad or redundant code gets eliminated.

The external contributors are required to work with a member of the internal team to include their code in the released version. The minimum set required from them is: (1) code that meets coding standards, has been used or will be used for results reported in peer-reviewed publication, (2) at least one test that can be included in the test-suite for nightly testing, (3) documentation for user's guide, robodoc documentation for any API functions and inline documentation explaining the flow of the control, and finally (4) a commitment to answer questions on users mailing list. The contributors can negotiate the terms of release, a code section can be excluded from the release for a mutually agreed period of time to enable the contributor to complete their research and publish their work before the code becomes public. This policy permits the potential contributors to be freed from the necessity of maintaining their code independently, while still retaining control over their software until agreed upon release time. As a useful side effect their code remains in sync with the developments in the main branch between releases.

There is another model of external contribution to FLASH that is without any intervention from the core gate-keeping team. In this model anyone can

stage any FLASH compatible code on a site hosted by them. The code has no endorsement from the distributing entity, the Flash Center, which does not take any responsibility for its quality. The Flash Center maintains a list of externally hosted 'as-is' code sites, the support for these code sections is entirely the responsibility of hosting site.

The attribution practices in CSE are somewhat ad-hoc. For many developers, the only metric of importance is scientific publications that result from using the software. When a team is dominated by such developers proper attribution for code development is not given enough importance or thought. Other teams also employ computing professionals whose career growth depends upon their software artifacts, and publications describing their algorithms and artifacts, FLASH falls into the latter category. All contributors' names are included in the author list for the user's guide, the release notes explicitly mention new external contributions and their contributors, if any, for that release. Internal contributors rely upon software related publications for their attribution. This policy usually works well, one exception has been citations skewed in favor of early developers. Users of FLASH cite a paper published in 2000 [24] which does not include any of the later code contributors in its author list. So even though the current incarnation of FLASH has very little of the original code, the original authors continue to get the benefit while the later authors are deprived of legitimate citations for their work. Many major long running software projects have this problem, which is peculiar to the academic world where these codes reside and are used.

1.5.2 Amanzi/ATS

Amanzi and its sister code the Advanced Terrestrial Simulator (ATS), provide a good contrasting example to FLASH. Developed starting in 2012 as the simulation capability for the Department of Energy's Environmental Management program, Amanzi solves equations for flow and reactive transport in porous media, with intended applications of environmental remediation for contaminated sites [30]. Built on Amanzi's infrastructure, ATS adds physics capability to solve equations for ecosystem hydrology, including surface/subsurface hydrology, energy and freeze/thaw cycles, surface energy balance and snow, and vegetation modeling [34, 7]. Amanzi was initially supported by a development team of several people with dedicated development money. ATS was largely developed by one person, post-docs, and a growing set of collaborators from the broader community, and was supported by projects whose deliverables are ecosystem hydrology papers.

Amanzi/ATS's history makes it a good contrast to FLASH. Developed from the ground up in C++ using relatively modern software engineering practices, there are few legacy code issues. Unlike FLASH, Amanzi/ATS makes extensive use of "third party libraries", with associated advantages and disadvantages (currently Amanzi/ATS uses nearly 10k lines of cmake to build it and its libraries). However, they also share a lot of commonalities. Like

FLASH, version control has played a critical role in the development process, especially as developers are spread across multiple physical locations and networks. Like FLASH, Amanzi/ATS makes extensive use of module-level and regression-level testing to ensure correctness and enable refactoring. And like FLASH, Amanzi/ATS has found the open source strategy to be incredibly useful; in particular, the open source nature of the code has eliminated some of the natural competition between research groups at different DOE national laboratories and helped establish a growing community of users and developers.

1.5.2.1 Multiphysics management through Arcos

Recognizing early the wide variety of multiphysics applications that would be targeted with Amanzi/ATS, a formal multiphysics framework was designed, implemented, and adopted. This framework, later named Arcos [15], consists of three main components: a *process tree*, a *dependency graph*, and a *state/data manager*.

