Integrating Performance Analysis in the Uintah Software Development Cycle

J. Davison de St. Germain[1], Alan Morris[1], Steven G. Parker[1],
Allen D. Malony[2], and Sameer Shende[2]

[1] School of Computing,
University of Utah
{dav,amorris,sparker}@cs.utah.edu

[2] Department of Computer and Information Science,
University of Oregon
{malony,sameer}@cs.uoregon.edu

Abstract. Technology for empirical performance evaluation of parallel programs is driven by the increasing complexity of high performance computing environments and programming methodologies. This paper describes the integration of the TAU and XPARE tools in the Uintah computational framework. Performance mapping techniques in TAU relate low-level performance data to higher levels of abstraction. XPARE is used for specifying regression testing benchmarks that are evaluated with each periodically scheduled testing trial. This provides a historical panorama of the evolution of application performance. The paper concludes with a scalability study that shows the benefits of integrating performance technology in the development of large-scale parallel applications.

1 Introduction

Modern scientific simulations have become incredibly complex. It is not uncommon for high-performance software systems to have large development teams involving personnel across a broad range of expertise who work simultaneously on different parts of the system. In these programming environments, software developers increasingly turn to industrial tools for managing the complex software process. Tools for revision control, automated testing, and bug tracking are now commonplace. Unfortunately, tools to help achieve the highest performance possible over a broad range of inputs and hardware configurations are not commonly available. As a result, many software development efforts leave performance evaluation and improvement until the end of a long, many-stage development process. Even if performance is studied early in development, tracking the performance of the system as new features are added is often too time-consuming. While the complexity of the software development process may justify these engineering decisions. increased sophistication in high-performance parallel software and platforms rarely reduces performance complexity as development and use of the software proceeds.

Certainly, one very serious problem that arises is when developers of parallel scientific software make design decisions without knowledge or understanding of the performance ramifications. Any code decision, however localized, may have significant impact on performance overall. These performance influences can be difficult to observe and subtle to understand. If a performance engineering methodology is not incorporated in the software design and development process, it will be extremely difficult to achieve the high-performance goals of the project over its lifetime. Moreover, if the methodology is not adequately supported by flexible and robust performance tools, it will be difficult to address all performance problems that arise.

In this paper, we report on our efforts to integrate performance analysis capabilities into one such complex scientific software system: the Uintah Computational Framework. These capabilities support a performance engineering methodology that augments Uintah's current software design process. We describe the Uintah system in sufficient detail to highlight the challenges we have faced in performance measurement and analysis, and in tracking, maintaining, and improving Uintah performance. The TAU and XPARE tools we developed for Uintah performance engineering are then discussed in detail. We demonstrate their benefits to Uintah performance analysis and improvement with several examples. Finally, we outline our plans for future work.

2 Background and Motivation

In 1997, the Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [2] was created at the University of Utah to focus 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. C-SAFE was created by the Department of Energy's Accelerated Strategic Computing Initiative's (ASCI) Academic Strategic Alliance Program (ASAP) [1].

C-SAFE's objective is to build a problem-solving environment in which fundamental chemistry and engineering physics are coupled fully with non-linear solvers, optimization, computational steering, visualization and experimental data verification. Such a system would allow better evaluation of the risks and safety issues associated with fires and explosions. However, the software needed to model such real-world scientific and engineering problems is very complex, and is further compounded when multiple simulation codes must work together. Likewise, achieving high performance on large-scale computer systems is a necessary, but non-trivial goal.

C-SAFE's Uintah Problem Solving Environment [4] is a massively parallel, component-based, problem solving environment (PSE) designed to simulate large-scale scientific problems, while allowing the scientist to interactively visualize, steer, and verify simulation results. Uintah is derived from the SCIRun[3] PSE [9–12], adding support for a more powerful component model on distributed-memory parallel computers. The Uintah PSE is being developed specifically to study interactions between hydrocarbon fires, structures, and high-energy materials (explosives and propellants), such as those shown in Figure 1.

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 pos-

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

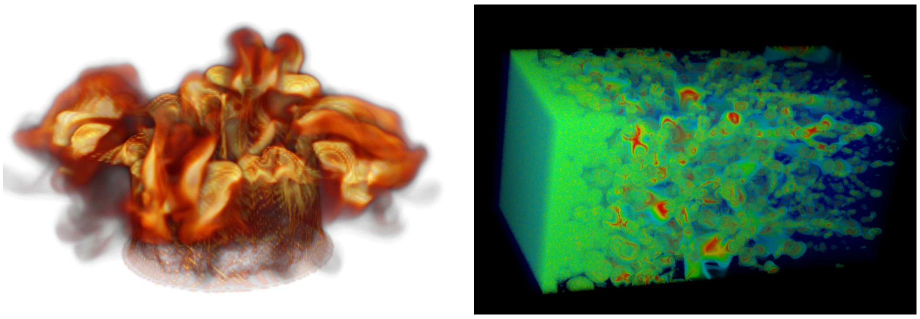

Fig. 1. Visualization of two different simulations from C-SAFE. On the left is a simulation of a heptane fire. On the right is a simulation of stress propagation through a block of granular material. Each of these simulations were performed using the Uintah Computational Framework and were executed on 1000 processors.

sible) be hidden from the scientist. Second, complex simulation components developed by third parties should be tools available for scientists to employ. And third, the scientist should be able to visually monitor and steer his or her simulation while it is running. A software environment that efficiently integrates these properties into a usable system allows scientists to effectively create and use complex simulations in an interactive, exploratory way. The Uintah 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 realize the full potential of large parallel computers.

