1 Adaptive Computations in the Uintah Framework

J. LUITJENS, J. GUILKEY, T HARMAN, B. WORTHEN and S. PARKER
SCI Institute and Department of Mechanical Engineering
University of Utah

1.1 INTRODUCTION

The University of Utah Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [15] is a Department of Energy ASC center that focuses on providing state-of-the-art, science-based tools for the numerical simulation of accidental fires and explosions. The primary objective of C-SAFE has been to provide a software system in which fundamental chemistry and engineering physics are fully coupled with nonlinear solvers, visualization, and experimental data verification, thereby integrating expertise from a wide variety of disciplines. The primary target scenario for C-SAFE is the full simulation of metal containers filled with a plastic-bonded explosive (PBX) subject to heating from a hydrocarbon pool fire, as depicted in Figure 1.1. In this scenario, a small cylindrical steel container (4" outside diameter) filled with plastic bonded explosive (PBX-9501) is subjected to convective and radiative heat fluxes from a fire to heat the outside of the container and the PBX. After some amount of time the critical temperature in the PBX is reached and the explosive begins to rapidly decompose into a gas. The solid→gas reaction pressurizes

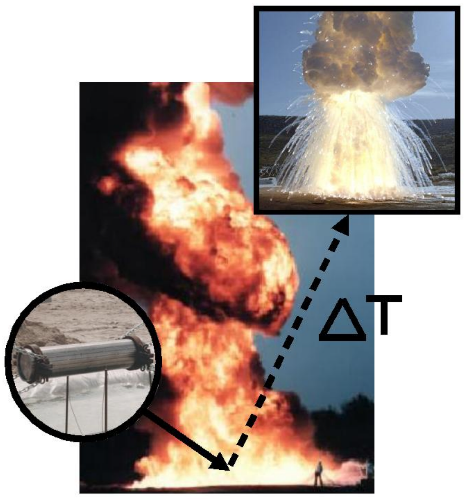

Fig. 1.1 Depiction of a typical C-SAFE scenario involving hydrocarbon fires and explosions of energetic materials

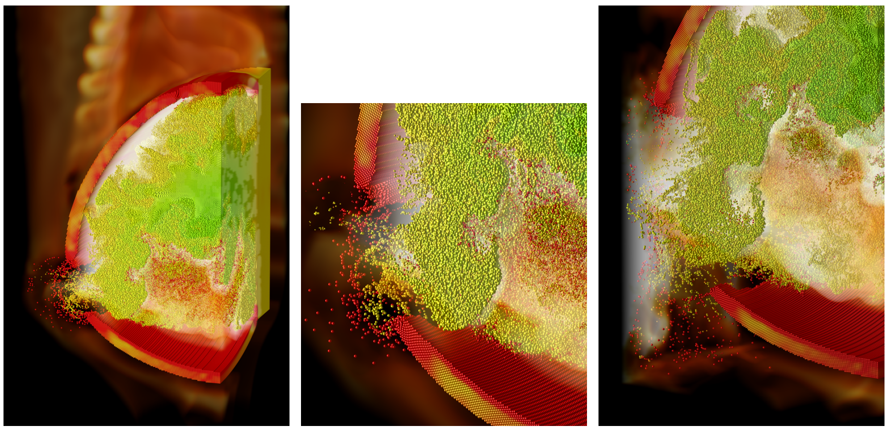

Fig. 1.2 Cross-section of an energetic device at the point of rupture

the interior of the steel container causing the shell to rapidly expand and eventually rupture. The gaseous products of reaction form a blast wave that expands outward along with pieces of the container and the unreacted PBX. The physical processes in this simulation have a wide range in time and length scales from microseconds and microns to minutes and meters. An example of this simulation is depicted in Figure 1.2.

The Uintah Computational Framework, developed as the main computational workhorse of the C-SAFE Center, consists of a set of parallel software components and libraries that facilitate the solution of Partial Differential Equations (PDEs) on Structured AMR (SAMR) grids. Uintah is applicable to a wide range of engineering domains that require fluid-structure interactions and highly deformable models. Uintah contains several simulation algorithms, including a general-purpose fluid-structure interaction code that has been used to simulate a wide array of physical systems, including stage-separation in rockets, the biomechanics of microvessels [12], the effects of wounding on heart tissue[17, 18], the properties of foam under large deformation [7], evolution of transportation fuel fires [25], in addition to the scenario described above.

The heart of Uintah is a sophisticated computational framework that can integrate multiple simulation components, analyze the dependencies and communication patterns between them, and efficiently execute the resulting multi-physics simulation. Uintah employs an abstract taskgraph representation to describe computation and communication. Through this mechanism, Uintah components delegate decisions about parallelism to a scheduler component, which determines communication patterns and characterizes the computational workloads, needed for global resource optimization. These features allow parallelism to be integrated between multiple components while maintaining overall scalability and allow the Uintah runtime to analyze the structure of the computation to automatically enable load balancing, data communication, parallel I/O, and checkpoint/restart.

We describe how these simulations are performed in the Uintah framework, beginning with a discussion of the Uintah framework (Section 1.2) and the techniques to support Structured AMR in this framework (Section 1.3). We will then discuss the adaptive ICE hydrodynamics solver employed (Section 1.4) and the particle-based Material Point Method (Section 1.5) that are used in this computation. Subsequently, we will discuss the results achieved in these computations (Section 1.6). Finally, we will discuss the status of Uintah and its future (Section 1.7).

1.2 UINTAH OVERVIEW

The fundamental design methodology in Uintah is that of software components that are built on the DOE Common Component Architecture (CCA) component model. Components are implemented as C++ classes that follow a very simple interface to establish connections with other components in the system. The interfaces between components are simplified because the components do not explicitly communicate with one another. A component simply defines the steps in the algorithm that will be performed later when the algorithm is executed, instead of explicitly performing the computation tasks. These steps are assembled into a dataflow graph, as described below.

The primary advantage of a component-based approach is that it facilitates the separate development of simulation algorithms, models, and infrastructure, such that components of the simulation can evolve independently. The component-based architecture allows pieces of the system to be implemented in a rudimentary form at first and then evolve as the technologies mature. Most importantly, Uintah allows the aspects of parallelism (schedulers, load-balancers, parallel input/output, and so forth) to evolve independently of the simulation components. This approach allows the computer science support team to focus on these problems without waiting for the completion of the scientific applications or vice-versa.

Uintah uses a non-traditional approach to achieving high degrees of parallelism, employing an abstract taskgraph representation to describe computation and communication that occur in the coarse of a single iteration of the simulation (typically a timestep or nonlinear solver iteration). Components, by definition, make local decisions. Yet parallel efficiency is only obtained through a globally optimal domain decomposition and scheduling of computational tasks. Consequently, Uintah components delegate decisions about parallelism to a scheduler component through a description of tasks and variable dependencies that describe communication patterns, which are subsequently assembled in a single graph containing all of the computation in all components. This taskgraph representation has a number of advantages, including efficient fine-grained coupling of multi-physics components, flexible load balancing mechanisms, and a separation of application concerns from parallelism concerns. However, it creates a challenge for scalability that we overcome by creating an implicit definition of this graph and representing the details of the graph in a distributed fashion.

This underlying architecture of the Uintah simulation is uniquely suited to taking advantage of the complex topologies of modern HPC platforms. Multi-core processors in SMP configurations combined with one or more communication networks are common petascale architectures, but applications that are not aware of these disparate levels of communication will not be able to scale to the large CPU counts for complex problems. The explicit taskgraph structure enables runtime analysis of communication patterns and other program characteristics that are not available at runtime in typical MPI-only applications. Through this representation of parallel structure, Uintah facilitates adapting work assignments to the underlying machine topology based on the bandwidth available between processing elements.

Tensor product task graphs Uintah enables integration of multiple simulation algorithms by adopting an execution model based on coarse-grained "macro" dataflow. Each component specifies the steps in the algorithm and the data dependencies between those steps. These steps are combined into a single graph structure (called a *taskgraph*). The taskgraph represents the computation to be performed in single timestep integration, and the data dependencies between the various steps in the algorithm. Graphs may specify numerous exchanges of data between components (fine-grained coupling) or few (coarse-grained coupling), depending on the requirements of the underlying algorithm. For example, the MPM-ICE fluid-structure algorithm implemented in Uintah (referenced in Section 1.4) requires several points of data exchange in a single timestep to achieve the tight coupling between the fluid and solids. This contrasts with multi-physics approaches that exchange boundary conditions at fluid-solid interfaces. The taskgraph structure allows fine-grained interdependencies to be expressed in an efficient manner.

The taskgraph is a directed acyclic graph of tasks, each of which produces some output and consumes some input (which is in turn the output of some previous task). These inputs and outputs are specified for each patch in a structured, possibly AMR, grid. Associated with each task is a C++ method which is used to perform the actual computation.

A taskgraph representation by itself works well for coupling of multiple computational algorithms, but presents challenges for achieving scalability. A taskgraph that represents all communication in the problem would require time proportional to the number of computational elements to create. Creating this on a single processor, or on all processors would eventually result in a bottleneck. Uintah addresses this problem by introducing the concept of a "tensor product taskgraph". Uintah components specify tasks for the algorithmic steps only, which are independent of problem size or number of processors. Each task in the taskgraph is then implicitly repeated on a portion of patches in the decomposed domain. The resulting graph, or tensor product taskgraph, is created collectively; each processor contains only the tasks that it owns and those that it communicates with. The graph exists only as a whole across all computational elements, resulting in a scalable representation of the graph. Communication requirements between tasks are also specified implicitly through a simple dependency algebra.