The process tree describes the hierarchical coupling between equations and systems of equations to be solved. Each leaf node of the tree is a single (partial) differential equation, such as conservation of mass. Each interior node of the tree couples the children below it into a system of equations. Every node presents a common interface to the nodes above it. Much of the coupling of internal nodes can be automated using this interface – sequential couplers can be fully automated, while globally implicit coupled schemes can be nearly automated (with off-diagonal blocks of preconditioners and globalization of nonlinear solvers the lone exceptions). This representation of multiphysics models is natural to the coupled physical system, and implicitly exists in most codes; Arcos makes this explicit while providing hooks for customization to the specific system.

A second view of the system of equations is stored in the dependency graph, much like that of [31]. The dependency graph is a directed, acyclic graph (DAG) where each node is a variable (either primary, secondary, or independent), and each edge indicates a dependency. The graph is built up from the leafs of the DAG, which are primary variables (those variables to be solved for) and independent variables (data provided to the model). Roots of the DAG are, for instance, corrections to the primary variable formed through a nonlinear solve, or time derivatives of primary variables used in time integrators. Between these, each interior node is a secondary variable, and consists of both data and an *evaluator*, or small, stateless (functional) unit of code that stores the physics or numerics of how to evaluate the variable given its dependencies.

Finally, a state object is a glorified container used to manage data. Through the state's interface, *const* access is allowed to all evalutors, and non-*const* access is strictly limited to the evaluator of that variable.

This framework, while at times seeming heavy-handed, results in several

important implications from a software engineering perspective. Here we focus on the dependency graph as used in Amanzi/ATS, and how it encourages and enables good software engineering practices.

1.5.2.2 Code Re-use and Extensibility

The first observation of this framework is that it results in extremely fine-grained modularity for physical equations. Most computational physics codes are modular at the level of equations; Amanzi/ATS is modular at the level of terms in the equation.

An example illustrates the usefulness of this in multiphysics applications. In a thermal hydrology code, the liquid saturation is a variable describing the volume fraction of pore space that is water. This is a secondary variable, and is a function of either liquid pressure (in isothermal cases) or of liquid pressure and temperature (in non-isothermal cases). By explicitly forming a dependency graph, terms that depend upon the liquid saturation need not know whether the model is isothermal or not. Furthermore, there is no concern of "order of equation evaluation" that is common in other multiphysics codes. As both the energy and mass conservation equations need liquid saturation, this model should be evaluated when it is first needed, but is likely not necessarily evaluated in both equations, as its dependencies have not changed. Optimization of evaluating a model only when its dependencies have changed results in tightly coupled, monolithic physics implementations. Often codes will have "modes" which reimplement the same mass conservation equation twice, once to support the isothermal case and once to support the non-isothermal case as coupled to an energy conservation equation.

By storing these dependencies in an explicit graph, the framework can keep track of when dependencies have changed, and lazily evaluate models exactly when needed. In tightly coupled multiphysics, a dependency graph eliminates the need for an omnipotent programmer to carefully orchestrate when and where each equation term is evaluated. As a result, a dependency graph eliminates code duplication and discourages monolithic physics code.

Furthermore, this strategy greatly improves code extensibility. Many variations in model structure are easily implemented by writing a new evaluator and using it in an existing conservation equation. Once past an initial developer learning curve, Amanzi/ATS developers are able to quickly adapt the code to new models.

1.5.2.3 Testing

Testing is an extremely sensitive subject in computational software engineering – so much so that it merits its own chapter: **??**. Few CSE codes are sufficiently tested by convential Software Engineering (SE) standards, and many CSE developers are aware of the shortcoming. As discussed above, frequently CSE codes are limited to component-level tests, as it can be difficult to write sufficiently fine-grained unit tests. SE techniques such as mocking ob-

jects are almost never practiced, as mocked objects would require nearly all of the same functionality of the real object in order to properly test the physics component. The claim is that most physics code cannot be tested without including discretizations, solvers, meshes, and other components.

