Extensible Component Based Architecture for FLASH, A Massively Parallel, Multiphysics Simulation Code

A. Dubey*[,a], Katie Antypas[b], Murali K. Ganapathy[c], Lynn B. Reid[a], Katherine Riley[d], Dan Sheeler[d], Andrew Siegel[d], Klaus Weide[a]

[a] *ASC/Flash Center, The University of Chicago, 5640 S. Ellis Ave, Chicago, IL 60637*
[b] *Lawrence Berkeley National Laboratory, 1, Cyclotron Road, Berkeley, CA 94720*
[c] *Google Inc.*
[d] *Argonne National Laboratory, 9700 S. Cass Ave, Argonne, IL, 60439*

Abstract

FLASH is a publicly available high performance application code which has evolved into a modular, extensible software system from a collection of unconnected legacy codes. Its newest incarnation, FLASH3, consists of interoperable modules that can be combined to generate different applications. Such component-based architectures have historically met with varying degrees of success. FLASH has been successful because its capabilities have been driven by the needs of the scientific applications, without compromising maintainability, performance, and usability. The FLASH architecture allows arbitrarily many multiple alternative implementations of its components to co-exist and interchange with each other, resulting in greater flexibility. Further, a simple and elegant mechanism exists for customization of code functionality without the need to modify the core implementation of the source. A built-in unit test framework providing verifiability, combined with

*Corresponding author
Email address: dubey@flash.uchicago.edu (A. Dubey)

a rigorous software maintenance process, allow the code to operate simultaneously in the dual mode of production and development. In this paper we describe the FLASH3 architecture, with emphasis on solutions to the more challenging conflicts arising from solver complexity, portable performance requirements, and legacy codes. We also include results from user surveys conducted in 2005 and 2007, which highlight the success of the code.
Key words: Software Architecture, Portability, Extensibility, Massively parallel, FLASH

1. Introduction

The ASC/Flash Center at the University of Chicago has developed a public domain astrophysics application code, FLASH (8; 4). FLASH is component-based, parallel, and portable, and has a proven ability to scale to tens of thousands of processors. The FLASH code was developed under contract with the Department of Energy ASC/Alliances Program. It is the flagship Computer Science product of the Flash Center, resulting from over 10 years of research and development. One of the mandates of the Flash Center was the delivery of a parallel, scalable, and highly-capable community code for astrophysics. Motivation for the code effort lay in the increasing complexity of astrophysical simulations. The traditional academic approach of developing numerical software in piecemeal was deemed inadequate to meet the science needs.

Another aim of the Flash Center was to shift the paradigm of theoretical research towards working in multidisciplinary teams with scientific codes that are developed with modern software practices prevalent in the commercial

world. The FLASH code has now reached a level of maturity where it has a large number of users, many external to the University of Chicago. Moreover, it also has a substantial number of external code contributors. The number of requests for download, and the number of publications using the FLASH code, have grown superlinearly in recent years. This success was achieved by carefully balancing the often conflicting requirements of physics, software engineering, portability, and performance.

From its inception, FLASH has simultaneously been in development and in production mode. Its evolution into a modern component-based code has taken a path very different from that of most scientific computing frameworks such as SAMRAI, CACTUS, and POOMA (21; 5; 10; 11; 13; 18; 16). Those efforts developed the framework first, followed by the addition of solvers and other capabilities. An alternative path taken by scientific application codes such as Enzo, SWMF, and Athena (17; 20; 9) is to grow into a large application from smaller solvers and applications. Both models of development have their advantages and disadvantages: codes initialized with frameworks have superior modularity and maintainability, while codes begun with solvers generally deliver better performance for their target applications. FLASH straddles both approaches.

In the first released version, the development followed the solvers-first model, but later versions place more emphasis on modularity, maintainability, and extensibility. The outcome of this duality in development is that FLASH has more capabilities and customizability, and it reaches a much wider community than most scientific application codes. FLASH has gained wide usage because the capabilities of the code have been driven by physics,

while its architecture is driven by extensibility and maintainability. The addition of new solvers to FLASH is almost always dictated by the needs of users' applications. The solvers for multiphysics applications tend to put severe strain on any modern object-oriented software design. Lateral data movement is normally required between different solvers and functional units, which makes resolving data ownership and maintaining encapsulation especially challenging. Also, many of the core physics solvers are legacy third-party software written in Fortran, which are rarely modular. While modularity, flexibility, and extensibility are some of the primary guiding principles in the code architecture design, these goals often conflict with the equally important considerations of efficiency and performance. Additionally, since high performance platforms usually have a relatively short lifespan, the need for performance portability places even more constraints on the design process. Achieving a balance between these conflicting goals while retaining the very complex multiphysics capabilities has been the biggest contributor to the widespread acceptance of the FLASH code.

