Improving the Performance of Uintah: A Large-Scale Adaptive Meshing Computational Framework

Justin Luitjens
School of Computing University of Utah
Salt Lake City, Utah 84112
luitjens@cs.utah.edu

Martin Berzins
School of Computing
University of Utah
Salt Lake City, Utah 84112
mb@sci.utah.edu

***Abstract*—Uintah is a highly parallel and adaptive multi-physics framework created by the Center for Simulation of Accidental Fires and Explosions in Utah. Uintah, which is built upon the Common Component Architecture, has facilitated the simulation of a wide variety of fluid-structure interaction problems using both adaptive structured meshes for the fluid and particles to model solids. Uintah was originally designed for, and has performed well on, about a thousand processors. The evolution of Uintah to use tens of thousands processors has required improvements in memory usage, data structure design, load balancing algorithms and cost estimation in order to improve strong and weak scalability up to 98,304 cores for situations in which the mesh used varies adaptively and also cases in which particles that represent the solids move from mesh cell to mesh cell.**

***Keywords*-Adaptive Mesh Refinement, Parallelism, Load Balancing**

I. Introduction

The University of Utah Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [1] is a Department of Energy ASC center that focuses on providing state-of-the-art, science-based tools for the numerical simulation of accidental fires and explosions. The primary objective of C-SAFE has been to provide a software system in which fundamental chemistry and engineering physics are fully coupled with nonlinear solvers and visualization tools, thereby integrating expertise from a wide variety of disciplines. The creation of Uintah has furthered C-SAFE's understanding of fires, explosions, and other problems involving complex fluid-structure interactions.

For example, on August 11, 2005 a truck carrying 35,500 pounds of explosives down Utah's Spanish Fork Canyon overturned and caught fire. Within minutes the truck detonated with a force much larger than expected leaving behind a 70 foot crater. Fortunately no one was hurt. Why did a detonation occur as opposed to a deflagration, which is several orders of magnitude less violent? Could the packing of the individual explosive charges influence the propagation of the combustion wave, or the amount of energy released? These are the types of questions that C-SAFE is addressing and hopes to further address through the use of future petascale simulations. Large-scale simulations have allowed C-SAFE to further the understanding of explosions by providing the ability to look more closely at the underlying physical phenomena than is possible through experimental tests.

The target simulation scenario for C-SAFE is a small cylindrical steel container filled with plastic bonded explosive (PBX-9501) subjected to convective and radiative heat fluxes from a fire which heats the container and the PBX. After some amount of time the critical temperature in the PBX is reached and the explosive begins to rapidly decompose into a gas. The solid-to-gas reaction pressurizes the interior of the steel container causing the shell to rapidly expand and eventually rupture. The gaseous products of reaction form a blast wave that expands outward along with pieces of the container and the unreacted PBX.

Simulating this problem requires expertise from a wide variety of disciplines including combustion, structural mechanics, and fluid dynamics. In addition, such a problem requires a large amount of processing power necessitating the need for both adaptive mesh refinement (AMR) [2] and parallelism. AMR focuses the computational resources where needed by adding refinement in areas where rapidly evolving physical processes are occurring. For example, in the case of the exploding container mentioned above; the container, the explosive, and the pressure wave all need to be highly resolved, where as the surrounding atmosphere has a lower resolution requirement. Even with AMR, the processing requirements for such a problem are still large necessitating the use of parallelism.

The need for parallelism, AMR, and a wide variety of physics has led to the development of the Uintah Computational Framework [1], [3], [4]. Uintah, which was developed by C-SAFE provides a large degree of encapsulation that allows scientists to focus on their area of expertise without fully understanding complexities outside of their domain.

In preparation for petascale architectures and simulations, the performance of frameworks like Uintah must be analyzed and optimized. Poor performance in any portion of the framework can have a significant impact on the overall performance. Achieving a high degree of scalability for AMR based simulations is challenging due to poor scalability associated with the changing grid. With AMR, whenever the grid changes, a number of operations must be performed.

For example, the new grid must be created, work must be load balanced and migrated to the owning cores, and the communication schedule must be created. In the past, optimizations to these operations have led to improvements in scalability within general purpose AMR frameworks [5]–[7]. Previously application specific AMR codes have been shown to scale up to 60,000 cores [8], [9]. The challenge is to see if a general purpose framework such as Uintah is capable of scaling to such numbers of cores. We address this challenge by: describing Uintah and its novel approach to parallelism, describing a tool used to identify inefficiencies in memory usage and data structures, and presenting a new method to estimate costs used in load balancing, which together have led to substantial improvements in Uintah's scalability.

II. Uintah

The Uintah computational framework, is a set of parallel software components and libraries built upon the DOE Common Component Architecture (CCA) that facilitate the solution of partial differential equations (PDEs) on structured AMR grids. Uintah is a sophisticated framework that can integrate multiple simulation components, analyze the dependencies and communication patterns between them, and efficiently execute the resulting multi-physics simulation. Uintah employs an abstract task graph representation to describe computation and communication [3], [4]. Through this mechanism, Uintah components delegate decisions about parallelism to a framework component, which determines communication patterns and characterizes the computational workloads needed for global resource optimization. This allows parallelism to be integrated between multiple components while maintaining overall scalability. Uintah also analyzes the structure of the computation and automatically enables load balancing, data communication, parallel I/O, checkpointing and restarting capabilities.

One of the primary strengths of Uintah is that application designers can develop large-scale parallel AMR simulations with little understanding of the underlying parallelism. To do this the designers must specify their algorithm as a series of serial tasks that run on a hexahedral mesh patch. Each task specifies the computation to be performed for a single time step and the related variable dependencies. Variable dependencies state what variables the task requires for the computation (along with the stencil width) and what variables the task modifies or computes. Using these dependencies, Uintah creates a directed acyclic task graph that specifies the task execution order and the required communication for the simulation. This design shields developers from the parallelism while allowing Uintah to utilize highly sophisticated communication patterns including a large amount of asynchronous communication and message coalescing. By using these advanced communication techniques Uintah is able to hide the cost of some of the communication by overlapping it with computation. As we move to petascale architectures advanced frameworks such as Uintah that can implement advanced parallel techniques independent of the simulation components will be increasingly necessary.