Each execution of a taskgraph integrates a single timestep, or a single non-linear iteration, or some other coarse algorithm step. A portion of the MPM timestep graph is shown in Figure 1.3. Taskgraphs may be assembled recursively, with a typical Uintah simulation containing one for time integration and one for nonlinear integration. An AMR simulation may contain several more for implementing time subcycling and refinement/coarsening operators on the AMR grid.

The idea of the dataflow graph as an organizing structure for execution is well known. The SMARTS [34] dataflow engine that underlies the POOMA [2] toolkit shares similar goals and philosophy with Uintah. The SISAL language compilers [10] used dataflow concepts at a much finer granularity to structure code generation and execution. Dataflow is a simple, natural and efficient way of exposing parallelism and managing computation, and is an intuitive way of reasoning about parallelism. What distinguishes implementations of dataflow ideas is that each caters to a particular higher-level presentation. SMARTS caters to POOMA's C++ implementation and stylistic template-based presentation. Uintah's implementation supports dataflow (task) graphs of C++ and Fortran based mixed particle/grid algorithms on a structured adaptive mesh.

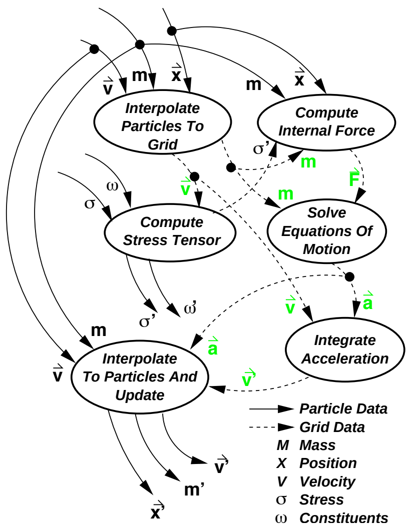

Fig. 1.3 An example Uintah task graph for MPM

Particle and Grid Support Tasks describe data requirements in terms of their computations on Node, Cell and Face-centered quantities. A task that computes a cell-centered quantity from the values on surrounding nodes would establish a requirement for 1 layer of nodes around the cells at the border of a patch. This is termed "nodes around cells" in Uintah terminology. Likewise, a task that computes a node-centered quantity from the surrounding particles would require the particles that reside within 1 layer of cells at the patch boundary. The region of data which is required is termed the "halo region". Similarly, each task specifies the non-halo data that it will compute. By defining the halo region in this way, one can specify the communication patterns in a complex domain without resorting to an explicit definition of the communication needed. These "computes & requires" lists for each task are collected to create the full taskgraph. Subsequently, the specification of the halo region is combined with the details of the patches in the domain to create the tensor product taskgraph. Data dependencies are also specified between refinement levels in an AMR mesh using the same approach, with some added complexity. This can often result in complex communication, but is still specified using a simple description of data dependencies.

Executing task programs Each component specifies a list of tasks to be performed and the data dependencies between them. These tasks may also include dependencies on quantities from other components. A scheduler component in Uintah sets up MPI communication for data dependencies and then executes the tasks that have been assigned to it. When the task completes, the infrastructure will send data to other tasks that require that task's output.

On a single processor, execution of the taskgraph is simple. The tasks are simply executed in the topologically sorted order. This is valuable for debugging, since multi-patch problems can be tested and debugged on a single processor. In most cases, if the multi-patch problem passes the taskgraph analysis and executes correctly on a single processor, then it will execute correctly in parallel.

In a multi-processor machine the execution process is more complex. There are a number of ways to utilize MPI functionality to overlap communication and computation. In Uintah's current implementation we process each task in a topologically sorted order. For each task, the scheduler posts non-blocking receives (using MPI_Irecv) for each of the data dependencies. Subsequently, we call MPI_Waitall to wait for the data to be sent from neighboring processors. After all data has arrived, we execute the task. When the task is finished, we call MPI_Isend to initiate data transfer to any dependent tasks. Periodic calls to MPI_Waitsome for these posted sends ensure that resources are cleaned up when the sends actually complete.

To accommodate software packages that were not written using the Uintah execution model, we allow tasks to be specially flagged as "using MPI". These tasks will be gang-scheduled on all processors simultaneously, and will be associated with all of the patches assigned to each processor. In this fashion, Uintah applications can use available MPI-based libraries, such as PETSc [3] and Hypre [9].

Infrastructure features The taskgraph representation in Uintah enables compiler-like analysis of the computation and communication steps in a timestep. This analysis is performed at runtime, since the combination of tasks required to compute the algorithm may vary dramatically based on problem parameters. Through analysis of the taskgraph, Uintah can automatically create checkpoints, perform load balancing and eliminate redundant communication. This analysis phase, which we call "compiling" the taskgraph, is what distinguishes Uintah from most other component-based multi-physics simulations. The taskgraph is compiled when the grid changes, when the nature of the algorithm changes, or when load imbalance is detected; basically whenever the structure of the communication may change. Uintah also has the ability to modify the set of components in use during the course of the simulation. This is used to transition between solution algorithms, such as a fully explicit or semi-implicit formulation, triggered by conditions in the simulation.

Data output is scheduled by creating tasks in the taskgraph just like any other component. Constraints specified with the task allow the load balancing component to direct those tasks (and the associated data) to the processors where data I/O should

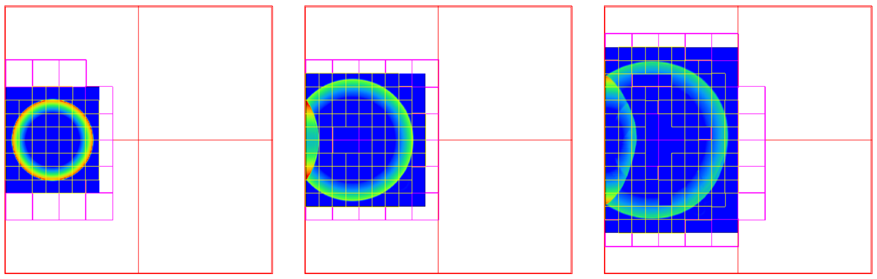

Fig. 1.4 A pressure blast wave reflecting off of a solid boundary

occur. In typical simulations, each processor writes data independently for the portions of the dataset which it owns. This requires no additional parallel communication for output tasks. However, in some cases this may not be ideal. Uintah can also accommodate situations where disks are physically attached to only a portion of the nodes, or a parallel filesystem where I/O is more efficient when performed by only a fraction of the total nodes.

Checkpointing is obtained by using these output tasks to save all of the data at the end of a timestep. Data lifetime analysis ensures that only the data required by subsequent iterations will be saved automatically. During a restart, the components process the XML specification of the problem that was saved with the datasets, and then Uintah creates input tasks that load data from the checkpoint files. If necessary, data redistribution is performed automatically during the first execution of the taskgraph. As a result, changing the number of processors when restarting is possible.

1.3 AMR SUPPORT

Adaptive Mesh Refinement Many multi-physics simulations require a broad span of space and time scales. Uintah's primary target simulation scenario includes a large scale fire (size of meters, time of minutes) combined with an explosion (size of microns, time of microseconds). To efficiently and accurately capture this wide range of time and length scales, the Uintah architecture has been has been designed to support AMR in the style of Berger and Colella [6], with great initial success. Figures 1.4 show a blast wave reflecting off of a solid boundary with an AMR mesh using the explicit ICE algorithm and refinement in both space and time.

The construction of the Uintah AMR framework required two key pieces: multi-level grid support and grid adaptivity. A multi-level grid consists of a coarse grid with a series of finer levels. Each finer level is placed on top of the previous level with a spacing that is equal to the coarse level spacing divided by the refinement ratio, which is specified by the user. A finer level only exists in a subset of the domain of the coarser level. The coarser level is used to create boundary conditions for the finer level.

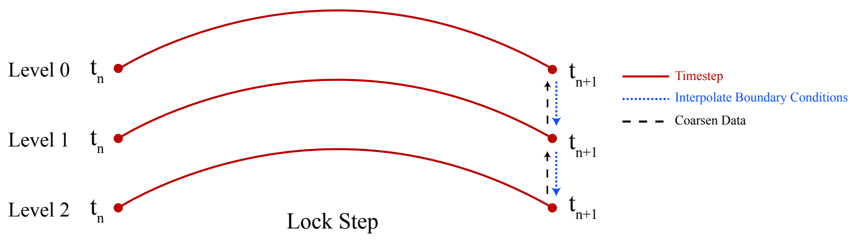

Fig. 1.5 The Lockstep time integration model

The framework supports multi-level grids by controlling the inter-level communication and computation. Grid adaptivity is controlled through the use of refinement flags. The simulation components generate a set of refinement flags which specify the regions of the level that need more refinement. As the refinement flags change a regridder uses them to produce a new grid with the necessary refinement.

Whenever the grid changes a series of steps must be taken prior to continuing the simulation. First the patches must be distributed evenly across processors through a process called load balancing. Second all patches must be populated with data either by copying from the same level of the previous grid or by interpolating from a coarser level of the previous grid. Finally the task graph must be recompiled. These steps can take a significant amount of runtime [36, 28] and affect the overall scalability at large numbers of processors.

Multi-Level Execution Cycle There are two styles of multi-level execution implemented in Uintah: the Lockstep (typically used for implicit or semi-implicit solvers) and the W-cycle (typically used for explicit formulations) time integration models. The Lockstep model, as shown in Figure 1.5, advances all levels simultaneously. After executing each timestep the coarsening and boundary conditions are applied. The W-Cycle, as shown in Figure 1.6, uses larger timesteps on the coarser levels than the finer levels. On the finer levels there are at least n subtimesteps on the finer level for every subtimestep on the coarser level, where n is equal to the refinement ratio. This provides the benefit of performing less computation on the coarser levels but also requires an extra step of interpolating the boundary conditions after each subtimestep. In both cycles at the end of each coarse level timestep the data on a finer level is interpolated to the coarser levels ensuring that the data on the coarser level reflects the accuracy of the finer level. In addition, the boundary conditions for the finer level are defined by the coarser levels by interpolating the data on a coarse level to the domain boundaries of the finer level [6]. These inter-level operations are described in detail in Section 1.4.1.

