A Component-based Architecture for Parallel Multi-Physics PDE Simulation

Steven G. Parker¹

Scientific Computing and Imaging Institute
School of Computing
50 S. Central Campus Dr., Room 3490
University of Utah
Salt Lake City Utah 84112, USA,
sparker@cs.utah.edu,
<http://www.cs.utah.edu/~sparker>

Abstract. We describe the Uintah Computational Framework (UCF), a set of software components and libraries that facilitate the simulation of Partial Differential Equations (PDEs) on Structured AMR (SAMR) grids using hundreds to thousands of processors. The UCF uses a non-traditional approach to achieving parallelism, employing an abstract taskgraph representation to describe computation and communication. This representation has a number of advantages that affect the performance of the resulting simulation. We demonstrate performance of the system on a solid mechanics algorithm, two different Computational Fluid-Dynamics (CFD) algorithms, and coupled CFD/solids algorithms. We illustrate the performance of the UCF for jobs requiring up to 2000 processors.

1 Introduction

Computational scientists continue to push the capabilities of current computer hardware to the limits in order to simulate complex real-world phenomena. These simulations necessitate the use of ever increasing computational resources. Furthermore, the software written to model real-world scientific and engineering problems is typically very complex. Grid generation, non-linear and linear solvers, visualization systems, and parallel runtime systems all combine to provide a very powerful environment for solving complex scientific and engineering problems. Such complexities are further compounded when multiple simulation codes are combined to simulate the interaction of multiple phenomena.

The Uintah Computational Framework (UCF) is a set of software components and libraries that facilitate the simulation of Partial Differential Equations (PDEs) on Structured AMR (SAMR) grids using hundreds to thousands of processors. The UCF uses a non-traditional approach to achieving parallelism, employing an abstract taskgraph representation to describe computation and communication. This 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.

Fig. 1. A typical C-SAFE problem involving hydrocarbon fires and explosions of energetic materials.

In this paper, we will describe the taskgraph concept, along with a number of the implementation details. We will discuss the advantages and disadvantages of this methodology, and will present results from UCF simulations.

2 Overview

The system described here involves several connections between a number of different projects, including C-SAFE, CCA, SCIRun and Uintah. In order to adequately describe how they all interact, we first give an overview of the individual pieces in this section. The remainder of the paper will discuss how those pieces fit together to achieve large-scale parallel simulations.

2.1 C-SAFE

In 1997, the University of Utah created an alliance with the DOE Accelerated Strategic Computing Initiative (ASCI) to form the Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [1, 2]. C-SAFE focuses specifically

on providing state-of-the-art, science-based tools for the numerical simulation of accidental fires and explosions, especially within the context of handling and storage of highly flammable materials. The primary objective of C-SAFE is to provide a software system comprising a PSE in which fundamental chemistry and engineering physics are fully coupled with non-linear solvers, optimization, computational steering, visualization and experimental data verification. The availability of simulations using this system will help to better evaluate the risks and safety issues associated with fires and explosions. Our goal is to integrate and deliver a system that is validated and documented for practical application to accidents involving both hydrocarbon and energetic materials. Efforts of this nature requires expertise from a wide variety of academic disciplines. A typical C-SAFE problem is shown in Figure 2.1.

2.2 CCA

The CCA forum [3, 4] was established by a group of researchers from several Department of Energy national laboratories, and several universities to address the need for a software component architecture that fulfilled the needs of high-performance computing. The CCA architecture aims to provide higher performance, explicit support for multi-dimensional arrays, and explicit support for parallelism. Uintah, described below, is one implementation of the CCA architecture which we use as a research vehicle for experimenting with these ideas and for exercising their efficacy on a complex scientific application, such as C-SAFE simulations.

2.3 SCIRun

SCIRun¹ is a scientific problem solving environment that allows the interactive construction and steering of large-scale scientific computations [5–7]. A scientific application is constructed by connecting computational elements (modules) to form a program (network). This program may contain several computational elements as well as several visualization elements, all of which work together in orchestrating a solution to a scientific problem. Geometric inputs and computational parameters may be changed interactively, and the results of these changes provide immediate feedback to the investigator. SCIRun is designed to facilitate large-scale scientific computation and visualization on a wide range of machines from the desktop to large supercomputers.