While Uintah is provides 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 2. In order to produce realistic simulations of these problems, we must utilize large-scale parallel computers at maximum efficiency. For the largest simulations, we use DOE ASCI computing resources consisting of thousands of processors. A typical simulation consists of billions of degrees of freedom or more.

During simulation software development at C-SAFE, the need for performance analysis became very apparent. In particular, performance measurement and analysis tools were required for three main tasks:

1. Optimization of code kernels for maximum serial performance (micro tuning).
2. Analysis of parallel execution bottlenecks (scalability tuning).
3. Understanding the performance impacts of code modifications over the course of development (performance tracking).

By integrating tools to address these tasks in the Uintah PSE development process, we have created a scalable simulation environment for C-SAFE problems where performance of the overall environment is high and will not diminish unexpectedly due to evolution of the Uintah code.

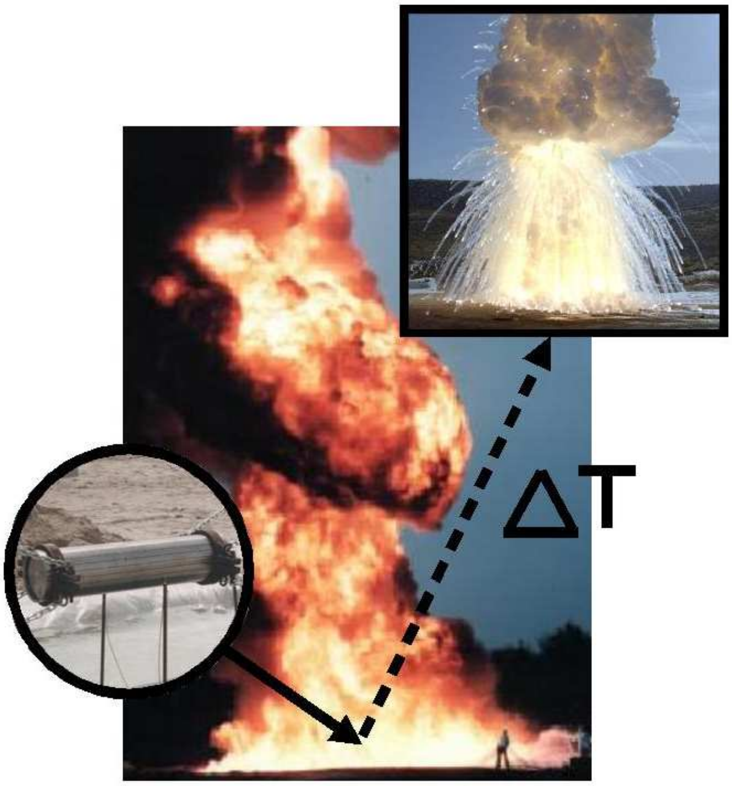

Figure 2: A Typical C-SAFE Problem

3 Uintah Architecture

The Uintah PSE provides a component-based environment for developing parallel scientific applications. Uintah is based on the component architecture being developed by the Common Component Architecture (CCA) Forum. The CCA Forum [3] was established to specify a software component architecture that could address the needs of high-performance computing. The CCA architecture aims to provide higher performance, explicit support for multi-dimensional arrays, and support for parallelism. Uintah is a research vehicle for implementing these ideas and for exercising their efficacy on complex scientific applications, such as the C-SAFE simulations.

Solving a typical C-SAFE problem involves running multiple large-scale physically coupled simulations. For example, to investigate the effects of fire on metal structures, a fluid-dynamics-based combustion model might be coupled with a particle-based solid mechanics simulation. The simulation models may involve representations of size 10^9 finite volume cells and 10^8 solid material points. To handle the large number of operations necessary to process such immense datasets, we have designed the *Uintah Com-*

putational Framework (UCF). The UCF is the foundation upon which all C-SAFE simulation components are developed.

The UCF is a set of components and classes that build on the Uintah component model, adding capabilities such as semi-automatic parallelism, automatic checkpoint/restart, load-balancing mechanisms, resource management, and scheduling. The UCF exposes flexibility in dynamic application structure by adopting an execution model based on software or "macro" dataflow. Computations are expressed as directed acyclic graphs of *tasks*, each of which consumes some input and produces some output (input of some future task). These inputs and outputs are specified for each patch in a structured grid. Tasks are organized in a UCF data structure called the *task graph*.