Refinement Flags After the execution of a timestep the simulation runs a task that marks cells that need refinement. The criteria for setting these flags is determined by

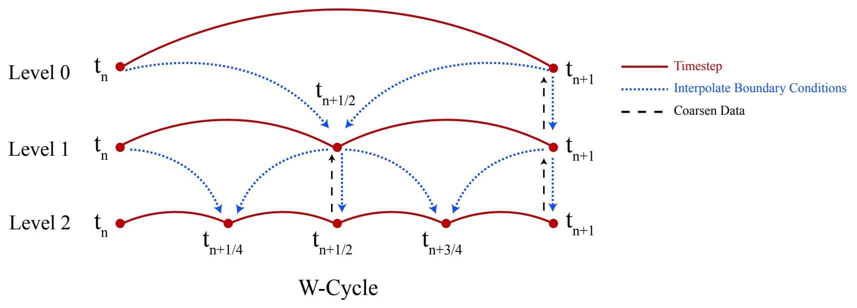

Fig. 1.6 The W-cycle time integration model

the simulation component, and is typically determined by a gradient magnitude of a particular quantity or other error metrics.

Regridding When using AMR the areas of the domain that need refinement can change every timestep necessitating a full recomputation of the grid. A regrid occurs whenever the simulation components produce flags that are outside of the currently refined regions. Regridding creates a new grid over the refinement flags.

Regridding is a necessary step in AMR codes. The grid must move in order to fully resolve moving features. A good regridder should produce large patches whenever possible. If the regridder produces small patches the number of patches will be higher than necessary and can cause significant overhead in other components leading to large inefficiencies. However, it is also important that the regridder does not produce too large of patches because large patches cannot be load balanced effectively. Thus an ideal regridder should produce patches that are as large enough to keep the number of patches small but small enough to be effectively load balanced.

A common algorithm used in AMR codes for regridding is the Berger-Rigoutsos algorithm [5]. The Berger-Rigoutsos algorithm uses edge detection methods on the refinement flags to determine decent areas to place patches. The patch sets produced by the Berger-Rigoutsos algorithm contains a mix of both large and small patches. Regions are covered by as large patches as possible and the edges of those regions are covered by small patches producing tight covering of the refinement flags. In addition the Berger-Rigoutsos algorithm can be run quickly in parallel [37, 13] making it an ideal algorithm for regridding.

However, the current implementation of Uintah has two constraints on the types of patches that can be produced that prevent direct usage of the Berger-Rigoutsos algorithm. First, patch boundaries must be consistent; i.e., each face of the patch can contain only coarse-fine boundaries or neighboring patches at the same level of refinement, not a mixture of the two. Second, Uintah requires that all patches be at least four cells in each direction. One way around the neighbor constraint is to locate patches that violate the neighboring constraint and split them into two patches,

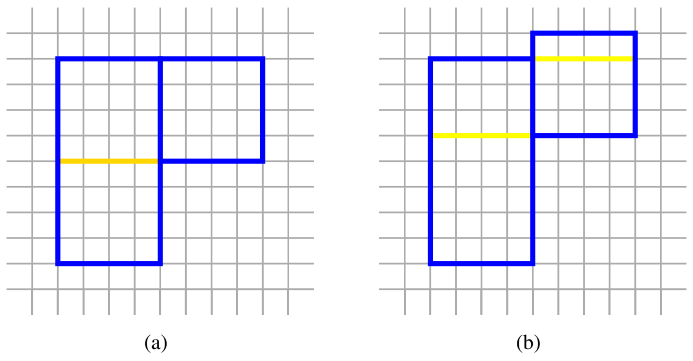

Fig. 1.7 The blue patches are invalid; in (a) they can be split by the yellow line to produce a valid patch set but in (b) they cannot.

thus eliminating the constraint violation. Figure 1.7(a) shows an example of fixing a constraint violation via splitting. Unfortunately splitting patches can produce patches that violate Uintah's size constraint as shown in Figure 1.7(b).

To make the Berger-Rigoutsos algorithm compatible with Uintah, a minimum patch size of at least four cells in each dimension is specified. The minimum patch size is used to coarsen the flag set by dividing every flag's location by the minimum patch size and rounding down producing a coarse flag set. The Berger-Rigoutsos algorithm is then run on the coarse flag set producing a coarse patch set. Next the patch set is searched for neighbor constraint violations and patches are split until the neighbor constraints are met. An additional step is then taken in order to help facilitate a decent load balance. Large patches do not load balance well thus splitting patches larger than a specified threshold helps maintain load balance. The threshold used is equal to the average number of cells per processor multiplied by some percentage. We have found through experimentation that that a threshold of 6.25% works well. Finally the coarse patches are projected onto the fine grid by multiplying the location of patches by the minimum patch size producing a patch set that does not violate the neighbor constraint or the size constraint.

1.4 LOAD BALANCING

A separate load balancer component is responsible for assigning each detailed task to one processor. In patch-based AMR codes the load balance of the overall calculation can significantly affect the performance. Cost models are derived from the size of the patches, the type of physics, and in some cases the number of particles within the patch. A load balancer component attempts to distribute work evenly while minimizing communication by placing patches that communicate with each other on the same processor. The taskgraph described above provides a mechanism for analyzing the computation and communication within the computation to distribute work appropriately. One way to load balance effectively is by using this graph directly, where the nodes of the graph are weighted by the amount of work a patch has and the edges are weighted by the cost of the communication to other processors. Graph algorithms or matrix solves are then used to determine an optimal patch distribution. These methods do a decent job of distributing work evenly while also minimizing communication but are often too slow to run often.

Another method that has been shown to work well for load balancing computations is the use of space-filling curves [8, 32, 31]. The curve is used to create a linear ordering of patches. Locality is maintained within this ordering by the nature of the space-filling curve. That is, patches that are close together on the curve are also close together in space and are likely to communicate with each other. Once the patches are ordered according to the space-filling curve they are assigned a weight according to how the cost model. Then the curve is split into segments by recursively splitting the curve into two equally weighted segments until there is an equal number of curve segments as processors. Then each segment is assigned to a processor.

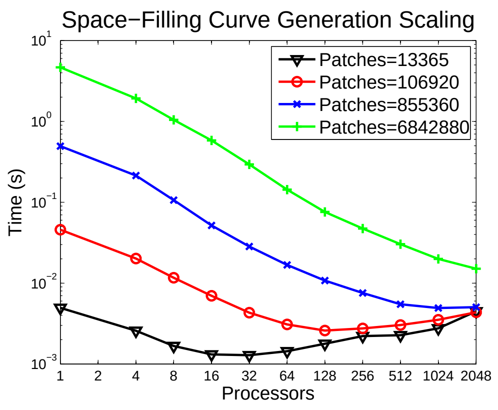

Fig. 1.8 The scalability of the SFC curve generation for multiple problem sizes

With AMR space-filling curves are preferable to the graph-based and matrix-based methods because the curve can be generated in $O(N \log N)$. In addition the curve can be generated quickly in parallel using parallel sorting algorithms [27]. Figure 1.8 shows the scalability of the curve generation up to thousands of processors for various problem sizes.

Data Migration After a new grid is created and load balanced the data must be migrated to the owning processors. The data is migrated by creating a taskgraph for the copying data from the old grid to the new grid. If a patch in the new grid is covering a region that does not exist in the old grid then a refinement task is scheduled that performs the interpolation from the coarser level to populate the data in the patch. The next task is to copy data from the old grid. Any patches within the old grid that overlap the new grid are copied to the new grid. The infrastructure handles copying data between processors whenever needed.

Task Graph Compilation As discussed earlier, the taskgraph is responsible for determining the order of task execution and communication that needs to occur between them. This information changes whenever the grid changes, so the taskgraph needs to be recompiled. Compiling the task graph can take a significant amount time. In the case of the W-cycle the compile time can be reduced by reusing redundant sections of the task graph. Each subtimestep has the same data dependencies as the previous subtimestep and thus only needs to be compiled once. This was done by creating task-subgraphs for each level. Each level then executes it's task-subgraph multiple times. This dramatically reduces the compile time for the W-cycle.

Grid Reuse Changing the grid and the corresponding steps that follow is expensive and does not scale as well as the computation. Regridding too often can greatly impact the overall scalability and performance. One way to reduce the overhead associated with changing the grid is to generate grids that can be used for multiple timesteps. This can be done by predicting where refinement will be needed and adding refinement before it is needed. In Uintah this is done by dilating the refinement flags prior to regridding. The refinement flags are modified by applying a stencil that expands the flags in all directions. The regridder then operates on the dilated refinement flags producing a grid that is slightly larger than is needed but can be reused for multiple timesteps. Prior to the dilation and regridding each timestep the undilated refinement flags are compared to the grid. If the refinement flags are all contained within the current grid regridding is not necessary.