2.4 Uintah

C-SAFE's Uintah Problem Solving Environment [8, 9] is a massively parallel, component-based, PSE designed to simulate large-scale scientific problems, while

¹ Pronounced “ski-run.” SCIRun derives its name from the Scientific Computing and Imaging (SCI) Institute at the University of Utah.

allowing the scientist to interactively visualize, steer, and verify simulation results. Uintah is a derivative of the SCIRun PSE and adds support for the more powerful CCA component model and support for distributed-memory parallel computers.

3 Component Architecture

Due to space constraints, we will only briefly describe the overall Uintah architecture here. Further details can be found in [4, 8, 9], as well as in future publications from C-SAFE.

The Uintah Component Architecture is based on the CCA component architecture. Since the CCA component architecture is an evolving standard, we focused on a subset of the overall standard in order to focus on C-SAFE simulations. In particular, we created a C++ only implementation and ignored the multi-language features that were still under development.

The primary feature of the CCA is its port model. The port model, also called gPorts (for generalized ports), is the mechanism by which components communicate. In C++, this looks like little more than an abstract class on which the caller can perform method invocations. Components *provide* a set of interfaces, which other components can *use*. At component creation time, the component declares a set of Provides and Uses ports. An external entity, called a builder, connects the provides port of one component to the uses port of another component. For the simulations described here, this connection is provided by a stand-alone main program. However, in general, the builder can be a script, a graphical user interface, or even another component.

The CCA architecture utilizes the Scientific Interface Definition Language (SIDL) [10]. SIDL is the mechanism by which component interfaces are described. A SIDL compiler generates code for inter-language operation, as well as, for remote method invocations in a distributed memory environment. Although Uintah contains support for such distributed memory operations, the component interactions described here all occur within a single address space. Operation in a distributed-memory parallel environment occurs with MPI calls within individual components.

4 Uintah Computational Framework

The Uintah Computational Framework (UCF) is implemented in the context of the Uintah PSE Environment (PSE). The Uintah PSE Architecture is very general, facilitating a wide range of computational and visualization applications. On top of the Uintah architecture, we have designed a set of components and supporting libraries that are targeted toward the solution of PDEs on massively parallel architectures (hundreds to thousands of processors).

4.1 Overview

The UCF employs a non-traditional approach to achieving parallelism. Instead of explicit MPI calls placed throughout the program, applications are cast in terms of a *taskgraph*, a construct that describes the data dependencies between various steps in the problem.

The UCF exposes flexibility in dynamic application structure by adopting an execution model based on software-based “macro” dataflow. Computations are expressed as directed acyclic graphs 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 AMR grid. Tasks form a UCF data structure called the *taskgraph*, which represents imminent computation. Associated with each task is a C++ method, which is used to perform the actual computation. UCF data structures are compatible with Fortran arrays, so that the application writer can use Fortran subroutines to provide numeric kernels on each patch.

Each execution of a taskgraph integrates a single timestep, or a single non-linear iteration, or some other coarse algorithm step. Tasks “communicate” with each other through an entity called the *DataWarehouse*. The DataWarehouse is accessed through a simple name-based dictionary mechanism, and provides each task with the illusion that all memory is global. If the tasks correctly describe their data dependencies, then the data stored in the DataWarehouse will match the data (variable and region of space) needed by the task. In other words, the DataWarehouse is an abstraction of a global single-assignment memory, with automatic data lifetime management and storage reclamation. Values stored in the DataWarehouse are typically array-structured.

Communication is scheduled by a local scheduling algorithm that approximates the true globally optimal communication schedule. Because of the flexibility of single-assignment semantics, the UCF is free to execute tasks close to data or move data to minimize future communication.

