Experience Report: Refactoring the Mesh Interface in FLASH, a Multiphysics Software

Jared O'Neal
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL 60439
Email:joneal@anl.gov

Klaus Weide
Flash Center for Computational Science
University of Chicago
Chicago, IL 60637
Email:klaus@flash.uchicago.edu

Anshu Dubey
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL 60439
Flash Center for Computational Science
University of Chicago, Chicago, IL 60637
Email:adubey@anl.gov

***Abstract*—FLASH is a highly-configurable multiphysics software designed for solving a large class of problems that involve fluid flows and need adaptive mesh refinement (AMR). FLASH has been in existence for two decades and has undergone four major revisions. It is now undergoing its fifth major revision to deal with increasingly heterogeneous platforms. The architecture of previous versions of the code and the AMR package at its core, Paramesh, are inadequate to meet the challenges posed by heterogeneity. In this paper we describe our experience with refactoring the mesh interface of the code to work with a more modern AMR library, AMReX. The focus of the paper is the refactoring methodology and the attendant software process that we have found useful to ensure that code quality is maintained during the transition.**

***Index Terms*—scientific computing, software engineering, refactoring**

I. Introduction

The multiphysics simulation software FLASH is a highly-configurable code that is designed for solving a large class of problems ranging from compressible and incompressible flows to N-body problems. The flows are often reactive, where the reactive terms can be nuclear or chemical. The primary discretization of the physical model is by Eulerian finite volume, but the code also supports finite difference methods and some Lagrangian modes of operation. The mesh, which is the discretization of the problem's spatial domain, can be uniform, where the distance between two consecutive discrete points is the same throughout the mesh. It can also be adaptive, where higher or lower mesh resolution is applied to regions of the domain as needed. In this latter mode, known as adaptive mesh refinement (AMR) [3], [4], these regions change as the simulation advances in time.

FLASH has undergone four major revisions in the past. It started as an amalgamation of three pre-existing code bases meant to provide orthogonal functionalities: Paramesh [14] for AMR, Prometheus [10] for reactive hydrodynamics, and a collection of functions to compute equations of state (EOS) and nuclear burning [22]. The first three revisions clarified the architecture including data ownership and scoping [7], [6]. The fourth revision primarily added a collection of capabilities to make the code ready for a new domain, high-energy-density physics (HEDP) [12].

We have now embarked upon another major revision, FLASH5, to prepare the code for running efficiently on forthcoming heterogeneous platforms. The architecture of previous versions of the code and the AMR package at its core, Paramesh, are inadequate to make good use of computational resources offered by these platforms. In this paper we describe our experience and methodology in refactoring FLASH's internal mesh interface, at which the code that creates and manages the mesh is coupled to the code that uses the mesh. The objectives of this effort are: (1) to transition the base AMR capability from Paramesh to the more modern AMR library AMReX [1]; (2) continue to maintain separation between mesh and physics units through appropriate abstractions; and (3) prune old and redundant portions of the interface.

The remainder of the paper is organized as follows. In section II we summarize available literature on refactoring in scientific software to the best of our knowledge. We emphasize the caveat because literature on this topic is sparse and difficult to find since there aren't many publications or conference venues available for publishing such work. In section III we describe the motivation behind the choices made for the transition. Section IV describes details of FLASH relevant to the discussion in this paper. Sections V and VI describe our approach in detail. In section VII we discuss the outcomes of this effort and present our conclusions.

II. Background

As understood by the software engineering community, the term refactoring usually applies to a careful process where the internals of the implementations are "cleaned up" without changing the behavior. There is plenty of literature about refactoring methods and tools. Among them, [17] provides a good survey and [15] considers motivations for refactoring. Literature for commercial software refactoring is large and not very germane to this work. Therefore, we focus our attention on publications related to software restructuring and refactoring in the scientific world. Several of these papers are case studies from specific software projects, e.g. [13], [5]. Some of the literature discusses the lack of software engineering practices in scientific software development, where refactoring is one of the considered features, i.e. [21], [20], [11], [9].