The FLASH model of development and architecture is informed by the literature from the common component architecture effort (12; 1). Since the project's inception, FLASH has undergone two major revisions, both of which included significant architectural and capabilities improvements. FLASH has always striven for a component-based architecture, but this goal was not realized in the first version because of a strong emphasis on producing early scientific results using legacy codes. However, foundations for a component-based architecture were firmly laid in the first version FLASH1.6 (8) by providing wrappers on all the solvers and minimizing lateral com-

munication between different solvers. The second versions, FLASH2.0 – FLASH2.5, built upon this foundation by addressing data ownership and access, resulting in a centralized data management approach. Finally, the current version, FLASH3, has realized a true component-based architecture with decentralized data management, clean interfaces, and encapsulation of functional units. FLASH3 also has well-defined rules for inheritance within a unit and for interactions such as data communication between units.

This latest release contains over 380,000 lines of code (with over 138,000 additional lines of comments). More than 35 developers and researchers have contributed to all versions of the FLASH code. As the complexity of the code and the number of developers have grown, code verification and management of the software development process have become increasingly important to the success of the project. The FLASH3 distribution now includes a unit test framework and its own test-suite, called FlashTest, which can be used for professional regression testing.

In this paper we describe the FLASH3 architecture, with emphasis on solutions to the more challenging conflicts arising from solver complexity, portable performance requirements, and legacy codes. We also include results from user surveys conducted in 2005 and 2007, indicating how the architecture choices have led to the widespread acceptance of the FLASH code.

2. Architecture Cornerstones

FLASH is not a monolithic application code; instead, it should be viewed as a collection of components that are selectively grouped to form various

applications. Users specify which components should be included in a simulation, define a rough discretization/parallelization layout, and assign their own initial conditions, boundary conditions, and problem setup to create a unique application executable. In FLASH terminology, a component that implements an exclusive portion of the code's functionality is called a *unit*. A typical FLASH simulation requires a proper subset of the units available in the code. Thus, it is important to distinguish between the entire FLASH source code and a given FLASH application.

The FLASH architecture is defined by four cornerstones: unit, configuration layer, data management, and interaction between units. Here we describe the four cornerstones briefly.

2.1. Unit

A FLASH unit provides well-defined functionality and conforms to a structure that facilitates its interactions with other units. A unit can have interchangeable implementations of varying complexity, as well as subunits that effect subsets of the unit's functionality. Each unit defines its Application Programming Interface (API), a collection of routines through which other units can interact with it. Units must provide a null implementation for every routine in their API. This feature permits an application to easily exclude a unit without the need to modify code elsewhere. For example, the input/output unit can be easily turned on and off for testing purposes.

FLASH units can be broadly classified into five functionally distinct categories: infrastructure, physics, driver, monitoring, and simulation. The infrastructure category includes the units responsible for housekeeping tasks such as the management of runtime parameters, the handling of input and output

to and from the code, and the administration of the solution mesh. Units in the physics category implement algorithms to solve the equations describing specific physical phenomena, and include units such as hydrodynamics, equations of state, and gravity. The driver unit implements the time advancement methods, initializes and finalizes the application, and controls most of the interaction between units included in a simulation. The monitoring units track the progress and performance of a simulation. The simulation unit is of particular significance; it defines how a FLASH application will be built and executed. It also provides initial conditions and the simulation-specific runtime parameters for the application. The simulation unit has been designed to enable customization of the FLASH code for specific applications without modifying other units, as explained in Section 3.5. Additional detail on the unit architecture is provided in Section 3.

