Uintah: A Massively Parallel Problem Solving Environment

J. Davison de St. Germain, John McCorquodale, Steven G. Parker, Christopher R. Johnson
Scientific Computing and Imaging Institute
University of Utah, Salt Lake City, UT 84112
{dav,mcq,sparker,crj}@cs.utah.edu
http://www.cs.utah.edu/sci and http://www.csafe.utah.edu

Abstract

This paper describes Uintah, a component-based visual problem solving environment (PSE) that is designed to specifically address the unique problems of massively parallel computation on terascale computing platforms. Uintah supports the entire life cycle of scientific applications by allowing scientific programmers to quickly and easily develop new techniques, debug new implementations, and apply known algorithms to solve novel problems. Uintah is built on three principles: 1) As much as possible, the complexities of parallel execution should be handled for the scientist, 2) software should be reusable at the component level, and 3) scientists should be able to dynamically steer and visualize their simulation results as the simulation executes. To provide this functionality, Uintah builds upon the best features of the SCIRun PSE and the DoE Common Component Architecture (CCA).

Introduction

Due to concerns about safety and environmental impact, as well as the difficulty inherent in measuring certain real world physical phenomena, many scientists have turned to computer simulations to model the real world. As these simulations become larger and more complex, and as more accurate results in shorter amounts of time are required, scientists continue to require more powerful computers. However, harnessing the power of today's latest supercomputers is a non-trivial task and depends largely on the computational software system in which the physical processes are modeled and simulated.

In designing the Uintah software system, we focused on three guiding properties. First, the complexities of code creation for parallel machines should (as much as possible) be hidden from the scientist. Second, complex simulation components developed by third parties should be available tools scientist can choose to employ. And third, the scientist should be able to visually monitor and steer her simulation while it is running. A software environment that efficiently integrates these properties into a usable system will allow scientists to effectively create and use complex simulations in an interactive, exploratory way. The Uintah Problem Solving Environment (PSE) is such a system. It allows scientists and engineers to focus on algorithm development and data analysis rather than details of the underlying software architecture without sacrificing the ability to effectively realize the full potential of large parallel computers.

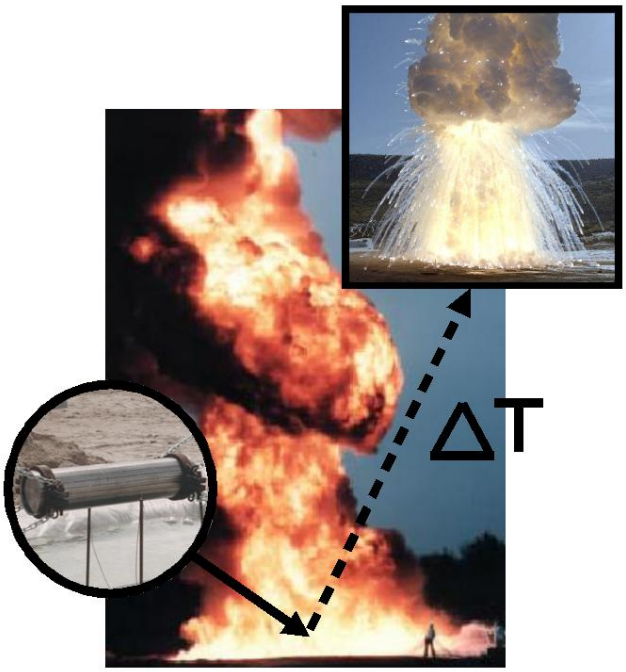

Figure 1: A Typical C-SAFE Problem

While Uintah is designed to provide a general framework in which a wide variety of large scale, massively parallel simulations can be conducted, the specific problem that has driven its creation is the modeling of the interactions between hydrocarbon fires, structures and high-energy materials (explosives and propellants), as shown in Figure 1. Exploring this problem is the mission of the Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [1], located on the campus of the University of Utah. C-SAFE was created by the

Department of Energy's Accelerated Strategic Computing Initiative's (ASCI) Academic Strategic Alliance Program (ASAP) [5].

The problems studied within C-SAFE require the ability to efficiently run physically coupled computations on over a billion particles contained in a grid of over a billion cells, with time and length scales spanning over ten orders of magnitude, decomposed into over a 100,000 computational/spatial regions running on an 8000 processor distributed-memory supercomputer. This paper describes the architecture currently under development in the continuing pursuit of this goal.