Among these [9] is notable in that it makes an attempt to provide a systematic study of scientific software development. Literature is sparse on general studies of refactoring in scientific software with a few exceptions such as [18], [19], which are about Fortran code refactoring, and [16], which is about defining a methodology for code development taking into account ongoing changes in specifications that are an inevitable part of scientific software development.

III. Motivation

As mentioned in section I, the primary motivator for restructuring FLASH is the heterogeneity in future platforms that simultaneously increases the degree of parallelism with the diversity in methods and approaches needed for exploiting it effectively. This requires introduction of hierarchy in data structures and parallelism granularity, and moving away from the bulk-synchronous parallel model that has been the mainstay of complex applications such as FLASH. FLASH had an additional incentive to consider deeper restructuring of its base infrastructure because of a confluence of circumstances. In the current version of FLASH, AMR is enabled through the incorporation of the AMR library Paramesh [14]. However, this library is no longer in active development. Therefore, in the interest of moving this dependency to a maintained and supported library, we needed an alternative. AMReX was an obvious choice once it became the centerpiece for the exascale co-design for AMR under the US DOE Exascale Computing Project (ECP)[2]. AMReX's development path is favorable to the architecture design approach that FLASH desires for becoming an exascale-capable application. It includes an architecture that allows for separation of parallelism concerns from numerical and algorithmic concerns, while also hiding the details of the target platform's macro-architecture from the physics kernels in the code. For more details about AMReX and FLASH next generation architecture see [1] and [8] respectively.

IV. FLASH

FLASH is a component-based code where different permutations and combinations of components constitute different applications. Multiple alternative implementations exist for some of the components, which adds to the degree of composability in the code. While FLASH, which is primarily written in Fortran, is implemented with procedural programming, the organization of files, its architecture, and its build system were designed and implemented to take advantage of the benefits of object-oriented programming (OOP) without incurring the runtime overhead, such as virtual tables, associated with OOP.

Specifically, a set of related functionality and data that could be grouped as a single class is organized into a code unit in FLASH. Each unit is isolated in the FLASH source directory as a folder hierarchy containing all related routines. This set of code offers one subset of routines whose names begin with the name of the unit, and another whose names begin with a common two letter abbreviation of the unit. For example, the "Grid" unit provides routines whose names have the respective

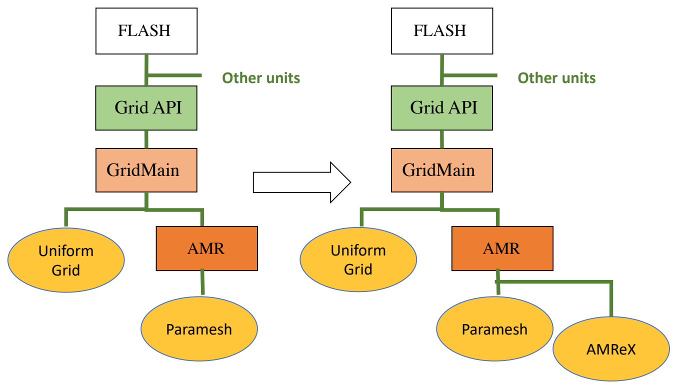

Fig. 1. Transition of the FLASH unit architecture with an emphasis on the Grid unit structure. Each Grid element is a subfolder in the FLASH folder hierarchy. A rectangular shape indicates an abstract layer in the inheritance hierarchy; an oval shape, a concrete implementation of the Grid unit. The refactoring begins by adding an AMReX-based implementation that co-exists with the Paramesh-based implementation.

prefixes `Grid_` and `gr_`. The former are routines that define and implement the public interface of the unit. The latter are routines implementing functionality for use only within the unit and are therefore analogous to protected class methods. At the top level of the folder there are null implementations for the entire interface of the unit. The subfolder and file structure of a given unit is designed and constrained so that different implementations of the unit may be written and used. In particular, the design allows for inheritance so that common code may be written and either used or overwritten by any particular implementation. See Figure 1 for a cartoon visualization of an example folder hierarchy and refer to [7] for a more detailed description of this structure.