2.2. Configuration Layer

FLASH implements its inheritance, extensibility, and object-oriented approach through its configuration layer. This layer consists of a collection of text *Config* files that reside at various levels of the code organization, and the *setup* tool which interprets the Config files. The two primary functions of the this layer are to configure a single application from the FLASH source tree, and to implement inheritance and customizability in the code. The Config files for a unit contain directives that apply to everything at, or below, that hierarchical level, and describe its dependencies as well as variables and runtime parameters requirements. The setup tool parses the relevant Config files, starting with the one for the Simulation unit described in Section 3.5. Dependencies are recursively resolved to configure individual units

needed for the application. Remember that each application requires different sections of code and produces a distinct executable. This method of configuration avoids an unnecessarily large binary and memory footprint, as only the needed sections of code are included. It also enables extensibility, since the inclusion of a new unit, or a new implementation of a unit, need become known only to the Config file of the specific problem setup in the simulation unit.

FLASH exploits the Unix directory structure for inheritance, instead of relying on language-provided infrastructure. Due to the relatively short life of high performance machines, and the code's wide external user base, portability is of paramount importance for FLASH. However, many of the more important physics solvers in the code are legacy third-party codes implemented in Fortran, and Fortran interfaces with other higher-level languages compromise portability. Additionally, supporting easy import of new third-party software into the code also precludes reliance on a specific language to provide inheritance. FLASH's solution of using the Unix directory structure with text annotations in the Config files has the triple advantage of being simple, extensible, and completely portable.

2.3. Data Management

In a large multiphysics code with many solvers, management and movement of data is one of the biggest challenges. Legacy solver codes rarely address resolving the ownership of data by different sections of code, a necessity for encapsulation and modularity. During the first round of modernization in the second version of FLASH, the data management was centralized into a separate unit to unravel the legacy code. This technique is also the data

management model followed by SAMRAI (5). The centralized data management extracted all the data from the individual units, and ensured data coherency by eliminating any possibility of replication. The main drawback of this approach was that it gave equal access to all units for data fetching and modification. Thus a unit could get mutator access to data that it should never have modified. The onus was on the developer to find out the scope of each data item being fetched and to make sure that the scope was not violated. This responsibility limited the ability to add more functionality to the code to those who knew the code very well, a serious handicap to extensibility.

FLASH Version 3 takes the next and final step in modularizing data management by decentralizing the data ownership. Every data item in the code belongs to exactly one unit. The owner unit has complete control over the scope and modifiability of the data item while the non-owner units can access or mutate the data only through the unit's API functions. Additionally, the scope of data within a unit can vary. Thus for example a data item specific to a subunit is visible only to that subunit, while unit scope data is visible to all functions in the unit.

2.4. Interactions between Units

The interactions between units are governed by both the *Driver* unit and the published APIs of the individual units. The Driver unit is responsible for initializing all the included units and the meta-data for the application as a whole. The Driver unit implements the time-stepping scheme of the application, and hence dictates the order in which the units are initialized and invoked, and how they interact with each other. Recall that units have

null implementations, a feature that allows a comprehensive implementation of the Driver unit. Once a unit is invoked by the driver, it can also interact with other units through their API. The Driver unit also cleanly closes the units and the application when the run is complete.

3. Unit Architecture

Of the four cornerstones of the FLASH architecture, the unit structure is the most complex. Unit architecture separates the computational kernel from the public interfaces, and controls the scope of various data items owned by the unit. A detailed description of the unit architecture is therefore critical to understanding the overall structure and software methodology of the FLASH code. Subunits are an important and novel feature of the unit architecture detailed below. In addition to the unit architecture, we also describe some of the infrastructure units, and the *Simulation* unit, since these play an important role in the code architecture.

The unit itself has three layers. The outer layer, the API, defines the full functionality of the unit. A unit's API can be viewed as having two sections: one for making its private data available to the other units, and another which defines its capabilities for modifying the state of the simulation. The inner layer of the unit is known as the kernel, and implements the full functionality. The middle layer implements the architecture, and acts as conduit between the outer and inner layers. It hides the knowledge of the FLASH framework and unit architecture from the kernel, and vice-versa, by providing wrappers for the kernel. The wrapper layer thus facilitates the import of third party solvers and software into FLASH.

