View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

The Astrophysical Journal Supplement Series, 201:27 (11pp), 2012 August doi:10.1088/0067-0049/201/2/27
© 2012. The American Astronomical Society. All rights reserved. Printed in the U.S.A.

IMPOSING A LAGRANGIAN PARTICLE FRAMEWORK ON AN EULERIAN HYDRODYNAMICS INFRASTRUCTURE IN FLASH

A. Dubey[1,4,5], C. Daley[1,4,5], J. ZuHone[2], P. M. Ricker[3], K. Weide[1], and C. Graziani[1]
[1] Flash Center for Computational Science, Astronomy & Astrophysics, University of Chicago, Chicago, IL 60637, USA
[2] Astrophysics Science Division, Code 662, NASA/Goddard Space Flight Center, Greenbelt, MD 20771, USA
[3] Department of Astronomy, University of Illinois, Urbana, IL 61801, USA
Received 2012 February 28; accepted 2012 June 13; published 2012 July 20

ABSTRACT

In many astrophysical simulations, both Eulerian and Lagrangian quantities are of interest. For example, in a galaxy cluster merger simulation, the intracluster gas can have Eulerian discretization, while dark matter can be modeled using particles. FLASH, a component-based scientific simulation code, superimposes a Lagrangian framework atop an adaptive mesh refinement Eulerian framework to enable such simulations. The discretization of the field variables is Eulerian, while the Lagrangian entities occur in many different forms including tracer particles, massive particles, charged particles in particle-in-cell mode, and Lagrangian markers to model fluid–structure interactions. These widely varying roles for Lagrangian entities are possible because of the highly modular, flexible, and extensible architecture of the Lagrangian framework. In this paper, we describe the Lagrangian framework in FLASH in the context of two very different applications, Type Ia supernovae and galaxy cluster mergers, which use the Lagrangian entities in fundamentally different ways.

Key words: galaxies: clusters: general – hydrodynamics – methods: numerical – supernovae: general

Online-only material: color figures

1. INTRODUCTION

Simulations of astrophysical phenomena are some of the most challenging multiphysics computations. For example, supernova (SN) simulations primarily rely upon compressible hydrodynamics for modeling gas, but they also need specialized equations of state, elliptic solvers for computing gravitational potentials, source term networks for computing nuclear reactions, and many other solvers depending upon the specific target of the simulation. All of the above models work well with Eulerian discretization; however, explicitly advancing large nuclear networks inline is prohibitively expensive because of its impact on the time step. One alternative is to use tracer particles to record the thermodynamic history of mass elements and then post-process the time history to compute the nucleosynthetic yield from the simulation. This necessitates support for Lagrangian entities in the code. A similar requirement for Lagrangian support arises in cosmological simulations where there is need to model gas, stars, and dark matter.

Many commonly used codes in the astrophysics community opt to have a purely particle-based treatment: examples include Gadget (Springel 2005), Gasoline (Wadsley et al. 2004), Hydra (Couchman et al. 1995; Pearce & Couchman 1997), and SEREN (Hubber et al. 2011).[6] Other codes have support for both Eulerian and Lagrangian entities, for example FLASH (Fryxell et al. 2000; Dubey et al. 2009), COSMOS (Ricker et al. 2000), Enzo (O'Shea et al. 2005a; Norman et al. 2008), RAMSES (Teyssier 2002), ART (Kravtsov et al. 1997), NYX (A. S. Almgren et al. 2012, in preparation), and CHARM (Miniati & Colella 2007). These two classes of codes have been employed to model a variety of physical situations, including the cosmological structure formation, the formation of solar system objects, SN explosions, star formation, and binary collisions of compact stellar objects, to name only a few of many possible examples. Many codes of each class focus on one class of applications and optimize their implementations accordingly. The advantage is that many simplifying assumptions become possible, thereby decreasing the complexity of the application infrastructure design. However, FLASH takes the approach of targeting a wider community of researchers, ranging from simulations of novae, SNe, cosmological structure formation, and galaxy clusters to high energy density physics and fluid–structure interactions. Each of these communities has its own requirements for the Lagrangian infrastructure in the code, although some features are common to all. Because of this, the Lagrangian framework in FLASH has to be general, flexible, and easily customizable. In this paper, we describe the design considerations and architecture of FLASH's Lagrangian framework. We also demonstrate the usability and effectiveness of the design through its use in two vastly different applications. The Lagrangian entities are referred to as *particles* for convenience from here on.

The framework unifies the general, mostly similar characteristics of various incarnations of particles in FLASH into a few flexible, reusable, and well-encapsulated code modules. Examples of such characteristics include methods of initializing particle positions and velocities, time advancement schemes, ways of mapping physical variables to and from the Eulerian mesh, and movement of data structures that store the particles. It also provides hooks for those characteristics that are unique to a specific mode of particle usage, such as communicating the forces exerted by particles on the fluid when used in certain active modes. In keeping with the overall philosophy of the FLASH architecture, the Lagrangian framework also provides several representative alternative implementations of relevant code units, while permitting customization of any of these units by individual applications. The framework and capabilities described in this

[4] Computation Institute, University of Chicago and Argonne National Laboratory, USA.
[5] CELS, Argonne National Laboratory, USA.
[6] An alternative approach involves "moving-mesh" Lagrangian codes, AREPO (Springel 2011) being a primary example.

paper have evolved over several years, therefore many features are missing from earlier versions of the code. FLASH4-beta has the complete framework described here, though FLASH3.3 also has most of the features.

Section 2 describes the different classes of applications that use the Lagrangian capabilities of FLASH and the demands they place upon the Lagrangian framework in the code. Section 3 gives a synopsis of the code architecture features necessary for explaining the framework, followed by a description of the data structures and relevant code units that constitute the Lagrangian framework in Section 4. In Section 5, we describe the features and operation of the framework. Section 6 describes the use of the Lagrangian framework in simulations of Type Ia supernovae (SNe Ia) and galaxy cluster mergers, highlighting the commonalities and the differences in the usage. Finally, we conclude in Section 7.

2. LAGRANGIAN ENTITIES IN FLASH

The Lagrangian framework gets used in many different ways in the code. The simplest use is in the form of tracer particles that are carried along passively with the fluid flows and sample and record the state of their surrounding fluid elements (Beaudoin et al. 2007; Fisher et al. 2008; Federrath et al. 2008). Effectively this provides a Lagrangian view of physics within a model advanced using Eulerian equations. In terms of code components, this mode needs support for simple initialization of particle positions, a time advancement method, and mapping of mesh quantities to the appropriate particles. The mapping requires interpolation to particle positions from the cell-centered or face-centered coordinates of the mesh cells within which they lie.

The SN simulations use particles in a slightly more demanding tracer mode. During a simulation, the particles collect the thermodynamic time history of their associated mass elements, which is then post-processed to determine the nucleosynthetic yield (Jordan et al. 2008). The aim of this particle post-processing is to obtain nucleosynthetic abundances in photospherically significant regions of the final outflow, so that radiation transport calculations can accurately predict light curves and spectra. The final spatial distribution of the particles is of crucial importance, since it affects the accuracy with which the nucleosynthetic abundance distribution is known. However, the final particle distribution is not easy to predict or control based on position initialization. Hence, these simulations add the requirement of a more sophisticated position initialization method. Additionally, because particles can congregate in a very localized section of the domain during evolution, the demands on memory require that the mesh be able to adaptively refine based on particle count.

Particles can also operate in an active mode in which they influence the evolution of the mesh variables. For example, in many cosmological simulations, mass-endowed particles represent dark matter that interacts gravitationally with fluids representing baryonic matter (Efstathiou et al. 1985; Dolag et al. 2008). In this mode, several additional code capabilities are needed. Here, in addition to the mapping from the mesh variables described above, mapping from particles to mesh "smears" the mass carried by the particles on to a mesh variable. When used in parallel environments, the smearing may occur in parts of the domain that are on processor boundaries, thereby causing communication. Additionally, the forces exerted and experienced by the particles must be computed. A hybrid particle-in-cell (PIC) method[7] is another example of coupling between particles and mesh variables for evolution that requires some of the capabilities mentioned above. Here, electrons are treated as massless fluid and ions as discrete macroparticles, where each macroparticle represents a large number of real particles (Holmström et al. 2012). The method uses mapping to mesh capabilities to deposit ion charges and currents to the mesh, which are used in calculation of electrical and magnetic fields by the mesh.