Before discussing the Uintah PSE in more detail, we provide some background on two important building blocks upon which Uintah is based: SCIRun[1] and the DOE Common Component Architecture (CCA).

The SCIRun Problem Solving Environment

Problem-solving in scientific computing typically involves symbolic computation, numeric computation and visualization of data. Historically, these tasks have been carried out by separate tools which share common data file formats. In 1987, the Visualization in Scientific Computing (ViSC) workshop made some forward-looking observations [7]: "Scientists ... want to drive the scientific discovery process; they want to *interact* with their data. *Interactive visual computing* is a process whereby scientists communicate with data by manipulating its visual representation during processing. The more sophisticated process of *navigation* allows scientists to *steer*, or dynamically modify computations while they are occurring. These processes are invaluable tools for scientific discovery."

An interactive scientific Problem Solving Environment (PSE) [15] provides a complete set of tools for a scientist to solve a class of problems. In our opinion, a PSE integrates a domain-specific library with a high-level visual user interface via a common software infrastructure supporting dynamic data and program modification. As an application runs in a PSE, a scientist can dynamically visualize the data to assist in the debugging process, as well as modify input conditions, algorithms or other parameters of the simulation's running state.

We believe these abilities contribute to a rich and fundamentally superior environment for all phases of computational science, from initial algorithm development to performance tuning and debugging to application steering. Tight integration of visualization with steerable computation [13, 14] allows the cause-effect relationships within a problem domain to become more evident, allowing a scientist to develop more intuition about not only the effects of problem parameters but also about fundamental algorithmic approach.

The SCIRun scientific problem solving environment has evolved to address this need for interactive, visual computational steering in problems of bioelectric field modeling [16] and computational medicine [10] (among other applications). Its primary goal is to provide the scientist with a comprehensive environment with interfaces to control and interact with a simulation at both application and system levels, and to use scientific visualization in all aspects of the computational endeavor.

SCIRun makes use of a programming model based on *functional dataflow*. Atoms of computation in SCIRun are called *modules* and are assembled in a visual programming environment into *dataflow networks* by connecting together modules' inputs and outputs, as shown in figure 2.

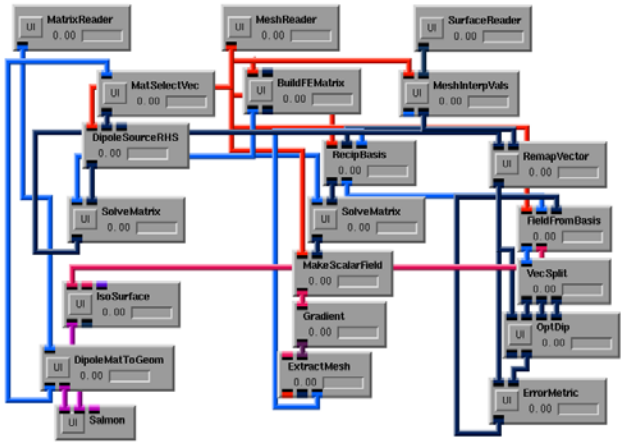

Figure 2: A SCIRun application represented visually as a dataflow network

Edges in this network represent streams of typed objects flowing asynchronously among modules. SCIRun makes heavy use of threads within a single address space to both support this asynchrony and provide substantial parallel speedups on a shared-memory multiprocessor computer.

The interactive visual programming mechanisms developed for SCIRun have proven to be a convenient and natural approach to both application construction and runtime steering. However, the pure thread-based dataflow module composition mechanism of SCIRun lacks the ability to operate in a distributed-memory

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

environment. Furthermore, many scientific algorithms are difficult to cast into a pure dataflow programming model.

The Common Component Architecture

The Common Component Architecture Forum [2, 4] is working to define a language-independent software interoperability standard targeted specifically to the needs of massively parallel scientific computing. Driving forces in its definition are the need for fast connections among components that perform numerically intensive work and for parallel collective interactions among components that use multiple processes or threads. A compliant implementation enables the experimental combination of components from disparate sources into large simulation codes without sacrificing the performance advantages of parallel communication and synchronization patterns across component boundaries.

Central to this achievement is the CCA *port model*. A CCA *port* is a communication abstraction carefully designed to simultaneously capture distributed-memory parallel communication patterns (the *collective port*) while allowing implementations to optimize shared address-space inter-component communications as a single indirect function call (the *directly-connected port*). A simple example of a CCA application is shown in Figure 3.