In natural agreement with the functional nature of its pure macro-dataflow execution model, the UCF presents developers with an abstraction of a global single-assignment memory, with automatic data lifetime management and storage reclamation. Storage is abstractly presented to the scientific programmer as a dictionary mapping names to values. The value associated with a name can be written only once, and once written is communicated by UCF to all tasks awaiting that value. Values 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 mapped efficiently onto both message-passing and share-memory communication mechanisms. Threads sharing a memory can access their input data directly; single-assignment dataflow semantics eliminate the need for complex locking of values. The UCF is free to optimize allocation of physical memory to minimize remote memory accesses. 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, the UCF has complete flexibility in control and data placement to optimize communication both between address spaces and within the shared ccNUMA memory hierarchy of the Origin 2000 (or other SMP-based distributed memory supercomputers). Solving this optimization problem for C-SAFE simulations is difficult and is a subject of ongoing investigation.

An example UCF taskgraph is shown in Figure 3. Ovals represent tasks, each of which is a simple array algorithm and easily treated by traditional compiler array optimizations. Edges represent named values stored by the 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 been updated during the time step. The figure shows a slice of the actual Uintah Material Point Method (MPM) task graph concerned with advancing Newtonian material point motion on a single patch for a single timestep.

4 Performance Technology Integration

The Uintah PSE and the UCF present interesting challenges to performance analysis technology and its integration. The diversity of the Uintah software, including the UCF

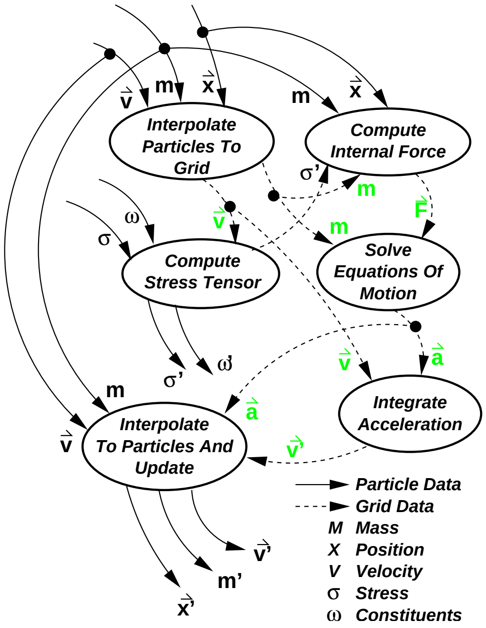

Figure 3: An Example UCF Task Graph

middleware and simulation code modules, and Uintah's portability objectives requires performance instrumentation and measurement tools that are both cross-language and cross-platform. The performance system must also work at large scales, and be able to analyze performance data captured for the different execution modes (shared-memory, message passing, mixed-mode) that Uintah supports. Perhaps the most important concern is being able to relate multi-level performance data to the high-level task abstractions used within Uintah for simulation programming and during execution by the UCF for task graph scheduling and storage management. Without this capability, it would be extremely difficult to piece apart performance effects across UCF levels and to identify the simulation components responsible for different performance behaviors.

4.1 TAU Performance System

Performance technology integration in the Uintah PSE is based on the TAU performance system [7]. TAU provides robust technology for performance instrumentation, measurement, and analysis for complex parallel systems. It targets a general computation model consisting of shared-memory computing *nodes* where *contexts* reside, each providing a virtual address space shared by multiple *threads* of execution. The model is general enough to apply to many high-performance scalable parallel systems and programming paradigms. Because TAU enables performance information to be captured at the node/context/thread levels, this information can be mapped to the particular parallel software and system execution platform under consideration.

