Scalable Large-scale Fluid-structure Interaction Solvers in the Uintah Framework via Hybrid Task-based Parallelism Algorithms

Qingyu Meng, Martin Berzins

UUSCI-2012-004

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

May 7, 2012

Abstract:

Uintah is a software framework that provides an environment for solving fluid-structure interaction problems on structured adaptive grids on large-scale science and engineering problems involving the solution of partial differential equations. Uintah uses a combination of fluid-flow solvers and particle-based methods for solids, together with adaptive meshing and a novel asynchronous task-based approach with fully automated load balancing. When applying Uintah to fluid-structure interaction problems with mesh refinement, the combination of adaptive meshing and the movement of structures through space present a formidable challenge in terms of achieving scalability on large-scale parallel computers. With core counts per socket continuing to grow along with the prospect of less memory per core, adopting a model that uses MPI to communicate between nodes and a shared memory model on-node is one approach to achieve scalability at full machine capacity on current and emerging large-scale systems. For this approach to be successful, it is necessary to design data-structures that large numbers of cores can simultaneously access without contention. These data structures and algorithms must also be designed to avoid the overhead involved with locks and other synchronization primitives when running on large number of cores per node, as contention for acquiring locks quickly becomes untenable. This scalability challenge is addressed here for Uintah, by the development of new hybrid runtime and scheduling algorithms combined with novel lockfree data structures, making it possible for Uintah to achieve excellent scalability for a challenging fluid-structure problem with mesh refinement on as many as 260K cores.

Scalable Large-scale Fluid-structure Interaction Solvers in the Uintah Framework via Hybrid Task-based Parallelism Algorithms

Qingyu Meng, Martin Berzins
{qymeng, mb}@cs.utah.edu
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, Utah 84112, USA

May 7, 2012

Abstract

Uintah is a software framework that provides an environment for solving fluid-structure interaction problems on structured adaptive grids on large-scale science and engineering problems involving the solution of partial differential equations. Uintah uses a combination of fluid-flow solvers and particle-based methods for solids, together with adaptive meshing and a novel asynchronous task-based approach with fully automated load balancing. When applying Uintah to fluid-structure interaction problems with mesh refinement, the combination of adaptive meshing and the movement of structures through space present a formidable challenge in terms of achieving scalability on large-scale parallel computers. With core counts per socket continuing to grow along with the prospect of less memory per core, adopting a model that uses MPI to communicate between nodes and a shared memory model on-node is one approach to achieve scalability at full machine capacity on current and emerging large-scale systems. For this approach to be successful, it is necessary to design data-structures that large numbers of cores can simultaneously access without contention. These data structures and algorithms must also be designed to avoid the overhead involved with locks and other synchronization primitives when running on large number of cores per node, as contention for acquiring locks quickly becomes untenable. This scalability challenge is addressed here for Uintah, by the development of new hybrid runtime and scheduling algorithms combined with novel lock-free data structures, making it possible for Uintah to achieve excellent scalability for a challenging fluid-structure problem with mesh refinement on as many as 260K cores.

1 Introduction

A significant part of the challenge in moving from petascale to exascale problems is to ensure that multi-physics multi-scale codes that reflect real applications can be run in a scalable way on parallel computers with large core counts. The multi-scale challenge involved arises from the need to use approaches such as adaptive mesh refinement while the multi-physics challenge is typified by different physics being used in different parts of the domain with different computational loads in these different spatial domains. In addition it is necessary to couple these domains with the different physics and this coupling is an interesting challenge in itself as the coupling algorithm may be more complex than the algorithms used away from the interface. These challenges are further compounded by the anticipated relative memory per core potentially shrinking [1]. Furthermore the node architectures are becoming more complex with ever-increasing core counts and with the potential for the use of accelerators as part of the node [1], on machines under construction such as Titan [1] and Stampede. [2]