Inheritance is managed and runtime overhead is avoided by adding into the use of FLASH an additional setup stage. At this level, FLASH users express to the setup system the different implementation and runtime parameters they desire for a particular instance of their simulation. The setup system determines which versions of each routine are needed and aggregates these into a single folder. The build system is then used to build that particular version of the simulation.

One example of a unit is the Driver unit, which is responsible for orchestration of sequencing and interaction among units during runtime, and also executes timestepping. All units that contain algorithms responsible for implementing different aspects of the physical model are sensibly grouped under the category of physics units. For instance, hydrodynamics codes are located in the Hydro physics unit, and during this refactoring effort we have worked with two implementations of this unit named "simpleUnsplit" and "Unsplit". The former is a simplified implementation meant for easy testing of FLASH *via* simulation. The latter is a full-featured implementation that is meant to be used for production.

At runtime, FLASH's Grid unit is configured with the desired domain discretization. If the simulation is configured to run AMR, then the Grid unit is also configured to know how and when to refine the resolution across the spatial domain in accordance with the evolution of the solution. 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. Parallel computation is achieved by distributing the blocks across processes. While each block contains identically-shaped rectangular cells, the adaptivity in FLASH's AMR meshes is obtained by composing regions of different resolution with blocks of appropriately-sized cells.

Besides managing the mesh definition and evolution of mesh resolutions in time, the Grid unit also 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. In the current production version of FLASH, for instance, physics operators typically loop over blocks by first obtaining a list of blocks on which to operate *via* the `Grid_getListOfBlocks` public interface routine. Each element in the returned list is an index for a block that can be given to the Grid unit to obtain metadata for that block. Thanks to this parametrization of block metadata, the operators remain oblivious to the physical location of the blocks in the spatial domain. A common use of the Grid unit by another unit is shown in Listing 1.

Listing 1
THE FIRST DO LOOP ILLUSTRATES LOOPING OVER BLOCKS IN PRE-REFACTORING FLASH. THE SECOND ILLUSTRATES THE USE OF THE NEW BLOCK ITERATOR AND ITS ASSOCIATED BLOCK METADATA STRUCTURE

```
! FLASH4.4 looping over blocks
call Grid_getListofBlock(blocklist, blockcount)
Do i = 1, blockcount
     blockid = blocklist(i)
     call Grid_getBlkLimits(blockid, limits)
     call Grid_getBlkPtr(blockid, solnData)
     call Physics(blockid, solnData)
End Do

! FLASH5 looping over tiles
call Grid_getIterator(itor, tiling=.TRUE.)
Do while (itor%is_valid())
   call itor%blkMetaData(blk)
   limits = blk%limits
   call Grid_getBlkPtr(blk, solnData)
   call Physics(blk, solnData)
   call itor%next()
End Do
call Grid_releaseIterator(itor)
```

V. Methodology

The first step in developing methodology for restructuring complex software like FLASH is the phase where one considers and plans the degree and scope of change. This is followed by exploratory prototyping to either confirm the design choices or evolve them as needed. Once the design choices are validated, implementation can proceed with more rigor followed by verification of every code module affected directly or indirectly by this effort.

A. *Scope of refactoring*

Before embarking on the refactoring, it was decided that the Paramesh-based Grid unit implementation should not be removed from FLASH immediately. Rather, an AMReX-based implementation would be added to FLASH as an alternate AMR implementation and would therefore coexist with the Paramesh version (Figure 1). This decision implied that as the Grid unit interface evolved to accommodate AMReX, the Paramesh implementation would be co-evolved to satisfy the updated interface. Therefore, on shorter timescales this effort progresses as an extension of FLASH through the addition of a new feature.