The UCF storage abstraction is sufficiently high-level that it can be efficiently mapped onto both message-passing and shared-memory communication mechanisms. Threads sharing a memory can access their input data directly; single-assignment dataflow semantics eliminate the need for any locking of values. Threads running in disjoint address spaces communicate by message-passing protocol, and the UCF is free to optimize such communication by message aggregation. Tasks need not be aware of the transports used to deliver their inputs and thus UCF has complete flexibility in control and data placement to optimize communication both between address spaces or within a single shared-memory SMP node. Latency in requesting data from the DataWarehouse is not an issue; the correct data is deposited into the DataWarehouse before the task is executed.

Consider the taskgraph in Figure 4.1. Ovals represent tasks, each of which are a simple array program and easily treated by traditional compiler array optimizations. Edges represent named values stored by UCF. Solid edges have values defined at each material point (Particle Data) and dashed edges have values defined at each grid vertex (Grid Data). Variables denoted with a prime (') have

Fig. 2. An Example UCF Taskgraph

been updated during the time step. The figure shows a portion of the actual Uintah Material Point Method (MPM) [11–13] taskgraph concerned with advancing Newtonian material point motion on a single patch for a single timestep.

The idea of the dataflow graph as an organizing structure for execution is well known. The SMARTS [14] dataflow engine that underlies the POOMA [15] toolkit shares goals and philosophy with the UCF. SISAL compilers [16] 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. The SISAL compiler was of course developed to support the SISAL language. UCF is implemented to support a presentation catering to C++ and Fortran based mixed particle/grid algorithms on a structured adaptive mesh. The primary algorithms of importance to C-SAFE are the Material Point Method (MPM), and Eulerian CFD algorithms.

4.2 Taskgraph Advantages and Disadvantages

This dataflow-based representation of parallel computation fits very well with the SAMR grid, and with the nature of the computations that C-SAFE is performing. In particular, we decided to use this approach in order to accommodate a number of important needs.

First, the taskgraph helps accommodate flexible multi-physics integration needs. In particular, integration of a particle-based solid mechanics algorithm

(MPM) with a state-of-the-art CFD code was a primary C-SAFE goal. However, scientists still wanted to execute the CFD algorithm by itself, or the MPM algorithm by itself or even with a different CFD algorithm. The taskgraph facilitates this integration by allowing each application component (MPM and CFD in this example) to describe their tasks independently. The scheduler connects these tasks where data is exchanged between the different algorithm phases. In this fashion, a fine-grained interleaving of these different algorithms is obtained.

Second, the taskgraph can accommodate a wide range of unforeseen workloads. In C-SAFE simulations, load imbalances arise from a variety of situations: with particle-based methods particles may exist only in small portions of the domain; or ODE-based reaction solvers may be more costly in some regions of space than in others. Using the taskgraph, the UCF can map patches to other processors to minimize overall load imbalance. Communication is performed automatically, and the system has the information necessary to predict whether data motion is likely to pay off. These features would be more difficult to implement if each scientist were burdened with these complexities when writing simulation components.

Third, the taskgraph helps manage the complexity of a mixed threads/MPI programming model. Many modern supercomputing architectures employ a number of SMP nodes connected together by a fast interconnect. These architectures are often programmed with a flat MPI model, ignoring the fact that 2-128 of the nodes actually share a single memory. Using the UCF, tasks can be mapped to threads in order to achieve multi-threaded execution within a node. The semantics of the DataWarehouse enable true sharing of the data, eliminating explicit communication and data redundancy between neighboring SMP processors. Once again, this would not be possible if simulation scientists were burdened with these additional complexities.

Fourth, the taskgraph can accommodate a mix of static and dynamic load balancing mechanisms. In particular, we are developing a mechanism that uses fine-grained dynamic load balance within an SMP node, and a relatively static coarse-grained mechanism between address spaces.

Fifth, the taskgraph facilitates development of simulation *components* that allow pieces of the simulation to evolve independently. In many respects, this is the most important advantage for a large interdisciplinary project such as C-SAFE. Since C-SAFE is a research project, we need to accommodate the fact that most of the simulation components are still under development. The component-based architecture allows pieces of the system to be implemented in a basic form at first, and then to evolve as the technologies mature. Most importantly, the UCF allows the aspects of parallelism (schedulers, load-balancers, parallel I/O, and so forth) to evolve independently of the simulation components. This allows the computer science effort to focus on these problems without waiting for the completion of the scientific applications or vice-versa.