In many simulations, it is desirable to differentiate between active particles (e.g., stars and dark matter). Some of these simulations also need to include tracer particles to track the Lagrangian characteristics. In some SN simulations, tracer particles need to be differentiated based upon whether they follow a mass section or a volume section. This would be simple if all particle types could use the same initialization, integration, and mapping methods. In reality, the required integration methods differ, as do the mapping and initialization methods. Therefore, the framework needs to allow for maximum interoperability among various alternatives for initialization, mapping, integration, and forces for multiple particle types to exist in one simulation.

3. FLASH ARCHITECTURE

FLASH is a scientific simulation code framework with a collection of modules featuring adaptive mesh refinement (AMR), efficient parallelization with good scaling behavior, and portability. FLASH is not a single application code, but rather provides modules that can be combined to generate many different multiphysics simulation applications. At the core of many FLASH applications are a Grid unit, typically representing an Eulerian view of a physical domain, and physics units representing the evolution of physical variables of interest on this grid. Together, they model a time-dependent solution of the equations of interest. A detailed description of the current FLASH architecture is given in Dubey et al. (2009). In this section, we include a synopsis of the features that are necessary to follow the discussion in this paper.

FLASH's functional components are organized in code modules called *units*. The units can be infrastructural such as Grid, I/O, or Multispecies, or they can be solvers for specific physics such as Hydro, Gravity, or Flame. Each unit publishes its application programming interface (API), which is used by all other units to interact with it. Units can be further organized into sub-units, such that each sub-unit implements a subset of the parent unit's API. The parts of the API implemented in different sub-units are mutually exclusive, i.e., each API function has one or more implementations in exactly one sub-unit. The union of all its sub-units implements the whole API of the unit. A sub-unit may have many alternative implementations, and each of these implementations can choose whether or not it can co-exist with any of the remaining implementations.

The unit architecture defines the scoping of various data items in use by the unit. Unit scope data items are placed in a single data module that is available to all functions in the unit, public or private. Other data items may have a more restricted scope, for example they may be limited to a sub-unit or to a specific implementation of a sub-unit. FLASH architecture is implemented through a combination of directives in FLASH specific syntax encoded in *Config* files, a setup script that parses

[7] Contributed by Mats Holmström (see Holmström et al. 2012) and the FLASH User's Guide for implementation details.

these files, and a set of naming conventions and inheritance rules imposed on top of the filesystem directory structure. Config files can exist at any level in the code. They encode the architectural and configuration information relevant to the level at which they appear. For example, a Config file in the *UnitMain* sub-unit may specify other required units, suggest some other units for inclusion, specify units/sub-units with which it is mutually exclusive, request storage for variables and fluxes, and define runtime parameters.

The *Simulation* unit has a special role; it defines individual applications by specifying the needed code units and providing other application-specific information such as initial conditions. Each individual application has its own directory in the Simulation unit where all the relevant data and application implementations are placed. The Simulation unit also provides a customization mechanism whereby an implementation of any function anywhere in the source tree of FLASH placed into the application's simulation directory replaces the default implementation during the configuration step. Thus, any section of the code can be customized without having to modify the base source tree.

While FLASH can support many different mesh packages, the common requirement is that they be block structured and Eulerian. This is because the Grid unit divides the physical domain into a set of blocks, the union of which covers the entire domain. The Grid unit presents a single block surrounded by a halo of guard cells as a self-contained computational domain to the solvers. Solvers get the list of blocks from the Grid unit and loop over and process them one at a time. If a solver needs to perform a global operation on the mesh, then it temporarily passes control to the Grid unit, which in turn returns the control back to the solver when it is done. Currently, FLASH supports PARAMESH (MacNeice et al. 2000), which is an oct-tree based AMR package as default; a Uniform Grid, which has uniformly spaced grid cells across the entire domain; and Chombo (Colella et al. 2009), a patch-based block-structured AMR package. Chombo is likely to become the default AMR package in FLASH in the future, but its implementation in FLASH is not production grade at the time of this writing.

4. CODE UNITS

The code sections relevant to the Lagrangian framework are distributed among three FLASH units. The *Particles* unit is the primary code unit that houses the bulk of the implementation. In addition, the *Grid* unit (the unit that manages the discretized Eulerian mesh) has a sub-unit, *GridParticles*, where all particles related functions that need knowledge of the Eulerian mesh reside. Similarly, the *IO* unit has a sub-unit, *IOParticles*, that handles all the I/O (checkpoint and analysis) for the Particles data structures.

The Particles unit is organized into four sub-units: *ParticlesMain*, *ParticlesInitialization*, *ParticlesForces,* and *ParticlesMapping*. As their names suggest, the division into sub-units reflects four broad categories into which particle capabilities can be organized. ParticlesMain contains implementations for time advancement of particles during simulation evolution. It also contains a unit-scope data module which has the base particle data structure, scratch space needed for processing, and other unit-scope data items, such as a processor's particle count. The data module is also made available to all other sub-units.

The ParticlesInitialization sub-unit contains several implementations for initial placement of discrete particles in the physical domain. In addition to the lattice and density based distribution described earlier, there is an example implementation for reading the initial positions from a file. The hybrid PIC method initializes particle positions from a uniform distribution and initial particle velocities from a Maxwellian distribution. The ParticlesMapping sub-unit provides interpolation functions to map physical quantities such as velocities from the Eulerian mesh to the particles. When used in active mode, it also provides means for mapping particle attributes to the appropriate mesh variables. Finally, the ParticlesForces sub-unit includes functions that communicate forces back and forth between the particles and the mesh.

The Particles sub-units need access to grid-specific information in order to function properly. In order to maintain unit encapsulation, this information is not available to the Particles unit directly. The Grid unit infrastructure has to provide needed information and services to all the Particles sub-units in a way that is independent of specific mesh implementations. The latter requirement arises from the presence of multiple mesh packages in the code. The GridParticles sub-unit addresses this challenge by further classifying the services provided into *GridParticlesMove*, *GridParticlesMapToMesh,* and *GridParticlesMapFromMesh* sub-units. These sub-units have common as well as mesh-specific implementations of the uniform interface through which the Particles sub-units receive service from the Grid unit. Services are provided by temporarily transferring control to the appropriate Grid sub-unit, passing Particles data structure as an argument, and returning the control back to the requesting Particles sub-unit when done.

The IOParticles sub-unit hides the details of both I/O libraries and the process of restarting from a checkpoint file from the Particles unit. For efficiency reasons, unlike other units in FLASH, the IO unit is given access to other units' internal data structures. Thus, the IOParticles sub-unit can directly access the particle data structure for the purpose of checkpointing. Otherwise, the Particles unit has very limited direct interaction with the IOParticles sub-unit. Moreover, in this instance, the Particles unit provides some services to the IOParticles sub-unit just as IOParticles sub-unit provides services to the Particles unit. More details of the IOParticles sub-unit can be found in the FLASH User's Guide.

4.1. Data Structures

Particles are organized in a two-dimensional array of double-precision floating-point numbers. This simplified choice of data structure facilitates easy movement both between code units for particles' evolution, and between blocks and processors when operating in a parallel environment. Additionally, the simple data structure enables multiple particle types to co-exist in a simulation.