This plan was adopted because fundamental changes like mesh are best done in small steps, with each step thoroughly verified for regression. Having a functional Paramesh-dependent Grid unit implementation permits cross-verification of any new AMReX-based functionality introduced. As the Paramesh implementation will be removed from FLASH once the AMReX-based implementation is fully-implemented and verified, the long term goal is refactoring and the option of AMReX and Paramesh coexisting is motivated by making the refactoring easier and with clear verification of success.

To date this refactoring effort has been largely restricted to the level of redefining public interfaces and adapting the code on either side of these interfaces. Fortunately, the original architectural and interface design of FLASH had resulted in a good amount of encapsulation and modularity, so that it was not necessary to alter the kernels of the scientific portions of FLASH.

A graphical overview of the advancement of the refactoring effort is presented in Figure 2.

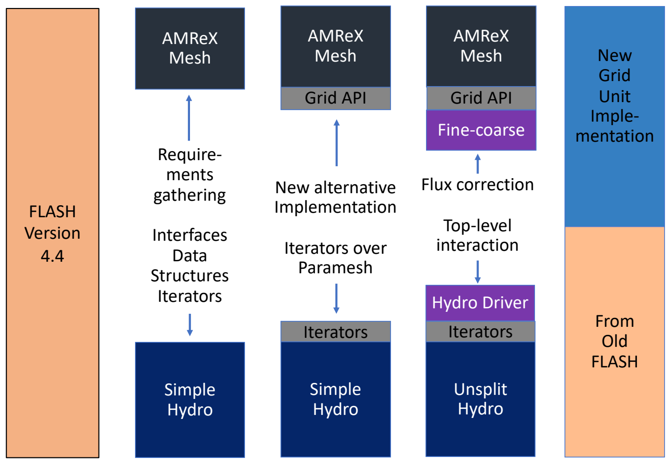

Fig. 2. Temporal evolution of implementation of AMReX-based Grid unit and changes made to this unit and pre-existing FLASH code to accommodate growth. Time runs left to right.

B. *Design phase*

An important question confronting us at the beginning of the project was the organization of the Driver unit, which typically invokes the highest-level interface of each physics unit it is driving. These, in turn, carry out looping over the mesh blocks. This method works because there is operator splitting in the time-step. However, in the future we do not expect to be able to maintain strict operator splitting for all physics. One consideration, therefore, was to put looping over blocks in Driver and let the physics units strictly only see a block at a time. We did an exploratory implementation of this functionality and soon discovered that this not only makes the driver too cumbersome, but also conflicts with the needs of units such as Hydro, which intersperse global operations with per-block operations. Our solution, therefore, is to hoist the highest-level interface in each physics unit into a separate function that doesn't apply any operators but rather loops over blocks and applies functionality on each block through lower-level calls.

To accommodate the hierarchical parallelism through tiling and asynchronization being implemented in AMReX, this looping over blocks could no longer be facilitated by the Grid unit routine `Grid_getListOfBlocks` mentioned in section IV. Rather, the inclusion of AMReX required changes to fundamental data structures exposed through the Grid unit interface and therefore to the interface as well. In particular, arrays of block indices have been replaced with iterators over blocks. For the current block, the iterator provides a data structure that encapsulates a minimal set of metadata, such as the block's mesh refinement level and spatial extent. This data structure can also be used to access the data stored on the associated block. This differs from version 4 of FLASH, which operates in pull mode such that physics units query the Grid unit for any metadata it needs through explicit interfaces.

C. Prototyping phase

The second phase of implementation was to begin refactoring the Paramesh-based FLASH implementation to use the iterator and metadata data structures as well as to subsequently begin integrating AMReX into FLASH through unittests. This process was exploratory and iterative in the sense that the interface of these data structures was evolved to satisfy the needs and constraints of both the Paramesh- and AMReX-based implementations of the Grid unit. Note that this portion of the refactoring as well as the following sections did not evolve other mesh implementations, such as the uniform grid mode.