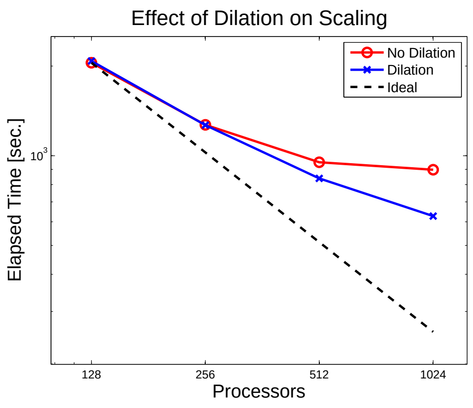

Fig. 1.9 The effect of dilation on scalability

Dilation can have a large impact on performance when using large numbers of processors. The time to compute on the grid scales better than the time for changing the grid. As the number of processors is increased the time to change the grid becomes the dominant component preventing further scalability. By dilating the refinement flags prior to regridding the time spent changing the grid can be significantly reduced keeping the computation on the grid the dominant component. The amount of dilation is controlled at runtime by measuring how much time is spent computing on the grid versus how much time is spent changing the grid. If the percentage of time spent changing the grid is not within a target range the dilation is either raised or lowered. Figure 1.9 shows the scalability with and without dilation for an expanding blast wave similar to the problem seen in Figure 1.4. In this simulation the regridder was configured to produce more patches than would typically be used in order to simulate the overhead due to patches that would be expected when running with larger numbers of processors.

1.5 AMR APPLIED TO A MULTI-MATERIAL EULERIAN CFD METHOD

Uintah contains four main simulation algorithms: 1) the Arches incompressible fire simulation code, 2) the ICE [24] compressible (both explicit and semi-implicit versions), 3) the particle-based Material Point Method (MPM) for structural modeling, and 4) a fluid-structure interaction method achieved by the integration of the MPM and ICE components, referred to locally as MPM-ICE. In addition to these primary algorithms, Uintah integrates numerous sub-components including equations of state, constitutive models, reaction models, radiation models and so forth. Here we provide a high-level overview of our approach to "full physics" simulations of fluid-structure interactions involving large deformations and phase change. By "full physics" we refer to problems involving strong coupling between the fluid and solid phases with a full Navier-Stokes representation of fluid phase materials and the transient, nonlinear response of solid phase materials, which may include chemical or phase transformation between the solid and fluid phases. Details of this approach, including the model equations and a description of their solution can be found at [11].

Multi-material dynamics The methodology upon which our software is built is a full "multi-material" approach in which each material is given a continuum description and defined over the computational domain. Although at any point in space the material composition is uniquely defined, the multi-material approach adopts a statistical viewpoint whereby the material (either fluid or solid) resides with some finite probability. To determine the probability of finding a particular material at a specified point in space, together with its current state (i.e., mass, momentum, energy), multi-material model equations are used. These are similar to the traditional single material Navier-Stokes equations, with the addition of two intermaterial exchange terms. For example, the momentum equation has, in addition to the usual terms that describe acceleration due to a pressure gradient and divergence of stress, a term that governs the transfer of momentum between materials. This term is typically modeled by a drag law, based on the relative velocities of two materials at a point. Similarly, in the energy equation, an exchange term governs the transfer of enthalpy between materials based on their relative temperatures. For cases involving the transfer of mass between materials, such as a solid explosive decomposing into product gas, a source/sink term is added to the conservation of mass equation. In addition, as mass is converted from one material to another, it carries with it its momentum and enthalpy, and these appear as additional terms in the momentum and energy equations, respectively. Finally, two additional equations are required due to the multi-material nature of the equations. The first is an equation for the evolution of the specific volume, which describes how it changes as a result of physical process, primarily temperature and pressure change. The other is a multi-material equation of state, which is a constraint equation that requires that the volume fractions of all materials in a cell sum up to unity. This constraint is satisfied by adjusting the pressure and specific volume in a cell in an iterative manner. This approach follows the ideas previously presented by Kashiwa and colleagues [20, 21, 22, 24].

These equations are solved using a cell-centered, finite volume version of the ICE (for Implicit, Continuous-fluid, Eulerian) method [14], further developed by Kashiwa and others at Los Alamos National Laboratory [23]. Our implementation of the ICE technique invokes operator splitting in which the solution consists of a separate Lagrangian phase where the physics of the conservation laws, including intermaterial exchange, is computed, and an Eulerian phase, where the material state is transported via advection to the surrounding cells. The method is fully compressible, allowing wide generality in the types of scenarios that can be simulated.

Fluid-Structure Interaction using MPM-ICE The fluid-structure interaction capability is achieved by incorporating the Material Point Method (described briefly in the following section) within the multi-material CFD formulation. In the multi-material formulation, no distinction is made between solid or fluid phases. Where distinctions do arise is in the computation of the material stress and material transport or advection. In the former, the use of a particle description for the solid provides a convenient location upon which to store the material deformation and any relevant history variables required for say, a plasticity model. Regarding the latter, Eulerian advection schemes are typically quite diffusive, and would lead to an unacceptable smearing of a fluid-solid interface. Thus, performing advection by moving particles according to the local velocity field eliminates this concern.

At the beginning of each timestep, the particle state is projected first to the computational nodes, and then to the cell centers. This co-locates the solid and fluid data, and allows the multi-material CFD solution to proceed unaware of the presence of distinct phases. As stated above, the constitutive models for the solid materials are evaluated at the particle locations, and the divergence of the stress at the particles is computed at the cell centers. The Lagrangian phase of the CFD calculation is completed, including exchange of mass, momentum and enthalpy. At this point *changes* to the solid materials' state are interpolated back to the particles, and their state, including position, is updated. Fluid phase materials are advected using a compatible advection scheme such as that described in [35]. Again, a full description can be found in [11].

1.5.1 Structured AMR for ICE

In addition to solving the multi-material equations on each level there are several basic operators or steps necessary to couple the solutions on the various levels. These steps include:

- Refine: the initial projection of the coarse level data onto to the fine level.
- Refine coarse-fine interface: projection of the coarse level solution to the fine level, only in the ghost cells at the coarse-fine interface.
- Coarsen: projection of the fine level solution onto the coarse level at the end of the timestep.
- Refluxing: correction of the coarse level solution due to flux inconsistencies.
- Refinement criteria: flagging cells that should be refined.

Refine: When a new fine level patch is generated, data from the coarse level is projected onto the new patch using either a tri-linear or tri-quadratic interpolation. Figure 1.10 shows the computational footprint or stencil used during a tri-quadratic interpolation in one plane. The dashed red lines show the edges of the fine level ghost cells, along the coarse-fine interface. The large open circles show the underlying coarse level cells used during the projection of data to the pink diamond. Note that multi-field CFD algorithm requires one layer of ghost cells on all patch sides that do not have an adjacent neighbor. These ghost cells provide boundary conditions for the fine level solution.

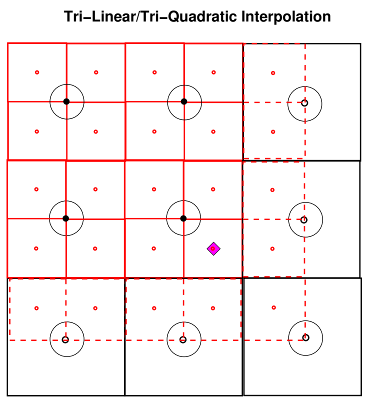

Fig. 1.10 Tri-quadratic interpolation stencil used during the projection of coarse level data on to a fine level

Refine coarse-fine interface: At the end of each timestep the boundary conditions or ghost cells on the finer levels are updated by performing both a spatial and a linear temporal interpolation. The temporal interpolation is required during the W-cycle on all finer levels, shown as blue lines in Figure 1.6. When the lock-step cycle is used there is no need for temporal interpolation. Spatially, either a tri-linear or tri-quadratic interpolation method may be used. Details on the quadratic interpolation method are described in [30].

Coarsen: At the end of each timestep the conserved quantities are conservatively averaged from the fine to coarse levels using

$$Q_{CoarseLevel} = \frac{1}{Z} \sum_{c=1}^{RefinementRatio} Q_{FineLevel} \tag{1.1}$$

Refluxing: As mentioned above, the solution to the governing equations occurs independently on each level. During a timestep, at the coarse-fine interfaces, the computed fluxes on the fine level may not necessary equal the corresponding coarse level fluxes. In Figure 1.11 the length of the arrows indicate a mismatch.

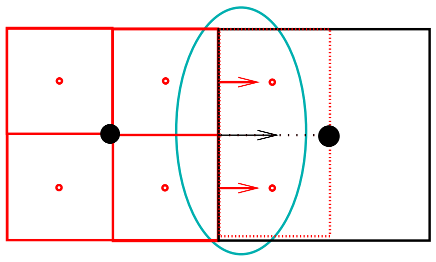

Fig. 1.11 Illustration of a flux mismatch at the coarse-fine interface

To maintain conservation of the conserved quantities, a correction, as described in [6] is added to all coarse level cells which are adjacent to the coarse-fine interfaces. For the W-cycle the fluxes of mass, momentum, energy and any number of conserved scalars are corrected using

$$Q_{correction} = Q^{C}_{flux} - \frac{1}{Z} \sum_{t=1}^{subtimesteps} \sum_{f=1}^{faces} Q_{Flux} F \tag{1.2}$$

where superscripts C and F represent the coarse and fine levels, respectively. For the lockstep execution cycle the correction is given by

$$Q_{correction} = Q^{C}_{flux} - \frac{1}{Z} \sum_{f=1}^{faces} Q^{F}_{Flux} \tag{1.3}$$

Experience has shown that the most difficult part of computing the correction fluxes lies in the bookkeeping associated with the summation terms in equations 1.2 and 1.3. For each coarse level cell there are n_x, n_y, n_z overlying fine level cells with faces that overlap the coarse cell face. Keeping track of these faces and fluxes, on multiple levels, in a highly adaptive 3D mesh, where new fine level patches are constantly being generated, has proven to be difficult, see Figure 1.12.