This viewpoint, however, is somewhat narrow. Physics at the level of a differential equation cannot be tested at the granularity of a unit test. Physics at the level of a term within an equation, however, is much easier to test. By forcing componentization at the level of an evaluator, Amanzi/ATS allows a large portion of its physics implementation to be unit tested. Evaluators (and their derivatives) are stateless, and therefore can be tested without additional components. Pushing the majority of the physics implementations into evaluators and out of monolithic, equation-level classes greatly improves the code coverage of fine-grained unit tests. Amanzi/ATS still makes extensive use of medium-grained component tests for discretizations, solvers, etc, but a significantly large portion of physics testing is done at the unit-test granularity.

Amanzi/ATS additionally maintains a large list of coarse-grained system-level tests. These test the full capability, and serve additionally as documentation and example problems for new users. This strategy, of providing large suites of sample problems for both testing and documentation has become extremely common in the CSE community, and is widely considered a CSE best practice. By greatly lowering the bar for new users, this collection of dual-purpose examples encourages community; in ATS, each new physics contribution must be accompanied by a new system-level test for inclusion in these test suites.

In Amanzi/ATS, unit and component granularity tests are automated using ctest and run sufficiently fast to be run prior to every commit. While Amanzi/ATS does not practice true continuous integration, it is expected that all developers run this testsuite prior to commiting to the main repository.

1.5.2.4 Performance Portability

Amanzi/ATS was designed from the ground up with awareness of ongoing, disruptive architecture changes. Performance portability in the face of these changes is an issue that is and will continue to confront all codes. As such, Amanzi/ATS takes several strides to buffer itself from disruptive change.

First, by leveraging popular, supported, open source libraries with significant community, Amanzi/ATS is able to immediately leverage advances in these codes. For instance, a significant portion of time is spent in inverting a preconditioner using approximate linear solvers. By using a common interface and a wide variety of existing solver libraries through Trilinos, Amanzi/ATS is able to immediately leverage advances in any one of these libraries.

Next, by encouraging overdecomposition of physics into smaller, more heterogeneous units, Amanzi/ATS is set up to leverage task-based programming models with novel runtime systems. While not currently implemented, using

one of the several emerging "coarse task" runtime environments [9, 28, 26] is an exciting research area for the community.

Finally, Arcos's evaluators are a functional programming concept, and are stateless functors with no side effects. As a result, they abstract "what is evaluated" from "on what data and chip is it evaluated." This design decision allows the implementation of multiple versions of the latter (i.e. GPU, many-core) without touching any of the former code.

1.6 Generalization

Not all of the solutions described in the earlier sections for CSE specific challenges are generalizable to all scientific software, but the vast majority of them are. This is borne out by the fact that at a workshop on community codes in 2012 [21], all represented codes had nearly identical stories to tell about their motivation for adopting software engineering practices and the ones that they adopted. This was true irrespective of the science domains these codes served, the algorithms and discretization methods they used and communities they represented. Even their driving design principles were similar at the fundamental level though the details differed. The codes represented state-of-the-art in their respective communities in terms of both model and algorithmic research incorporated and the software engineering practices. Note that these are the codes that have stood the test of time and won the respect in their respective communities. They are widely used and supported, and have more credibility for producing reproducible reliable results than smaller individualistic efforts. Therefore, it is worthwhile to discuss those practices in this chapter. At a minimum they provide a snapshot of the state of large scale computing and its dependence on software engineering in the era of relatively uniform computing platforms.

One practice that is universally adopted by all community codes and other large scale codes is versioning repositories. That is worthy of mention here because even this practice has not penetrated the whole computational science community. There are many small projects that still do not use versioning, though their number is steadily decreasing. Other common practice is that of licensing for public use and many of the codes are freely available to download along with their source. Testing is also universal, though the extent and methodologies for testing vary greatly. A general verification and validation regime is still relatively rare, though regression testing is more common. Unit tests are less common than integration tests and bounded-change tests. Almost all codes have user level documentation and user support practices in place. They also have well defined code contribution policies.