To understand the effect that changes to the Grid unit had on interactions with other units, the minimalistic simpleHydro implementation mentioned in Section IV was used. It was hoped that the pressures that the physics units impose on the Grid unit interface is well-represented by this implementation. In practice, standard use patterns of the iterator, the metadata data structure, and the associated Grid interface were revealed with the take away that the iterator appears to be a welcome change to FLASH. In particular, the fact that the block data structure not only functions as a unique index but also contains metadata allows for passing fewer parameters to routines operating on blocks.

By the end of this phase, a single 2D simulation could be run with the AMReX implementation of the Grid unit and with simpleHydro. Rigorous verification of the quality of the result was not yet possible so that verification was only qualitative proof of concept. In fact, the implementation was missing several corrections needed to achieve essential physics such as energy conservation.

While this method of exploring and prototyping was reasonable as the portion of FLASH touched was limited, this phase was understandably marked by an accumulation of technical debt. For instance, normal work overhead such as maintaining inline documentation and using testing with substantial code coverage were avoided with the obvious consequences. This difficulty was anticipated and the development team was given a period of approximately one month to study what had been done, derive lessons learned, and apply these through an effort of cleaning and maturing the changes.

D. Implementation phase

Once a fair portion of Paramesh had been updated to use the iterator, refactoring entered its third phase. Here, each author was responsible for one of the following development tasks:

1) update the fully-featured Unsplit hydrodynamics implementation for inclusion in FLASH5,
2) demonstrate the successful execution of a fully-functional 2D simulation with AMReX by continuing to integrate AMReX into FLASH with simpleUnsplit, and
3) achieve a fully-functional hybrid version of FLASH that used AMReX data structures for storing data but Paramesh to drive the refinement of the mesh.

This partitioning of the work into non-overlapping tasks had the benefit of minimizing merge conflicts in the version control system and was likened to one team building a set of tracks from one direction, another team a set of tracks in the opposite direction, and the third team building the station where the two tracks should meet.

This divided AMR-based FLASH into distinct operational modes for which AMR refinement is done with either

1) iterator-based Paramesh,
2) hybrid Paramesh/AMReX implementation, or
3) iterator-based AMReX.

As detailed in Section VI, this phase coincided with a more rigorous approach to verification and specifically moved toward continuous integration by designing and implementing policies along with an automated testing system. In addition, work progressed such that documentation is now actively maintained as code is refactored and such that prototype code is cleaned, out-of-date documentation is updated, and code that is not needed for FLASH5 is pruned.

VI. Verification

Regular verification of FLASH5 was integrated into the refactoring work early in the prototyping stage. The amount of code coverage and the level of testing automation matched the scope and character of the different development phases.

Throughout the process, baselines for regression testing of the Paramesh implementation were first established with FLASH4.4, which pre-dates the refactoring effort. For the subset of FLASH5 covered by our present test suite, the parallelization of the code is such that bit-by-bit reproducibility of results generated by different runs and by tests run with a different number of MPI processes is expected. Fortunately, this allows for regression testing without requiring a threshold to gauge success, which can be hard to determine.

However, certain refactoring steps did lead to different round-off errors, which necessitated the manual establishment of new baselines. The process of verifying these new baselines produced as a byproduct a history of expected variations in simulation results due to round-off noise. Estimating this level is useful as verification of AMReX-based results can be done by direct comparison against associated Paramesh-based results. As these two implementations do use their own versions of floating point computations (e.g. interpolation), this verification does require a threshold for gauging sufficient similarity.

A. *Prototyping phase*

Verification of refactoring efforts related to introducing the iterator and metadata data structures into the Paramesh implementation of the Grid unit was based on manual regression testing with a single 2D simulation. At the same time, no specific git workflow policies were in place to help maintain code correctness. While this allowed for quick exploration, it is clear that refactoring might have created bugs in the implementation that exist only with configurations not tested. In addition, the lack of code coverage might have resulted in undetected bugs in other units.