Regridding Criteria: A cell on a coarser level is flagged to be refined whenever a user defined threshold has been exceeded. Currently, for fluid calculations, the magnitude

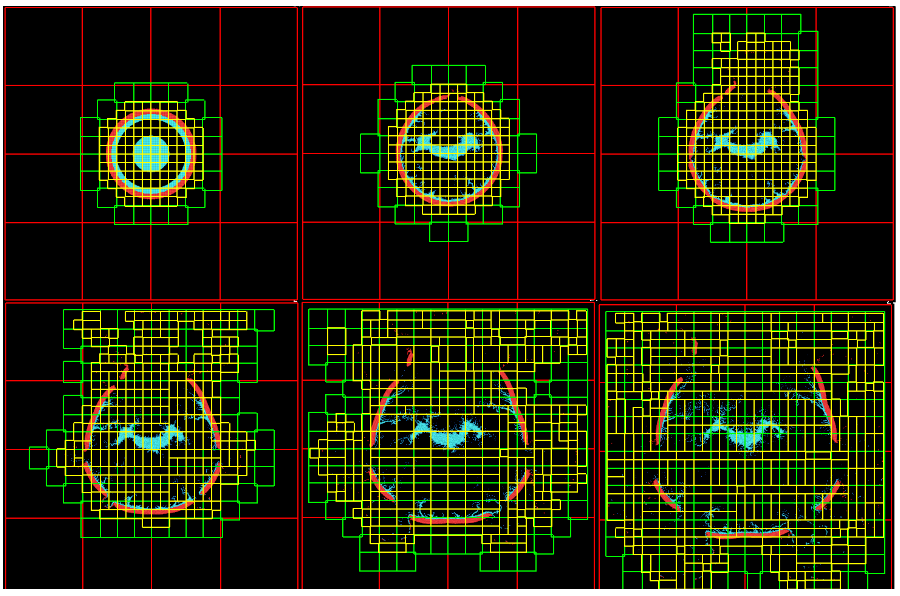

Fig. 1.12 Simulation of an exploding container on a 3-level adaptive mesh. The solid lines outline patches on the individual levels, red: level 0, green: level 1, yellow: level 2

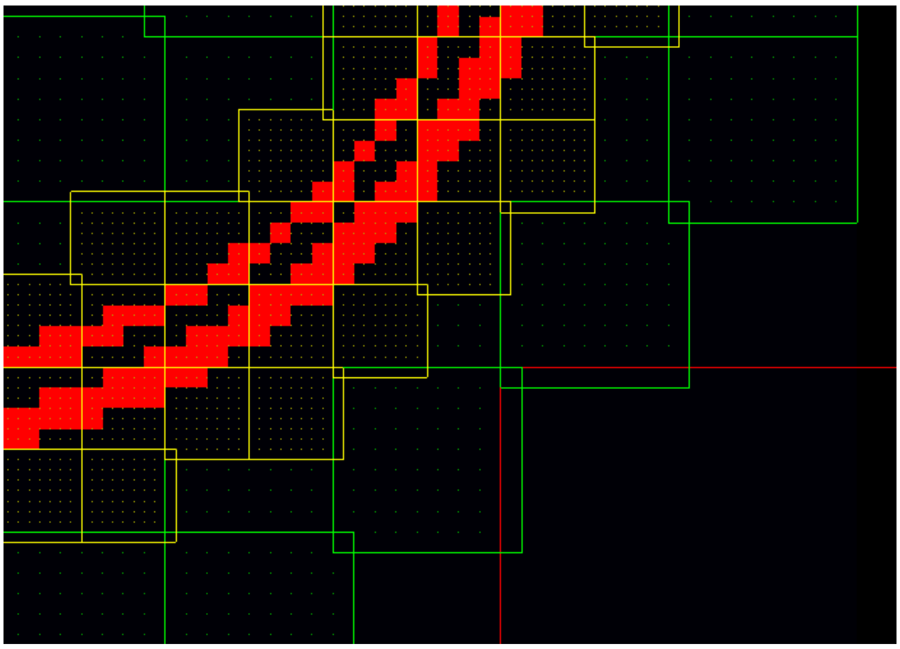

Fig. 1.13 Close up of a 3 level adaptive grid. The colored lines show the outline of the individual patches, red: level 0, green: level 1, yellow: 2. The solid red squares show cells that have been flagged for refinement

of the gradient: density, temperature, volume fraction, pressure, or a passive scalar may be used. More sophisticated techniques for flagging cells are available, but have not been implemented, an example is described in [1]. In Figure 1.13 the red squares show cells that have exceeded a user specified threshold for the magnitude of the gradient of the pressure and are flagged.

Pressure Field: There are two methods implemented for solving for the change in pressure (ΔP) which is used to evolve the multi-material governing equations. The explicit formulation, used for compressible calculations, uses the W-cycle where ΔP is computed in a point-wise fashion on each of the individual levels, without inter-level communication. The governing equations evolve on each of the levels independently with inter-level communication only taking place at the end of a timestep or whenever a new patch is generated. The price paid for this simplicity is the small timestep that is required to guarantee stability. The timestep size is computed based on the convective *and* acoustic velocities.

The semi-implicit technique, which is still being actively developed for multi-level grids, utilizes the lockstep execution cycle. A Poisson equation for ΔP is first solved on a composite grid and then the solution is projected onto the underlying coarse level cells. Experience has shown that any mismatch in the pressure gradient

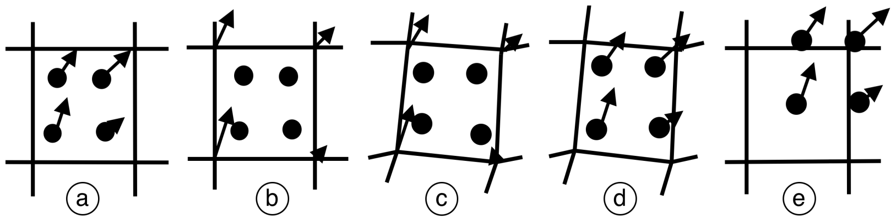

Fig. 1.14 Graphical depiction of the steps in the MPM algorithm

at the coarse-fine interfaces will produce non-physical velocities, thereby creating a feedback mechanism for polluting the pressure field in the subsequent timestep. The potential benefit of this method is the larger timestep. In this case the timestep required to remain stable is computed based on *only* the convective velocity.

1.5.2 MPM with AMR

Uintah contains a component for solid mechanics simulations known as the Material Point Method (MPM)[33]. MPM is a particle based method that uses a (typically) Cartesian mesh as a computational scratch-pad upon which gradients and spatial integrals are computed. MPM is an attractive computational method in that the use of particles simplifies initial discretization of geometries and eliminates problems of mesh entanglement at large deformations (see, e.g.[7]), while the use of a Cartesian grid allows the method to be quite scalable by eliminating the need for directly computing particle-particle interactions. A graphical description of MPM is shown in Figure 1.14. There, a small region of solid material, represented by four particles, overlayed by a portion of grid is shown in panel (a). The particles, or material points, carry minimally, mass, volume, velocity, temperature, and state of deformation. Panel (b) reflects that this particle information is projected to the nodes of the overlying grid, or scratch-pad. Based on the nodal velocity, a velocity gradient is computed at the particles, and is used to update the state of deformation of the particle, and the particle stress. Based on the internal force (divergence of the stress) and external forces, as well as the mass, acceleration is computed at the nodes, as is an updated velocity. This is shown in panel (c), where the computational nodes have been deformed based on the new velocity and the current timestep size. Note that the deformation of the grid is never explicitly performed. Panel (d) reflects that the changes to the field quantities, specifically velocity and position in the case, are interpolated back to the particles. Once the particle state has been updated, the grid is reset in preparation for the next timestep (Panel (e)).

Development of multi-level MPM is an active area of research that is still in the early stages. Ma, Lu and Komanduri [29] reported the first implementation, in which they used the Structured Adaptive Mesh Application Interface (SAMRAI)[16] as a vehicle to handle the multi-level grid information. While the Ma, et al. approach makes a huge step towards realizing general purpose, multi-level MPM, it does so in a manner that is quite different from standard SAMR algorithms. The brief description of their work, given below, will highlight some of these differences, and illustrate some of the remaining challenges ahead.

A region of a simple 2-level grid, populated with one particle in each cell, is shown in Figure 1.15. Two of the nodes at the coarse-fine interface, denoted by filled diamonds, are highlighted for discussion. First consider the node at (4,5). The work of Ma describes a treatment of these nodes, which could be considered mid-side nodes on the coarse mesh, using what they term "zones of influence". This refers to the region around each node, within which particles both contribute to nodal values, and receive contributions from the nodal values. The zones of influence for the node at (4,5) and (8,4) are highlighted. What these are meant to indicate is that the data values at fine level nodes and particles depend on coarse level nodes and particles, and vice versa. This is further complicated by the fact that interactions between particles and grid occur multiple times each timestep (projection to grid, calculation of velocity gradient at particles, calculation of divergence of particle stress to the nodes, update of particle values from nodes).

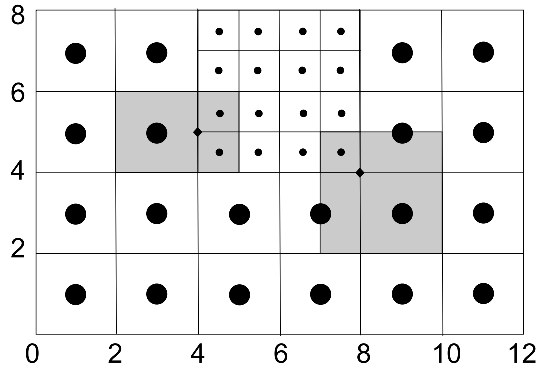

Fig. 1.15 Sample region of a multi-level MPM grid

The result of this is that the multi-level MPM algorithm behaves much more as if it exists on a composite grid than on a traditional SAMR grid. For instance, there is no coarse level solution in regions where a finer level exists. Additionally, dataflow across the coarse-fine interface is much less hierarchical. That is, the solution on one level depends equally on data contributions from both finer and coarser levels. This loss of hierarchy has thus far also frustrated efforts at developing a "W-cycle" type approach to temporal refinement. To date, no temporal refinement approaches have been described in the literature.