Another feature that stands out is similarity in high level design of all multiphysics codes. Every code exercises separation of concerns between math-

ematical and structural parts and between sequential and parallel parts. In almost all cases this separation is dictated by the need to reduce complexity for efforts needing specific expertise. Also, all the codes have basic backbone frameworks which orchestrate the data movement and ownership. This is usually driven by the need for maintenance and flexibility. And where it is realized well it provides extensibility - the ability to add more physics and therefore greater capabilities and fidelity in the models being computed. Majority of frameworks are component based with composability of some sort. This is because different models need different capability combinations. Most codes use self-describing IO libraries for their output to facilitate the use of generally available analysis and visualization tools.

The degree to which teams from vastly different scientific domains producing community codes have arrived at essentially similar solutions points to a possibility that seemingly diverse problems can have a uniform solution if they are trying to achieve similar objectives. For the codes highlighted in this section, the objectives were capabilities, extensibility, composability, reliability, portability and maintainability. They were achieved through design choices conscious of trade-offs, most often with raw performance that individual components or specific platforms were capable of. The lesson here is that similar objectives can yield a general solution even if there is great diversity in the details of the individual problem. It is not beyond the realm of possibility that similar generalized solution will emerge for the next generation software faced with heterogeneous computing described in the next section.

1.7 Additional Future Considerations

An aspect of software design that is a unique requirement of the CSE domain is fast becoming a great challenge - performance portability. In the past, machine architectures were fairly uniform across the board for large stretches of time. The first set of effective HPC machines in routine use for scientific computing were all vectors machines. They later gave way to parallel machines with *risc* processor as their main processing element. A code written for one machine of its time, if portable, would have reasonable performance on most of its contemporary machines. The abstract machine model to which the codes of the era were programming was essentially the same for all machines of that era. It is true that wholesale changes had to occur in codes for transitioning from vector to risc-parallel machines, but it was a transition from one long-term stable paradigm to another long-term stable paradigm. In addition, the codes were not as large as the multiphysics codes of today. So although the transitions took time, the codes that adapted well to the prevailing machine model thrived for several years.

The computing landscape is undergoing significant changes. Now there

are machines in the pipeline that have deep enough architectural differences among them that one machine model cannot necessarily describe their behavior. A look at the top supercomputers in the world shows a variety of architectures from accelerator models, to many-core systems. Even though many different vendors are moving to architectures with lighter and smaller cores, the different cache and memory hierarchies on these systems make portability across architectures difficult. In addition, the lack of a high performing, common programming model across architectures poses an even greater challenge for application developers. And, because the codes are significantly larger than they were during the last architecture paradigm shift, the transition will be even more challenging. More importantly, some aspects of the challenges are not unique to the large multiphysics codes. Because the deep architectural changes are occurring at the level of nodes that will go into all platforms, the change is ubiquitous and will affect everyone. Portability in general, and performance portability in particular, is an issue for everyone. At this writing the impact of this paradigm shift is not fully understood. Means of combating this challenge are understood even less. There is a general consensus that more programming abstractions are necessary not just for the extreme scale, but also for small scale computing. The unknown is which abstraction or combination of abstractions will deliver the solution. Many solutions have been proposed, for example [42] (also see [3] for a more comprehensive and updated list). Of these, some have undergone more testing and exercise under realistic application instances than others. Currently, no approach has been shown to provide a general solution that can be broadly applicable in the ways that optimizing compilers and MPI were in the past. This is an urgent serious challenge facing the CSE community today, future viability of CSE codes depends upon significant help from software engineering expertise and motivation within the community.

Bibliography

[1] Agile methodology. http://agilemethodology.org/.

[2] The dividends of investing in computational software design - a case study.

[3] Ideas productivity - howto documents. https://ideas-productivity.org/resources/howtos/.

[4] Waterfall model. https://www.techopedia.com/definition/14025/waterfall-model.

[5] The FLASH code. http://flash.uchicago.edu/flashcode, 2000.

[6] A. Dubey, K. Weide, D. Lee, J. Bachan, C. Daley, S. Olofin, N. Taylor, P.M. Rich, and L.B. Reid. Ongoing verification of a multiphysics community code: FLASH. *Software: Practice and Experience*, 45(2), 2015.