3.1. Subunits

Units can have one or more subunits which are groupings of self-contained functionality. The wrapper layer starts with the definition of subunits. Subunits implement disjoint subsets of a unit's API, where none of the subsets can be a null set. The union of all subsets constituting various subunits must be exactly equal to the unit API. Every unit has at least a *Main* subunit that implements the bulk of the unit's functionality, including its initialization. The Main subunit is also the custodian of all the unit-scope data. The wrapper layer arbitrates on locating functions common to many alternative implementations of subunits such that code duplication is minimized and flexibility is maximized. The concept of subunits was developed to constrain the complexity of the code architecture, and to minimize the fragmentation of code units, which would result in proliferation of data access functions.

These difficulties are best illustrated with an example of interdependencies between the *Grid* unit, which manages the Eulerian mesh, and the *Particles* unit. Particles may be massless and passive, used to track the Lagrangian features of the simulation, or active particles with mass which can affect the gravitational fields. FLASH has four different functionalities related to particles, each of which can have multiple alternative implementations. The current FLASH release provides three methods for initial distribution of particles, four methods of mesh/particle mapping, two types of gravitational field interaction, and seven methods of time interpolation. This level of complexity is not limited to the Particles unit. The time integration of particles could result in their migration between blocks, or even processors. Similarly, regridding of the active mesh might require migration of particles. These

particle-related movements, are best handled by the Grid unit since it knows the topology of the Eulerian mesh, thereby retaining encapsulation of alternate unit implementations.

If FLASH were to solely follow the unit model of architecture described in Section 2.1, then separate units for particles distribution, mapping, integration and migration would be needed. Each of these units would need access to large amounts of data in the other units, thereby requiring many accessor-mutator functions. The concept of subunits very elegantly solves both the problems of data access and unit fragmentation through the introduction of a level of hierarchy in the unit's architecture. Thus in the *Particles* unit the *ParticlesInitialization* and *ParticlesMapping* subunits respectively deal with the initial spatial distribution and with mappings to and from the Eulerian grid, while the *ParticleMain* unit keeps the unit scope data and implements time integration methods. Each subunit can have several alternative implementations. Hence, subunits not only organize a unit into distinct functional subsets that can be selectively turned off, but also expand the flexibility of the code since implementations of different subunits can permute with each other and therefore can be combined in many different ways.

3.2. Lateral Data Movement

In addition to the subunits level functionality, the other major challenge posed by the interaction between solvers for multiphysics simulations is the need for lateral data movement, which makes resolution of data ownership and encapsulation extremely difficult. For instance, the calculation of the hydrodynamics equations is dependent upon the equation of state, and if gravity is included in the simulation, upon gravitational acceleration. Simi-

larly, within the hydrodynamics calculation, there is a need to reconcile the fluxes at a global level when adaptive meshing is being used. All of these operations require access to data which is owned by different units, and the obvious solution of giving a copy of the data to each unit as needed would severely compromise the performance. FLASH's solution to this challenge is to provide interfaces that allow for transfer back and forth between units, so that data copy can be replaced with arguments passing by reference. The challenge is then reduced to arbitration between units as to which one is best suited to implement the needed functionality.

3.3. Example Unit

As an example of the unit architecture, Figure 1 shows a section of the *Particles* unit in the FLASH code. The figure illustrates a combined use of directory structure for organizational and architectural purposes. Here, the square boxes represent a directory used as a namespace, while the shaded ovals represent directories used for organizing the code. The example does not represent the full implementation of the unit; it includes only those few sections that best highlight the features of the unit architecture. Two subunits shown in the figure are *ParticlesMain*, and *ParticlesInitialization*. ParticlesMain includes time-integration methods which are broadly categorized into methods for active and passive particles. Here the two subdirectories, active and passive, are used for organizational purpose only, and the figure shows examples of two different methods for each particle type. The *ParticlesInitialization* subunit has two implementations that are functionally grouped together, and two more that are not. The two implementations grouped together are based on density distribution, while the remaining two provide

either a lattice based distribution or a file-read initial distribution. Note that the subdirectory *WithDensity* is not an organizational directory because the two implementations below it share common functionality that they each inherit. Hence a function implemented in the *ParticlesInitialization* directory will be inherited by all four of the implementations, while one implemented in the *WithDensity* directory will be inherited only by the two methods that use density distribution. If another implementation of the same function occurs at any of the lower levels, it overrides the inherited implementation. Thus the inheritance in FLASH eliminates code duplication while retaining all the flexibility of having alternative implementations of the same function co-exist at several levels in the source tree.