Finally, the approach described in Ma, et al., did not address adaptivity, only a static, multi-level solution. An adaptive technique would require further considerations, including particle splitting and merging. Work by Lapenta and Brackbill[26] provides a starting point for this task, although quantities such as history dependent plasticity variables are not addressed there.

Given the significant challenges associated with a general adaptive Material Point Method, its development within Uintah is at the early stages. The strategy of the Uintah developers has been to maintain particles only at the finest level in the solution.

In simulations only involving MPM, this strategy does not provide substantial time savings, given that computational cost is most strongly a function of number of particles, and only weakly depends on the number of computational cells/nodes. Where this strategy does prove to be advantageous is in the fluid-structure interaction simulations carried out within Uintah, in which MPM is incorporated within the multi-material CFD code described in the previous section (see, e.g.[11]). For the Eulerian CFD code, cost is entirely dependent upon number of computational cells, so refining only in the vicinity of the solid objects allows for large regions of the domain to be at lower resolutions. This strategy also makes sense from a physical point of view, in that regions near the solid object generally demand the highest resolution in order to capture the fluid-solid interaction dynamics. In addition, for the types of simulations that Uintah was built to address, resolution requirements *within* the solid are usually the most demanding, in order to capture phenomena such as solid phase reactions and metal failure. Normal error cell flagging still occurs within the rest of the domain, such that regions of the domain that contain only fluid will be refined as well should they require it. Examples of success with this strategy are provided in the next section.

1.6 RESULTS

Within the C-SAFE project, we utilize Uintah to perform simulations of both fires and explosions using hundreds to a few thousand processors. One scenario of two containers stored in close proximity is shown in Figure 1.16. In this simulation the adaptive grid (not shown) was comprised of three levels, with a refinement ratio of 4 in each direction. The results from this simulation are counterintuitive in that the container with a smaller initial explosive charge (right) produced a more violent explosion. This is due to the rapid release of energy that results from the structural collapse of the explosive into the hollow bore.

Figure 1.19 shows another 2D simulation of one quarter of two steel (red) containers filled with burning explosive (blue). Symmetry boundary conditions are applied on all sides. The containers and contents are preheated to the ignition temperature at t=0. As the explosive decomposes, the containers are pressurized and break apart shortly thereafter. Eventually, the container fragments collide in the center. This simulation demonstrates how AMR allows us efficiently simulate arrays of explosive devices, separated by a significant distance. For multiple containers in 3D, the cost of such simulations would be prohibitive if the whole domain required the finest resolution everywhere during the entire simulated time.

Figure 1.18 depicts a rigid 20 degree wedge, represented by MPM particles, moving through initially still air. Contours indicate pressure, while the pink boxes reflect regions of refinement in the 2 level solution. In addition to refinement near the wedge, regions of large pressure gradient around the leading shock and trailing expansion fan are also automatically refined. This simulation also served as an early validation of ICE method and the fluid-structure interaction formulation, as the

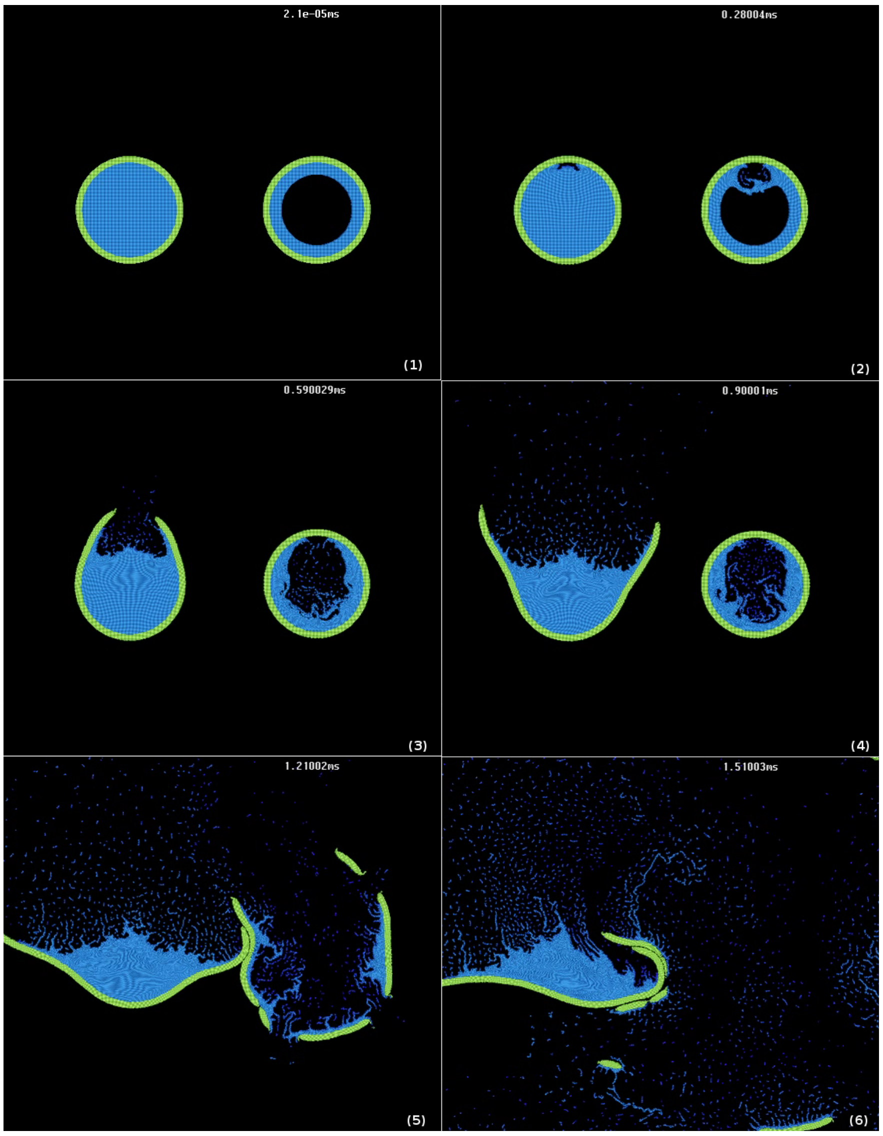

Fig. 1.16 Simulation of two exploding containers with different initial configurations of the explosive. The left container was initially full of explosive while the right container had a hollow core.

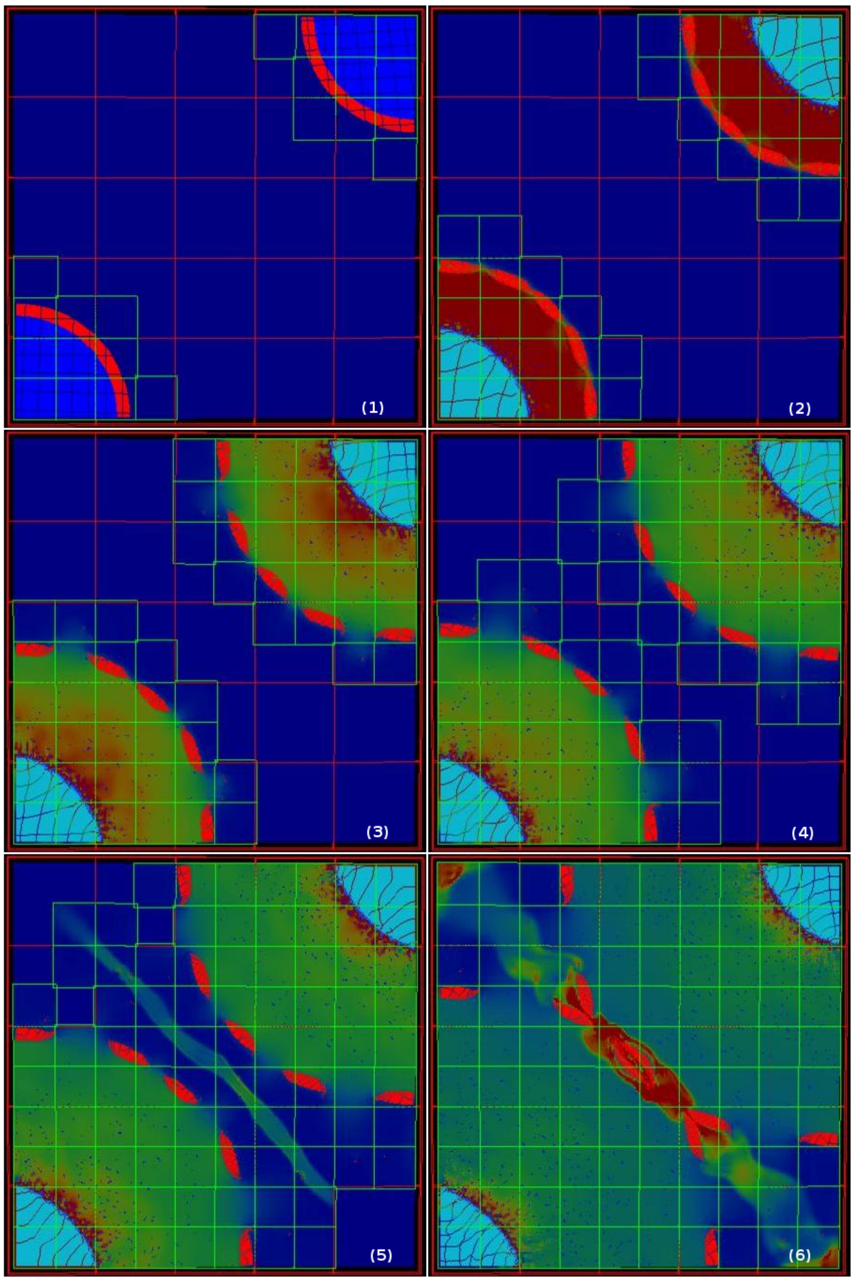

Fig. 1.17 Temporal evolution of two steel containers initially filled with a high explosive at a temperature above the threshold ignition temperature. The background contour plot shows the magnitude of pressure and the particles are colored by mass. The solid green lines show the outline of the fine level patches and the red lines correspond to the coarse level patches.