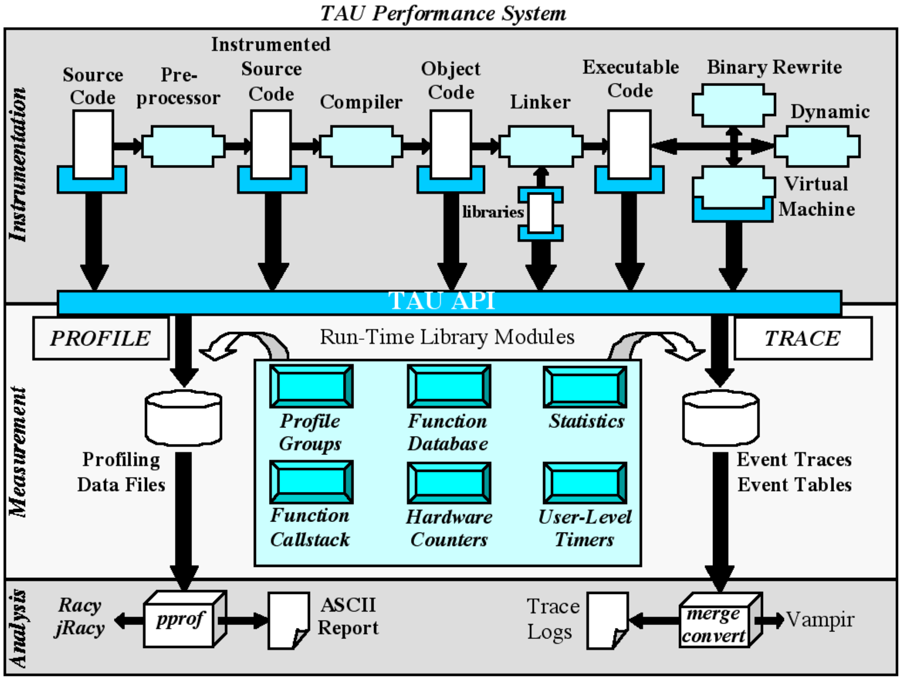

Figure 4: TAU Performance System Architecture

As shown in Figure 4, TAU supports a flexible instrumentation model that applies at different stages of program compilation and execution. The instrumentation targets multiple code points, provides for mapping of low-level execution events to higher-level performance abstractions, and works with multi-threaded and message passing parallel computation models. Instrumentation code makes calls to the TAU measurement API. The TAU measurement library implements performance profiling and tracing support for performance events occurring at function, method, basic block, and statement levels during execution. Performance experiments can be composed from different measurement modules (e.g., hardware performance monitors) and measurements can be collected with respect to user-defined performance groups. The TAU data analysis and presentation utilities offer text-based and graphical tools to visualize the performance data as well as bridges to third-party software, such as Vampir [8] for sophisticated trace analysis and visualization.

4.2 TAU Performance Mapping in Uintah

To evaluate the performance of Uintah applications, we selectively instrument at the source level and the message passing library level. Source-level instrumentation occurs at subroutine and method boundaries, as well as at important code sections using TAU user-defined timers (with *start/stop* semantics) to highlight the time spent in

Figure 5: TAU Performance Profiles Without Mapping (top) and With Mapping (bottom)

groups of statements. Message passing instrumentation (using a MPI interposition library based on PMPI [6]) shows both execution time spent in message communication and messaging behavior with respect to application level routines. Figure 5 shows two profiles of the execution time of different tasks within the UCF's parallel scheduler for an MPI-only run. The displays were created by TAU's parallel profile visualizer, *Racy*, which can show full profile details across all threads of execution. Here, the right views show the detailed performance profile on "n,c,t (node,context,thread) 0,0,0" (i.e., MPI process with rank 0). The left views show performance for all of the MPI processes in bargraph form.

To generate the top two views, we placed instrumentation in the *MPIScheduler* class and the MPI library. Clearly, *Task execution [MPIScheduler:execute()]* (green bar) takes up a significant chunk of the overall execution time, 79.91% of the total (exclusive) on MPI process 0. The time spent in *MPI_Waitall()* and *MPISchedule::gatherParticles* is also of significance, but the other routines are of less consequence. Unfortunately, these top two views give only a rough breakdown of UCF performance. While it is important to see a high percentage of time being spent executing tasks, what the scientist wants to know additionally is the distribution of the overall task execution time among the different types of tasks performed. While more detailed instrumentation (using user-defined events and tracing) can show each instance of task execution, standard instrumentation mechanisms have no means to identify task semantics (i.e., from what simulation component the tasks were produced). To understand TAU's solution to this problem, we need to describe how UCF operates in more detail.

During the computation, many individual particles are being partitioned across processing elements (processes or threads) and worked on by the simulation components represented in the task graph. As work is performed on the particles, a *task instance* is created and scheduled. Each task instance corresponds to some simulation operation (task), such as interpolating particles to the grid in the Material Point Method, and its execution is controlled by its task graph dependencies. We can give each task instance a name (e.g., *SerialMPM::interpolateParticlesToGrid*) that identifies its domain-specific character in the computation (i.e., its specific simulation task relationship). The the number of task types is finite and is typically less than twenty in Uintah applications. In contrast, there are a large number of task instances created and executed during the computation. The association of a task type with a task instance occurs at a time different from when the task instance is finally scheduled and executed.

Thus, to provide the desired performance view, we must map the performance of each individual task instance to the task type to which it belongs and then accumulate the performance data at the task level. Using TAU's *Semantic Entity, Association, and Attributes* (SEAA) model of performance mapping [13], we form an association during initialization between a timer for each task (the task semantic entity) and the task name (its semantic attribute). Then, while processing each task instance in the scheduler, a method to query the task name (stored within the task instance object) is invoked and the address of the task name (a static character string) is returned. Using this address, we do an external map lookup (implemented as a hash-table) and retrieve the address of the timer object (i.e., a runtime semantic association). Once the timer is known, it can be started and stopped around the code segment that executes the task instance.

The bottom two views in Figure 5 show the results of this task mapping performance analysis in Uintah. Clearly, there is a significant benefit of the SEAA approach in presenting performance data with respect to high-level semantics of the Uintah application. The performance of all five simulation model components (i.e., tasks) are now clearly distinguished in the profile. With the generation of event traces, the benefits are even more dramatic as this task mapping allows distinct phases of computation to be highlighted based on task semantics. This can be seen in the trace visualization in Figure 7. Although we are looking at individual task instances being executed, the color-coded mapping allows us to view their performance data at a higher level.

4.3 Performance Experiment Reporting and Alerting

With the integration of performance measurement support in the Uintah software system comes the ability to analyze performance throughout Uintah's development lifetime. Typically, performance analysis is done ad hoc, at the convenience of the developer, and only when time permits. When such performance practice is applied across a large, multi-person effort such as C-SAFE, the resulting "performance portfolio" becomes scattered and tends to report performance information only after significant stages of development have been accomplished and software committed. The downside of such a performance methodology is a disengagement of performance knowledge from key software design decisions. The goal of our work is to more tightly couple the reporting of performance experimentation results with timely software testing and alerting to performance problems. We have created the XPARE (eXPeriment Alerting and REporting) system for this purpose.