Introducing AMReX into FLASH5 required adding a significant amount of complex code. Therefore, the likelihood of successfully advancing the work without errors to the point where a full simulation could be run was low. To address this difficulty, several AMReX-specific unittests were created to verify correctness of the complex code.

B. *Implementation phase*

To help overcome technical debt and to reduce the accumulation of more debt, verification of FLASH5 during this phase centered on adopting a git workflow that allows for continuous integration and therefore protects the master branch through code reviews and different levels of automated testing with a Jenkins-based test server.

While creating AMReX-based simulation results was a main goal of the previous phase, writing these results to FLASH-format HDF5 files was not. In fact, code at the prototype stage was not able to save computation results to disk storage and, as a result, regression testing of simulation results was limited to qualitative, visual verification, which was both insufficient in terms of quality and efficiency. Therefore, an initial task was to write AMReX-based results to file with the standard format so that pre-existing file comparison tools in FLASH could be used for improved regression testing.

The addition of new capabilities into the AMReX implementation drove the growth of the test suite. This process to add a new capability

1) identifies a new end-to-end simulation that covers the new capability,
2) establishes a Paramesh baseline for the simulation using FLASH4.4,
3) determines whether changes to the Paramesh-based Grid implementation in FLASH5 are required and, if necessary, refactors this implementation in conjunction with regression testing, and
4) adds the new capability in the AMReX-based Grid implementation in FLASH5, in conjunction with verification testing by comparing results against the Paramesh baseline.

As mentioned above, this final step requires the identification of a threshold for determining if the AMReX-derived result is sufficiently similar to the Paramesh-derived result.

For each end-to-end regression test included in the test suite, results are generated in pseudo-uniform grid (pseudo-UG) mode and in normal AMR mode. The former mode uses the same AMR implementation of Grid but limits the AMR functionality to a single mesh refinement at the finest resolution. The pseudo-UG results are cleaner in the sense that the Grid unit does not carry out floating point computations such as interpolation and averaging of data across different mesh resolutions. Therefore, comparisons of AMReX-based pseudo-UG results are expected to be bit-by-bit exact with the Paramesh-based results.

The Pseudo-UG results are therefore the true numerical baseline for the simulation, and the AMR results are manually verified against this baseline by calculating the error between the AMR approximation and the baseline on those regions of the domain that were resolved to the finest resolution.

VII. Summary and Conclusions

The AMR interface refactoring in FLASH has gone relatively smoothly because we started with a software where the abstraction of mesh infrastructure from physics solvers was effective. The methodology adopted during the previous transition from version 2 to version 3, where development of the framework was done in isolation from the physics [6], has proved to be an excellent design choice in retrospect. This separation allowed the first phase of exploration/prototyping to remain lightweight and rapidly produce executable code that could be exercised for validating design choices. As expected, during this phase some technical debt was acquired in the form of degradation of documentation and rigor of testing. Although we allocated a period for cleaning the code once the design choices were validated and implementation began in earnest, the technical debt is not paid entirely yet. Code cleaning and catching up on documentation is an ongoing process at this writing.

Team discussions revealed that the aforementioned technical debt is still manageable for experienced FLASH developers. However, the transition from a prototyping phase to a more rigorous development phase was necessary for new developers as the debt was becoming unsustainable. In particular, manual regression testing and qualitative visual verification became a significant portion of the cognitive load during development, especially as more tests were added to the test suite. Also, documentation of requirements, which was often built into FLASH in the form of inline documentation, can be missing or wrong. This necessitates reverse engineering of requirements from the code, which itself could be in a prototyped state.

The most important cost-benefit trade-off during this process was the amount of development to be done with Paramesh given that the ultimate goal was to eliminate the use of Paramesh. The coexistence of the two implementations has allowed for designing and implementing an effective test suite and testing process. Even the investment in creating a hybrid mode where Paramesh and AMReX operate in parallel has been helpful in both debugging and gaining confidence in the refactored code. We conclude that it has been worthwhile investing in the refactoring of the Paramesh implementation to use iterators and to satisfy changes in the Grid interface driven by integrating AMReX.