One of the necessary particle properties is a permanent tag value which uniquely identifies a particle within a simulation. Though this property is not important during the simulation itself, it is critical in post-processing and analysis. Additional identifier properties include the block identification number (blockID) and processor number (procID) for the finest resolution block containing a particle. Since time advancement of particles depends upon interpolating relevant physical quantities from the closest mesh cell to the particle position (see Section 5.3 for details), for reasons of efficiency it is desirable to attach each particle to its overlapping block and have the particle reside on the same processor where its block is located. The GridParticlesMove sub-unit uses the combination of blockID and procID for migrating the particle's data structure

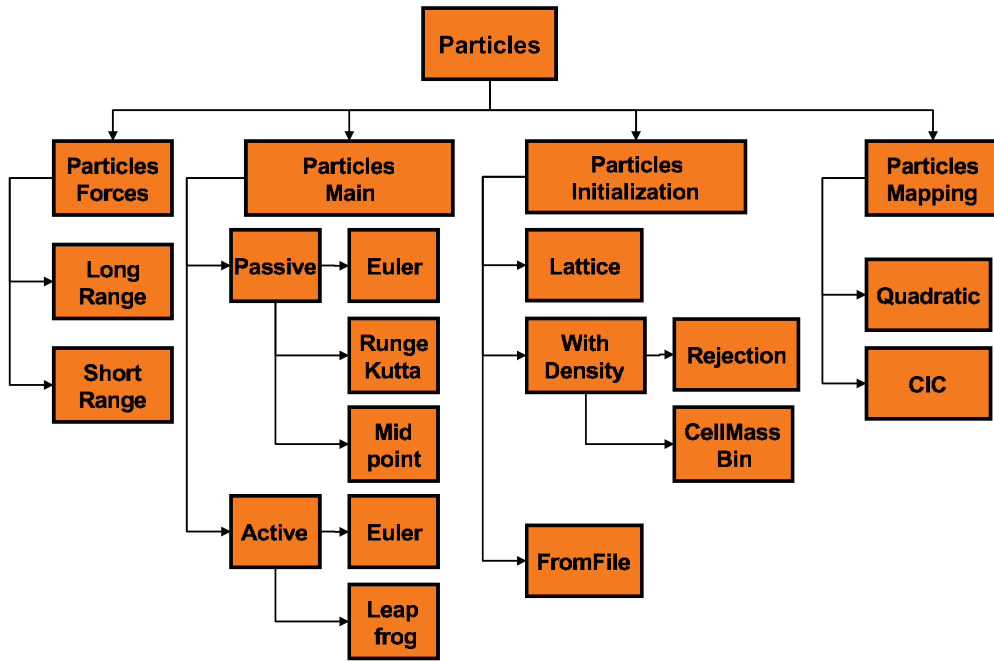

Figure 1. Schematic of the Particles unit.
(A color version of this figure is available in the online journal.)

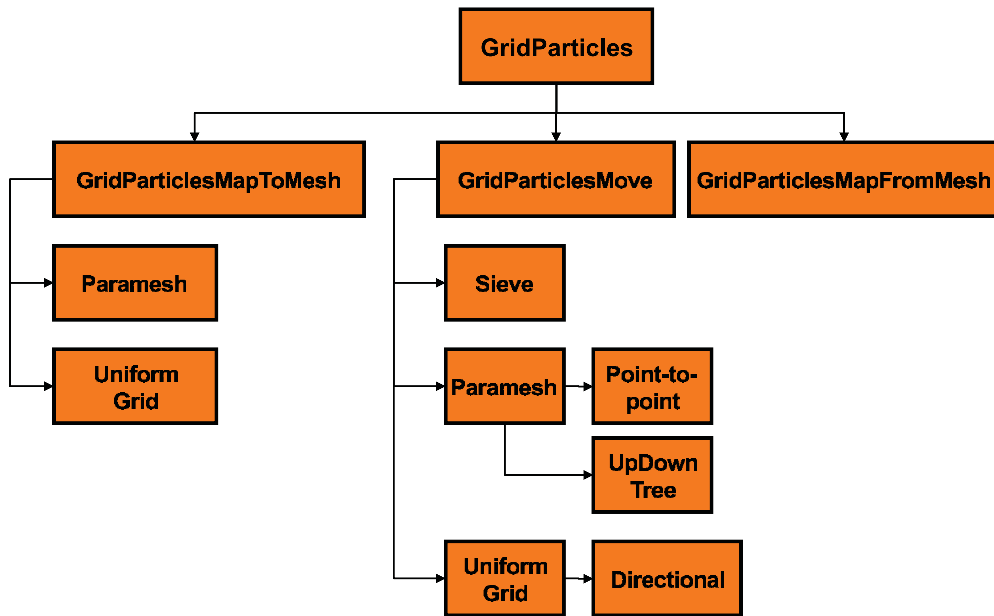

Figure 2. Schematic of the GridParticles sub-unit that manages all particle-related functions that need access to Grid owned data.
(A color version of this figure is available in the online journal.)

to the appropriate processor/block combination as its physical position changes with time evolution (Dubey et al. 2011).

The particle coordinates are represented by another three (for three-dimensional simulations) properties. Usually, there are also properties that store velocities and, for some time advancement schemes, additional sets of previous, estimated, or predicted coordinates and/or velocities. Other properties depend on the physics being modeled. Active particles interacting via gravity usually carry a mass property. Particles in hybrid PIC mode carry charge and several electrical and magnetic field properties. Passive particles carry additional properties for the fluid variables they are intended to trace. All the needed properties can be specified in appropriately placed Config files. When multiple particle types are used in a simulation, each particle carries a union of all properties requested by individual particle types. While this is not the most efficient storage choice, it precludes the necessity of multiple data structures which might lead to possible code duplication and parallel inefficiency. This choice also prevents proliferation of conditional branches in treating particle attributes. Branches are known to be error prone and in general add significant complexity in code verification.

5. FRAMEWORK

From the description above, it is clear that the constituent features of the Lagrangian framework are a data structure that

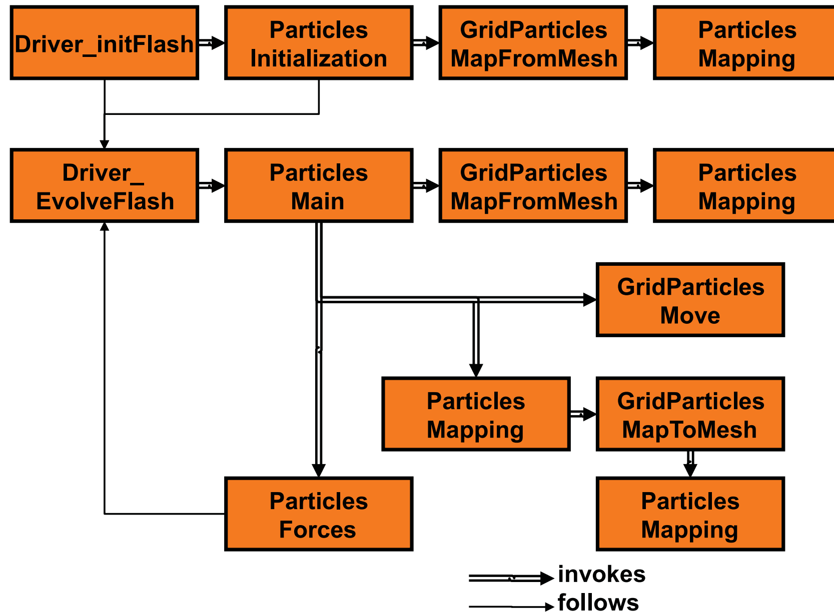

Figure 3. Schematic of control flow between Particles and Grid units during a simulation. Single line arrows indicate a routine that follows, while double line arrows represent a routine that is invoked.

(A color version of this figure is available in the online journal.)

encodes identifiers and properties into a single entity and a collection of code units that between them initialize, integrate, move, and map the particles and their attendant forces if applicable. In this section, we describe how these constituent features come together to provide a versatile Lagrangian framework atop an Eulerian mesh.