The Uintah software system was engineered with a regression testing harness to regularly evaluate correctness. At these times, minimal performance benchmarking would be conducted to determine if total execution time was seriously degraded. If so, the tester would notify software developers, but left it up to them to manually run specific instrumented tests to investigate where the performance problems lay. The XPARE system augments the regression tester to conduct a range of performance experiments with fully-instrumented code modules. Multiple experiments can be conducted with different instrumentation layouts to exercise different code regions and behaviors. The TAU performance tools are used for measurement and analysis, allowing execution time and hardware statistics to be used to construct a complete performance portrait.

Once the performance experiments have been conducted, XPARE will automatically interrogate the performance data to determine not only if the overall code has run for longer than expected, but also which tasks and profiled procedures are potential suspects. XPARE accomplishes this by applying alerting "rulesets" (performance difference thresholds) to a historical, multiple experiment performance database. Experiment sets can be selected by the user from the database for evaluation. For each experiment set, specific performance data can be chosen for analysis. Performance regression testing is then done by comparing the current performance with that in the experiment set, using the alerting rulesets constructed by the user to determine performance violations worthy of report.

The XPARE system architecture is shown in Figure 6, with images of the web-based interfaces for experiment selection, performance data selection, and ruleset definition. As also shown, results of regression analysis are automatically reported to the software developers, who can explore the performance data more fully through the performance reporter, whether or not significant performance shifts have been detected. Because the performance database contains prior performance history, a panoramic view of performance change can be scrutinized based not only on code alteration, but also platform, choice of compiler, different optimizations, and other performance factors.

By scheduling regular performance regression tests, performance knowledge can be closely linked with the Uintah software development cycle. Currently, we use XPARE to run weekly performance tests of small to medium-scale experiments, and monthly evaluations of full-scale experiments. The general construction of XPARE will allow it to easily extend to changes in the Uintah code base and to incorporate new simulation components as they become available.

5 Performance Studies

Contemporary efforts in gathering performance data have focused on function by function analysis. C-SAFE has taken the somewhat novel approach of gathering performance statics on an *algorithmic* basis. This approach provides four major benefits.

1. Due to the use of the task abstraction in the UCF, it is straightforward to manually insert the profiling code at one location in the code to capture data on the performance of all tasks.

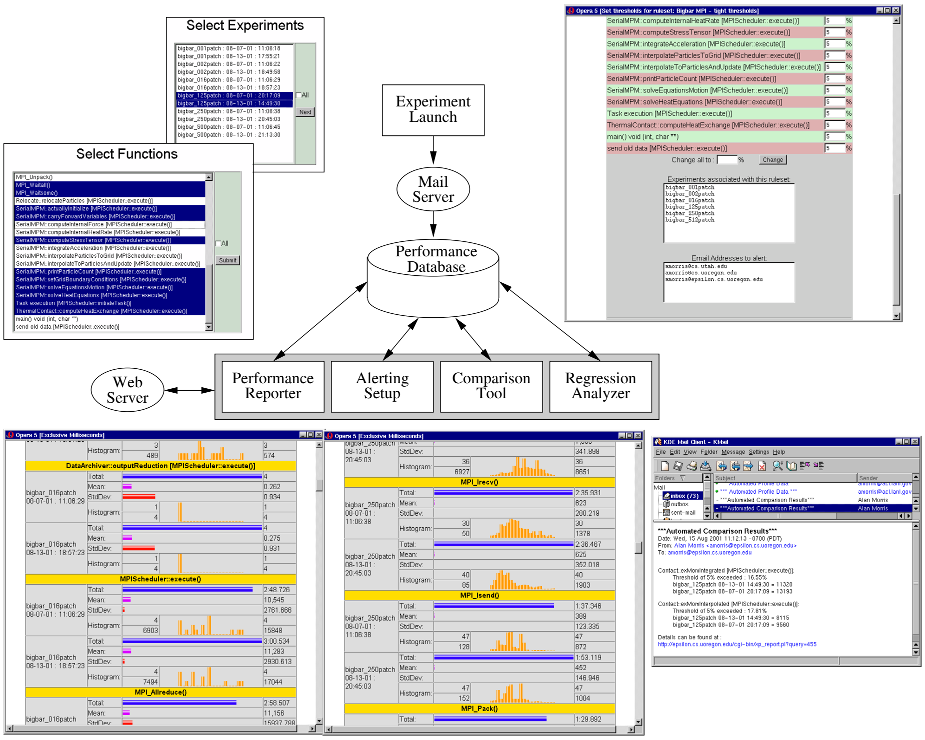

Figure 6: XPARE System Architecture and Tools

2. The performance characteristics of each of the algorithmic tasks is clearly displayed in relation to the other simulation tasks.
3. Scientific programmers are allowed to focus on making performance improvements at an algorithmic level.
4. Uintah Computational Framework developers can easily find performance bottlenecks that are not directly associated with application codes (e.g.; MPI communications, task scheduling overhead, and data I/O).