However, in addition to these advantages there are some disadvantages to the approach that we chose. First, creating an optimal schedule for the taskgraph is known to be an NP-hard problem. However, we have found that using

Fig. 3. UCF Simulation Components

some simple heuristics, and exploiting the regularity in the problem, that we can obtain respectable performance using straightforward scalable algorithms. Further refinements in these algorithms will receive more attention in the near future. Second, creation of the schedule can be costly. We take advantage of the fact that the schedule does not need to be recomputed for each execution. It does need to be recomputed if the algorithm changes, or if the grid changes. In addition, the schedule may need to be recomputed periodically to maintain an optimal load-balance. Third, the taskgraph requires a mental shift for parallel application programmers. Nevertheless, we found that the programmers were able to easily take a description of their algorithm and cast that into a set of tasks. The application was then able to run in parallel, even on hundreds of processors. For our applications, this benefit outweighed the cost of casting the algorithms in the dataflow execution model.

For C-SAFE, the advantages far outweighed the disadvantages. We note that for a typical purely structured-grid computation, the taskgraph may be overkill. For a purely unstructured-grid computation, the granularity would be too small and data dependencies would be more complex. The SAMR grids employed by C-SAFE seem to have just the right granularity for this approach to be successful.

4.3 Components Involved

Figure 4.3 shows the main components involved in a typical C-SAFE simulation. The SimulationController is the component in charge of the simulation. It will manage restart files if necessary, and control the integration through time. First, it reads the specification of the problem from an XML input file. After setting up the initial grid, it passes the description to the simulation component. The

Fig. 4. A simple computational domain and a 4-patch decomposition of that domain

simulation component can be a number of different things, including one of two different CFD algorithms, the MPM algorithm, or a coupled MPM-CFD algorithm. The simulation components define a set of tasks to the scheduler. In addition, the DataArchiver component describes a set of output tasks to the scheduler. These tasks will save a specified set of variables to disk. Once all tasks are known to the scheduler, the load-balancer component uses the configuration of the machine (including processor counts, communication topologies, etc.) to assign tasks to processing resources. The scheduler uses MPI to communicate the data to the right processor at the right time and then executes callbacks into the simulation or DataArchiver components to perform the actual work. This process continues until the taskgraph is fully executed. The the execution process is repeated to integrate further timesteps.

Each of these components run concurrently on each processor. The components communicate with their counterparts on other processors using MPI. However, the scheduler is typically the only component that needs to communicate with other processors.

4.4 Definitions

Consider Figure 4.4. We define several terms which we use in discussing Structured AMR grids:

- **Patch:** A contiguous rectangular region of index space and a corresponding region of simulated physical space. The domain on the right of Figure 4.4 is the same as the domain on the left, except that it has been decomposed into two patches.
- **Cell:** A single coordinate in the integer index space, also corresponding to the smallest unit in simulated physical space. A variable centered at the cells in the simulation would have a value corresponding to each of the **X**'s in Figure 4.4.
- **Node:** An entity at the corners of each of the cells. A variable centered at the nodes in the simulation would have a value corresponding to each of the **O**'s in Figure 4.4.
- **Face:** The faces join two cells. The UCF represents values on X, Y, and Z faces separately.
- **Ghost cell:** Cells (or nodes) that are associated with a neighboring patch, but are copied locally to fulfill data dependencies from outside of the patch.

4.5 Variable Types

UCF simulations are performed using a strict “owner computes” strategy. This means that each topological entity (a node, cell or face) belongs to exactly one patch. There are several variable types that represent data associated with these entities. An **NCVariable** (Node-centered variable) contains a single value at each **Node** in the domain. Similarly, **CCVariables** contain values for each cell, and **XFCVariables**, **YFCVariables** and **ZFCVariables** are Face-centered values for the faces corresponding to the X, Y and Z axes. Each of these variables types are C++ template classes, therefore a node/cell/face-centered value can be any arbitrary type. Typically values are a double-precision number representing pressure, temperature, or some other scalar throughout the field, but values may also be a more complex entity, such as a vector representing velocity or a tensor representing a stress.