Uintah achieves parallelism by dividing the grid into hexahedral mesh patches, which are uniquely assigned to cores. Each core executes the tasks on its assigned patches achieving a domain-based parallelism. When a task requires a variable with a non-zero stencil width, communication between neighboring patches is required before the task can execute. Using the task graph and the core assignments for each patch, Uintah determines the communication necessary and schedules communication and computation at the appropriate times. All communication, including intra-level (within a single level), inter-level (between AMR levels), and data migration after load balancing is included in this schedule. The schedule is then executed repeatedly with each execution corresponding to a single time step of the simulation. In static grid computations, this schedule is created once and reused for the entire simulation. However, in AMR computations the schedule must be recreated whenever the patch set changes (regridding) or whenever the patch assignments change (load balancing). The framework also utilizes parallel I/O to store simulation data for use in checkpointing, restarting, and visualization within VisIt [10].

Uintah was originally designed for a few thousand cores and has been used regularly for AMR simulations with up to 2,000 cores. However, scalability at larger numbers of cores was problematic because the memory utilization, data structures, and load balancing did not scale well on 4,000 or more cores. In particular, memory utilization was an issue due to data structures that consumed memory on the order of the number of cores or the number of patches. As the number of cores or patches increased the size of these data structures would also increase eventually exceeding the available resources. Recently these inefficiencies were resolved through the creation of a tool that aides in the tracking memory allocations over time. These improvements are described in Section III.

The component design has allowed Uintah to excel as a research platform. Components can be swapped in and out, allowing them to be developed and tested within the entire framework, without affecting other components. This has led to a highly flexible simulation package which has been able to simulate a wide variety of problems including shape charges, stage-separation in rockets, the biomechanics of microvessels [11], the properties of foam under large deformation [12], and the evolution of large pool fires caused by transportation accidents [13], in addition to the exploding container scenario described in Section I.

Uintah currently contains three main simulation algorithms, or components, that are capable of using AMR: i) the ICE compressible multi-material CFD formulation [14]–[16], ii) the particle-based Material Point Method (MPM) [17] for structural mechanics, and iii) the combined fluid-structure interaction algorithm MPMICE [18]. In addition, Uintah integrates numerous sub-components including equations of state, constitutive models, and reaction models.

ICE is a "multi-material" CFD algorithm that was developed by Kashiwa and others at LANL [14]–[16]. This technique can be used in both incompressible and compressible flow regimes, which is necessary when modeling fires and explosions. This method conserves mass, momentum, energy, and the exchange these quantities between materials.

The *Material Point Method* is a particle method that is used to evolve the equations of motion for the solid materials. MPM is a powerful technique for computational solid mechanics, and has found favor in many applications involving complex geometries [11], large deformations [12], and fracture [19]. Originally described by Sulsky, et al., [20], MPM is an extension to solid mechanics of FLIP [21], [22], which is a particle-in-cell (PIC) method for fluid flow simulation [23]. MPM simulations have added complexity over ICE simulations because particles move throughout the simulation which can causes load imbalance.

III. Peformance & Memory Improvements

The original implementation of the Uintah framework used a few data structures that had memory complexity on the order of the number of patches or cores. As the problem size or number of cores increased the memory requirement of these data structures would also increase.

Uintah has been in development for over ten years and now contains around a half a million lines of code. Uintah was initially designed for upwards of a thousand cores. The data structures and algorithms used in Uintah worked well for a few thousand cores but became inefficient when moving to tens of thousands of cores. Inefficiencies like these will be common in many codes attempting to move onto petascale platforms. Identifying these inefficiencies in large legacy codes like Uintah is a challenging task that has been a significant focus of some C-SAFE personnel. Many inefficiencies within Uintah have been identified using parallel profiling tools like TAU [24]. In particular, TAU was recently used to identify a performance bottleneck within the task scheduling algorithm that was preventing scalability to large numbers of cores for some problems. However, we have been unable to use TAU on very large problems on large numbers of cores due to large overheads in both memory and processing time. In particular, on some problems we would be unable to run at large scales due to lack of memory. Similar problems with overhead on have been reported in [25]. These overheads have necessitated different methods to identify these inefficiencies.

One method that proved to be successful in Uintah was to look at memory allocation. By looking at memory allocation between strong scaling runs we were able to quickly identify portions of the code that did not strong scale in terms of memory usage. Poor scalability within memory utilization can indicate poor scalability in application time. For example, if memory allocation is proportional to the number of processors then at minimum a component of the runtime must also be proportional to the number of processors.

The need for this type of analysis has led to the development of MallocTrace [26]. MallocTrace is a low-memory-overhead tool for tracking memory usage within an application. This tool logs memory allocations in C++ programs through a series of macros and library hooks. The logs contain information including the file name and line number where the allocation occurred which is useful for tracking memory usage. The tool also provides a basic mechanism to parse the logs and provides a summary of memory usage at any given time in the simulation. This allows a user to see where memory is allocated at any point in the simulation. The low memory overhead allows the tool to be used with a large number of cores on programs like Uintah.

This tool has allowed us to rapidly identify and eliminate inefficiencies in memory usage at large numbers of cores. The effect of eliminating these inefficiencies can be seen in Figure 1. This graph shows an algorithmic decrease in memory usage. Prior to the optimizations the memory usage would sometimes increase with the number of cores. The same increase was not seen after the optimizations.

