Journal of
open research software

Dubey, A and Van Straalen, B 2014 Experiences from Software Engineering of Large Scale AMR Multiphysics Code Frameworks. *Journal of Open Research Software,* 2(1): e7, pp. 1-5, DOI: http://dx.doi.org/10.5334/jors.am

ISSUES IN RESEARCH SOFTWARE

Experiences from Software Engineering of Large Scale AMR Multiphysics Code Frameworks

Anshu Dubey* and Brian Van Straalen*

Many research problems are being pursued through simulations that require multi-physics capabilities in codes, that are also able to run on HPC platforms. Multiphysics implies many solvers with divergent, and sometimes conflicting, demands on the software infrastructure. Additionally many multiphysics simulation codes make use of structured adaptive mesh refinement to achieve maximum resolution where needed within resource constraints, which places even more demands on software infrastructure. The software architecture and process for these codes, therefore, is a challenging task. In this experience report we detail the challenges faced, design choices made, and insights from two such major software efforts, FLASH and Chombo.

Keywords: structured AMR; frameworks; experience

1 Introduction

Of the many research problems being pursued through simulations many are multiphysics multiscale in nature. Such problems typically need high performance computing (HPC) resources, and require more than one solver to properly model all phenomena of interest. The solvers often have divergent requirements of data layout, and place different demands on the underlying hardware and system software. However, for an integrated simulation, all the involved solvers need to be able to interoperate with one another. When we take into account the diversity of computing platforms and their typical shelf life into account, and compare that with the number of person years it takes to build a reliable multiphysics software, it becomes obvious that software engineering of such codes is as important as it is non trivial.

Among the present generation of multiphysics HPC simulation codes there are many that are built upon general infrastructural frameworks. This is especially true of the codes that make use of structured adaptive mesh refinement (SAMR) because of unique demands placed on the framework of the code. They have varying degrees of abstractions between the infrastructure such as mesh management and I/O and the numerics of the physics solvers. In this experience report we detail the challenges faced, design choices made, and insights from two of such major software efforts, FLASH [1] and Chombo [2].

Both Chombo and FLASH are built on top of the same SAMR [3] principles, however, their architecture, purpose and reach are very different. Chombo is primarily an AMR library that comes with built in solver technologies. Typical Chombo users build application codes for their domains by treating Chombo as a toolbox. Therefore, Chombo has been the basis of such application codes as Bisicles[4], used for ice sheet modeling, ChomboCrunch[5], used for modeling pore scale reactive transport in carbon sequestration, CHARM [6, 7], used for cosmology, and several others (see [8]). About 15 active users have built their own applications on top of Chombo, and there are roughly 200 casual users. FLASH, on the other hand, is a complete application code that can use Chombo as one of its mesh packages. FLASH has been applied to a variety of astrophysics problems including super-novae, X-ray bursts, galaxy clusters, and stellar structure. It has also been used to model laser experiments and problems involving fluid-structure interactions. More than 850 papers have been published that used FLASH to obtain all or part of their results for a combined authorship of roughly 1200. The basic component in FLASH is code unit that provides a specific functionality. Typical FLASH users combine capabilities provided by the code in many different ways, customize some of them and/or add relatively small amount of code of their own. Chombo expects its sophisticated users to modify some of its lower levels, FLASH takes a great deal of trouble to avoid such occurrences. Because of these differences in approach there are differences in the architecture of the two codes, however, their software engineering and sustainability practices are very similar. The target platforms for the two codes range from small applications running on workstations to large simulations running at scale on supercomputers such as Edison, the Cray machine at NERSC, and Mira, the BG/Q machine at

* Lawrence Berkeley National Laboratory, Berkeley CA, United States
adubey@lbl.gov, bvstraalen@lbl.gov

ANL. Both the codes support hybrid MPI/OpenMP parallelization, neither code supports accelerators.

2 History

Chombo started as not-backward-compatible branch of the BoxLib Framework, a collection of tools to manage adaptive mesh. The reason for this bifurcation was to serve the divergent needs of two groups using BoxLib as their basic source. Chombo's objective was to provide a general purpose tool that users could build upon without necessitating significant interactions with the core developers of the code. Therefore, it was found to be necessary to articulate the interfaces clearly and explicitly. The Chombo team made significant changes to the API layers above BoxLib that were best suited for the purpose of generalization and to be able to leverage other supporting libraries and software packages (for example parallel I/O and visualization).

FLASH, on the other hand, started its life as an amalgamation of three independent codes written predominantly in f77 style [9]. From the beginning the main purpose was to be able to simulate phenomena of scientific interest as early as possible. The scientific objectives were the drivers for the code development, and in the early stages less effort was expended on code architecture than on capability addition. However, since the same code base was to be used for several different but similar applications, the concept of reusable alternative code components became the basis for code architecture design. As the code and its user base grew, the need for a more coherent code architecture became apparent. The architecture and modularity was achieved by unraveling the data structures and lateral dependencies over several design iterations. The current software infrastructure of the code has been stable for close to a decade.