3.4. Infrastructure Units

The infrastructure units in FLASH are responsible for discretization of the physical domain; reading, writing, and maintaining the data structures related to the simulation data; and other housekeeping tasks such as handling physical constants and runtime parameters. Of these, the most extensive responsibilities lie with the *Grid* unit, which manages the discretized mesh, and the input/output *I/O* unit, which reads and writes the data. These two units are also unique in that they share their data with each other; this exception to unit encapsulation is allowed for performance reasons. Here we describes these two units briefly (further discussion in (3) and (7)).

The Grid unit is the custodian of all the data structures related to the physical variables necessary for advancing the simulations. Every discrete point in the mesh has associated with it a number of physical variables, logical and physical coordinates, and an indexing number. On each processor,

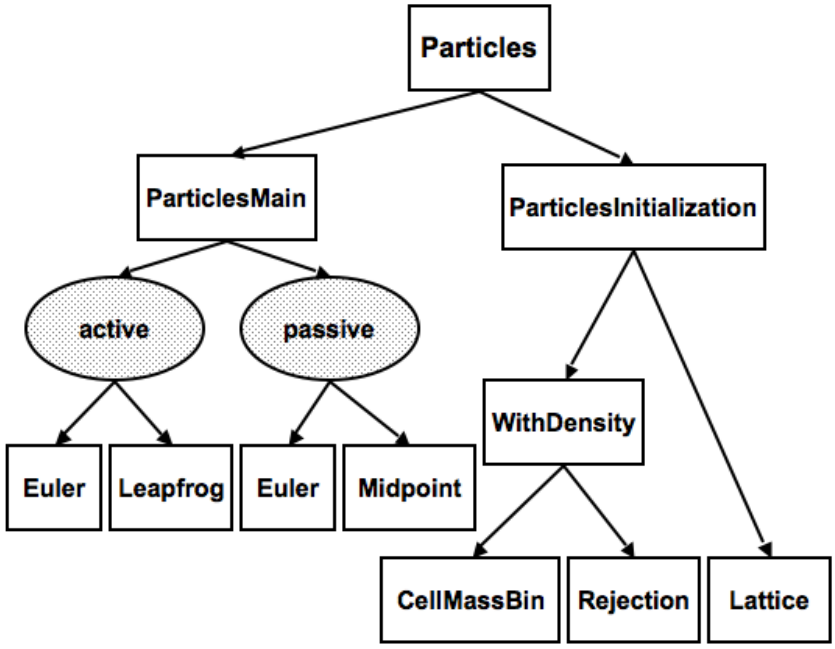

Figure 1: The unit hierarchy and inheritance. Square boxes represent unit namespaces; ovals represent organizational directories.

meta data exists such as the location in the physical domain and the number of discretization points per parallel grouping. FLASH3 has two different Grid implementations: a simple grid uniform in space and a block-structure adaptive oct-tree mesh. If Adaptive Mesh Refinement (AMR) is being used, blocks are created, destroyed, and distributed dynamically, and different blocks exist at varying levels of resolution, all of which must be tracked by the unit. The Grid unit is also responsible for keeping the physical variables consistent throughout the simulation. For example, when two adjacent blocks are at different resolutions, interpolation and prolongation ensure that conservation laws are not violated. Hence, the Grid unit is the most complex and extensive unit in the code, and most of the scaling performance of the code is determined by the efficiency of its parallel algorithms.

In FLASH, more than 90% of the reading or writing of data to the disk is controlled by the I/O unit. FLASH outputs data for checkpointing and analysis. The checkpoints save the complete state of the simulation in full precision so that simulations can transparently restart from a checkpoint file. The analysis data is written in many formats. The largest of these are the plotfiles, which record the state of the physical variables. Quantities integrated over the entire domain are written from the master processor into a simple text file. The only input controlled by the I/O unit is the reading of checkpoint files. Other forms of input, such as reading in a table of initial conditions needed by a specific simulation, are managed by the unit in question. FLASH is one of the relatively few applications codes that have support for multiple I/O libraries, such as HDF5 (15) and PnetCDF (14; 6), where all processors can write data to a single shared file.