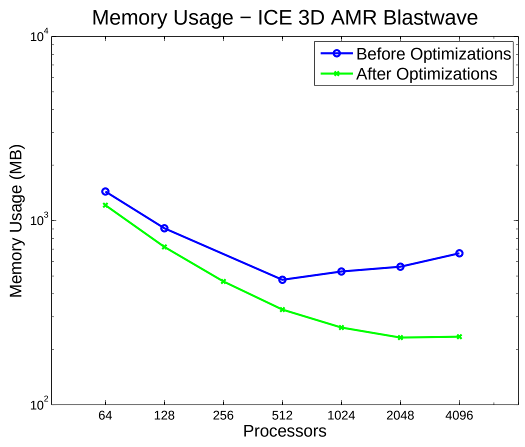

Fig. 1. A comparison of the memory usage of Uintah before and after the elimination of inefficiencies identified with MallocTrace.

Figure 2 shows the corresponding increase in scalability due to the elimination of inefficiencies identified with MallocTrace.

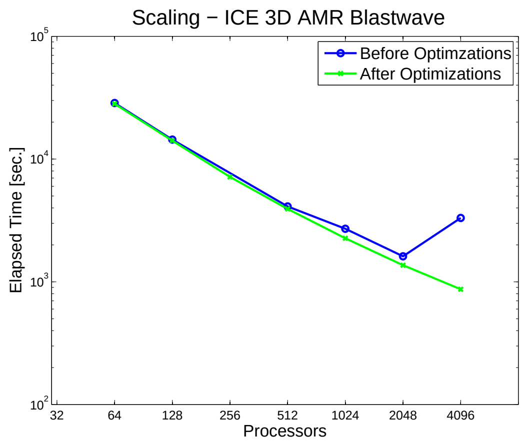

Fig. 2. A comparison of the scalability of Uintah before and after the elimination of inefficiencies identified with MallocTrace.

By analyzing the memory utilization of Uintah we were able quickly identify and eliminate memory inefficiencies leading to significant increases in performance. It took less than a week from the deployment of this tool to achieve the results shown here. Inefficiencies like those found in Uintah are not uncommon in large legacy codes. Undoubtedly similar issues will exist when moving to hundreds of thousands of cores. Determining which portions of Uintah need to be redesigned is a challenging task that is made easier with tools that are capable of running on large-scale problems like MallocTrace, see [26].

IV. Dynamic Load Balancing

The variety of available simulation components within Uintah necessitates a sophisticated load balancer that is flexible enough to handle all of Uintah simulations. Dynamic load balancing can be described as the minimization of three competing costs: The cost of load imbalance, the cost of communication, and the cost to generate the load distribution.

A load imbalance occurs when one or more cores are assigned more work than other cores. A large load imbalance will cause cores to wait for other cores to finish their computation leading to poor utilization of system resources.

In addition, too much communication can also cause performance issues. Communication across the network is slow relative to the time for computation and can easily dominate the time to reach a solution. In many simulations, communication is predominantly local, meaning that only a small area around each patch must be communicated from physically neighboring patches. By clustering neighboring patches together the framework can greatly reduce the necessary communication and significantly affect the overall runtime.

Finally, with AMR methods the workload changes as the mesh changes. In addition, with particle methods the workload can change on each time step as particles move throughout the domain. This can necessitate that load balancing occurs often, making it important that the time to generate the patch distribution is small relative to the overall computation. If a slow load balancing algorithm is used and load balancing occurs often, the time to load balance can dominate the overall runtime. In this case, it may be preferable to use a faster load balancing algorithm that produces more load imbalance.

The need for fast and effective load balancing techniques has led to the development of widely used load balancing applications like Metis [27], Jostle [28], and Zoltan [29], [30]. Uintah has recently added support for the Zoltan load balancing package, providing easy access to a number of algorithms. In addition, Uintah can use its own highly parallel load balancing algorithm [31] that utilizes space-filling curves, which has been shown to be better than Zoltan's space-filling curve load balancer within Uintah [32].

To balance the computation effectively load balancers need an estimate of the cost (execution time) of the computation. A poor estimate of the cost will lead to a decrease in load balance. Thus it is important that this estimate be accurate. One method to estimate these costs is to use algorithmic cost models.

A. Algorithmic Cost Models

Algorithm cost models (ACM) attempt to model the underlying algorithms. For example, the ICE algorithm is a cell-based algorithm that performs a constant amount of work per cell and as such the cost is proportional to the number of cells. Equation (1) below, describes an accurate ACM for ICE, where C_p is the cost of a patch, N_c is the number of cells in that patch, and c_1 is the constant time execution time the ICE algorithm on a single cell.

$$C_p = c_1 N_c \tag{1}$$

In addition to performing cell-based computations, MPMICE also has particle-based computations in regions where solid materials exist. Equation 2 below describes a possible ACM for MPMICE, where N_p is the number of particles within the patch and c_2 is the constant execution time on a particle.

$$C_p = c_1 N_c + c_2 N_p \tag{2}$$

This model is not as accurate as the ICE model because the work performed by MPMICE is not constant per particle or cell. In MPMICE, during the equilibration pressure solve, the simulation performs a local iterative solve on a per-cell basis [18]. This solve may converge at different rates throughout the domain depending on the underlying physics. Capturing such behavior in an ACM is a challenging task.

In addition to developing these models, estimates for the constants must be determined on a per-problem basis. The constants can vary greatly depending on the underlying physical processes. To make matters worse these constants can also vary according to system architectures, compilers, and compiler optimization flags. In order to achieve an effective load balance, these constants must be proportionally accurate. For models with a single constant, like the model used for ICE, estimating the constant is trivial. However, the difficulty in estimating these constants increases significantly with the number of constants in the model. Maintaining an accurate list of these constants for each possible problem, architecture, and compiler combination is not feasible, thus placing the challenge of estimating these constants on the user.