3 Software Design Choices

The software design choices of the two codes were dictated by a combination of their expected roles and target communities, and their starting point. As mentioned earlier, Chombo set out to be a library of AMR technologies and solvers on top of which application codes could be built, whereas FLASH aimed to provide end-to-end simulation solution with customizability. In keeping with these different aims Chombo's APIs can interact with the client code at many different levels from the highest to a fairly low level. FLASH, on the other hand, publishes all of its API at the highest level and hides the hierarchy within a unit from the users. Chombo's architecture is layered while in FLASH the units are treated as peers which interoperate with one another. Within a unit though, there is layering within FLASH's architecture as well. This was done to facilitate possible exploitation of third party software for some desired capabilities. The FLASH architecture, therefore, recognizes the concept of a "kernel", which lies at the lower layer and is not subject to the code's architecture rules or coding standards. An intermediate wrapper layer provides a separation between the unit's API and the kernel. In Chombo customization gets easier as one moves up in the layered architecture, it is most challenging at the level of base classes. In FLASH customizability is achieved through a specialized unit designed to provide an encapsulated space for user code, and the degree of customizability is flat across the entire code base. In Chombo, the API of a class rarely changes, in FLASH additions to the API of a unit can happen when a capability addition demands such a modification. In Chombo a capability addition usually means a new class, in FLASH it can mean many different things. It could be a new unit, a new subunit within a unit or a new alternative implementation of a unit.

In addition to their different roles, one other major difference influenced the design choices of the two codes, and that was the language. Chombo, being a C++ code, has the advantage of language supported object oriented features, while FLASH, having a great deal of Fortran legacy code in its core, had to devise customized ways of imposing principles of object oriented design on top of non-object-oriented code kernels. This was achieved through the use of unix directory structure and a limited domain-specific-language (DSL) for configuration. This difference is reflected in ways that encapsulation and abstractions play out in the two codes. In Chombo, the classes are direct translations of mathematical abstractions, while the units in FLASH are specific capabilities whether physics such as hydrodynamics, or infrastructure such as mesh management. A unit in FLASH can encompass multiple mathematical abstractions, which often become the basis for separating sub-units within a unit. Chombo enjoys the benefits of strong typing provided by the language in keeping the code relatively clean and free of bugs. FLASH relies on a collection of homegrown scripts to do the same, though not as effectively. However, sometimes the advantage is that coding standard can be short circuited for debugging purposes; something that is much harder to do with language imposed constraints.

Because of having the framework in C++, and being a library for scientists, Chombo has to ensure interoperability between Fortran and C++. The reasons are a mix of the user profiles (many have an already existing Fortran code with which they want to use AMR), and the relative ease of obtaining better performance with multidimensional arrays in Fortran. The generic interface, of passing values by reference, works but tends to be error-prone. Chombo solves this problem by providing a DSL only for the C++/Fortran interface. FLASH does not face this problem to the same extent because majority of its production grade mesh infrastructure and solvers are Fortran. Only the I/O is predominantly C, but because users rarely interface with it directly, providing Fortran wrappers proves to be an adequate solution.

4 Software Process

It is in the software engineering and the software process that these two codes, and many other codes of similar vintage, have a lot in common. Most codes started with CVS for version control and then transitioned to subversion.

Neither Chombo nor FLASH have yet moved to newer decentralized versioning systems, primarily because they have not yet found a centralized repository to be a serious hindrance in their code management. Neither code has seriously considered decentralization of the repository for two reasons. One is gate-keeping for quality control and the other is the expectation that the next iteration of both codes will be far more disruptive, and therefore a more logical choice for adopting new development and distribution models.

Almost all the current generation multiphysics capable codes have composability with interoperating components built into them. FLASH and Chombo are no exceptions. They both aim to serve science domains with many divergent solver needs and therefore the ability to orchestrate applications out of a combination of many moving parts is a necessity. Both the codes have been largely successful in realizing the separation of concerns between the numerical and parallel complexity through modularization and adoption of such component based architectures.

Both the codes have ongoing verification practices that ensure code quality and robustness. They both have nightly regression testing on multiple platforms and policies about addressing any failures in testing [9]. Also, regular testing is not the complete verification story for either code. There are different levels of testing for different purposes, for example targeted verification on a platform before a simulation campaign. Both codes publish and enforce their coding standards and have well defined policies about external contributions. Such policies are an important part of the gate-keeping strategies that balance the code growth and community investment with maintaining the code quality. Both codes conduct periodic training for new users and developers because the underlying AMR and solver technologies are inherently complex. Both codes firmly resist the notion of providing black-box solutions for any except the most trivial applications because it is very easy to obtain bad scientific results if the user does not understand the involved methods or is not careful with the quirks of the involved technology.