In addition to the topological based variables described above, there is one additional variable type: **ParticleVariable**. This variable contains values associated with each particle in the domain. A special particle variable contains the position of the particle. Other particle variables are defined by the simulations, and in the case of the MPM algorithm include quantities like temperature, acceleration, stress and so forth. For the purposes of the discussions below, particles can be considered a fancy type of cell-centered variable, since each particle is associated with a single cell. It is important however, to point out that explicit lists of particles within a cell are not maintained. We have found it more efficient to determine particle/cell associations as they are needed instead of paying the high cost of maintaining lists of particles for each cell.

4.6 X around Y

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 within a patch. This is termed “nodes around cells” in UCF terminology. As shown in Figure 4.6, a layer of *ghost nodes* would be copied from neighboring patches to the top and right edges of the lower-left patch. In a four-processor simulation this copy would involve MPI messages from each of the other three processors. It is important to note the asymmetry in this process; data is often not required from all 26 (in 3D) neighbors to satisfy a computation. Symmetry comes when a subsequent step uses “cells around nodes” to satisfy another data dependency.

In this fashion, each task in the algorithm specifies a set of data requirements. Similarly, each task specifies a set of data which it will compute, but in this case no ghost cells are necessary (or allowed). These “computes & requires” lists for each task are collected to create the full taskgraph.

A task could specify that it requires data from the entire computational domain. However, for typical scalable algorithms, the tasks ask for only one (or possibly two) layers of data outside of the patch.

Fig. 5. Communication of ghost nodes in a simple 4-patch domain

4.7 Compilation

Using the data dependencies described above, the UCF scheduler compiles a coarse-representation of the taskgraph. This compilation process proceeds simultaneously on each processor without communication between processors. This representation contains all of the data dependency information, but contains only a single vertex (graph bubble) for each task. The full taskgraph will contain a vertex for each patch/task combination. However, this full representation is not fully instantiated on each processor.

Initially, a topological sort is performed on the coarse taskgraph. The scheduler then creates a *detailed taskgraph* for the subregion of the total graph which covers the neighborhood of a particular processor. It will instantiate graph vertices for the task/patch combinations that are assigned to the processor, and will instantiate graph edges which connect to or from those vertices. Finally, it will create vertices that connect to those edges. In this fashion, the detailed taskgraph contains the vertices for tasks owned by this processor and for those tasks on other processors with which it will communicate. The detailed taskgraph uses a very compact representation since the number of detailed tasks can be significant. However, there are $O(1)$ detailed tasks on each processor for scalable simulations.

After instantiating the detailed tasks, the UCF scheduler performs a set of analysis functions on the resulting taskgraph. It ensures that the application programmers have used every variable that it computes, and that they do not expect variables that are never produced. It also ensures that the types of variables match between the computes and requires, including the type of the underlying templated data. Finally, the scheduler analyzes the lifetimes of each variable used throughout the execution of the taskgraph. Variables that hold intermediate quantities are scheduled for deletion when no more tasks require them.

4.8 I/O and Checkpointing

Data output is scheduled using the taskgraph just like any other computation. Constraints specified with the task allow the load balancing components to di-

rect those tasks (and the associated data) to the processors where data I/O should 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. The UCF 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 in the data warehouse at the end of the timestep. Data lifetime analysis ensures that only the data required by subsequent iterations will be saved. If the simulation components have been correctly written to store all of their data in the data warehouse, restart is a trivial process. During restart, the components process the XML specification of the problem that was saved with the datasets, and then the UCF creates input tasks that load the data warehouse from the checkpoint files. If necessary, data redistribution is performed automatically during the first execution of the taskgraph. In a similar fashion, changing the number of processors is possible. The current implementation does not redistribute data among the patches when the number of processors are changed. Patch redistribution is a useful component even beyond changing the processor count, and will be implemented in the future.

4.9 Legacy MPI Compatibility

In order to accommodate software packages that were not written using the UCF 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, UCF applications can use available MPI libraries, such as PETSc [17] and hypre [18].