[7] A. L. Atchley, S. L. Painter, D. R. Harp, E. T. Coon, C. J. Wilson, A. K. Liljedahl, and V. E. Romanovsky. Using field observations to inform thermal hydrology models of permafrost dynamics with ATS (v0.83). *Geosci. Model Dev. Discuss.*, 8:3235–3292, 2015.

[8] Victor R Basili, Jeffrey C Carver, Daniela Cruzes, Lorin M Hochstein, Jeffrey K Hollingsworth, Forrest Shull, and Marvin V Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE software*, 25(4):29, 2008.

[9] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*, page 66. IEEE Computer Society Press, 2012.

[10] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C.A. Wight, and J.R. Peterson. Uintah - a scalable framework for hazard analysis. In *TG '10: Proc. of 2010 TeraGrid Conference*, New York, NY, USA, 2010. ACM.

[11] Marek Blazewicz, Ian Hinder, David M Koppelman, Steven R Brandt, Milosz Ciznicki, Michal Kierzynka, Frank Löffler, Erik Schnetter, and Jian Tao. From physics model to results: An optimizing framework for cross-architecture code generation. *Scientific Programming*.

[12] A. C. Calder. Laboratory astrophysics experiments for simulation code validation: A case study. *Astrophysics and Space Science*, 298:25–32, July 2005.

[13] Jeffrey C Carver. Software engineering for computational science and engineering. *Computing in Science & Engineering*, 14(2):8–11, 2012.

[14] DA Case, V Babin, Josh Berryman, RM Betz, Q Cai, DS Cerutti, TE Cheatham Iii, TA Darden, RE Duke, H Gohlke, et al. Amber 14. 2014.

[15] E. T. Coon, J. D. Moulton, and S. L. Painter. Managing complexity in simulations of land surface and near-surface processes. *Environmental Modelling & Software*, 78:134–49, 2016.

[16] G. Dimonte, D. L. Youngs, A. Dimits, S. Weber, M. Marinak, S. Wunsch, C. Garasi, A. Robinson, M. J. Andrews, P. Ramaprabhu, A. C. Calder, B. Fryxell, J. Biello, L. Dursi, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. Timmes, H. Tufo, Y.-N. Young, and M. Zingale. A comparative study of the turbulent Rayleigh-Taylor instability using high-resolution three-dimensional numerical simulations: The Alpha-Group collaboration. *Physics of Fluids*, 16:1668–1693, May 2004.

[17] A. Dubey, K. Antypas, M.K. Ganapathy, L.B. Reid, K. Riley, D. Sheeler, A. Siegel, and K. Weide. Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10-11):512–522, 2009.

[18] A. Dubey, A.C. Calder, C. Daley, R.T. Fisher, C. Graziani, G.C. Jordan, D.Q. Lamb, L.B. Reid, D. M. Townsley, and K. Weide. Pragmatic optimizations for better scientific utilization of large supercomputers. *International Journal of High Performance Computing Applications*, 27(3):360–373, 2013.

[19] A. Dubey and T. Clune. Optimization techniques for pseudospectral codes on MPPs. In *Proceedings of Frontiers 99*, 1999.

[20] A. Dubey, C. Daley, J. ZuHone, P. M. Ricker, K. Weide, and C. Graziani. Imposing a Lagrangian particle framework on an Eulerian hydrodynamics infrastructure in FLASH. *ApJ Supplement*, 201:27, aug 2012.

[21] A. Dubey, D.Q. Lamb, and E. Balaras. Building community codes for effective scientific research on HPC platforms. http://flash.uchicago.edu/cc2012, 2012.

[22] A. Dubey, L.B. Reid, and R. Fisher. Introduction to FLASH 3.0, with application to supersonic turbulence. *Physica Scripta*, T132, 2008. Topical Issue on Turbulent Mixing and Beyond, results of a conference at ICTP, Trieste, Italy, August 2008.