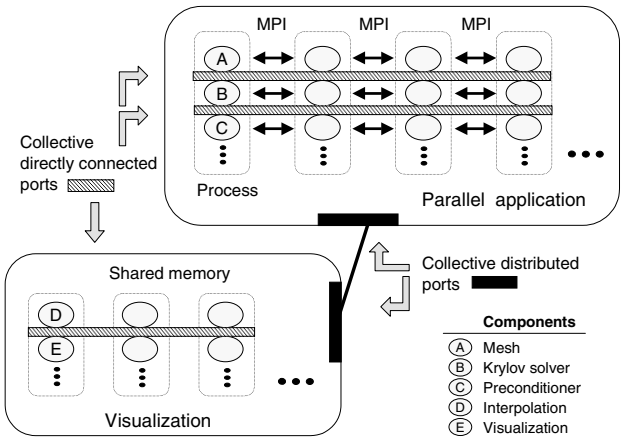

Figure 3: Parallel CCA Component Interactions

The CCA is an ongoing effort to define standard parallel data transport mechanisms and canonical wire formats thus enabling rapid prototyping by pure component composition without hand-written middleware data translation code. The *scientific interface definition language* (SIDL) [6] and its prototype implementation automate the generation of data transport wrappers in a language-neutral way, thus insuring components written in many languages will seamlessly interoperate. Unlike other interface definition languages such as the CORBA IDL [3], SIDL is sufficiently expressive to effeciently represent the abstractions and data types common to scientific computing such as dynamically sized multidimensional arrays and complex numbers.

Further, the CCA will ultimately specify a *component repository* that collects and manages available components. Component interactions with the repository, and tasks of component cataloging and interface registration, are automated by the SIDL tools. The component repository is a mechanism to accumulate components from disparate sources by a common cataloging and distribution mechanism, to support component proliferation and reuse.

The CCA is an enabling technology for computational steering in that it has been specifically designed to allow the dynamic restructuring of massively parallel applications. This enables researchers to introduce new components or change component compositions during the course of an ongoing simulation. However, the presentation of a steering-capable environment to the user is outside its primary domain of concern. Uintah directly addresses the need for visual steering of CCA applications.

The Uintah Architecture

Uintah is a new software system for computational science that combines the proven exploratory visual computing and computational steering capabilities of the SCIRun PSE with the high-performance and naturally parallel component composition mechanisms of the CCA design to allow a visual, steerable problem solving environment for tera-scale scientific computing. While Uintah focuses on coupled chemistry and physics simulations, it is capable of serving as a base framework for a large number of scientific applications.

Component Model

Uintah generalizes component programming by allowing different *kinds* of components to be connected through different *kinds* of ports. A Uintah application can be implemented by composing existing visualization components which use the SCIRun dataflow communication model with computational components which adhere to the CCA. Component connections are presented to the user uniformly by the Uintah PSE interface. Data translations and control emulations are performed through specific adapter components when two or more components of different types are composed. Uintah can be thought of as implementing a

component integration framework whose component architecture is an extensible superset of both the Common Component Architecture and the SCIRun module architecture.

The Uintah approach to software components addresses several key challenges of large-scale scientific computing. The mechanism can provide steerability to component families or entire component architectures which could not previously attain this degree of interactivity. This ability is, of course, predicated upon technical details of each component architecture related to such problems as dynamic recomposition. At the very least, Uintah provides a uniform application building and data interoperability environment for multiple component architectures. We hope, in the future, to extend Uintah to support other component models, such as CORBA and COM, among others, as needs arise.

This ability also encourages component reuse, since programmers are not forced to rewrite large libraries written in the "wrong" component architecture. Further, CCA offers the promise of large bodies of existing code being "wrapped" as CCA components and thus able to be composed within the Uintah environment in an efficient and parallel way. It is hoped that the future availability of Uintah will provide a tangible incentive to others to evolve potentially useful codes into CCA components.

Resource Mapping and Simulation Startup

The current trend in supercomputing is to use a large number of full function processors, grouped into shared memory nodes of 2 to 128 processors. These nodes communicate with each other using high speed data interconnects such as HiPPI. To take full advantage of this type of architecture, processes running on a single node take advantage of hardware-assisted implicit communication via shared memory to reduce communication overhead and implementation complexity, while also utilizing distributed-memory-style communications across the nodes of the largest parallel machines.