B. Forecasting Cost Model

Since developing an accurate algorithmic cost model is challenging, we have added an alternative approach to Uintah which utilizes a forecasting cost model (FCM) to predict the cost of each patch based on time series. During task execution, the time to complete each task on a region of the domain is recorded and used to update a simple forecasting model. That model is then used to predict the execution time on that region in the future. This provides a mechanism to accurately predict the cost of each patch and eliminates the need to estimate constants for an ACM. Uintah uses simple exponential smoothing, which is also known as a fading memory filter, as its forecasting model [33]. The model is as follows:

$$W_{r,t+1} = \alpha E_{r,t} + (1-\alpha) W_{r,t}, \quad (3)$$

where $W_{r,t}$ is the predicted cost at time step t on region r, $E_{r,t}$ is the actual execution time at time step t on region r, and α is a weighting factor in the range of [0,1] which represents the rate of decay on past data. This method can also be viewed as a weighted moving average where the weight on past observations decreases exponentially [33]. A smaller value for α causes the algorithm to put more weight on recent observations causing the forecast to respond more quickly to changes in the actual value but also causes the forecast to become more susceptible to noise. A larger value for α will cause that data to be smoother eliminating noise but also causes the forecast to react more slowly to changes in the actual value. α can be defined in terms of the size of a moving average window using the following equation:

$$\alpha = \frac{2.0}{T+1}, \quad (4)$$

where T is the number of time steps that will contain 99.9% of the total weight in the weighted average [33]. Uintah uses a default value of 10 for T.

On the first time step of the simulation $W_{r,t}$ is unavailable, requiring an estimation of the initial value. For the initial time step Uintah load balances using the algorithmic cost models described above. The initial measurements are then used to set the initial value by setting $W_{r,0} = E_{r,0}$. Doing this helps the forecast to rapidly converge.

A different initialization approach is used when new regions of refinement are created during the regridding process. During this process refinement may be added in regions where it previously did not exist. When these regions are created, $W_{r,t}$ must be estimated. Using an ACM would likely produce a poor estimate that would not be proportionally accurate to the forecasted values elsewhere in the domain. In order to estimate the cost while maintaining proportional accuracy, Uintah sets $W_{r,t}$ for the new regions equal to the average value of $W_{r,t}$ for all regions. This ensures that the initial value for the new region is at least close to the actual value which also ensures that the estimation will be accurate within a few time steps and that load imbalance caused by this estimation will be limited.

To allow for changing patch sets forecasting is performed on a per-region basis instead of a per-patch basis. The difference between regions and patches is shown in Figure 3. Regions are constant-sized portions of the domain that are contained within a single patch. Patches on the other hand are variable-sized portions of the domain that may contain many regions.

By forecasting on a per-region basis, the patch set can change without needing to migrate forecasting data between the changing patch sets. This necessitates mechanisms to interpolate the data between regions and patches. Since regions

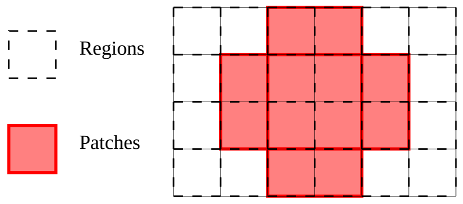

Fig. 3. The difference between regions and patches. Patches are composed of one or more fixed size blocks referred to as regions.

are completely contained within patches these mechanisms are straight forward to describe and implement. The measured cost for each region is equal to the cost of the patch, times the proportion of the patch that the region encompasses, as described in the following equation:

$$E_{r,t} = E_{p,t} \frac{V_r}{V_p}, \quad (5)$$

where t is the current time step, $E_{r,t}$ is the measured computation time for region r, $E_{p,t}$ is the measured execution time for patch p, V_p is the volume of patch p, and V_r is the volume of region r. In addition, $W_{p,t+1}$ can be defined as the sum of the weights of all regions contained in patch p:

$$W_{p,t+1} = \sum_{r}^{r \in p} (W_{r,t+1}). \quad (6)$$

Uintah stores the forecasting data while minimizing both storage and communication. The forecasting data is stored locally on each core. When a core executes a task on a patch, it adds the contribution to its local forecast data using Equation (5). If a region was owned by a different core in the past, then local forecast data will exist on multiple cores but each core will only update its local data. At the end of each time step the simulation finalizes the forecast data by applying Equation (3). Updating the forecast data each time step is a local operation which does not require any communication. However, communication is required when load balancing occurs. During load balancing, each core must know the cost of each patch. This is done by applying Equation (6) locally and then performing a MPI_Allreduce to get the global sum.

In order to keep the data structures for forecasting as small as possible, contributions are stored in a Standard Template Library (STL) map, which is a sparse data structure. This causes the storage per core to be proportional to the number of patches per core. In addition, when a core has not updated a region in its map for over T time steps, the contributing weight for that region is less than 0.1% of the total weight, at this point we consider the weight to be insignificant and delete it from the map. This prevents the size of the maps from slowly increasing over time.

C. Forecasting Results

The effect of forecasting on Uintah's runtime was tested using two different simulation components. The first test used the ICE, multi-material algorithm with explicit time stepping to simulate the transport of two fluids with a prescribed initial velocity. For this problem the conservation of mass, momentum, and energy equations are solved for two inviscid fluids. The fluids exchange momentum and heat through the exchange terms in the governing equations. This problem exercises all of main features of ICE and amounts to solving eight P.D.E's, along with two point-wise solves, and one iterative solve for more information see [18].

The second simulation was C-SAFE's target problem using the MPMICE algorithm with explicit time stepping seen in Section I. In this problem, a steel container filled with an explosive material is suspended over a fire. As the simulation progresses, the explosive heats up and ignites causing the container to rupture resulting in a violent explosion. This is a complex problem whose computational cost is difficult to predict. As the container ruptures, the performance characteristics of the problem rapidly change as pressurized gasses and explosive materials move across the domain. These problems were selected because of the complexity of the relevant physics.

The computational cost of the ICE problem is predictable and developing an accurate ACM is straight forward. In contrast, the computational cost of the MPMICE problem is difficult to predict, hindering the creation of an accurate ACM. For the ICE simulation, Equation (1) above was used for the ACM with $c_1 = 1$. For the MPMICE simulation, Equation (2) above was used for the ACM with $c_1 = 1$ and $c_2 = 1.25$. While these values are not representative of actual machine constants, they are proportionally accurate, which is sufficient for the load balancing process.