3.5. Simulation Unit

The *Simulation* unit effectively defines the scientific application. Each subdirectory in the simulation unit contains a different application, which can be viewed as a different implementation of the Simulation Unit. This unit also provides a mechanism by which users can customize any part of their application without having to modify the source code in any other unit.

An application can assume very specific knowledge of units it wants to include and can selectively replace functions from other units with its own customized ones by simply placing a different implementation of the function in its *Simulation* subdirectory. At configuration time, the arbitration rules of the setup tool cause an implementation placed in the simulation unit to override any other implementation of that function elsewhere in the code. Similarly, the simulation unit can also be aware of the runtime parameters defined in other units and can reset their default values. Additionally, FLASH does not limit applications to the functionality distributed with the code; an application can add functionality by placing its implementation in the Simulation subdirectory. The setup tool has the capability to include any new functionality thus added at the configuration time, without any prior knowledge of the functionality. Accordingly, by allowing great flexibility to the Simulation unit, FLASH makes it possible for users to quickly and painlessly customize the code for their applications. A typical use of this flexibility is in user-defined boundary conditions that may not have standard support in FLASH. Another frequently customized functionality is control of refinement when using the AMR adaptive grid mode.

4. Code Maintenance

While a clear architecture design is the first step in producing a useful code, the FLASH code is not static and continues to develop based on internal pressures and external requests and collaborations. As the code gains maturity, regular testing and maintenance become crucial. Maintenance of the FLASH code is assisted by guidelines for all stages in the code lifecycle, some of which are enforced and others are strongly encouraged.

4.1. Unit Test Framework

In keeping with good software practice, FLASH3 incorporates a unit test framework that allows for rigorous testing and easy isolation of errors.The implementation of a new code unit or subunit is usually accompanied by the creation of one or more corresponding unit tests. Where possible, the unit tests compare numerical results against known analytical or semi-analytical solutions which isolate the new code module.

The components of the unit test reside in two different places in the FLASH source tree. One is a dedicated path in the Simulation unit, where the specific unit test acts as an ordinary Simulation. The other is a subdirectory called unitTest, located within the hierarchy of the corresponding unit, which implements the actual test and any helper functions it may need. These functions have extensive access to the internal data of the unit being tested. By splitting the unit test into two locations in the source tree, unit encapsulation is maintained.

4.2. Documentation

FLASH's clean architecture is well documented, which enables easy extention by external contributors (3). For all routines defining the interface of a module, a well documented header is a code requirement. The developers are also strongly encouraged to include extensive in-line documentation in addition to a header describing each routine they implement. FLASH uses Robodoc (22; 19) for automatic generation of documentation from internal headers. Compliance with code regulations such as documentation and good coding practice is checked through scripts that run nightly.

In addition, rapidly executing example problems are provided in the public release of FLASH. Availability of a collection of example problems that a first-time user can set up and run in an hour or less has been cited as one of the more attractive features of FLASH in a code survey (see Section 5. FLASH comes with a User's Guide, a Developer's Guide, on-line howtos, on-line quick reference tips, and hyperlinks to full descriptions with examples of all the routines that form the public interfaces of various units (2; 3). All of these user-assistance components are available on-line, as is the current release. In addition, there is an active email User's Group where support questions are addressed by both developers and knowledgeable active users.

5. User Survey

The FLASH Code has attracted a wide range of users and has become a premier community code preeminent in, but not limited to, the astrophysics community. Many users cite FLASH's capabilities, ease of use, scalability, modularity, and extensive documentation as the key reasons for their use of

FLASH. A code survey performed in 2005 followed by another in 2007 found that the close to three hundred users who cumulatively responded to the survey use the code in three major ways. The first group uses FLASH as a primary research tool for a broad range of application areas, including high-energy astrophysics, cosmology, stars and stellar evolution, computational fluid dynamics (CFD), and algorithm development. The second group of users employ the FLASH code for verification and validation (V&V). These users primarily attempt to compare FLASH to other codes or use FLASH as a benchmark. Still others in this V&V group port FLASH to new machines to test compilers, libraries, and performance. Finally, the third group uses FLASH as a sample code or for educational purposes.