A Uintah application runs under the management of a distributed steering-aware runtime environment. For a complex application, this runtime environment might ultimately span thousands of processes and thousands of processors located in a large collection of compute nodes connected by networks of widely varying capabilities. Managing such a computation, especially when a scientist's steering decisions might cause large perturbations to the topology of the runtime environment during computation, is a difficult challenge.

To address this challenge, Uintah begins by defining a *master control process* (MCP) which is the conceptual "handle" on the computation as a whole. When the MCP exits, the computation shuts down and all computational resources are surrendered. Component compositions and steering decisions are carried out under the control of the MCP. The first step in computing with Uintah is to start an MCP for the computation.

As the scientist builds her application and specifies computing resources to use, the MCP will cause *slave controller processes* (SCPs) to be started on particular compute nodes. We are currently investigating the Globus [8] Toolkit as a mechanism to help deal with the complexity of remote startup at remote sites in a consistent and general way. The current Uintah prototype uses ad hoc startup mechanisms tailored to the idiosyncrasies of well-known computing resources.

Staging and Component Composition

Once SCPs are started and have a reliable communication channel with the MCP, component binaries are sent over this channel and dynamically linked into the SCP. Thread-parallel components may result in the SCP starting groups of *worker threads* (WTs) which share an address space and fast communication with each other, with the SCP and possibly with other components running in the same address space.

Additionally, a single component may be logically broken up into multiple address spaces running in multiple processes. This is the case for MPI applications which have been wrapped as CCA components, and for components that desire to use a mixed threads/MPI programming model.

Finally, the SCP is responsible for all staging necessary for component operation and may copy files, set up environments for other processes, and so forth. An example of this process structure is given in figure 4.

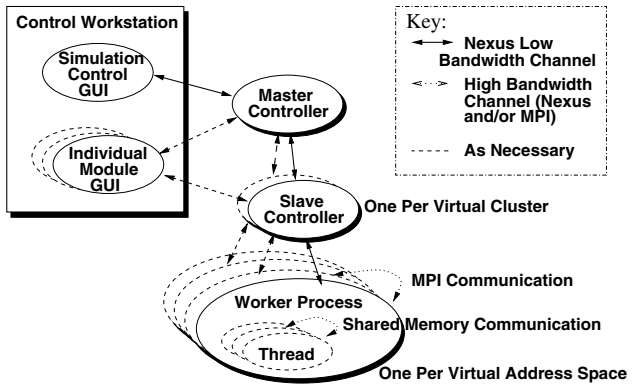

Figure 4: Uintah Parallelization Strategy

Component Communication

As the final stage of the component composition process, the SCPs and the MCP cooperate to establish the potentially quite complex communication pathways necessary for the threads of each component to talk both to each other, and to the threads of other potentially multithreaded components. This task is made more difficult by the presence of many kinds of components (SCIRun, CCA, etc.) which do not necessarily share common expectations about communication.

Intercomponent communication: In a Uintah simulation, threads belonging to two separate components running in separate address spaces communicate transparently through SIDL-based method calls. These calls are transformed into messages layered on the Nexus [9] parallel communication system. Nexus provides an efficient, reliable asynchronous messaging service that is well-suited to parallel component communication and transparently handles the details of primitive data translation in heterogeneous computing environments. As a vital optimization, if two components' threads are located in the same address space, Uintah implements communication between them at the cost of a single (indirect) function call. Communication between two parallel components is still an open research topic, but Uintah provides a flexible environment for prototyping these communication mechanisms.

Intracomponent communication: Components themselves may be internally parallel. If this parallelism is implemented within a single address space, the components' *slices* may communicate using shared memory and indirect function calls as in the case of intercomponent communication. Alternatively, components may be implemented as collections of MPI processes, in which case the slices of the component might communicate with each other using MPI, provided a thread-safe MPI implementation is available. Uintah makes no restrictions on intracomponent communication, and the component programmer is free to optimize to suit her particular needs.