Figure 4 shows the difference in load imbalance for the ICE simulation using a FCM versus an ACM. The load imbalance varies between 3% and 10% with an average imbalance of 5.3% using the FCM and 6% using the ACM. In this case the difference in runtime was marginal. This shows that Uintah's performance using a FCM is similar to performance using an accurate ACM.

The load imbalance for the MPMICE simulation can be seen in Figure 5. When using an ACM, the load imbalance varies between 13%-35% with an average imbalance of 20%. This variance is due to rapid changes in the performance characteristics that are not captured by the current model. At the same time the load imbalance when forecasting was relatively constant with an average imbalance of 4%.

The improved load balance led to a substantial increase in performance seen in Figure 6. When forecasting, the MPMICE simulation was approximately 15% faster than when using the ACM.

These results show that forecasting can produce accurate cost estimations that are at least as effective as an accurate ACM. In addition, forecasting is able to predict complex interactions that may be difficult to capture in an ACM leading to improved cost estimations and reduced runtimes. Higher order forecasting methods described in [33] have also been used but provided no benefit over the first order forecasting

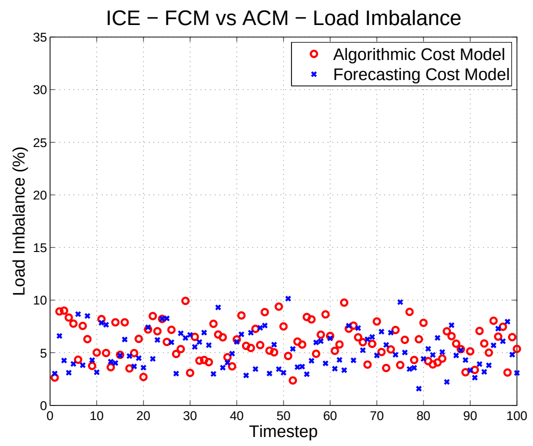

Fig. 4. A comparison of the load imbalance when using an ACM versus a FCM for ICE. This figure shows there is little difference between the ACM and the FCM.

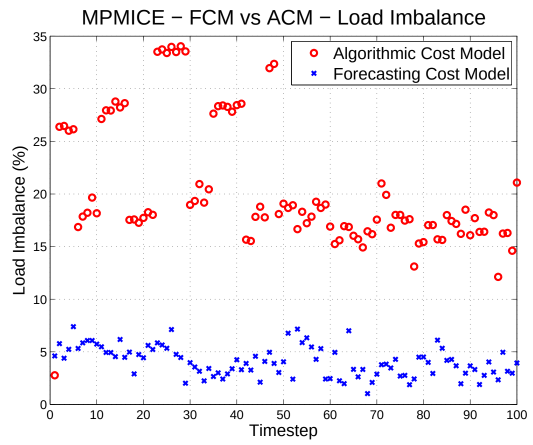

Fig. 5. A comparison of the load imbalance when using an ACM versus a FCM for MPMICE. This figure shows that using the FCM provides a much lower load imbalance than an ACM.

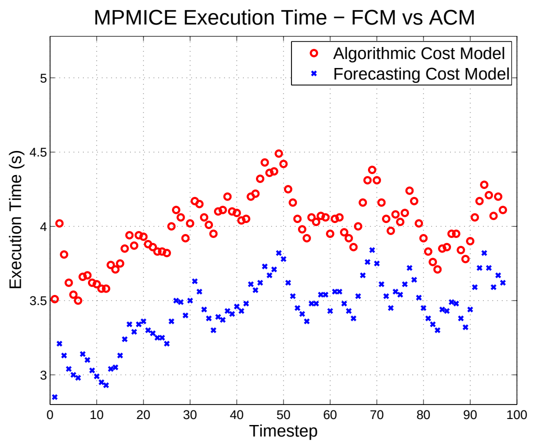

Fig. 6. A comparison of the time spent in task execution in MPMICE using an ACM versus a FCM. This figure shows that the using the FCM led to a large improvement in overall runtime.

methods described above. In the future, it may be worthwhile to use a Kalman filter [34], [35] or other advanced forecasting methods.

V. Scalability of AMR in Uintah

The scalability of Uintah's AMR infrastructure was tested in both the weak and strong sense using both the ICE problem described in section IV-C and a similar MPMICE problem that transports a solid explosive through air at mach 2. The MPMICE has the added complexity of representing the solid with particles. These problems were chosen because the location of refinement rapidly changes but the total size of the grid remains fairly constant allowing the scalability to be accurately measured. The tests were ran on Kraken[1]. For **weak scaling** the problem size per core was held constant as the number of cores increases and in **strong scaling** the total problem size was held constant while the number of cores increases.

For the ICE scaling test the time to complete 50 timesteps of a full AMR simulation was recorded. This problem contained three mesh levels with each level being a factor of four more refined than the coarser level. Patches were uniformly sized with 16^3 cells in each patch. Regridding and load balancing were performed as needed and occurred around 5 times in each problem. The performance was tested for five problem sizes with each problem size containing approximately four times as many cells as the previous problem. The smallest problem contained 1.7 million cells and the largest problem contained 435 million cells.

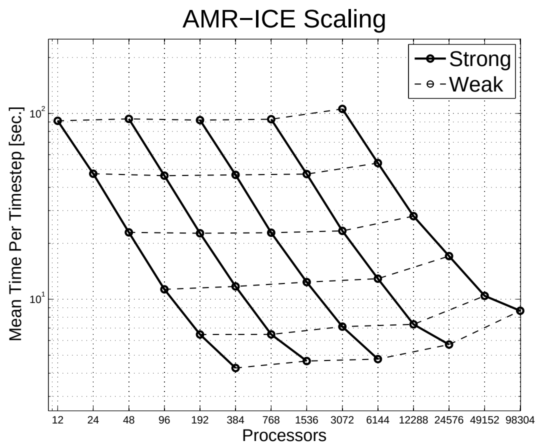

Fig. 7. The strong and weak scalability up to 98,304 cores of AMR in Uintah using ICE.