4.10 Execution

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 processes is more complex. In an MPI-only implementation, there are a number of ways to utilize MPI functionality to overlap communication and I/O. We describe one way that is currently implemented in the UCF.

We process each detailed 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.

The mixed MPI/thread execution is somewhat different. Initially, non-blocking `MPI_Irecv`s are posted for *all* of the tasks assigned to the processor. Then each thread will concurrently call `MPI_Waitsome` and will block for internal data dependencies (i.e. from other tasks) until the data dependencies for any task are complete. That task is executed and data that it produces is sent out. The thread then goes back and tries to complete a next task. This implements a completely asynchronous scheduling algorithm. Preliminary results for this scheduler indicate that a performance improvement of approximately 2X is obtainable. However, thread-safety issues in vendor MPI implementations have slowed this effort.

It can be seen that dramatically different communication styles can be employed by simply changing out the scheduler component. The application components are completely insulated from these variations. This is a very important aspect that allows the Computer Science team members to focus on the best way to utilize the communication software and hardware on the machine without requiring sweeping changes in the application. Each scheduler implementation consists of less than 1000 lines of code, so it is relatively easy to write one that will take advantage of the properties of the communication hardware available on a machine. Often, the only difficult part is getting the correct information from the vendor in order to determine the best strategy for communicating data.

4.11 Load Balancing

The Load Balancer component is responsible for assigning each detailed task to one processor. To date, we have implemented only simple static load-balancing mechanisms. However, the UCF was designed to allow very sophisticated load-balance analysis algorithms. In particular, a cost-model associated with each task will allow an optimization process to determine the optimal assignment of tasks to processing resources. Cost models associated with the communication architecture of the underlying machine are also available. One interesting aspect of the load-balance problem is that integrated performance analysis in the UCF will allow the cost-models to be corrected at run-time to provide the most accurate cost information possible to the optimization process.

The mixed thread/MPI scheduler described above implements a dynamic load-balancing mechanism (i.e. a work queue) within an SMP node, and uses a static load-balancing mechanism between nodes. We feel that this is a powerful combination that we will pursue further.

Careful readers will pick up on the fact that the creation of detailed tasks require knowledge of processor assignment. However, sophisticated load-balance components may require this detailed information before they can optimize the task/processor assignments. We use a two-phase approach where tasks are assigned arbitrarily, then an optimization is performed and the final assignments are made to the tasks. Subsequent load-balance iterations use the previous approximation as a starting point for the optimization process.

Fig. 6. Parallel Performance of a typical C-SAFE problem. This is a 16 million particle Material Point Method (MPM) computation, running on up to 2000 processors of the Los Alamos National Laboratory Nirvana machine. The straight line represents ideal performance, while the boxes show data points of performance actually obtained. There is a slight slowdown as the computation spans more than one 128 processor Origin 2000, but continues on linearly to the maximum number of processors on the machine.

5 Applications and Results

The system described here has been used to implement a variety of simulations. Two different CFD algorithms, one MPM (solid mechanics) algorithm, and two different coupled CFD-MPM algorithms have been implemented to date. Each of these simulations run quite well in parallel. Figure 5 shows the scalability of the MPM application up to 2000 processors of the ASCI Nirvana machine at Los Alamos National Laboratory. Nirvana is a cluster of 16 SGI Origin 2000 ccNUMA machines, each consisting of 128 MIPS R10K processors running at 250 Mhz. The machines are connected with a 6.4 Gigabit GSN (Gigabyte System Network). It should be noted that the problem shown is relatively small, with timesteps completing in much less than one second for the large processor configurations. One of the CFD components (Arches) has similar scaling properties to MPM; the others we have not had the opportunity to push to large numbers of processors. Figure 7 shows results from 1000 processor simulations using the Arches CFD component and the MPM component.