An example illustrating some of this complexity is found in figure 5. Here, a group of components (in this case SCIRun modules) implementing a fire simulation are running as two groups of 128 MPI processes on two nodes of a large machine at Los Alamos National Lab. Because these MPI processes are stand-alone entities, no SCP is shown on the second node, but in practice there may be one present to support other components on that node. These processes communicate with each other using MPI, and with the 32 processes of a parallel visualization component using Nexus [9]. The results of the visualization are delivered to the scientist's screen

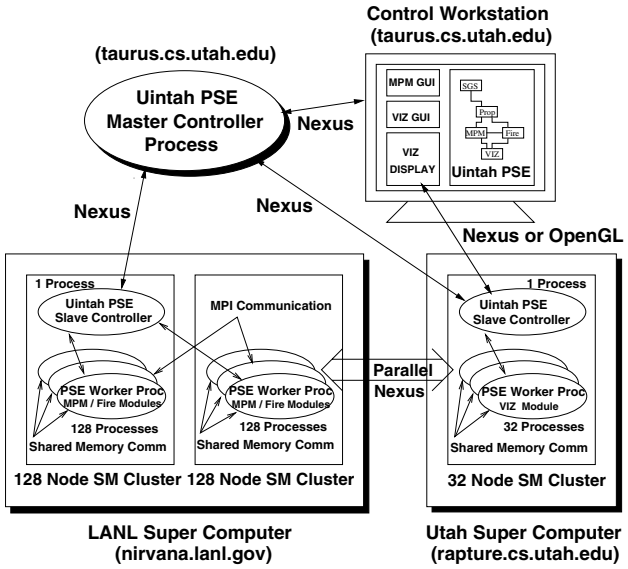

Figure 5: Uintah Architecture Example

with the help of the visualization node's SCP.

User Interaction

Uintah detaches the visual application steering interface from the simulation itself (which exists as the MCP), allowing steering infrastructure to be easily written in languages appropriate to particular tasks, and potentially allowing limited application steering via a web browser.

Steering interfaces can be connected to and disconnected from a running simulation at will, and multiple scientists can be simultaneously steering disjoint aspects of the simulation (hopefully in a non-competitive way).

An Example Uintah Application

Because Uintah supports multiple types of components and allows numerous communication mechanisms and internal component behaviors, it is not possible to describe a single, canonical Uintah simulation. Rather, we describe here one of many possible Uintah component implementation styles and a simulation session from initial specification, startup and execution through eventual visualization and steering. This example is chosen in order to more concretely illustrate the operation of the Uintah component system.

Problem Specification

Consider a typical C-SAFE problem: a capped metal cylinder filled with high-energy material is suspended above a pool of hydrocarbon fuel burning with an open flame, as shown in Figure 1. Energy from the flame

is transported though the metal cylinder to the high-energy material, causing it to undergo complex chemical changes. Solid deformations, deterioration and cracking occur in both the cylinder and the high-energy material as pressure within the cylinder builds, eventually leading to rupture and detonation.

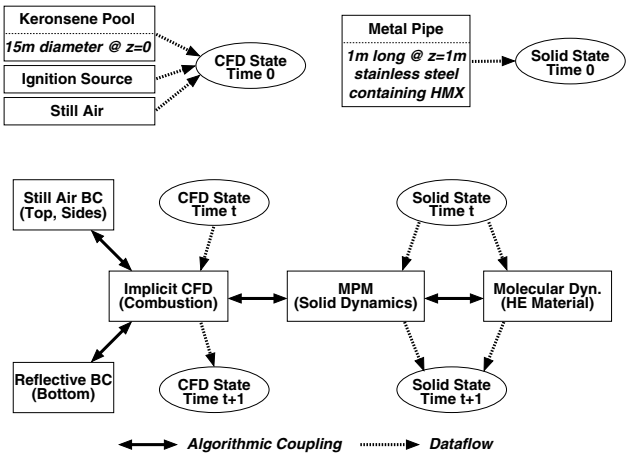

Figure 6: An Example Uintah Computation

As shown in Figure 6, we implement this simulation by combining a collection of existing components and providing some initial data on which they compute. In this case, we combine a computational fluid dynamics (CFD) [11, 12] component that simulates hydrocarbon combustion and reactant transport with a component that uses the material point method (MPM) [18] to simulate the mechanics of solid deformation and energy transport within the cylinder. The MPM component uses constitutive micromodels for the high-energy material parameterized on its temperature- and pressure-dependent thermophysical and viscoelastic properties. These parameters are computed from quantum molecular dynamics simulations [17] performed under the conditions within the cylinder. We represent the CFD boundary conditions as separate components to facilitate a discussion of algorithmic steering below.

The CFD and MPM components' visualization and control GUIs are used to display, specify and steer these components' internal data. These GUIs are used to specify the initial conditions for the simulation, including an initial distribution of combustible material: in this case a pool of kerosene on the ground, an ignition source, and initially still air for oxidation. Also, an initial distribution of material points and constitutive models is constructed to represent a stainless steel pipe 1 meter long suspended above the ground containing the explosive HMX in a polymer binder.