The comprehensive weak and strong scaling up to 98,304 cores for ICE can be seen in Figure 7 and the corresponding data can be found in Table I. The elimination of efficiencies within Uintah and improvements to the load balancer have led to marked improvements to Uintah's scalability. Good strong

ICE Scaling Data (s)					
	Strong 1	Strong 2	Strong 3	Strong 4	Strong 5
Weak 1	91.26	93.48	92.04	93.05	105.76
Weak 2	47.35	46.24	46.71	47.18	54.08
Weak 3	22.91	22.65	22.81	23.33	28.01
Weak 4	11.30	11.72	12.37	12.90	17.06
Weak 5	6.46	6.47	7.12	7.32	10.43
Weak 6	4.26	4.65	4.76	5.70	8.67

TABLE I
THE WEAK AND STRONG SCALABILITY DATA SHOWN IN FIGURE 7.

scaling occurred for every problem size tested. In each test scaling occurred down to approximately one patch per core. Decent weak scaling also occurred for each test though the scaling was not ideal.

For the MPMICE test the time to complete 100 timesteps of the MPMICE simulation using full AMR was recorded. This problem contained three mesh levels with each level being a factor of four more refined than the coarser level. Patches were uniformly sized with 8^3 cells in each patch. Regridding and load balancing were performed as needed and occurred 2 times in each problem. The performance was tested for three problem sizes with each problem size containing approximately eight times as many cells and particles as the previous problem. The smallest problem contained approximately 450 thousand cells and 1.07 million particles and the largest problem contained 11.6 million cells and 68.8 million particles.

The comprehensive weak and strong scaling up to 98,304 cores for MPMICE can be seen in Figure 8 and the corresponding data can be found in Table II. This figure shows decent strong scalability for each problem size tested. At smaller amounts of work per core the weak scaling was not ideal and showed an increase in runtime as the processors increased.

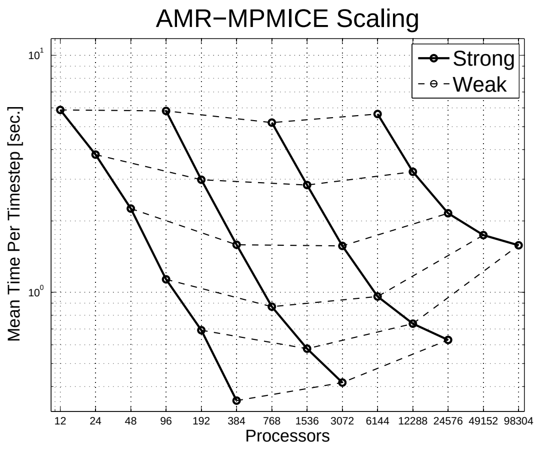

Fig. 8. The strong and weak scalability up to 98,304 cores of AMR in Uintah using MPMICE.

In order to show where Uintah's performance can be further improved a breakdown of the scaling results above will now be presented. Figures 9 and 10 shows a break down of Uintah's

[1]Kraken is a supercomputer located at the University of Tennessee with 99,072 cores. More information is available at http://www.nics.tennessee.edu/computing-resources/kraken.

MPMICE Scaling Data (s)				
	Strong 1	Strong 2	Strong 3	Strong 4
Weak 1	5.88	5.82	5.20	5.65
Weak 2	3.81	2.98	2.83	3.22
Weak 3	2.26	1.59	1.57	2.15
Weak 4	1.13	0.87	0.96	1.74
Weak 5	0.69	0.58	0.74	1.58
Weak 6	0.35	0.42	0.63	–

TABLE II
THE WEAK AND STRONG SCALABILITY DATA SHOWN IN FIGURE 8.

strong and weak scalability using ICE. In this figure the narrow bar represents the maximum time across all cores and the wide bar represents the average time, with the difference between those two bars representing the load imbalance. The black line represents the total time, which is approximately equal to the sum of the average times. This figure breaks down the simulation timings into five categories. The execution time is the time spent executing tasks, the global communication time is the time within collective MPI operations like MPI_Allreduce, the local communication is the time spent posting MPI messages, the AMR time is the time spent load balancing, regridding, and scheduling, and the wait time is the time spent within MPI_Wait. The wait time is a combination of time spent waiting for other tasks to complete so that communication can be sent (time for synchronization) and waiting for the communication to occur (time for communication).

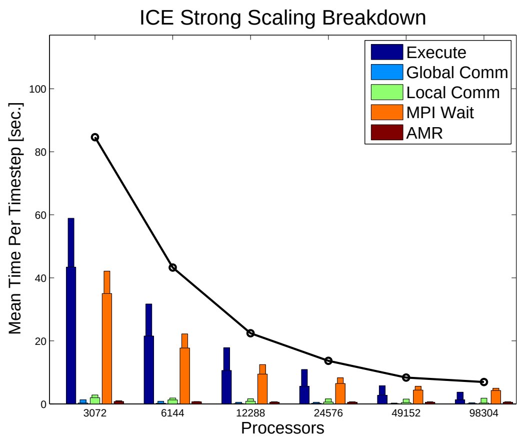

Fig. 9. A break down of the ICE strong scaling in Uintah. The thick bar is the average time per core and the narrow bar is the maximum time across all cores.