Figures 1 and 2 show the organization of the Particles unit and the particles related Grid sub-units, respectively. The schematic of the Particles unit does not show exhaustive implementations. Many time-integration schemes and the implementations related to charged particles are not included for the sake of clarity. Figure 3 shows an example of the control flow back and forth between the Particles unit and the GridParticles subunit during initialization and evolution. In the following sections, we describe these features and interactions in detail. The algorithms used in the GridParticlesMove and GridParticlesMapToMesh sub-units are described in detail in Dubey et al. (2011).

5.1. Initialization

In general, a particle comes into existence in the simulation environment during the initialization phase of a FLASH simulation. Particles can be created later in the simulation, though an explicit implementation of that ability is not included in the distribution. In order to add a new particle during a simulation, it is necessary to provision for extra space in the particle data structure, which is statically allocated at initialization. Once that is done, a particle can be added any time during a simulation by providing it with a new and unique tag number and populating all the necessary attributes such as position, velocity, block number, etc., as needed. Some methods of position initialization are local to the Particles unit, while some others require interaction with the Grid. The lattice-based method is an example of the former. An example of the latter is the method which reads in particle information from a file. There is no a priori knowledge about positions, and therefore block and processor association, when particle positions are read in. At a minimum, this method requires a redistribution of particles among blocks and processors after they are read in, invoking the GridParticlesMove sub-unit to migrate the particles to their destinations. In a more complex scenario, the particles may be congregated close together in the physical domain, overwhelming the memory of the processor during the read unless the mesh is refined. FLASH provides a particle-based refinement trigger that allows an interleaving of mesh and particles initialization similar to the following pseudocode:

```
Grid_initDomain {
 create initial blocks in mesh
 expand mesh :
   refine(on mesh criterion)
   apply initial conditions(all blocks)
 until (highest refinement level reached)}
 Particles_initPositions(done)
 while(not done)
   expand mesh:
      refine(on particles count)
      apply initial conditions(all blocks)
      Particles_initPositions(done)}

Particles_initPositions (done){
  get (maxLocalCount)
  for upto maxLocalCount
     read particle from file
  done = EOF(file)}.
```

The methods of particle position initialization included in the FLASH distribution are by no means exhaustive, and they can be easily customized by the user.

5.2. Integration

The Particles unit provides four methods for time advancement of tracer particles, and three methods for that of active particles in the massive and/or cosmology mode. The hybrid PIC method does not directly advance particles in time. The

trajectories of ions are computed from Lorentz forces which are then used to update the particle positions. The method implements a predictor-corrector leapfrog method with subcycling for the field update. All passive and massive particle time advancement implementations are standard first- or second-order time integration methods for solving ordinary differential equations. An interesting finding from our various implementations is that some predictor-corrector type second-order methods are not suitable when Δt varies between time steps. The difficulty arises when the integration method predicts the velocities and positions for t_{n+1} at time t_n based on the current Δt, but the time step limiters in the simulation change Δt between the two steps. When this happens, the predicted values are no longer valid, and integration must revert to the first-order Euler method. This can be especially bad in simulations with frequent changes in Δt where the time integration becomes closer to first order than second order. The implementations of other second-order methods, such as mid-point and Runge–Kutta, included in the FLASH distribution, do not have this problem.

5.3. Mapping

The Eulerian mesh in FLASH is discretized in the form of cells which combine together to form a block. The mesh variables are either cell centered or face centered, while particles can be located anywhere in a cell. Therefore, the Particles unit provides interpolation functions to map quantities from the coordinates of the overlapping cell to each particle's position. The two available methods are the quadratic interpolation and cloud-in-cell methods. Because the blocks have surrounding guard cells, this operation is completely local to a block. However, the interpolation methods need a lot of mesh-specific information, such as the cell widths Δx, Δy, Δz, and the physical coordinate bounds of the relevant block, which is most readily available in the Grid unit. Additionally, for efficiency, the particles are sorted on their blockID before processing. Here, sorting and gathering of block specific information is done in the GridParticlesMapFromMesh sub-unit, and then control is transferred to the ParticlesMapping sub-unit to apply the interpolation method.

When particles are used in active mode as massive particles, their contribution to the gravitational potential is computed by mapping the mass onto a mesh variable as an add-on density component. The mapping is achieved by smearing the mass of the particle onto the overlapping cell and some of the cells surrounding the overlapping cell. Other quantities of interest, depending on the problem, may be mapped to the mesh as well. Unlike mapping from the mesh, the mapping to mesh operation is not localized to a block. When a particle is close to a block boundary, some of the surrounding cells could be guard cells, and that information must be conveyed back to the neighboring block which has the corresponding interior cells. This is in effect a subset of an inverse guard-cell fill operation, and therefore it invokes additional coupling between the Particles and Grid units through the GridParticlesMapToMesh sub-unit. Reverse guard cell fill is not native to the mesh packages, and it is non-trivial for AMR meshes. The parallel algorithm has to find the destination while accounting for the fine-coarse boundary between blocks where applicable. FLASH includes two parallel algorithms, *Sieve* and *Point-To-Point*, for this purpose. Both the algorithms build upon similar algorithms used for moving the particle data structures between blocks and processors (see Dubey et al. 2011) and are described below.

5.3.1. Parallel Algorithms in Mapping

A particle's mass is deposited in the cells on the source block. When a guard-cell region (the halo of guard cells described in Section 3) has non-zero mass, its data are sent to the block containing the corresponding interior cells. Before sending, the source block makes use of locally available metadata to determine if the destination block is at a different refinement level. If so, then the appropriate restriction/interpolation is also performed at the source block. The destination blocks collect all applicable contributions from the source blocks and accumulate them in the appropriate interior cells. The destination blocks may or may not be off-processor.

Both the algorithms begin by identifying guard-cell regions that are off-processor. Where they differ is the communication pattern used to send the cells to their destinations. The sieve algorithm places all the cells along with their metadata into a sieve. The metadata includes the identity of the destination block and the location of the cells within that block. The "sieve" is rotated among processors until it is emptied. The specifics for creating sieves and the mechanics for rotating the sieves among processors efficiently are described in more detail in Dubey et al. (2011). The process of emptying the sieves involves finding a matching BlockID for the cells contained in the sieve. Every time a match is found, the corresponding cells are "dropped" from the sieve. The process terminates when all sieves have zero cells left. This method is relatively easy to understand and implement, and it is quite general. It does not rely upon the mesh package metadata for generating the communication pattern. For efficiency, it uses the mesh package specific information to encode the identity of the destination block at the source, though that encoding is not necessary for correct operation. However, it is not always very efficient. At best, its performance is comparable to that of the point-to-point algorithm, although in general, it is much less efficient.

The point-to-point algorithm exploits the knowledge of the mesh package's own communication pattern to deliver better performance. The disadvantage is that its implementation is tied to a specific mesh package. This algorithm identifies the neighboring processors that have the cells corresponding to the non-local guard-cell regions of any of the local blocks that have non-zero mass. A global operation lets each processor know the number of messages it should expect to receive. Each guard-cell region destined for an off-processor neighboring block is sent in a separate message, so that there are many small messages. Our strategy involves executing a busy-wait loop until sending and receiving criteria are satisfied. A process sends messages until it has visited each node in its send linked list, and a process receives messages until the count of messages received equals the number it received in the global operation. More details of the communication pattern and the busy-wait loop based parallel algorithm are described in Dubey et al. (2011) and Daley et al. (2012).