The physical and chemical models implemented by the components in this example are themselves the subjects of novel research at C-SAFE and as such are undergoing continuing development. Their exact functionality is not relevant to the following discussion of the Uintah component system.

The actual component structure of a complete Uintah simulation will be much more complex than that indicated in Figure 6. This figure is contrived so that we can discuss in detail the interesting facets of the Uintah runtime environment without becoming bogged down in the untenable complexity of the complete system.

Data Warehouse

In Figure 6, ovals represent a component called the Data Warehouse, which is tightly coupled with the Uintah environment itself and provides storage management services useful for iterative parallel computations. The data warehouse presents developers with an abstraction of a global single-assignment memory, with automatic data lifetime management and storage reclamation. The data warehouse is aware of component slicing, and handles migration and distribution of ghost data across spatially decomposed computations in an efficient way. A Uintah simulation need not necessarily keep its data in the data warehouse, but those that do gain the advantage of these services.

The data warehouse is responsible for managing the long term storage of the simulation's data. At the scientist's request, the data warehouse will store, in a scalable and parallel way, interesting data to disk. Simulations that take full advantage of the data warehouse to store all their state can be checkpointed by simply storing a full snapshot of state. The single assignment semantics of the data warehouse assures that the data existing at a timestep boundary can be identified and used to safely restart the simulation, enabling the scientist to "pick up and steer" a previously completed simulation from an intermediate point.

Visualization components can take advantage of the fact that the data warehouse is a central repository for the simulation data. The visualization tool can ask the data warehouse for subsets of the simulation data (filtered in space and/or time), thus making the visualization tool's job of displaying informative much easier.

Resource Acquisition and Simulation Startup

The first step in starting a Uintah simulation session is to start a Master Controller Process (MCP) on a reliable machine and attach a Uintah PSE interface to it. The scientist specifies the computation visually within the PSE by selecting and connecting components from the list of those available, to obtain a rep-

resentation similar to that depicted in Figure 2.

To facilitate easy access, MCPs can dynamically register themselves with the C-SAFE web server, at which point, any detachable PSE graphical user interface (GUI) can request a list of registered MCPs. Once the detachable PSE GUI connects to an MCP, the MCP provides a snapshot of the current simulation state (if any) to the GUI, where it is displayed to the user.

When a user requests that a component be instantiated, the MCP (or the user if she so indicates) chooses the best machine(s) on which to run that component. The first step towards instantiating the component on the given machine is to create a Slave Controller Process (SCP) on the machine. If the component is to run on more than one machine (or shared memory node) then a SCP will be created on each machine/node. The SCP will then determine the resources available to it (e.g. the number of shared memory processors) and may allocate a number of threads which will handle the execution of each of the component slices assigned to the given SCP.

In this example we are running the simulation on 9 128-processor Origin 2000 boxes, resulting in 9 Slave Controller Processes each handling component startup on a 128 processor shared memory node.

Component Slicing and Communication

Arrows which can be thought of as representing physical coupling in Figure 6 in fact represent component interface connections within Uintah. In order for two components to be connected, they must implement compatible interfaces. For example, the particular CFD and MPM components selected can be coupled only because they both implement an interface abstraction designed for coupling combustion and solid mechanics algorithms.

It is important to keep in mind that the components manipulated by the scientist from the Uintah PSE interface are in fact *parallel* components which during execution will each be implemented by perhaps thousands of slices, where each slice may run with slices of other components on the same processor of a large parallel machine. Thus interface composition correlates with parallel communication during execution.

Since these interfaces are specified in SIDL, this communication can be optimized with no special programmer effort for situations where slices are communicating within the same address space, or across boxes via Nexus, MPI or some other message-based transport.

It is plain to see that the choice of assignment of slices of components to particular processors is a critical factor in determining simulation performance. One such assignment is shown in Figure 7. This assignment may not result in optimal performance because the coupling between CFD and MPM components may require bandwidth in excess of that available between processors. Placing communicating slices on different processors within the same node is often a better choice than choosing different nodes.