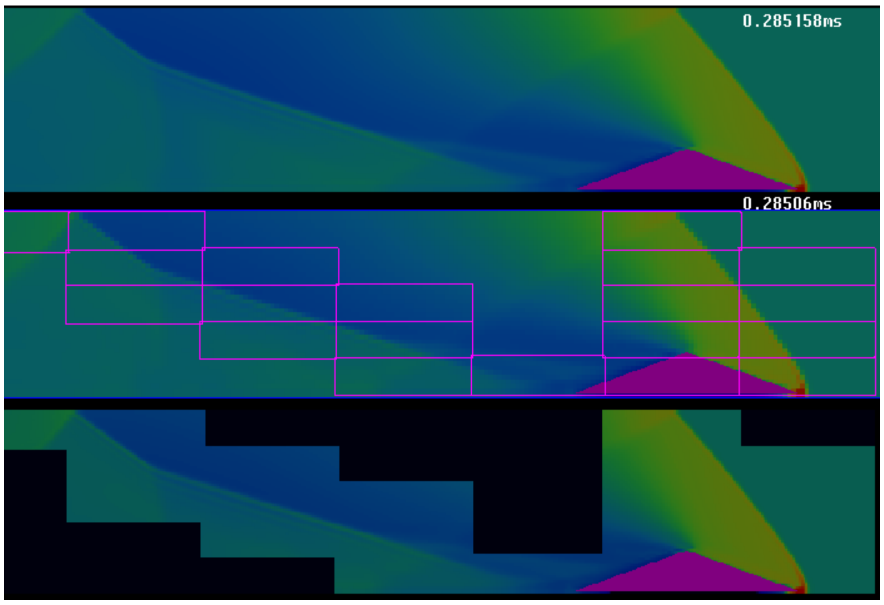

Fig. 1.18 A 2D wedge traveling at Mach 2, contour plot of the pressure field. Results from a single level and an 2 level simulation are shown, top: single level solution, middle: coarse level pressure field, bottom: fine level pressure field

expected shock angle for this configuration is known, and compared very favorably to the computed result.

Figure 1.20 shows slices of the temperature field, from a simulation of a JP8 jet flame, issuing at $50m/s$ from a $0.2m$ hole in the floor of the computational domain. The grid consisted of 4 static levels with 128^3 cells on each level and the computational domain spanned $12.8m$ in each direction. This simulation was performed on 600 processors.

One of the scientific questions that Uintah is currently being used to address is to predict how the response of an explosive device that is heated by a jet fuel fire changes with a variety of conditions. These include fire size, container location relative to the fire, and the speed and direction of a cross-wind. A total of 12 simulations are being performed to investigate this question. The current state of one of those cases is shown in Figure **??**. This simulation is being carried out on 500 processors of the Atlas cluster at LLNL. The grid utilizes four levels of refinement with a refinement ratio of 4, with the finest level at 1 mm. The figure shows a cut-away view of an isosurface of the steel container (blue) which has partially ruptured. The density of the products of reaction are volume rendered, and the void space inside the container is where the remaining unburned explosive lies. It was not explicitly shown in this view for the sake of clarity. As these simulations come to completion, data analysis of fragment size and velocity will be used as an indicator of violence of explosion.

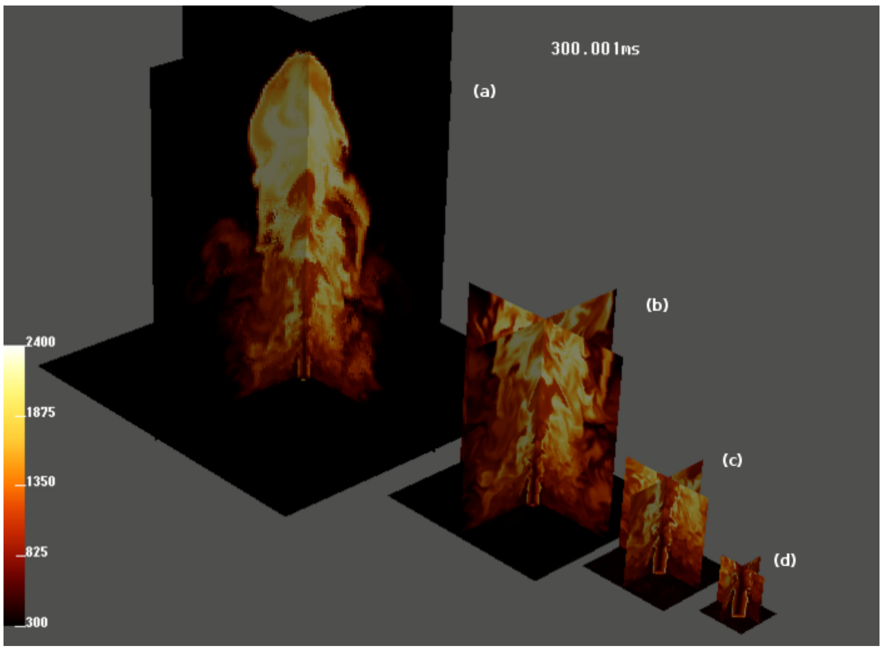

Fig. 1.19 Temperature distribution of a JP8 jet flame on 4 static levels, 128^3 cells per level, (a) L-1: $12.8m^3$, (b) L-2: $6.4m^3$, (c) L-3: $3.2m^3$, (d) L-4: $1.6m^3$.

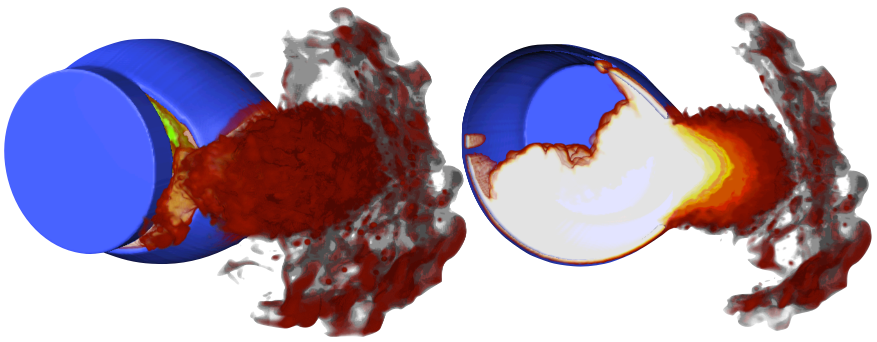

Fig. 1.20 Full adaptive 3D simulation of an explosive device subjected to a hydrocarbon fire, shown just as the pressure is release through a rapture in the steel container. The view on the left shows the outside of the container, while the view on the right shows a cross section of the same data.

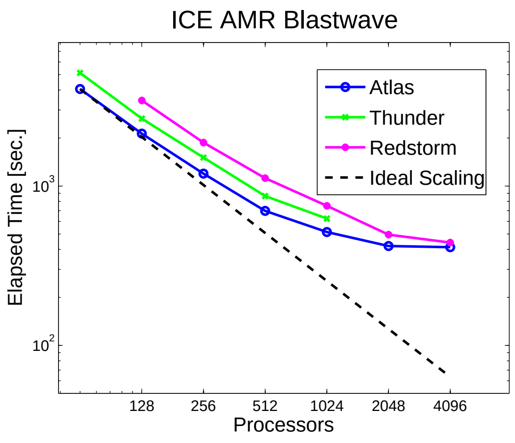

Fig. 1.21 The scalability of AMR ICE

Figure 1.17 shows the scalability of Uintah running a 3 level 3D ICE problem similar to the problem seen in Figure 1.4 on multiple architectures. Thunder is a Linux cluster located at LLNL with 1024 quad processor Intel Xeon nodes, connected via a Quadrics switch. Atlas is also Linux cluster located at LLNL with 1152 nodes each with eight AMD Opteron processors, also connected via a Quadrics switch. Red storm is an Opteron-based supercomputer located at Sandia with 12920 dual-core CPUs, connected with a proprietary Cray interconnect. Although the code does not scale perfectly for AMR problems, it still dramatically outperforms the non-AMR equivalents. Figure **??** (left) illustrates another challenge – that as the container expands, the number of refined cells increases consistently. However, the performance gain is still substantial as indicated by the right side of Figure **??** as the total number of cells over all levels is just over 1% of the number required by a single-level grid. A non-AMR version of the explosion simulations shown in Figure **??** would impose unacceptable limitations on the domain size and grid resolution, or would require nearly a two order of magnitude increase in computational resources.

1.7 CONCLUSIONS AND FUTURE WORK

We have presented a framework and application for using adaptive methods for complex multi-physics simulations of explosive devices. This framework uses an explicit representation of parallel computation and communication to enable integration of parallelism across multiple simulation methods.

We have also discussed several simulations that are performed routinely using this framework, including the heat-up, pressurization and consequent rupture of an incendiary device. Using adaptive methods, we are able to perform simulations that would otherwise be inaccurate or computationally intractable.