Figure 4 shows an example of the communication steps involved in the two algorithms through a two-dimensional setup with two different positions of a particle near a block boundary. The first column in the figure shows two locations of the particle with different regions of influence. The location at the top influences four blocks, while the one at the bottom influences only two blocks. The second column shows the communication steps in the sieve algorithm with the incoming sieve contents showing on every processor, while the third column shows communication in the point-to-point algorithm. The bold lines are the processor boundaries, and the gray area represents all

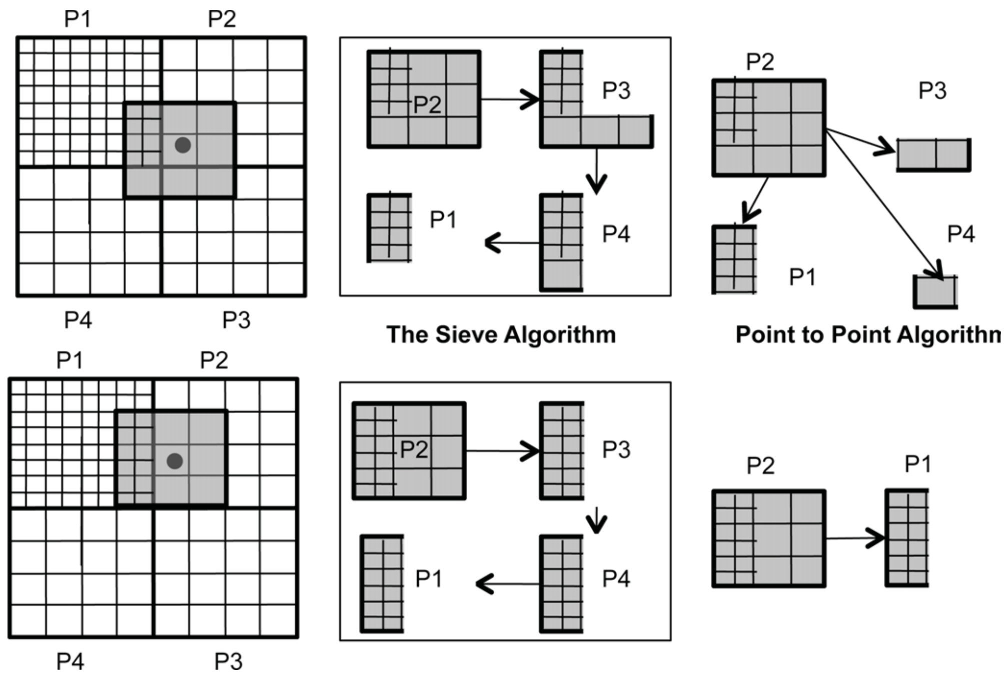

Figure 4. Mapping a particle to the surrounding cells in the mesh. The gray area represents the cells that are affected by the particle. The bold lines represent processor boundaries. The bold lines around the gray area enclose the region of influence.

the cells in the mesh that are affected by the particle. For the sieve algorithm, we may assume without any loss of generality that the order of rotation of the sieve is p2→p3→p4→p1. For clarity, only the sieve associated with p2 is shown in the figure. In general, the sequence of sieve rotation steps follows a fixed pattern, and does not depend on the actual destinations. We choose this particular order to illustrate the potential inefficiency of the sieve algorithm. The sieve algorithm has the three communication steps for both positions of the particle, even though the particle does not affect p3 and p4 in the bottom panel. In contrast, the point-to-point algorithm has each message being directly sent to their predetermined destinations. Here, the bottom panel has only one send–receive step.

5.4. Forces

The Particles unit provides for two broad categories of forces on active particles, "short-range" forces and "long-range" forces, which are split into the subunits *ParticlesForces/shortRange* and *ParticlesForces/longRange*, respectively. The division arises from algorithmic considerations; long-range forces (gravity, the electric force, etc.) arise from the matter distribution as a whole and are usually interpolated from the grid to the particle positions, whereas short-range forces (collisions, etc.) are between pairs of particles that come within close contact. The magnitude of the force on an individual particle is typically dependent on other properties of the particle, such as its mass, charge, cross-section, etc., depending on the force and the particle type in question.

In many cases of interest, the particles both exert and respond to forces from the grid. For example, in self-gravitating systems the particles serve as a source for the gravitational potential and then experience the gravitational force that is produced by the potential. In cases such as these, in order to ensure that particles do not experience a "self-force" and to locally preserve Newton's third law of motion, the same mapping scheme must be used to map particle properties between particle positions and mesh cell in both directions (Hockney & Eastwood 1988). Note that this only eliminates self-forces, and only for particles whose clouds do not overlap a refinement boundary. It does not address the violation of Newton's third law that comes about because of the different force truncation errors experienced by particle pairs in which each member is on a different refinement level. It also does not address the self-force that arises if a particle cloud overlaps a refinement boundary. In general though, the particle count is large enough that the mean field dominates these small errors. We will address methods for eliminating these errors in a future paper.

5.5. Multiple Particle Types

FLASH can support multiple particle types in a simulation through a combination of a unified data structure, layered sorting, enhanced configuration features, and interoperability of various alternative implementations of code sections in the unit. The only place where there is operational and performance trade-off in this approach is in the unified data structure. The choice of a two-dimensional double-precision array representation where the first dimension is the union of attributes from all particle types could, in theory, bloat the storage of particles. In practice, this is rarely true. The majority of attributes, such as identifiers, positions, velocities, etc., are common to all particles. The relatively small percentage increase in the storage due to non-common attributes is compensated by the ability to use the optimized single particle type infrastructure for the majority of operations. We keep the data structure sorted first by particle types, and within particle types by the associated block ID. We also maintain the starting and ending pointers for each particle type. Thus, a contiguous section of the particle data structure can be handed over for processing to each method without additional overhead. Ample scratch space is maintained by the GridParticlesMove sub-unit for communication; the same space can be recycled for out-of-place $O(n)$ sorting by letting the GridParticles sub-unit do it.

The second part of enabling multiple particle type capability is to allow for different mechanisms for initialization, mapping,

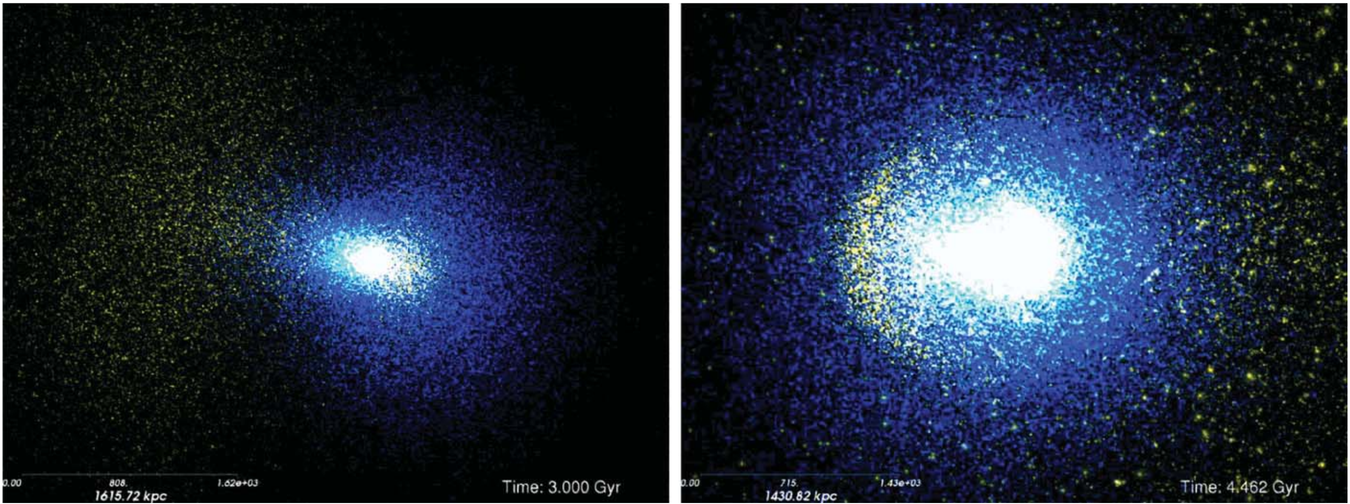

Figure 5. Volume renderings of active particles from FLASH galaxy cluster merger simulations.
(A color version of this figure is available in the online journal.)

and integration among different particle types. We opted to identify and attach the methods for each of the above-mentioned processing steps to particle types at the application configuration step rather than at runtime. The primary consideration for this choice is to prevent proliferation of switches in the code, which reduces the flexibility and maintainability of the code. The *Config* file syntax was augmented with directives to specify the desired methods, which are used to generate a map that can be queried by the Particles unit as needed at runtime.