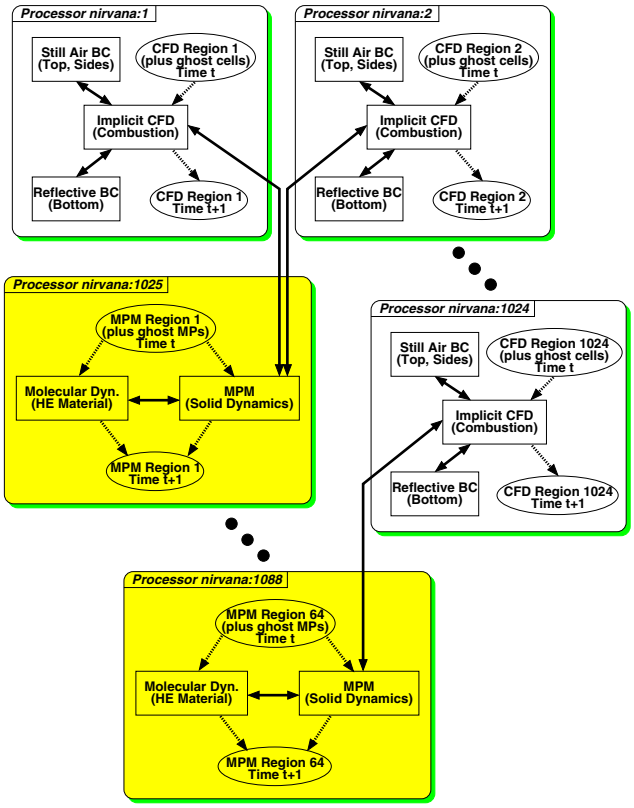

Figure 7: One Possible Slicing of the Computation in Figure 6

The slicing and processor assignment used for a particular computation is currently determined by the combination of inputs from the user and dynamic decisions made by a rudimentary scheduler. The task of automating and optimizing this assignment is a specialization of the more general task of dynamic load balancing by slice migration, and is a subject of ongoing work beyond the scope of this paper.

Component Startup and Composition

Once a slicing strategy has been determined and simulation execution is ready to begin, the master controller delivers to each node's slave controller information about what slices it is to execute. This information includes the actual component implementation binary, which is dynamically linked into the slave controller

process. It also includes any staging and initialization information the component requires, as well as all parameters and constants the component's operation may be parameterized upon that are not communicated over one of the component's interfaces.

The slave controller also receives information about the component compositions (component interface connections) its slices participate in. Those slices that are composed within a single address space communicate by (indirect) function call. Slice compositions between address spaces are implemented by intervening Nexus stub functions generated for the interfaces by the SIDL stub generator. A standard dynamic function dispatch table makes this selection possible, and enables steering by dynamic component recomposition, as described below.

Once the slave controllers have successfully assembled the desired simulation, slices begin executing. In the case of our example simulation, computation within each slice occurs as the slice's requested data becomes available in the data warehouse. The synchronization and propagation of data among slices is internal to and the primary purpose of the data warehouse component. Synchronization between the data warehouse and other components naturally follows from the normal caller-blocks semantics of SIDL method invocations.

Steering and Dynamic Component Recomposition

Once a simulation is running, there are several flavors of steering that are possible:

Data steering involves simply changing data values without affecting the algorithmic structure of the program. In our example, data steering would involve intercepting values as they are placed into the data warehouse and storing others in their stead. The data warehouse provides operations to make this possible. For example, heptane could be added to the pool fire simply by making the values for heptane species concentration nonzero in the combustion state near the ground plane. The same visual data manipulation tools that the scientist initially used to specify the kerosene are now used to steer the application.

Algorithmic extension involves adding orthogonal computation and data to a running simulation. For example, we might add a volume visualization to the simulation to study the effects of heptane addition on CO_2 concentration in the pool fire. We do this in the same PSE interface in which we originally performed component composition to construct the simulation. This time we select a volume visualization component and connect it to the desired data in the data warehouse. The slices of this component are best run on a 32-processor visualization machine located at Utah. This machine is acquired and the component slices are started exactly as before. The visualization slices request information from the data warehouse as the scientist animates visualizations of the ever growing set of simulation results. The single-assignment semantics of the data warehouse ensure that this inspection is does not affect the simulation progress.

Algorithmic steering involves changing the component composition in such a way that the simulation algorithm actually changes as it is running. For example, the scientist might wonder about the effects of more realistic outdoor air motion on the time to detonation. If there did not already exist a component to simulate gusty winds, and her programming skills were up to the task, she could quickly write one in C++ as a specialized kind of CFD boundary condition component. The SIDL tools would be used to generate appropriate communication stubs for her implementation from the already existing CFD boundary condition interface. Now having a gusty wind component, the scientist would load it into the PSE builder, and recompose the CFD component with it rather than with the still-air BC component. The MCP and SCP cooperate to deliver her new component to the appropriate boxes. On the next timestep boundary, the still-air BC component would decouple itself from the CFD and the framework would couple the gusty-wind BC component to the CFD in its place. In such a scenario, the components being uncoupled cooperate to determine when the uncoupling can take place without sacrificing the robustness of the overall simulation.