The first step in optimizing Uintah software was to manually instrument the code base with hooks to the TAU system. The event-traces generated were converted to the Vampir trace data format and visualized using Vampir. Figure 7 depicts one of the first visualizations of an early version of the Uintah code running an MPM simulation on 32 processors. The figure shows six time steps with the black lines between the time steps depicting the large MPI communications necessary to transmit boundary data. Listed on the right hand side of the window are each of the specific tasks, delineated by major software component (e.g.; SerialMPM, MPMICE, DataArchiver, Contact, etc.) followed by specific task name (e.g.; computeStressTensor, relocateParticles, etc.) Each task can be color coded to easily view its location in the time line. On the left hand

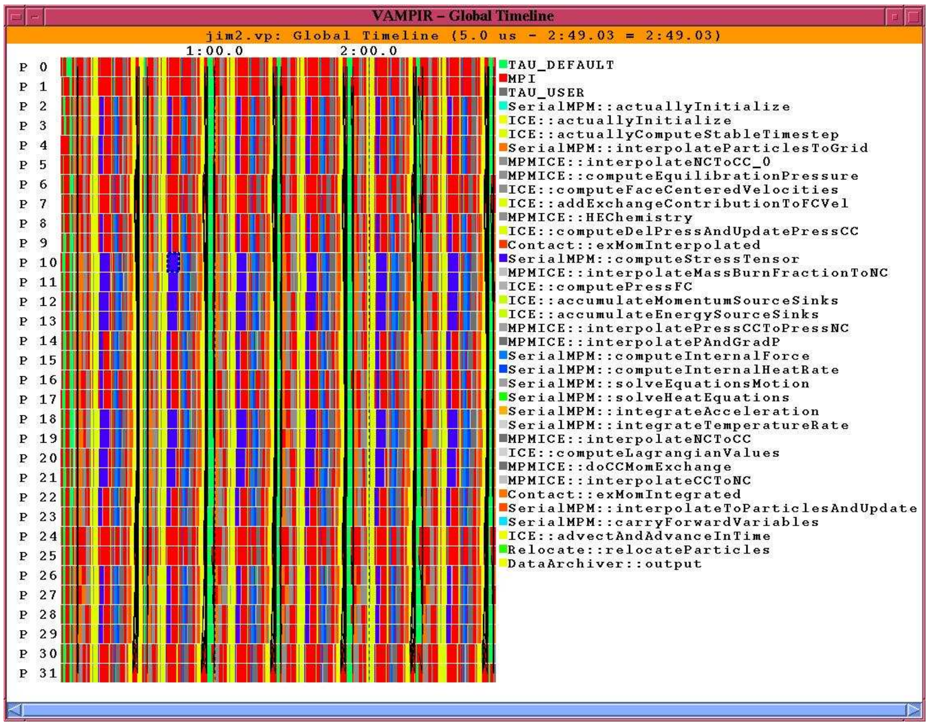

Figure 7: MPM Simulation Performance (TAU /Vampir)

side are rows displaying time lines for each process, running in parallel on individual processors in this simulation run.

When first viewed, this diagram provided a number of "Aha!" insights about the general behavior of the simulation. These insights included understanding:

1. the load imbalances we were experiencing with a rudimentary load balancer;
2. that the *computeStressTensor* task constituted a large portion of the execution time; and
3. that there was a significant amount of MPI overhead distributed throughout the computation.

Figure 8 is a zoomed-in view of a single time step in the MPM Simulation. This view provided insight into the parallelization of each of the tasks in a single time step. It also provided us with a visual feedback for how the processors where lining up and how much work each was doing.

Similarly, Figure 9 depicts five time steps of the "Arches" fire simulation within the UCF. This figure portrays explicitly how much time is being spent in the "pressure solving" portion of the simulation. (The pressure solve calculation utilizes a PETSc linear solver.) Figure 10 is a close up view of the *PressureSolver* task within the time step and reveals that a major portion of the solver's time is spent in MPI calls. This

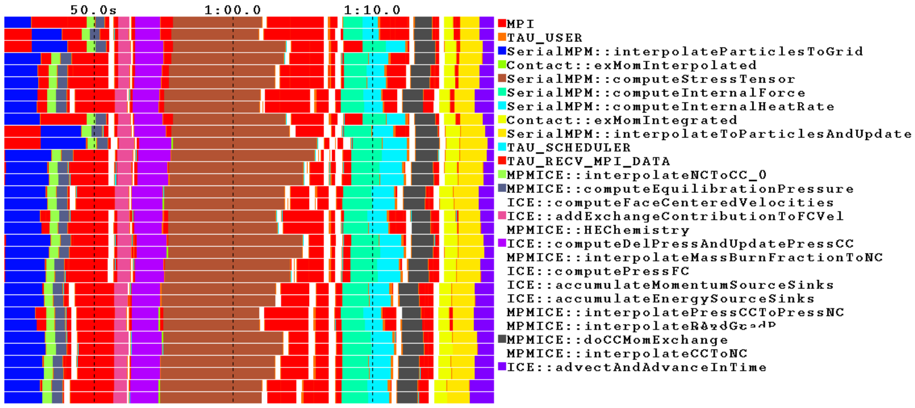

Figure 8: MPM Simulation (Single Time Step)

visualization has led to focusing performance enhancement resources on determining the best way to use PETSc solvers (including exploring different pre-conditioners).