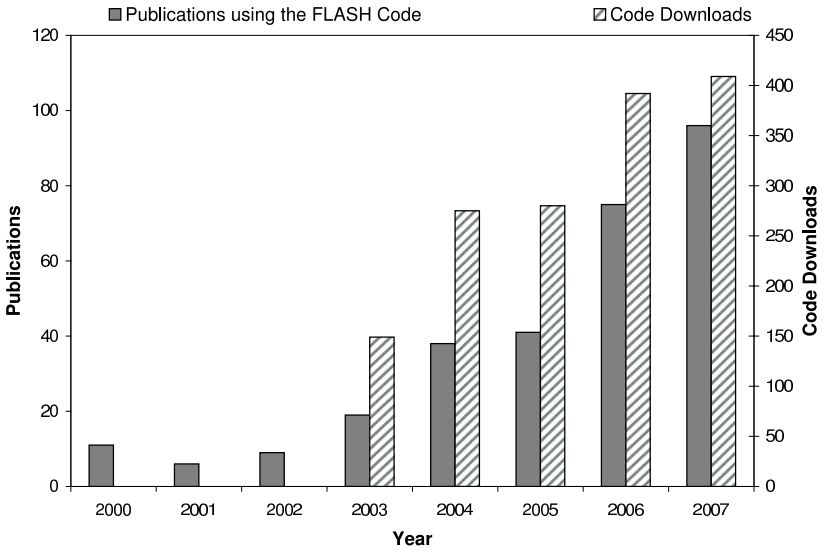

Figure 2: Yearly number of publications in which the FLASH code was used (left dark bars) and FLASH downloads (right striped bars). The jump in downloads in 2006 followed the release of the α version of FLASH3, the newest version of the code. The FLASH code has been downloaded more than 1700 times, and has been used in more than 320 publications.

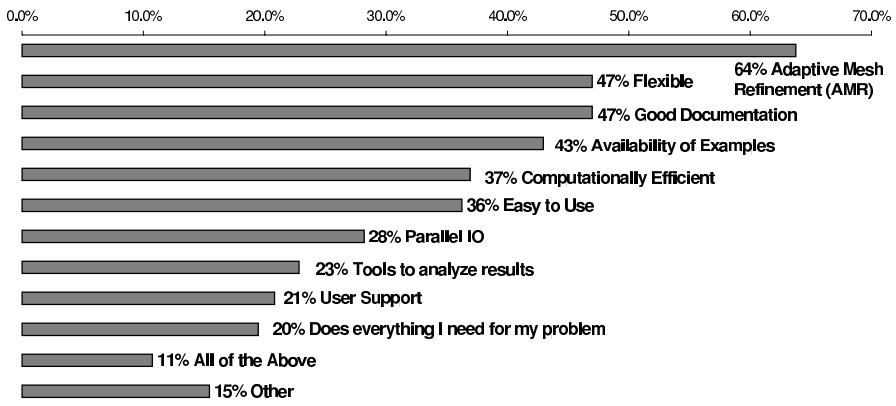

Figure 3: Results from a FLASH users survey in 2007: Reasons cited for FLASH usage.

The results of the survey clearly indicate that FLASH enjoys wide acceptance among the researchers from many fields. FLASH has been downloaded more than 1700 times and used in more than 320 publications, by both Center members and external users. Figure 2 shows that both the number of code downloads and the number of publications has steadily grown as the code has matured. Figure 3 shows that while the presence of adaptive mesh refinement is the top reason cited for using FLASH, it is the only one in the top six reasons that relates to the capabilities of the code. The remaining five top reasons pertain to the code architecture and its software process. These reasons include flexibility, ease of use, and performance, thus vindicating the architectural choices of FLASH.

6. Acknowledgments

We wish to thank all the past contributors to the FLASH code. The software described in this work was in part developed by the DOE-supported ASC / Alliance Center for Astrophysical Thermonuclear Flashes at the University of Chicago under grant B523820.

References

[1] Armstrong, R., Kumfert, G., McInnes, L., Parker, S., Allan, B., Sottile, M., Epperly, T., and Dahlgren, T. (2006). The CCA component model for high-performance scientific computing. Concurrency and Computation: Practice and Experience 18(2), 215–229.

[2] ASC Flash Center (2008). FLASH code support. `http://flash.uchicago.edu/website/codesupport/`.

[3] ASC Flash Center (2008). FLASH user's guide. `http://flash.uchicago.edu/website/codesupport/secure/flash3_ug/`.