In all forms of steering, the scientist must ensure that her steering decisions do not affect the stability of the computation in an undesired way. Discontinuous changes like the instantaneous addition of quantity of heptane to the fire's fuel source will produce discontinuous changes in pressure and temperature that can ripple out as a wave of non-physical side-effects throughout the computation. It is assumed that the scientist understands the magnitudes and relevance of these effects.

Conclusion

The Uintah PSE framework provides an environment that allows scientific programmers to more easily create coupled, parallel simulation components while at the same time allowing them to easily explore the effects of dynamically changing a large number of parameters during a simulation run. Because of Uintah's component architecture which automates the grunge work of communication and distribution, scientists can also more easily explore the use of different methods to

solve the same problem.

The Uintah PSE is a very powerful simulation tool that provides a number of advances over past tools. These include support of both distributed and shared memory computations, increasing the number and types of components that are interoperable within a single framework, adding additional data interfaces between components (Dataflow and Uses/Provides ports for parallel communication), and allowing a detachable user interface that supports a number of implementations (such as a TCL, Java, or Web Based GUI). Uintah combines the interaction capabilities of SCIRun with the parallel-communication capabilities of the Common Component Architecture. This allows Uintah to support a large number of interoperable, highly parallel components.

References

[1] Center for the Simulation of Accidental Fires and Explosions - Annual Report, Year 2. http://www.csafe.utah.edu/documents.

[2] Common Component Architecture Forum. http://z.ca.sandia.gov/ cca-forum.

[3] The Common Object Request Broker: Architecture and Specification. Revision 2.0. OMG Document, June 1995.

[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] Academic Strategic Alliances Program. http://www.llnl.gov/asci-alliances.

[6] 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.

[7] T. De Fanti et al. Special Issue on Visualization in Scientific Computing. *Computer Graphics*, 21(6), 1987.

[8] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 11(2):115–128, 1997.

[9] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[10] C. R. Johnson and S. G. Parker. Applications in Computational Medicine Using SCIRun: A Computational Steering Programming Environment. *Supercomputing '95*, pages 2–19, 1995.

[11] B. A. Kashiwa, N. T. Padial, R. M. Rauenzahn, and W.B. VanderHeyden. A cell-centered ice method for multiphase flow simulations. Technical Report LA-UR-93-3922, Los Alamos National Laboratory, 1994.

[12] S. A. Kumar. A Comparative Analysis of Solution Approaches for Steady-State Three-Dimensional Internal Flows. Ph.D. Dissertation, Department of Chemical and Fuels Engineering, University of Utah, 1999.

[13] M. Miller, C.D. Hansen, S.G. Parker, and C.R. Johnson. Simulation Steering with SCIRun in a Distributed Memory Environment. In *Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC-7)*, July 1998.

[14] S. G. Parker and C. R. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. *Supercomputing '95*, 1995.

[15] S.G. Parker, M. Miller, C.D. Hansen, and C.R. Johnson. An Integrated Problem Solving Environment: The SCIRun Computational Steering System. In *31st Hawaii International Conference on System Sciences (HICSS-31)*, 1998.

[16] J.A. Schmidt, C.R. Johnson, and R.S. MacLeod. An Interactive Computer Model for Defibrillation Device Design. In *International Congress on Electrocardiology*, pages 160–161, 1995.

[17] G. D. Smith, W. Paul, M Monkenbusch, L. Willner, D. Richter, X. H. Qui, and M. D. Ediger. Molecular dynamics of a 1,4 polybutadiene melt. comparison of experiment and simulation. Macromolecules: Submitted.

[18] D. Sulsky, Z. Chen, and H. L. Schreyer. A Particle Method for History Dependent Materials. *Comp . Methods Appl. Mech. Engrg*, 118, 1994.

Acknowledgments

This work was supported by an award from the DOE ASCI program. We would like to thank our C-SAFE colleagues for their valuable insights relating to the needs of scientists with respect to a general purpose problem solving environment.

We would also like to thank *Schonbucher Institute for Technical Chemistry, University of Stuttgart* for the photo of a pool fire used in Figure 1.