Once candidate tasks are identified as potential performance bottlenecks, the tasks are inspected from both an algorithmic view and from an implementation view. At this point, it is sometimes necessary to perform additional functional instrumentation of the code. We used this method of performance analysis from late 2000 through the first half of 2001 to investigate performance problems in the Uintah software. This lead to the parallel scaling improvements seen in Figure 11. Successive lines on the graph show the performance improvements after finding and fixing performance bottlenecks.

After directing our efforts at improving the Unitah scalability up to 2000 processors, our focus changed to other aspects of code development. It was at this point that we recognized the need for the XPARE system. Once implemented, it has allowed us to monitor the performance of individual simulation pieces in addition to the overall performance. XPARE has been developed with the goal of keeping the Uintah system efficient as we expand the system and add new features.

6 Lessons Learned and Future Work

The integration of performance measurement in the UCF scheduling component has been extremely useful in exposing bottlenecks and inefficiencies. While the performance analysis thus far has mainly been done post-mortem, Uintah applications will be increasingly adaptive in the future and will require UCF to implement dynamically adjusting scheduling policies. We plan to develop online performance query and feedback capabilities in TAU that will support adaptive Uintah execution. Also, to enhance online performance analysis, we are developing a runtime infrastructure to visualize dynamic, large-scale performance data using the SCIRun visualization environment.

We will also continue to build on the success of performance mapping in Uintah to attribute execution costs from the simulation component parts. We have recently encountered the need for more flexible performance mapping specification that allows

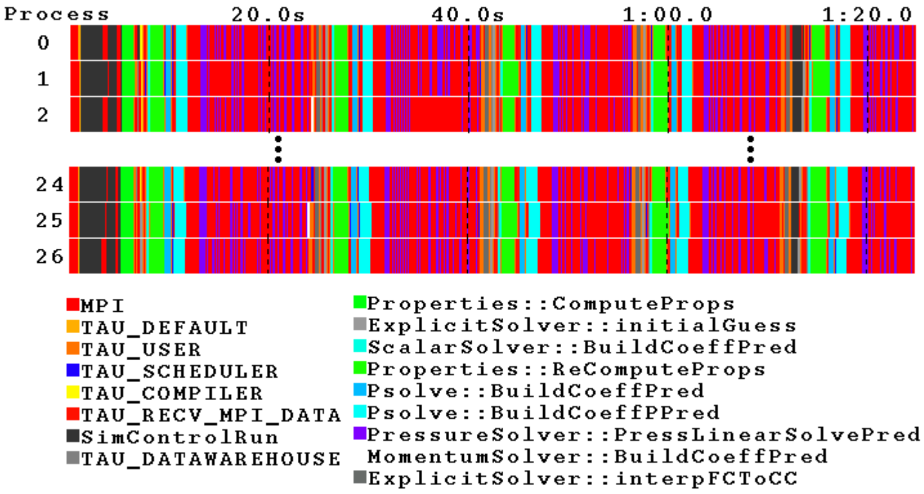

Figure 9: Arches Task Performance

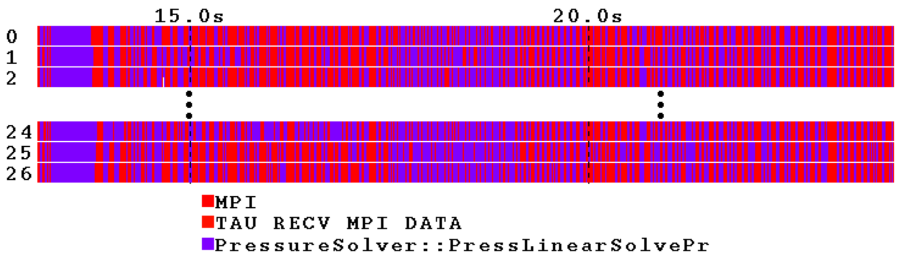

Figure 10: Arches Task Zoomed In

multiple mappings attributions (e.g., for mapping execution costs from component parts to higher-level tasks and patches) to be active simultaneously. The current rudimentary means to support these mappings will be implemented in more robust forms in the near future. Not only is the UCF a target for performance integration, but the individual simulation components can benefit from performance analysis. We will begin to work more closely with the developers of C-SAFE simulation component software to integrate performance measurement, analysis, and regression testing in their codes.

With the completion of a mixed-mode UCF implementation will come the need for performance analysis of integrated multi-threaded and message-based execution. While preliminary tests have demonstrated TAU's ability to observe thread and communication events in mixed-mode Uintah execution, it will be important to develop techniques for cross-mode sharing of instrumentation information so that integrative performance mapping and analysis is possible.