[23] Anshu Dubey, Ann Almgren, John Bell, Martin Berzins, Steve Brandt, Greg Bryan, Phillip Colella, Daniel Graves, Michael Lijewski, Frank Lffler, Brian OShea, Erik Schnetter, Brian Van Straalen, and Klaus Weide. A survey of high level frameworks in block-structured adaptive mesh refinement packages. *Journal of Parallel and Distributed Computing*, 74(12):3217–3227, 2014.

[24] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, J. W. Truran, and H. Tufo. Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical Journal, Supplement*, 131:273–334, 2000.

[25] Lorin Hochstein and Victor R Basili. The asc-alliance projects: A case study of large-scale parallel scientific code development. *Computer*, (3):50–58, 2008.

[26] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng. Programming petascale applications with Charm++ and AMPI. *Petascale Computing: Algorithms and Applications*, 1:421–441, 2007.

[27] JO Kane, HF Robey, BA Remington, RP Drake, J. Knauer, DD Ryutov, H. Louis, R. Teyssier, O. Hurricane, D. Arnett, et al. Interface imprinting by a rippled shock using an intense laser. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 63(5 Pt 2):055401, 2001.

[28] Q. Meng, J. Luitjens, and M. Berzins. Dynamic task scheduling for the uintah framework. In *Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10)*, 2010.

[29] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12, 2008.

[30] JD Moulton, Meza JC, and M et al Day. High-level design of Amanzi, the multi-process high performance computing simulator. Technical report, DOE-EM, Washington, DC, 2012.

[31] Patrick K. Notz, Roger P. Pawlowski, and James C. Sutherland. Graph-based software design for managing complexity and enabling concurrency in multiphysics pde software. *ACM Trans. Math. Softw.*, 39(1):1:1–1:21, November 2012.

[32] William L Oberkampf and Christopher J Roy. *Verification and validation in scientific computing*. Cambridge University Press, 2010.

[33] William L Oberkampf and Timothy G Trucano. Verification and validation in computational fluid dynamics. *Progress in Aerospace Sciences*, 38(3):209–272, 2002.

[34] SL Painter, JD Moulton, and CJ Wilson. Modeling challenges for predicting hydrologic response to degrading permafrost. *Hydrogeol. J.*, pages 1–4, 2013.

[35] S. G. Parker. A component-based architecture for parallel multi-physics PDE simulation. *Future Generation Comput. Sys.*, 22:204–216, 2006.

[36] James C Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D Skeel, Laxmikant Kale, and Klaus Schulten. Scalable molecular dynamics with namd. *Journal of computational chemistry*, 26(16):1781–1802, 2005.

[37] Robert G Sargent. Verification and validation of simulation models. In *Proceedings of the 30th conference on Winter simulation*, pages 121–130. IEEE Computer Society Press, 1998.

[38] L.I. Sedov. Similarity and dimensional methods in mechanics.

[39] Judith Segal. When software engineers met research scientists: a case study. *Empirical Software Engineering*, 10(4):517–536, 2005.

[40] Judith Segal and Chris Morris. Developing scientific software. *Software, IEEE*, 25(4):18–20, 2008.

[41] The Enzo Collaboration, G. L. Bryan, M. L. Norman, B. W. O'Shea, T. Abel, J. H. Wise, M. J. Turk, D. R. Reynolds, D. C. Collins, P. Wang, S. W. Skillman, B. Smith, R. P. Harkness, J. Bordner, J.-h. Kim, M. Kuhlen, H. Xu, N. Goldbaum, C. Hummels, A. G. Kritsuk, E. Tasker, S. Skory, C. M. Simpson, O. Hahn, J. S. Oishi, G. C So, F. Zhao, R. Cen, and Y. Li. Enzo: An Adaptive Mesh Refinement Code for Astrophysics. *ArXiv e-prints*, July 2013.

[42] Didem Unat, J Shalf, T Hoefler, T Schulthess, A Dubey, et al. Programming abstractions for data locality. In *Workshop on programming abstractions for data locality (PADAL 14)*, 2014.