These figures shows a large amount of time is spent within task execution and MPI_Wait. A majority of the MPI_Wait time is due to synchronization. The wait and execution times scale in the strong sense but as the number of processors increases the wait time becomes dominate. When weak scaling the load imbalance in the execution time increases with the number of processors which also increases the wait time. Reducing the load imbalance for the large problems would significantly improve the weak scalability of Uintah. In addition, advanced scheduling algorithms may be able to lower the wait

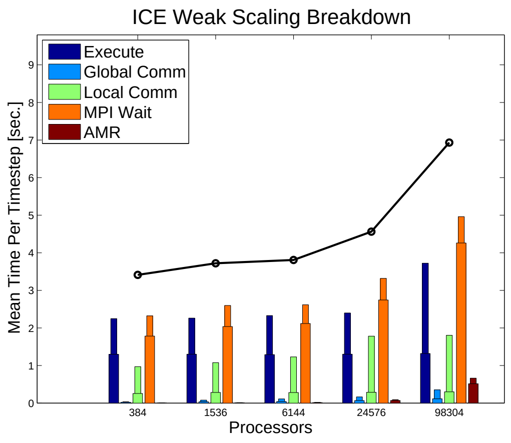

Fig. 10. A break down of the ICE weak scaling in Uintah. The thick bar is the average time per core and the narrow bar is the maximum time across all cores.

time and increase scalability. Research into better scheduling and load balancing algorithms is currently being undertaken. Finally this figure also shows that the time spent performing AMR functions increases rapidly at the last data point. For weak scaling to much larger problems it is likely that portions of the AMR infrastructure will need further improvements.

Figures 11 and 12 show the breakdown of the MPMICE scaling. The strong scaling breakdown shows the wait time becomes dominate eventually limiting the scalability. The weak scaling breakdown shows a large increase in the wait time which is due to increases in the load imbalance in the local communication and execution time. This increase is likely due to either synchronization issues or inefficiencies in the MPI library. The source of theses increase is currently being investigated.

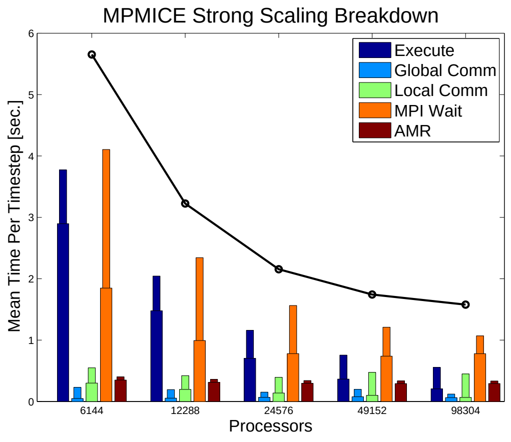

Fig. 11. A break down of the MPMICE strong scaling in Uintah. The thick bar is the average time per core and the narrow bar is the maximum time across all cores.

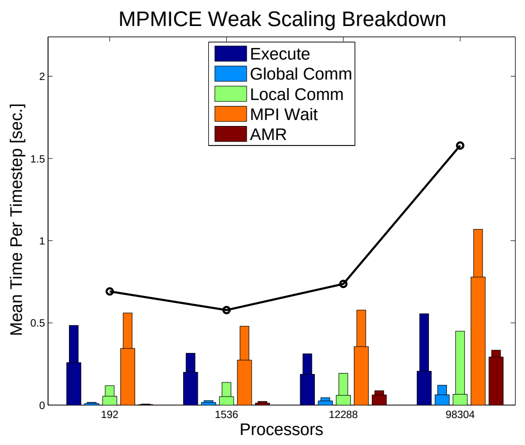

Fig. 12. A break down of the MPMICE weak scaling in Uintah. The thick bar is the average time per core and the narrow bar is the maximum time across all cores.

VI. Conclusions And Future Work

The primary strength of Uintah is that simulation designers can develop large-scale parallel AMR simulations with little understanding of the underlying parallelism. This allows for rapid development of large-scale simulations for a wide variety of problems using Uintah with features like, automated load balancing, parallel I/O, and checkpointing. Uintah's component design allows for the use of sophisticated algorithms without burdening users by complicating the individual components.

In preparation for emerging petascale architectures, it is important that the performance of frameworks like Uintah are analyzed for inefficiencies. Poor performance in any portion of the framework can hinder performance for the entire simulation. Because of this we have placed substantial effort into identifying, analyzing, and eliminating inefficiencies within Uintah using tools like TAU and MallocTrace. By analyzing Uintah's memory usage at large numbers of cores with MallocTrace, we were able to rapidly identify and eliminate multiple inefficiencies. This led to a substantial decrease in memory usage and corresponding increase in performance.

The effect of load balance on the overall performance of a simulation is substantial. A poor load balance will cause poor utilization of system resources preventing scalability. Because of this it is essential that we use effective load balancing algorithms. However, these algorithms will only be effective if the cost estimates provided to them are accurate. Poor cost estimates will cause a poor load balance regardless of what load balancing algorithm is used. While algorithmic cost models can be used to produce these estimates, they are often prone to large error and require the user to estimate model constants. We have shown that an alternative method for estimating these costs eliminates the constants while still providing an accurate estimate that is at least as effective as, and in many cases better than, algorithmic cost models. In the future, more sophisticated forecasting methods could be used to further improve estimates allowing for a greater utilization of system resources. We are currently working on automatically estimating the ACM constants using least squares methods in addition to utilizing Kalman filters [34], [35].

The scalability shown in this paper is due to work performed by the C-SAFE team over the last five years that has focused on algorithm and data structure design and efficiency [5], [31], [32]. In particular improvements to the memory utilization and load balancer have led to significant improvements to the overall scalability, which has been shown in both strong and weak sense for AMR problems within Uintah on up to 98,304 cores on Kraken. We expect even greater scalability when larger machines are made available.

VII. Acknowledgements

This work was supported by the University of Utah's Center for the Simulation of Accidental Fires and Explosions (C-SAFE) and funded by both the Department of Energy, under subcontract No. B524196 and the National Science Foundation under subcontract No. OCI0721659. We would like to thank TACC and NICS for computer access along with Jim Guilkey, Todd Harman, Qingyu Meng, Tom Henderson, and the rest of the C-SAFE team.

References

[1] J. D. de St. Germain, S. G. Parker, J. McCorquodale, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," in *HPDC*, 2000, pp. 33–42.

[2] M. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics." *Journal of Computational Physics*, vol. 82, pp. 64–84, 1989.

[3] S. G. Parker, J. Guilkey, and T. Harman, "A component-based parallel infrastructure for the simulation of fluid structure interaction," *Engineering with Computers*, vol. 22, no. 3-4, pp. 277–292, 2006.