6. APPLICATIONS

6.1. Example: Galaxy Cluster Mergers

A common use of the FLASH code has been to perform numerical studies of galaxy clusters. Galaxy clusters are the largest objects to have formed from gravitational collapse since the big bang, and as such provide excellent representations of the matter content of the universe as a whole. Besides the galaxies themselves, a cluster is filled with a diffuse, hot, magnetized, X-ray emitting plasma that is the dominant baryonic component (Sarazin 1988). The bulk of the cluster's mass is in the form of dark matter. Galaxy clusters are not only important objects of study in their own right, but they are being used to constrain cosmological models as well (e.g., Voit 2005; Vikhlinin et al. 2009). For this latter purpose, it is important to model accurately the physics of the clusters' gas and dark matter components.

FLASH's Eulerian grid framework, coupled with its suite of hydrodynamics solvers, is well suited to solve for the evolution of the gas in galaxy clusters. Dark matter is believed to be collisionless, and as such is modeled poorly by the Euler equations of hydrodynamics, which best represent collisional systems. The evolution of collisionless matter under gravitational interactions is best represented by the collisionless Boltzmann equation (CBE):

$$0 = \frac{df}{dt} = \frac{\partial f}{\partial t} + \mathbf{v} \cdot \frac{\partial f}{\partial \mathbf{x}} - \frac{\partial \Phi}{\partial \mathbf{x}} \cdot \frac{\partial f}{\partial \mathbf{v}}, \quad (1)$$

where $f(\mathbf{x}, \mathbf{v}, t)$ is the single-particle distribution function of the dark matter in phase space $\{\mathbf{x}, \mathbf{v}\}$ and Φ is the gravitational potential. Unfortunately, the direct numerical simulation of this equation on the FLASH grid is not feasible. First, the CBE is a nonlinear partial differential equation in seven dimensions with a vast phase space, and it is typically very inhomogeneous. Second, f develops strong gradients as it evolves, due to fluctuations in the field becoming mixed, leading to extremely thin layers of phase-space density. These issues are described in more detail in Binney & Tremaine (1987).

Since the above approach is not computationally feasible at this time given the currently available hardware capabilities, essentially all codes that model dark matter and other types of collisionless material in two and three spatial dimensions have chosen an alternative route. This approach, implemented by the Lagrangian particle framework in the FLASH code, is to model f in a Monte Carlo fashion by an ensemble of N randomly chosen phase-space points $\{\mathbf{x}_i, \mathbf{v}_i\}$, $i = 1 \ldots N$ with weights m_i corresponding to the particle masses (for a more comprehensive theoretical treatment of this representation of the distribution function, see the recent review by Dehnen & Read 2011). In this context, the Lagrangian mass entities are commonly referred to as "dark matter particles," but it is crucial to understand that the particles themselves do not represent physical dark matter particles or structures per se but act as discrete samples of the underlying dark matter distribution function f in phase space. Using a large number of massive particles per cluster (typically $\sim 10^5$–10^7), we may follow the evolution of the dark matter distributions of the clusters with sufficient accuracy to understand how their properties change during the merger as well as the effects on the gas of the clusters.

This model for the dark matter was used in a particular setup for the FLASH code in a number of works to simulate high-resolution binary mergers of galaxy clusters. ZuHone et al. (2009a) simulated a high-speed, head-on collision between two galaxy clusters to determine if such collisions could result in the creation of rings of dark matter particles as suggested by a recent observation (Jee et al. 2007). ZuHone et al. (2009b) re-simulated the same collision with gas included to estimate the masses of the clusters from the gas properties and compare them to the actual masses measured during the simulation. ZuHone et al. (2010) performed a number of simulations of mergers of clusters with small subclusters to show how gravitational perturbations initiate "sloshing" of gas in the cores of clusters and the heating of the cluster core. ZuHone (2011) presented a parameter space exploration of galaxy cluster mergers, examining the mixing

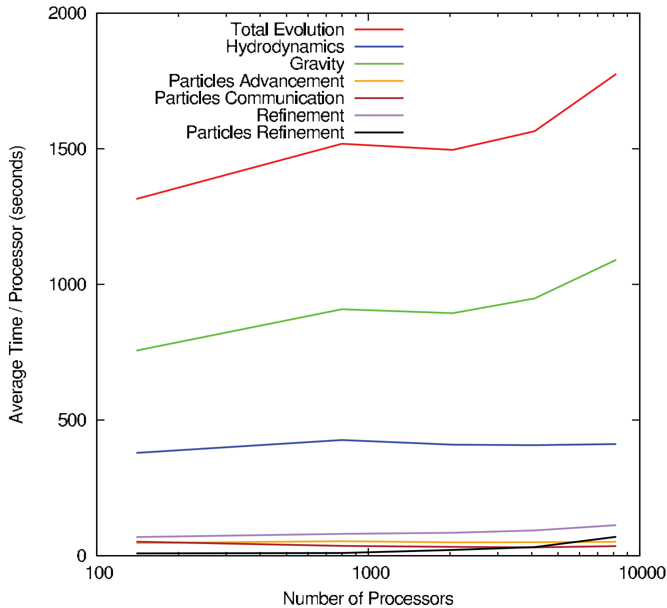

Figure 6. Weak scaling behavior of the galaxy cluster application on the ANL ALCF resource BlueGene/P "Intrepid," with the individual scaling performance of several code units plotted.

(A color version of this figure is available in the online journal.)

and thermal properties of the gas (example volume renderings of the particles from two galaxy clusters undergoing mergers are shown in Figure 5). Finally, Molnar et al. (2012) used the FLASH code to investigate the separation between the X-ray and Sunyaev–Zel'dovich peaks during galaxy cluster mergers. The setup used for all of these works relies crucially on the combination of the Lagrangian particle framework and the Eulerian grid framework in the FLASH code.

Some of these recent calculations were performed on "Intrepid," a BlueGene/P at the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory (ANL). Figure 6 shows the weak scaling performance of the main code units on Intrepid, including those involving active particles. The particle units not only scale well with increasing problem size, but also represent a small fraction of the computational time when compared with the hydrodynamics unit, which is the most fundamental physics unit of the code. This fact allows the user to include a large number of particles in the simulation without taking a significant performance hit or using up the memory on the compute nodes.

A number of recent works have demonstrated that grid-based and particle-based hydrodynamic codes produce very different results in physical situations involving fluid instabilities and turbulence (e.g., Dolag et al. 2005; O'Shea et al. 2005b; Agertz et al. 2007; Wadsley et al. 2008; Mitchell et al. 2009; Bauer & Springel 2011; Scannapieco et al. 2011). Specifically, in high Reynolds number situations, the codes based on smoothed particle hydrodynamics (SPH) fail to reproduce fluid instabilities, mixing, and small-scale turbulent motions. It is for this reason that for many applications grid-based methods are preferred. However, though the Eulerian approach to hydrodynamics may reproduce the properties of shocks, fluid instabilities, and turbulence better than SPH codes, it lacks the ability to trace the histories of individual gas parcels. A number of questions arise in the context of galaxy cluster mergers that require a Lagrangian representation of the fluid in order to obtain accurate answers. For example, mergers disrupt and heat the cool cores of relaxed galaxy clusters (Burns et al. 2008; ZuHone et al. 2010; ZuHone 2011) via shock heating and mixing of the cool gas from the core with hotter gas from the cluster outskirts. Eulerian methods represent the shock heating and gas mixing well, but they cannot give any information about the history of the gas that originally resided in the cluster centers. What happened to this gas? How have its thermodynamic properties changed as a result of the merger? Where does this gas end up in the cluster? How exactly did the metal-rich gas originating in the cluster cores get spread around by the merger event?