Documentation is extremely important in both the efforts and it exists in several forms. The importance of extensive documentation cannot be over-emphasized. Any complex code with many components cannot, by definition, have any developer that understands all aspects of the code. When there is a transient developer population, which happens often in academic and laboratory environments, the code base simply cannot be maintained without adequate documentation, because some of the expertise is also then transient. A well documented code section can be maintained by non-experts with general know-how of the code, whereas undocumented code will eventually die once the related expertise has gone from the team. Since both codes also routinely receive code contributions from external users, a clearly written developer's guide is also a necessity. It provides the added advantage of helping the learning curve of new team members.

5 Insights

In this section we share some of our more general insights gained during the evolution of the two code projects that might be of benefit to other projects. A more detailed discussion from FLASH can be found in [10], though many of the lessons learned described there apply to both the codes equally. The most important contributor to successful infrastructure building of both Chombo and FLASH was an earlier investment in framework research. That investment, erroneously, is assumed to have mostly been a failure because several of those frameworks did not survive. What is often overlooked is that even the failed frameworks contributed to the body of knowledge that was utilized in building backbone frameworks for all the large scale multiphysics codes that have had any degree of success in their respective communities. The codes themselves also benefitted from the availability of long term sustained funding in the initial stages to devote to framework development. Resources could be allocated for designing the code architecture and the appropriate mechanisms for ongoing code maintenance and verification. And there were experiences and literature to consult. The outcome in both instances has been software that has been available for about a decade and has been growing with its users community. It is fair to say that software projects without such support struggle a great deal more in achieving longevity. Both projects have also hugely benefitted from having members of the team that can communicate with domain experts, computer scientists and numerical analysts. Teams with such broad and cross-cutting expertise are consistently better able to absorb the loss of specific expertise.

A well designed software architecture that allows easy interoperability among various solvers is just the starting point in sustainable software design. No software being used is ever in a stationary state. As the acquired knowledge grows, so do the algorithms and the demands placed on the solvers. Sometimes the solvers need to change to accommodate the findings, at others new solvers need to be added as models are refined based upon the findings from the simulations. It is, therefore, equally important to design the software with extensibility and flexibility built into its architectural framework. From the reusability perspective also supporting many different applications is desirable even though it places more demands on the software design.

Another extremely useful insight from applications with components that place diverse demands on the system is that composability in software requires a careful balancing act. Deep optimization of individual components is rarely the best option, it can sometimes even be detrimental to the overall performance. One must also consider sub-optimal solutions for individual components in order to achieve optimal overall performance. Chombo and FLASH achieve good overall performance by dictating a common basic data layout to all participating solvers and mesh components even though it might be suboptimal for some of the components. FLASH's mesh infrastructure owns the data layout and the data, it only hands it over the

to the physics units to operate on. Chombo does not outright own the data itself, but by owning the data layout it imposes similar discipline on the client code. Such judicious trade-off between maintainability, portability and performance has been the hallmark of these and many other successful SAMR codes.

6 Future Prospects

The two codes discussed here, and other similar software packages, are in production in multiple disciplines, are prolific in producing scientific results, and are likely to continue to do so in the immediate future. They have reduced the barrier to entry into high performance computing for a large number of users. They have also enabled greater productivity among their user communities by eliminating the need for individual researchers to build their own infrastructure. The evidence of their usefulness is in the number of publications and dissertations that used these packages to obtain some or all of their results, and the constantly growing user communities. Though the packages are stable and extremely useful now, their future is less certain because of the ongoing revolution in hardware architecture.

In the era of cluster computing with fat nodes, distributed memory computing provided a near ideal programming model where these sometimes conflicting requirements of performance, portability and maintainability could be balanced. The overheads of communication primitives and bulk synchronizations could be amortized over large computational units to the point where they did not significantly compromise performance. Also, since the node architectures were mostly homogeneous even across vendors, general algorithmic or data structure optimizations provided benefits across the board. Therefore, although the two codes followed different paths, and differ significantly in their details, they have arrived at remarkably similar software architecture solutions conceptually. Their current success combined with the uncertainty in the HPC landscape at present has induced hesitancy in prioritizing the infrastructure refactoring by the funding agencies. This could not only halt, but possibly even reverse the gains made by these, and other codes in fostering a community approach to developing and using codes.