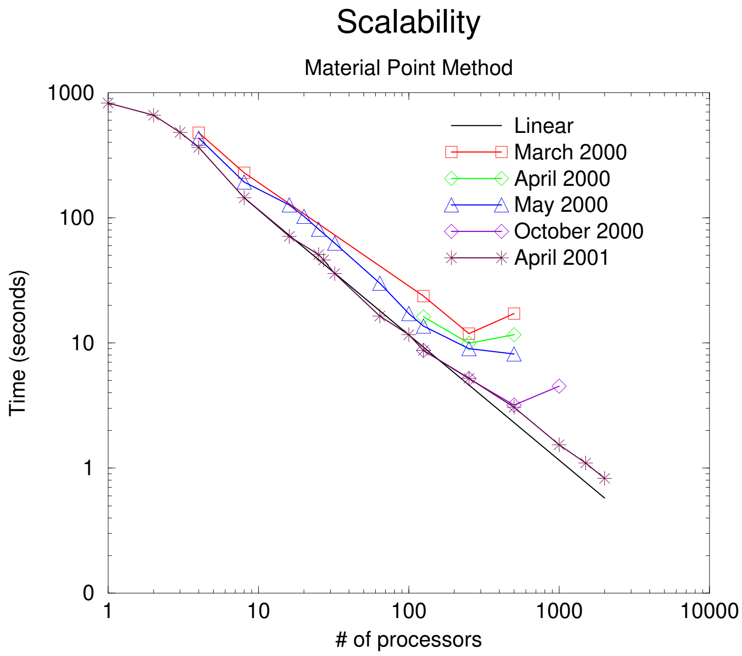

Figure 11: Parallel Performance Evolution

We will greatly enhance the existing prototype XPARE system to play an increasingly important role in Uintah software performance engineering in the future. In particular, we will concentrate on XPARE's performance database which is currently implemented in an ad hoc manner. The TAU project is building a performance database framework (PerfDBF) that will be employed by XPARE for more flexible cross-experiment data query and analyses. PerfDBF will allow for the set of analysis operations to be easily extended by UCF and simulation component developers. XPARE's alerting and reporting tools can then incorporate these expanded analysis options to construct more sophisticate threshold functions and performance data processing for generating performance reports.

7 Acknowledgments

This work was supported by the DOE ASCI ASAP Program. The work at Oregon was supported by a contract from the DOE 2000 program (Agreement No. DEFC 0398 ER 259 986) and a sub-contract from the University of Utah's DOE C-SAFE ASCI center (Agreement No. B341493). C-SAFE visualization images were provided by Kurt Zimmerman and Wing Yee. Datasets were created by Scott Bardenhagen, Jim Guilkey, and Rajesh Rawat. The DOE ASCI ASAP program also provided computing time for the simulations shown.

References

1. Academic Strategic Alliances Program. http://www.llnl.gov/asci-alliances.
2. Center for the Simulation of Accidental Fires and Explosions. http://www.csafe.utah.edu.
3. Common Component Architecture Forum. http://www.cca-forum.org.
4. Davison de St. Germain, J., McCorquodale, J., Parker, S.G., Johnson, C.R.: Uintah: A Massively Parallel Problem Solving Environment. HPDC'00: Ninth IEEE International Symposium on High Performance and Distributed Computing (2000)
5. Lindlan, K.A., Cuny, J., Malony, A.D., Shende, S., Mohr, B., Rivenburgh, R., Rasmussen, C.: Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. Proceedings SC'2000, (2000)
6. Message Passing Interface Forum: MPI: A Message Passing Interface Standard. International Journal of Supercomputer Applications (Special Issue on MPI) 8(3/4) (1994)
7. Malony, A., Shende, S.: Performance Technology for Complex Parallel and Distributed Systems. In: Kotsis, G., Kacsuk, P. (eds.): Distributed and Parallel Systems From Instruction Parallelism to Cluster Computing. Proc. 3rd Workshop on Distributed and Parallel Systems, DAPSYS 2000, Kluwer (2000) 37–46
8. Pallas GmbH: VAMPIR: Visualization and Analysis of MPI Resources. http://www.pallas.de/pages/vampir.htm.
9. Parker, S.G., Beazley, D.M., Johnson, C.R.: Computational steering software systems and strategies. IEEE Computational Science and Engineering, 4(4) (1997) 50–59
10. Parker. S.G., *The SCIRun Problem Solving Environment and Computational Steering Software System*. PhD thesis, University of Utah (1999)
11. Parker, S.G., Johnson, C.R.: SCIRun: A scientific programming environment for computational steering. Proc. Supercomputing '95. IEEE Press (1995)
12. Parker, S.G., Weinstein, D.M., Johnson C.R.: The SCIRun computational steering software system. In: Arge, E., Bruaset, A.M., Langtangen, H.P., (eds.): Modern Software Tools in Scientific Computing, Birkhauser Press (1997) 1–44
13. Shende, S.: The Role of Instrumentation and Mapping in Performance Measurement. Ph.D. Dissertation, University of Oregon (2001)
14. Shende, S., Malony, A., Ansell-Bell, R.: Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation. Proc. International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '2001, CSREA, (2001) 1150–1156
15. Shende, S., Malony, A., Cuny, J., Lindlan, K., Beckman, P., Karmesin, S.: Portable Profiling and Tracing for Parallel Scientific Applications using C++. Proc. SIGMETRICS Symposium on Parallel and Distributed Tools, SPDT'98, ACM, (1998) 134–145