These are questions answered best by the use of passive tracer particles, which represent individual gas parcels that can be tracked during the course of a simulation (see Vazza et al. 2010, for an example using tracer particles in the Enzo code). The particle framework in FLASH allows for the inclusion of both active particles to represent the dark matter and passive tracer particles to represent the gas. In certain situations, these particles are analogous to the gas particles in SPH methods. This functionality of the FLASH code offers a "best of both worlds" approach, providing both the advantages of the Eulerian approach to hydrodynamics and the ability to track the histories of individual gas parcels, the latter being normally offered only by SPH methods.

Ongoing simulation work with the FLASH code in the area of galaxy cluster mergers demonstrates the versatility of passive tracer particles. Beyond merely using the passive tracers to investigate the properties of the fluid parcels they represent over time, it is also possible to perform detailed post-processing calculations of other physical phenomena not represented by the physics modeled by the FLASH code. One such application is the use of the galaxy cluster merger setup to simulate turbulent reacceleration of cosmic-ray electrons in merging galaxy clusters. Cosmic-ray electrons are relativistic particles that produce synchrotron radio emission as they spiral around magnetic field lines. In a number of galaxy clusters, large-scale, diffuse radio emission is produced by these particles, which are believed to be continuously reaccelerated to the necessary energies by magnetohydrodynamic turbulence (Mazzotta & Giacintucci 2008; Cassano et al. 2010). It is necessary to take into account the energy gains and losses of the cosmic-ray electrons over time in order to determine their energies at the observed epoch. By associating populations of relativistic electrons with individual tracer particles, it is possible to determine the evolution of the electrons' energies as they are reaccelerated by turbulence and decelerated by energy losses from interactions with magnetic fields, background photons, and thermal electrons. The complete characterization of the gas properties along the trajectories followed by the tracer particles makes carrying out such a calculation possible. The first calculations of this kind using the FLASH code are presented in ZuHone et al. (2012).

6.2. *Example: Type Ia Supernova Explosions*

Another important application of the FLASH code has been the computation of explosion models of SNe Ia. Interest in modeling of SNe Ia has been stimulated by the discovery of the accelerating expansion of the universe using SNe Ia as standard-candle cosmographic markers (Riess et al. 1998; Perlmutter et al. 1999). The goal of using SNe Ia as probes of the equation of state of dark energy (and its evolution with redshift) awaits a reduction in the systematic errors in the distance determination to individual SN Ia from the current level of about 15% to about 1% (Kim et al. 2004; DETF 2006). The best hope for

improvements in distance modulus accuracy is more accurate modeling of SN Ia explosions.

The leading mechanism for SN Ia explosions is the single-degenerate model in which a progenitor white dwarf accretes material from a non-degenerate companion star (Whelan & Iben 1973; Nomoto 1982). In this model, an accreting white dwarf star, with a mass near the Chandrasekhar limit, manages to release enough nuclear energy by the fusing of carbon and oxygen into radioactive nickel and other lighter elements in the time span of a few seconds or less (Nomoto et al. 1984). This rapid fusion process deposits approximately 10^{51} erg of energy, unbinding the star and accelerating the stellar material to speeds of thousands of kilometers per second (Branch et al. 1982). This rapid fusion process must proceed in two phases (Röpke et al. 2007). The first phase begins with the initiation of a subsonic nuclear burning front (referred to as a deflagration or flame). The second phase consists of a supersonic burning front—a detonation—that consumes the remainder of the white dwarf.

The explosion itself is not detectable, since most of the nuclear energy evolved by the fusion reactions is converted to kinetic energy of the cooling and expanding nebula. The nebula starts to heat up and glow soon thereafter, however, due to radioactive decay of the unstable iron-group isotopes (mainly ^{56}Ni) produced by nuclear burning during the explosion. Prediction of observable multi-color light curves and spectra therefore requires radiation transport calculations of the light emitted by the inhomogeneously heated, inhomogeneously composed nebula. An accurate composition distribution of the ejecta is therefore required, in addition to the mass and velocity profiles.

A true, high-fidelity simulation of an SN Ia explosion would couple the hydrodynamic evolution of the degenerate plasma to a nuclear reaction network, in order to simultaneously obtain the correct energy release (and hence the correct flow properties) and the correct isotopic abundances. However, the expense of such a computation has proven prohibitive in three dimensions. The FLASH simulations of SN Ia explosions model the nuclear flame using a flame-capturing Advection-Diffusion-Reaction scheme (Khokhlov 1995), powered by a simplified, three-stage burning model representing carbon burning, oxygen burning, and relaxation to nuclear statistical equilibrium (Calder et al. 2007; Townsley et al. 2007). This model is calibrated to produce accurate energy release, but gives inadequate (and inaccurate) estimates of nucleosynthetic abundances. To compensate, passive scalar particles are used to record thermal history and are post-processed through a nuclear network to compute the final abundances (Brown et al. 2005). Those abundances are then placed on the grid together with the hydrodynamic outflow data for processing by radiation transport codes.

It seems natural to choose an initial particle distribution that is proportional to mass density in this application, since such a distribution has the desirable property of having a predictable evolution: passive advection necessarily preserves the mass-tracing property of such a particle distribution. This property may seem more desirable inasmuch as one might expect that most of the action of interest takes place at the locations where most of the mass is. However, this appearance is deceptive from the point of view of the radiation transport. In fact, the consequence of a mass-tracing particle placement strategy is that far too many particles take up residence in regions of the outflow that are "boring," in the sense that they are either too uniform in composition to justify high particle sampling or are too optically thick to contribute to the observed light. Furthermore, line-forming processes of crucial importance to the SN Ia spectrum often occur in (relatively) low-density regions of the outflow, and hence are severely undersampled by a mass-tracing particle distribution. As a result of these mismatches, we are forced to evolve many more particles than are strictly required to sample the outflow composition, in order to obtain adequate numbers in regions of interest from the perspective of radiation transport.

We address this problem by a combination of strategies. The simplest approach is to superpose two distributions of particles, one mass tracing, the other uniform in volume (over the initial stellar interior), keeping track of which parent distribution generated each particle. After a few pilot runs, we can estimate from the desired sampling rates in (photospherically) "interesting" regions the relative fractions of particles required from the two populations in a statistically optimal sense. The ability to customize particle location initialization in FLASH is obviously a necessity here, as is the ability to simultaneously evolve multiple particle types.

A more sophisticated strategy, currently under exploration, is as follows: take the final particle configuration from one of the above pilot simulations or perhaps of a cheaper, two-dimensional simulation of the same initial conditions. Break up the outflow itself into regions (e.g., radial shells, wedges, etc.) and estimate the local final particle density $f_{\rm final}(\boldsymbol{x})$ required to sample each region optimally. Suppose a particle started the simulation at position $\boldsymbol{x}_{\rm initial}$ with density $\rho_{\rm initial}(\boldsymbol{x}_{\rm initial})$, and ended it at position $\boldsymbol{x}_{\rm final}$ with density $\rho_{\rm final}(\boldsymbol{x}_{\rm final})$ (the densities ρ are recorded as part of each particle's thermal history). One can then obtain the required sampling density of the *initial* particle distribution $f_{\rm initial}(\boldsymbol{x}_{\rm initial})$ corresponding to the desired final distribution by the equation $f_{\rm initial}(\boldsymbol{x}_{\rm initial}) = [\rho_{\rm initial}(\boldsymbol{x}_{\rm initial})/\rho_{\rm final}(\boldsymbol{x}_{\rm final})] \times f_{\rm final}(\boldsymbol{x}_{\rm final})$, which follows from the advection of both the particles and the mass density. That point sampling of the initial distribution can then be interpolated to the initial grid, and the interpolated distribution sampled to produce a particle population of moderate size targeting the desired final distribution. This necessarily irregular particle distribution mandates the sort of flexible particle location initialization facilities available in FLASH.

7. CONCLUSIONS

We have successfully superimposed a fairly comprehensive Lagrangian framework on top of the FLASH code's primary Eulerian framework. The Lagrangian framework has all the modularity, extensibility, and customizability of the Eulerian framework. Also, because the data structures associated with the particles are independent of the physical quantities being represented, they have multiple different uses in different applications. Some of the more unorthodox uses of the Lagrangian framework include fluid–structure interaction and laser ray tracing. At the time of this writing, these are still works in progress, and therefore the details are not included in the discussion here. However, they both exploit the most important general contribution of the Lagrangian framework to the FLASH code: it provides a mechanism to support parallel data structures that are not native or known to the underlying mesh package, but need to interact with it.