6 Conclusions and Future Work

The past few years of this project have focused on developing a flexible framework in which to accomplish scalable parallel simulations. Now that we have the basic

Fig. 7. Visualization of two different simulations from C-SAFE. On the left is a simulation of a heptane fire. On the right is a simulation of stress propagation through a block of granular material. Each of these simulations were performed using the Uintah Computational Framework and were executed on 1000 processors.

infrastructure in place, there are a number of research projects worthy of pursuit, including: development of feedback-based load balancing components that utilize task performance information collected at runtime to optimize resource allocations; further development and tuning of the mixed thread/MPI scheduler; investigation of and development of optimization algorithms that operate on an incomplete representation of the global graph; and further development of the UCF's AMR capabilities.

We have presented a powerful way of building complex PDE simulation software. We have demonstrated that the UCF system can obtain good performance on up to 2000 processors. Using the UCF, C-SAFE scientists have been able to simulate complex physical phenomena that was not achievable using smaller-scale versions of the code.

We have described only a small fraction of a very complex system. Other interesting aspects, such as the template implementations of the data warehouse, the augmented run-time typing system, and other features will be described in more detail in the future.

7 Acknowledgments

This work was supported by the DOE ASCI ASAP Program. James Bigler, J. Davison de St. Germain, and Wayne Witzel are very valuable members of the C-SAFE PSE team that developed this software. C-SAFE visualization images were provided by Kurt Zimmerman and Wing Yee. Datasets were created by Scott Bardenhagen, Jim Guilkey, Seshadri Kumar, Rajesh Rawat, and John Schmidt. The DOE ASCI ASAP program also provided computing time for the simulations shown.

References

1. Center for the Simulation of Accidental Fires and Explosions - Annual Report, Year 2. <http://www.csafe.utah.edu/documents>.
2. T. C. Henderson, P. A. McMurtry, P. J. Smith, G. A. Voth, C. A. Wight, and D. W. Pershing. Simulating accidental fires and explosions. *Comp. Sci. Eng.*, 2:64–76, 1994.
3. Common Component Architecture Forum. <http://z.ca.sandia.gov/ccs-forum>.
4. R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, 1999.
5. S.G. Parker, D.M. Beazley, and C.R. Johnson. Computational steering software systems and strategies. *IEEE Computational Science and Engineering*, 4(4):50–59, 1997.
6. Steven Gregory Parker. *The SCIRun Problem Solving Environment and Computational Steering Software System*. PhD thesis, University of Utah, 1999.
7. S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.
8. J. Davison de St. Germain, John McCorquodale, Steven G. Parker, and Christopher R. Johnson. Uintah: A Massively Parallel Problem Solving Environment. In *Proceedings of the Ninth IEEE International Symposium on High Performance and Distributed Computing*, August 2000.
9. J. McCorquodale, S. Parker, J. Davison, and C. Johnson. The utah parallelism infrastructure: A performance evaluation. In *2001 High Performance Computing Symposium (HPC'01), Advanced Simulation Technologies Conference*, 2001.
10. A. Cleary, S. Kohn, S.G. Smith, and B. Smolinski. Language Interoperability Mechanisms for High-Performance Scientific Applications. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Interoperable Scientific and Engineering Computing*, October 1998.
11. D. Sulsky, Z. Chen, and H. L. Schreyer. A Particle Method for History Dependent Materials. *Comp. Methods Appl. Mech. Engrg*, 118, 1994.
12. S.G. Bardenhagen, J.E. Guilkey, K.M. Roessig, J.U. Brackbill, W.M. Witzel, and J.C. Foster. An improved contact algorithm for the material point method and application to stress propagation in granular material. *Computer Modeling in Engineering and Sciences*, 2(4):509–522, 2001.
13. J.E. Guilkey and J.A. Weiss. Implicit time integration with the Material Point Method (submitted). *International Journal for Numerical Methods in Engineering*, 2002.
14. 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.
15. S. Atlas, S. Banerjee, J.C. Cummings, P.J. Hinker, M. Srikant, J.V.W. Reynders, and M. Tholburn. POOMA: A high-performance distributed simulation environment for scientific applications. In *Supercomputing '95 Proceedings*, December 1995.
16. J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.
17. S. Balay, W. Gropp, L. McInnes, and B. Smith. Petsc home page, 1999.
18. hypre: high performance preconditioners. <http://www.llnl.gov/casc/hypre/>.