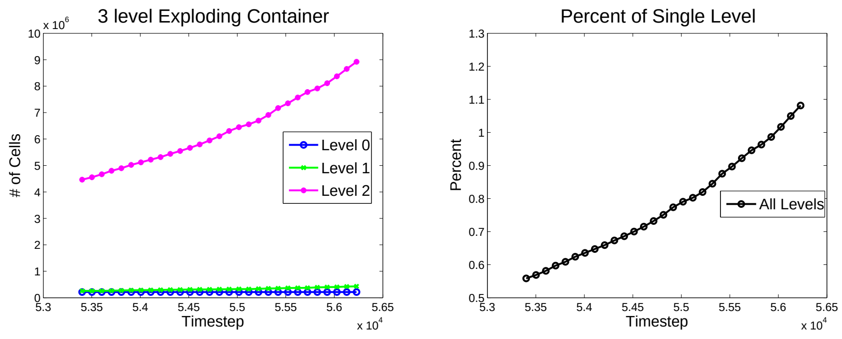

Fig. 1.22 **Left:** Number of cells at each level of refinement for the exploding container shown in Figure **??**, shown just as the container begins to rupture, **Right:** Total number of cells at all refinements levels, expressed as a percentage of the cells required for a non-adaptive simulation.

In addition to the adaptive explosion simulations describe above, Uintah is a general-purpose simulation capability that has been used for a wide range of applications. MPM has been used to simulate a penetrating wound in a comprehensive cardiac/torso model [19, 17, 18], with the goal of understanding mechanics of wounding in an effort to improve the chances of recovery from projectile wounds. It has also been used to study biomechanical interactions of angiogenic microvessels with the extracellular matrix on the microscale level, by developing and applying novel experimental and computational techniques to study a 3D in-vitro angiogenesis model [12]. The Uintah fluid-structure interaction component (MPM-ICE), is being employed in a study of phonation, or the production of sound, in human vocal folds. This investigation has determined the ability of Uintah to capture normal acoustical dynamics, including movement of the vocal fold tissue, and will consider pathological laryngeal conditions in the future. Uintah has also been used by collaborators at Los Alamos National Laboratory to model the dynamic compaction of foams [4, 7], such as those used to isolate nuclear weapons components from shock loading.

Achieving scalability and performance for these types of applications is an ongoing battle. We are continuing to to evaluate scalability on larger number of processors, and will reduce scaling bottlenecks that will appear when using these configurations. Intertask communication is a currently a known bottleneck on large numbers of processors. We aim to reduce this by using the taskgraph to transparently trade off redundant computation with fine-grained communication. We also have a few situations where extra data is sent, which may improve scalability for large processor counts. Finally, we are exploring methods to reduce communication wait time by dynamically reordering the execution of tasks as their MPI dependencies are satisfied leading to more overlap of communication and computation.

This work was supported by the U.S. Department of Energy through the Center for the Simulation of Accidental Fires and Explosions, under grant W-7405-ENG-48. We thank each of the members of the interdisciplinary C-SAFE team at the University of Utah for their contributions to the Uintah software.

References

1. M.J. Aftosmis and M.J. Berger. Multilevel error estimation and adaptive h-refiment for cartesian meshes with embedded boundaries. In *40th AIAA Aerospace Sciences Meeting and Exhibit*, number AIAA 2002-0863, Calagary, AB, 2002. AIAA.

2. S. Atlas, S. Banerjee, J.C. Cummings, P.J. Hinker, M. Srikant, J.V.W. Reynders, and M. Tholburn. A high-performance distributed simulation environment for scientific applications. In *Proceedings of Supercomputing*. Supercomputing, 1995.

3. Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A.M. Bruaset, and H.P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

4. S.G. Bardenhagen, A.D. Brydon, and J.E. Guilkey. Insight into the physics of foam densification via numerical solution. *J. Mech. Phys. Solids*, 53:597–617, 2005.

5. Marsha Berger and Isidore Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Trans. Systems Man Cybernet.*, 21(5):1278–1286, 1991.

6. M.J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:64–84, 1989.

7. A.D. Brydon, S.G. Bardenhagen, E.A. Miller, and G.T. Seidler. Simulation of the densification of real open-celled foam microstructures. *Journal of the Mechanics and Physics of Solids*, 53:2638–2660, 2005.

8. Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty, and Luis G. Gervasio. New challanges in dynamic load balancing. *Appl. Numer. Math.*, 52(2-3):133–152, 2005.

9. R.D. Falgout, J.E. Jones, and U.M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners, chapter in numerical solution of partial differential equations on parallel computers. *Bruaset and A. Tveito*, 51:267–294, 2006.

10. J.T. Feo, D.C. Cann, and R.R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10:349–366, 1990.

11. J.E. Guilkey, T.B. Harman, and B. Banerjee. An eulerian-lagrangian approach for simulating explosions of energetic devices. *Computers and Structures*, 85:660–674, 2007.

12. J.E. Guilkey, J.B Hoying, and J.A. Weiss. Modeling of multicellular constructs with the material point method. *Journal of Biomechanics*, 39:2074–2086, 2007.

13. Brian T. N. Gunney, Andrew M. Wissink, and David A. Hysom. Parallel clustering algorithms for structured amr. *J. Parallel Distrib. Comput.*, 66(11):1419–1430, 2006.

14. F.H. Harlow and A.A. Amsden. Numerical calculation of almost incompressible flow. *J. Comp. Phys.*, 3:80–93, 1968.

15. T.C. Henderson, P.A. McMurtry, P.G. Smith, G.A. Voth, C.A. Wight, and D.W. Pershing. Simulating accidental fires and explosions. *Comp. Sci. Eng.*, 2:64–76, 1994.

16. R.D. Hornung and S.R. Kohn. Managing application complexity in the samrai object-oriented framework. *Concurrency and Computation Practice and Experience*, 14:347–368, 2002.

17. I. Ionescu, J. Guilkey, M. Berzins, R.M. Kirby, and J. Weiss. *Computational Simulation of Penetrating Trauma in biological Soft Tissues using the Material Point Method.* IOS Press Amsterdam, 2005.

18. I. Ionescu, J. Guilkey, M. Berzins, R.M. Kirby, and J. Weiss. *Ballistic Injury Simulation using the Material Point Method.* IOS Press Amsterdam, 2006.

19. I. Ionescu, J.E. Guilkey, M. Berzins, R.M. Kirby, and J.A. Weiss. Simulation of soft tissue failure using the material point method. *Journal of Biomechanical Engineering*, 128:917–924, 2006.

20. B.A. Kashiwa. A multifield model and method for fluid-structure interaction dynamics. Technical Report LA-UR-01-1136, Los Alamos National Laboratory, Los Alamos, 2001.

21. B.A. Kashiwa and E.S. Gaffney. Design basis for cfdlib. Technical Report LA-UR-03-1295, Los Alamos National Laboratory, Los Alamos, 2003.

22. B.A. Kashiwa, M.L. Lewis, and T.L. Wilson. Fluid-structure interaction modeling. Technical Report LA-13111-PR, Los Alamos National Laboratory, Los Alamos, 1996.

23. B.A. Kashiwa and R.M. Rauenzahn. A cell-centered ICE method for multiphase flow simulations. Technical Report LA-UR-93-3922, Los Alamos National Laboratory, Los Alamos, 1994.

24. B.A. Kashiwa and R.M. Rauenzahn. A multimaterial formalism. Technical Report LA-UR-94-771, Los Alamos National Laboratory, Los Alamos, 1994.

25. G. Krishnamoorthy, S. Borodai, R. Rawat, J.P. Spinti, and P.J. Smith. Numerical modeling of radiative heat transfer in pool fire simulations. Orlando, Florida, 2005. ASME International Mechanical Engineering Congress (IMECE).

26. G. Lapenta and J.U. Brackbill. Control of the number of particles in fluid and mhd particle in cell methods. *Computer Physics Communications*, 87:139–154, 1995.

27. Justin Luitjens, Martin Berzins, and Tom Henderson. Parallel space-filling curve generation through sorting: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(10):1387–1402, 2007.

28. Justin Luitjens, Bryan Worthen, Martin Berzins, and Tom Henderson. *Petascale Computing Algorithms and Applications*, chapter Scalable parallel amr for the uintah multiphysics code. Chapman and Hall/CRC, 2007.

29. J. Ma, H. Lu, and R. Komanduri. Structured mesh refinement in generalized interpolation material point (GIMP). *Computer Modeling in Engineering and Sciences*, 12:213–227, 2006.

30. D. Martin. *An Adaptive Cell-centered Projection Method for the Incompressible EulerEquations*. PhD thesis, University of California, Berkeley, 1998.

31. Mausumi Shee, Samip Bhavsar, and Manish Parashar. Characterizing the performance of dynamic distribution and load-balancing techniques for adaptive grid hierarchies. In *IASTED: International Conference on Parallel and Distributed Computing and Systems*, Calagary, AB, 1999. IASTED.

32. Johan Steensland, Stefan Söderberg, and Michael Thuné. A comparison of partitioning schemes for blockwise parallel samr algorithms. In *PARA '00: Proceedings of the 5th International Workshop on Applied Parallel Computing, New Paradigms for HPC in Industry and Academia*, pages 160–169, London, UK, 2001. Springer-Verlag.

33. D. Sulsky, Z. Chen, and H.L. Schreyer. A particle method for history dependent materials. *Comput. Methods Appl. Mech. Engrg.*, 118:179–196, 1994.

34. S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith. Smarts: Exploiting temporal locality and parallelism through vertical execution, 1999.

35. W.B. VanderHeyden and B.A. Kashiwa. Compatible fluxes for van leer advection. *J. Comp. Phys.*, 146:1–28, 1998.

36. Andrew M. Wissink, Richard D. Hornung, Scott R. Kohn, Steve S. Smith, and Noah Elliott. Large scale parallel structured amr calculations using the

samrai framework. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 6–6, New York, NY, USA, 2001. ACM Press.

37. Andrew M. Wissink, David Hysom, and Richard D. Hornung. Enhancing scalability of parallel structured amr calculations. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 336–347, New York, NY, USA, 2003. ACM Press.