The frequent assertion that the code developers will rewrite their codes for the target platforms is valid only for software with relatively small code bases. When the algorithms are well understood, and refactoring the code is likely to take only a few person-months there is nothing to be gained by anticipating trouble and preparing for it ahead of time. However, codes like Chombo and FLASH have hundreds of thousands of lines of code, and therefore the amount of rewriting needed to obtain reasonable performance on each new heterogeneous target platform could take several person years. The only solution is to re-architect the codes in ways that can utilize higher level abstractions such as code transformation, auto-tuning and runtime management to shield them from platform specific details. Our combined experience indicates that bugs get eliminated from the code over several years of production use, therefore writing similar codes from scratch is not a solution either. Building, maintaining and orchestrating such codes has been challenging in the past, and their utilization of HPC resources has always been a balancing act between portability and performance. Increasing heterogeneity has moved this beyond a balancing act to a question of whether the codes will be able to effectively use future HPC resources at all.

Many efforts are underway to develop programming models and tools to help scientific software development. That may be a good solution for the far future, but the need for refactoring codes is more immediate, and new technologies take time to mature. What often gets overlooked in the discussion is that the present code base is already facing heterogeneity and hyper-parallelism. Therefore the redesign and re-architecting of code frameworks should begin now, especially because a consensus is beginning to emerge about the conceptual design that would allow the codes to work well on several generations of heterogeneous platforms without the constant need to rewrite. With the right kind of refactoring these codes can continue to serve their communities for many more years.

References

1. **Dubey, A, Antypas, K, Ganapathy, M K, Reid, L B, Riley, K, Sheeler, D, Siegel, A** and **Weide K** 2009 Extensible component-based architecture for FLASH, a massively parallel, multiphysics simulation code. *Parallel Computing*, 35(10–11): 512–522. DOI: http://dx.doi.org/10.1016/j.parco.2009.08.001
2. **Colella, P, Graves, D T, Keen, N D, Ligocki, T J, Martin, D F, McCorquodale, P W, Modiano, D, Schwartz, P O, Sternberg, T D** and **Van Straalen, B** 2009 Chombo software package for AMR applications design document. Technical report, Lawrence Berkely National Laboratory, Applied Numerical Algorithms Group, Computational Research Division.
3. **Berger, M J** and **Colella, P** 1989 Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82(1): 64–84. DOI: http://dx.doi.org/10.1016/0021-9991(89)90035-1
4. **Cornford, S L, Martin, D F, Graves, D T, Ranken, D F, LeBrocq, A M, Gladstone, R M, Payne, A J, Ng, E G** and **Lipscomb, W H** 2011 Adaptive mesh, finite-volume modeling of marine ice sheets. *Journal of Computational Physics*, [submitted].
5. **Molins, S, Trebotich, D, Steefel, C I** and **Shen, C** 2011 An investigation of the effect of pore scale flow on average geochemical reaction rates using direct numerical simulation. *Water Resources Research*, [submitted].
6. **Miniati, F** and **Colella, P** 2007 Block structured adaptive mesh and time refinement for hybrid, hyperbolic + N-body systems. *Journal of Computational Physics*, 227(1): 400–430, 2007. DOI: http://dx.doi.org/10.1016/j.jcp.2007.07.035
7. **Miniati, F** and **Martin, D F** 2011 Constrained-transport magnetohydrodynamics with adaptive mesh

refinement in CHARM. *The Astrophysical Journal Supplement Series*, 195(1): 5. DOI: http://dx.doi.org/10.1088/0067-0049/195/1/5

8. **Chombo** - software for adaptive solutions of partial differential equations. Available at: https://commons.lbl.gov/display/chombo/Applications
9. **Dubey, A, Antypas, K, Calder, A C, Daley, C, Fryxell, B, Gallagher, J B, Lamb, D Q, Lee, D, Olson, K, Reid, L B, Rich, P, Ricker, P M, Riley, K M, Rosner, R, Siegel, A, Taylor, N T, Timmes, F X, Vladimirova, N, Weide, K** and **ZuHone, J** 2013 Evolution of FLASH, a multiphysics scientific simulation code for high performance computing. *International Journal of High Performance Computing Applications*. DOI: http://dx.doi.org/10.1177/1094342013505656
10. **Dubey, A, Weide, K, Lee, D, Bachan, J, Daley, C, Olofin, S, Taylor, N, Rich, P M** and **Reid, L B** 2013 Ongoing verification of a multiphysics community code: FLASH. *Software: Practice and Experience*. DOI: http://dx.doi.org/10.1002/spe.2220.

How to cite this article: Dubey, A and Van Straalen, B 2014 Experiences from Software Engineering of Large Scale AMR Multiphysics Code Frameworks. *Journal of Open Research Software,* 2(1): e7, pp. 1-5, DOI: http://dx.doi.org/10.5334/jors.am

Published: 9 July 2014

]u[*Journal of Open Research Software* is a peer-reviewed open access journal published by Ubiquity Press.

OPEN ACCESS