[4] S. G. Parker, "A component-based architecture for parallel multi-physics pde simulation," *Future Gener. Comput. Syst.*, vol. 22, no. 1, pp. 204–216, 2006.

[5] J. Luitjens, B. Worthen, M. Berzins, and T. Henderson, *Petascale Computing Algorithms and Applications*. Chapman and Hall/CRC, 2007, ch. Scalable parallel amr for the uintah multiphysics code.

[6] A. M. Wissink, R. D. Hornung, S. R. Kohn, S. S. Smith, and N. Elliott, "Large scale parallel structured amr calculations using the samrai framework," in *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM Press, 2001, pp. 6–6.

[7] A. M. Wissink, D. Hysom, and R. D. Hornung, "Enhancing scalability of parallel structured amr calculations," in *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2003, pp. 336–347.

[8] C. Burstedde, O. Gattas, G. Stadler, T. Tu, and L. C. Wilcox, "Twoards adaptive mesh pde simulations on petascale computers," in *TeraGrid 08*, 2008.

[9] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L. C. Wilcox, and S. Zhong, "Scalable adaptive mantle convection simulation on petascale supercomputers," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–15.

[10] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, B. J. Whitlock, and N. Max, "A contract-based system for large data visualization," in *Proceedings of IEEE Visualization 2005*, 2005, pp. 190–198.

[11] J. Guilkey, J. Hoying, and J. Weiss, "Modeling of multicellular constructs with the material point method," *Journal of Biomechanics*, vol. 39, pp. 2074–2086, 2007.

[12] A. Brydon, S. Bardenhagen, E. Miller, and G. Seidler, “Simulation of the densification of real open–celled foam microstructures.” *J. Mech. Phys. Solids*, vol. 53, pp. 2638–2660, 2005.

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

[14] B. Kashiwa, M. Lewis, and T. Wilson, “Fluid-structure interaction modeling,” Los Alamos National Laboratory, Los Alamos, Tech. Rep. LA-13111-PR, 1996.

[15] B. Kashiwa, “A multifield model and method for fluid-structure interaction dynamics,” Los Alamos National Laboratory, Los Alamos, Tech. Rep. LA-UR-01-1136, 2001.

[16] B. Kashiwa and E. Gaffney, “Design basis for cfdlib.” Los Alamos National Laboratory, Los Alamos, Tech. Rep. LA-UR-03-1295, 2003.

[17] D. Sulsky, Z. Chen, and H. Schreyer, “A particle method for history dependent materials,” *Comput. Methods Appl. Mech. Engrg.*, vol. 118, pp. 179–196, 1994.

[18] J. Guilkey, T. Harman, and B. Banerjee, “An eulerian-lagrangian approach for simulating explosions of energetic devices,” *Computers and Structures*, vol. 85, pp. 660–674, 2007.

[19] Y. Guo and J. Nairn, “Calculation of j-integral and stress intensity factors using the material point method.” *Computer Modeling in Engineering and Sciences*, vol. 6, pp. 295–308, 2004.

[20] D. Sulsky, S. Zhou, and H. Schreyer, “Application of a particle-in-cell method to solid mechanics,” *Computer Physics Communications*, vol. 87, pp. 236–252, 1995.

[21] J. Brackbill and H. Ruppel, “Flip: A low-dissipation, particle-in-cell method for fluid flows in two dimensions,” *J. Comp. Phys.*, vol. 65, pp. 314–343, 1986.

[22] ——, “Flip: A method for adaptively zoned, particle-in-cell calculations of fluid flow in two dimensions.” *Journal of Computational Physics*, vol. 65, pp. 314–343, 1986.

[23] J. Brackbill, “Particle methods,” *International Jour. Numer. Meths. in Fluids*, vol. 47, pp. 693–705, 2005.

[24] S. S. Shende and A. D. Malony, “The tau parallel performance system,” *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[25] K. Mohror and K. L. Karavanic, “Towards scalable event tracing for high end systems,” in *HPCC*, 2007, pp. 695–706.

[26] J. Luitjens, “Malloc trace users guide.” [Online]. Available: http://www.csafe.utah.edu/wiki/index.php/Documentation/UsersGuide/MallocTrace

[27] G. Karypis and V. Kumar, *MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*.

[28] C. Walshaw, M. Cross, M. G. Everett, and S. Johnson, “Jostle: Partitioning of unstructured meshes for massively parallel machines,” in *Parallel Computational Fluid Dynamics: New Algorithms and Applications*. Elsevier, 1994.

[29] K. Devine, B. Hendrickson, E. Boman, M. S. John, and C. Vaughan, “Design of dynamic load-balancing tools for parallel applications,” in *ICS ’00: Proceedings of the 14th international conference on Supercomputing*. New York, NY, USA: ACM, 2000, pp. 110–118.

[30] E. Boman, K. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, C. Vaughan, U. Catalyurek, D. Bozdag, W. Mitchell, and J. Teresco, *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User’s Guide*, Sandia National Laboratories, Albuquerque, NM, 2007, tech. Report SAND2007-4748W.

[31] J. Luitjens, M. Berzins, and T. Henderson, “Parallel space-filling curve generation through sorting: Research articles,” *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 10, pp. 1387–1402, 2007.

[32] Q. Meng, J. Luitjens, and M. Berzins, “A comparison of load balancing algorithms for amr in uintah,” University of Utah, SCI Technical Report UUSCI-2008-006, 2008.

[33] D. C. Montgomery, L. A. Johnson, and J. S. Gardiner, *Forecasting and Time Series Analysis*, 2nd ed. Koga: Mcgraw-Hill, 1990.

[34] R. E. Kalman, “A new approach to linear filtering and prediction problems,” *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[35] P. Zarchan and H. Musoff, *Fundamentals of Kalman Filtering: A Practical Approach (Progress in Astronautics and Aeronautics)*. AIAA (American Institute of Aeronautics & Ast, December 2000.