Additionally, refactoring the mesh interface to meet AMReX requirements has led to a general improvement of the use of the Grid unit by other units. In particular, the iterator and its associated block metadata data structure provide an easy-to-use interface that pairs well with the Grid unit's interface such that similar simple use patterns emerge. In conclusion, the initial effort in designing the architecture of FLASH is proving its worth now where modularity and encapsulation have made refactoring clean.

Acknowledgments

This work was supported by the U.S. Department of Energy Office of Science Office of Advanced Scientific Computing Research. 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 nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government. 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] AMReX. https://amrex-codes.github.io/.

[2] Exascale computing project. https://www.exascaleproject.org/.

[3] M. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[4] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1):64–84, May 1989.

[5] B. Du Bois, S. Demeyer, and J. Verelst. Refactoring - improving coupling and cohesion of existing code. In *11th Working Conference on Reverse Engineering*, pages 144–151, Nov 2004.

[6] A. Dubey, K. Antypas, A. Calder, C. Daley, B. Fryxell, J. Gallagher, D. Lamb, D. Lee, K. Olson, L. Reid, P. Rich, P. Ricker, K. Riley, R. Rosner, A. Siegel, N. Taylor, F. Timmes, N. Vladimirova, K. Weide, and J. ZuHone. Evolution of FLASH, a multiphysics scientific simulation code for high performance computing. *International Journal of High Performance Computing Applications*, 28(2):225–237, 2013.

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

[8] A. Dubey and D. Graves. A design proposal for a next generation scientific software framework. HeteroPar'2015, colocated with Europar-2015.

[9] R. Farhoodi, V. Garousi, D. Pfahl, and J. Sillito. Development of scientific software: A systematic mapping, a bibliometrics study, and a paper repository. *International Journal of Software Engineering and Knowledge Engineering*, 23(04):463–506, 2013.

[10] B. Fryxell, E. Müller, and D. Arnett. *Numerical Methods in Astrophysics*, page 100. New York: Academic, 1989.

[11] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, May 2009.

[12] D. Lee, G. Xia, C. Daley, A. Dubey, S. Gopal, C. Graziani, D. Lamb, and K. Weide. Progress in development of HEDP capabilities in FLASH's unsplit staggered mesh MHD solver. *HEDLA 2010: Astrophysics and Space Science Special Issue*, 2010.

[13] Y. Li. Reengineering a scientific software and lessons learned. In *Proceedings of the 4th International Workshop on Software Engineering for Computational Science and Engineering*, SECSE '11, pages 41–45, New York, NY, USA, 2011. ACM.

[14] P. MacNeice, K. Olson, C. Mobarry, R. de Fainchtein, and C. Packer. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications*, 126(3):330–354, 2000.

[15] M. V. Mäntylä and C. Lassenius. Drivers for software refactoring decisions. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 297–306, New York, NY, USA, 2006. ACM.

[16] M. Méndez and F. G. Tinetti. Change-driven development for scientific software. *The Journal of Supercomputing*, 73(5):2229–2257, May 2017.

[17] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb 2004.

[18] J. Overbey, S. Xanthos, R. Johnson, and B. Foote. Refactorings for Fortran and high-performance computing. In *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 37–39. ACM, 2005.

[19] J. L. Overbey, S. Negara, and R. E. Johnson. Refactoring and the evolution of Fortran. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 28–34. IEEE Computer Society, 2009.

[20] M. T. Sletholt, J. E. Hannay, D. Pfahl, and H. P. Langtangen. What do we know about scientific software development's agile practices? *Computing in Science Engineering*, 14(2):24–37, March 2012.

[21] T. Storer. Bridging the chasm: A survey of software engineering practice in scientific programming. *ACM Comput. Surv.*, 50(4):47:1–47:32, Aug. 2017.

[22] F. Timmes. Integration of nuclear reaction networks. *The Astrophysical Journal Supplement Series*, 124:241–263, 1999.