The code described in this work was in part developed by the NNSA-DOE supported Flash Center under grant B523820, and NSF Peta-apps grant 5-27429 at the University of Chicago.

The PIC capabilities described in the paper are contributed and maintained by Mats Holmström.

REFERENCES

Agertz, O., Moore, B., Stadel, J., et al. 2007, MNRAS, 380, 963
Bauer, A., & Springel, V. 2011, arXiv:1109.4413
Beaudoin, A., de Dreuzy, J.-R., & Erble, J. 2007, in Euro-Par 2007 Parallel Processing, ed. A.-M. Kermarrec, L. Bougé, & T. Priol (Lecture Notes in Computer Science; Berlin: Springer), 717
Binney, J., & Tremaine, S. 1987, Galactic dynamics (Princeton, NJ: Princeton Univ. Press)
Branch, D., Buta, R., Falk, S. W., et al. 1982, ApJ, 252, L61
Brown, E. F., Calder, A. C., Plewa, T., et al. 2005, Nucl. Phys. A, 758, 451
Burns, J. O., Hallman, E. J., Gantner, B., Motl, P. M., & Norman, M. L. 2008, ApJ, 675, 1125
Calder, A. C., Townsley, D. M., Seitenzahl, I. R., et al. 2007, ApJ, 656, 313
Cassano, R., Ettori, S., Giacintucci, S., et al. 2010, ApJ, 721, L82
Colella, P., Graves, D., Keen, N., et al. 2009, Chombo Software Package for AMR Applications Design Document, Technical Report, Lawrence Berkeley National Laboratory, Applied Numerical Algorithms Group, Computational Research Division
Couchman, H. M. P., Thomas, P. A., & Pearce, F. R. 1995, ApJ, 452, 797
Daley, C., Vanella, M., Weide, K., Dubey, A., & Balaras, E. 2012, Concurrency Comput.: Pract. Exp., 10.1002/cpe.2821
Dehnen, W., & Read, J. 2011, The European Physical Journal Plus, arXiv:1105.1082
DETF. 2006, Report of the Dark Energy Task Force, Available Online at http://www.nsf.gov/mps/ast/aaac/dark_energy_task_force/report/detf_final_report.pdf
Dolag, K., Borgani, S., Schindler, S., Diaferio, A., & Bykov, A. M. 2008, Space Sci. Rev., 134, 229
Dolag, K., Vazza, F., Brunetti, G., & Tormen, G. 2005, MNRAS, 364, 753
Dubey, A., Antypas, K., & Daley, C. 2011, Parallel Comput., 37, 101
Dubey, A., Antypas, K., Ganapathy, M. K., et al. 2009, Parallel Comput., 35, 512
Efstathiou, G., Davis, M., White, S. D. M., & Frenk, C. S. 1985, ApJS, 57, 241
Federrath, C., Glover, S. C. O., Klessen, R. S., & Schmidt, W. 2008, Phys. Scr. T, 132, 014025
Fisher, R., Abarzhi, S., Antypas, K., et al. 2008, IBM J. Res. Dev., 52, 127
Fryxell, B., Olson, K., Ricker, P., et al. 2000, ApJS, 131, 273
Hockney, R. W., & Eastwood, J. W. 1988, Computer Simulation Using Particles (Bristol: Hilger)
Holmström, M., Fatemi, S., Futaana, Y., & Nilsson, H. 2012, Earth Planets Space, 64, 237
Hubber, D. A., Batty, C. P., McLeod, A., & Whitworth, A. P. 2011, A&A, 529, A27
Jee, M. J., Ford, H. C., Illingworth, G. D., et al. 2007, ApJ, 661, 728
Jordan, G., Fisher, R., Townsley, D., et al. 2008, ApJ, 681, 1448
Khokhlov, A. M. 1995, ApJ, 449, 695
Kim, A., Linder, E., Miquel, R., & Mostek, N. 2004, MNRAS, 347, 909
Kravtsov, A. V., Klypin, A. A., & Khokhlov, A. M. 1997, ApJS, 111, 73
MacNeice, P., Olson, K., Mobarry, C., de Fainchtein, R., & Packer, C. 2000, Comput. Phys. Commun., 126, 330
Mazzotta, P., & Giacintucci, S. 2008, ApJ, 675, L9
Miniati, F., & Colella, P. 2007, J. Comput. Phys., 227, 400
Mitchell, N. L., McCarthy, I. G., Bower, R. G., Theuns, T., & Crain, R. A. 2009, MNRAS, 395, 180
Molnar, S. M., Hearn, N. C., & Stadel, J. G. 2012, arXiv:1201.1533
Nomoto, K. 1982, ApJ, 253, 798
Nomoto, K., Thielemann, F.-K., & Yokoi, K. 1984, ApJ, 286, 644
Norman, M., Bryan, G., Harkness, R., et al. 2008, Petascale Computing: Algorithms and Applications (New York: Chapman and Hall), 83
O'Shea, B. W., Bryan, G., Bordner, J., et al. 2005a, in Adaptive Mesh Refinement—Theory and Applications, ed. T. Plewa, L. Timur, & V. Weirs (Lecture Notes in Computational Science and Engineering, Vol. 41; Springer), 341
O'Shea, B. W., Nagamine, K., Springel, V., Hernquist, L., & Norman, M. L. 2005b, ApJS, 160, 1
Pearce, F. R., & Couchman, H. M. P. 1997, New Astron., 2, 411
Perlmutter, S., Aldering, G., Goldhaber, G., et al. 1999, ApJ, 517, 565
Ricker, P. M., Dodelson, S., & Lamb, D. Q. 2000, ApJ, 536, 122
Riess, A. G., Filippenko, A. V., Challis, P., et al. 1998, AJ, 116, 1009
Röpke, F. K., Hillebrandt, W., Schmidt, W., et al. 2007, ApJ, 668, 1132
Sarazin, C. L. 1988, X-ray Emission from Clusters of Galaxies (Cambridge: Cambridge Univ. Press)
Scannapieco, C., Wadepuhl, M., Parry, O. H., et al. 2011, arXiv:1112.0315
Springel, V. 2005, MNRAS, 364, 1105
Springel, V. 2011, in IAU Symp. 270, Computational Star Formation, ed. J. Alves, B. G. Elmegreen, J. M. Girart, & V. Trimble (Cambridge: Cambridge Univ. Press), 203
Teyssier, R. 2002, A&A, 385, 337
Townsley, D. M., Calder, A. C., Asida, S. M., et al. 2007, ApJ, 668, 1118
Vazza, F., Gheller, C., & Brunetti, G. 2010, A&A, 513, A32
Vikhlinin, A., Kravtsov, A. V., Burenin, R. A., et al. 2009, ApJ, 692, 1060
Voit, G. M. 2005, Rev. Mod. Phys., 77, 207
Wadsley, J. W., Stadel, J., & Quinn, T. 2004, New Astron., 9, 137
Wadsley, J. W., Veeravalli, G., & Couchman, H. M. P. 2008, MNRAS, 387, 427
Whelan, J., & Iben, I., Jr. 1973, ApJ, 186, 1007
ZuHone, J., Markevitch, M., Brunetti, G., & Giacintucci, S. 2012, arXiv:1203.2994
ZuHone, J. A. 2011, ApJ, 728, 54
ZuHone, J. A., Lamb, D. Q., & Ricker, P. M. 2009a, ApJ, 696, 694
ZuHone, J. A., Markevitch, M., & Johnson, R. E. 2010, ApJ, 717, 908
ZuHone, J. A., Ricker, P. M., Lamb, D. Q., & Yang, H.-Y. 2009b, ApJ, 699, 1004