[4] Calder, A. C., Fryxell, B., Plewa, T., Rosner, R., Dursi, L. J., Weirs, V. G., Dupont, T., Robey, H. F., Kane, J. O., Remington, B. A., Drake, R. P., Dimonte, G., Zingale, M., Timmes, F. X., Olson, K., Ricker, P., MacNeice, P., and Tufo, H. M. (2002). On validating an astrophysical simulation code. Astrophysical Journal, Supplement 143, 201–229.

[5] CASC (2007). SAMRAI structured adaptive mesh refinement application infrastructure. `https://computation.llnl.gov/casc/SAMRAI/`.

Center for Applied Scientific Computing, Lawrence Livermore National Laboratory.

[6] Chilan, C., Yang, M., Cheng, A., and Arber, L. (2006). Parallel I/O performance study with HDF5, a scientific data package. http://www.hdfgroup.uiuc.edu/papers/papers/ParallelIO/ParallelPerformance.pdf.

[7] Dubey, A., Reid, L., and Fisher, R. (2008). Introduction to FLASH 3.0, with application to supersonic turbulence. Physica Scripta (To appear).

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

[9] Gardiner, T. A. and Stone, J. M. (2005). An unsplit Godunov method for ideal MHD via constrained transport. J. Comput. Phys. 205(2), 509–539.

[10] Hornung, R. and Kohn, S. (2002). Managing application complexity in the SAMRAI object-oriented framework. Concurrency and Computation: Practice and Experience 14(5), 347–368.

[11] Hornung, R. D., Wissink, A. M., and Kohn, S. R. (2006). Managing complex data and geometry in parallel structured AMR applications. Engineering with Computers 22, 181–195.

[12] Hovland, P., Keahey, K., McInnes, L. C., Norris, B., Diachin, L. F.,

and Raghavan, P. (2003). A quality of service approach for high-performance numerical components. In Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference. Toulouse, France.

[13] Ko, S., Cho, K. W., Song, Y. D., Kim, Y. G., Na, J., and Kim, C. (2005). Development of Cactus driver for CFD analyses in the grid computing environment. In Advances in Grid Computing - EGC 2005. vol. 3470, pp. 771–777.

[14] Li, J., Liao, W., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., and Zingale, M. (2003). Parallel netCDF: A high-performance scientific I/O interface. Supercomputing, 2003 ACM/IEEE Conference , 39–39.

[15] NCSA (2008). Heirarchical Data Format 5. `http://hdf.ncsa.uiuc.edu/HDF5/`.

[16] Oldham, J. (2002). Scientific computing using POOMA. C++ Users Journal 20(11), 6–23.

[17] O'Shea, B. W., Bryan, G., Bordner, J., Norman, M. L., Abel, T., Harkness, R., and Kritsuk, A. (2005). Introducing Enzo, an AMR cosmology application. In Plewa, T., Timur, L., and Weirs, V. (eds.) Adaptive Mesh Refinement – Theory and Applications. Springer, vol. 41 of *Lecture Notes in Computational Science and Engineering*.

[18] Reynders, J., Hinker, P., Cummings, J., Atlas, S., Banerjee, S., Humphrey, W., Karmesin, S., Keahey, K., Srikant, M., and Tholburn,

M. (1996). POOMA: A framework for scientific simulations on parallel architectures. Parallel Programming using C++ .

[19] Slothouber, F. (2007). Automating software documentation with ROBODoc. http://www.xs4all.nl/~rfsber/Robo/robodoc.html.

[20] Toth, G., Sokolov, I., Gombosi, T., Chesney, D., Clauer, C., De Zeeuw, D., Hansen, K., Kane, K., Manchester, W., Oehmke, R., et al. (2005). Space Weather Modeling Framework: A new tool for the space science community. J. Geophys. Res 110, 12–226.

[21] Wissink, A. and Hornung, R. (2000). SAMRAI: A framework for developing parallel AMR applications. In 5th Symposium on Overset Grids and Solution Technology. Davis, CA, pp. 18–20.

[22] Worth, D. and Greenough, C. (2005). A survey of available tools for developing quality software using Fortran 95. Technical report RAL-TR-2005, SFTC Rutherford Appleton Laboratory, SESP Software Engineering Support Programme. Available at http://www.sesp.cse.clrc.ac.uk/html/Publications.html.