This work will start to address some of these challenges by developing scheduling algorithms and associated software that will be used to tackle fluid-structure interaction algorithms in which mesh refinement is used to resolve the structure. The vehicle for this work is the Uintah Computational Framework [16, 34, 35]. Many challenging Uintah applications and the software itself are described in [4]. The Uintah code was designed to solve fluid-structure interaction problems arising from a number of physical scenarios. Uintah uses a combination of computational fluid dynamics algorithms in its ICE solver [20, 24] and couples ICE to a particle-based solver for solids known as the Material Point Method (MPM) [39, 40]. The Adaptive Mesh Refinement (AMR) approach used in Uintah is that of multiple levels of regularly refined mesh patches. This mesh is also used as a scratch pad for the MPM calculation of the movement and deformation of the solid, [29, 30]. Uintah has been shown to scale successfully with many fluid and solid problems with adaptive meshes, [9, 33], however the scalability of fluid-structure interaction problems has proven to be somewhat more challenging. This is at least partly because the particles that represent the solid in Uintah can freely move across mesh patch boundaries in a way that is not known beforehand and partly because not all the domain is composed of the solid or fluid. There are two key components in the approach that will be described here to improve the scalability of fluid-structure interaction. The first component concerns access to data stored at the level of a node

[1]Titan is a parallel computer under construction at Oak Ridge National Laboratory with about 299K CPU cores now with a large number of attached GPUs to be added in 2012.

[2]Stampede is a parallel computer under construction at Texas Advanced Computing Center with 2 petaflops of CPU performance and 8 petaflops of accelerator performance.

in Uintah. In Uintah a multicore node is treated similarly to a miniature shared memory system, and only one copy of Uintah's global data needs to be stored per multicore node in the data warehouse that Uintah uses, thus saving a considerable amount of memory overall [33]. This approach also makes it possible to migrate particles across the cores in a node without using MPI. However for this approach to be successful, it is necessary to design a data warehouse that large numbers of cores can simultaneously access without contention.

The second component concerns how the cores themselves request work. In the model proposed by the authors [33], a single centralized controller allocates work to the cores. This approach has worked well on the Kraken [3] and Jaguar XT5 [4] architectures. This approach breaks down when there are more cores per node, and when communications are faster, as on the new Jaguar XK6 machine that presently consists of the CPU part of the proposed Titan machine. The solution, as will be shown, is to move to a distributed approach in which the cores themselves are able to request work.

Both approaches will be shown to make a substantial improvement to the scalability of fluid-structure interaction problems. This improvement in scalability will be demonstrated by using a challenging problem that consists of the motion of a solid through a liquid that is modeled by the compressible Navier-Stokes equations. The solid is modeled by particles on the finest part of an adaptive mesh.

The challenge addressed here is similar to that addressed in the PRONTO work [2, 5] for the solution of contact problems in which the contact algorithm requires more work than takes place in the rest of the domain. There are many examples of other parallel fluid-structure interaction work [13, 14, 17, 21] but the approach adopted here is somewhat different to many as it relies heavily upon the asynchronous nature of Uintah.

In the remainder of this paper the above challenges are addressed as follows: Section 2 describes the Uintah software and the adaptive fluid-structure methodology used. In Section 3 the details of the fluid-structure algorithm are described. Section 4 describes the existing scalability of Uintah on fixed and adaptive meshes. The challenge of scaling fluid-structure algorithms is explained in Section 5 in the context of a challenging model problem. Section 6 describes the different approaches that may be used in scheduling tasks, while Section 7 shows how to improve the data warehouse to avoid contention when access is attempted by multiple cores. Section 8 shows that these improvements very much improve the scalability of the model problem on the new Jaguar XK6 architecture up to 260K cores.

[3]Kraken is an NSF Cray XT5 computer located at NICS, Oak Ridge, Tennessee with 112K compute cores and a peak performance of 1.17 PF

[4]Jaguar was until 2012 the Cray XT5 system at Oak Ridge National Lab. with 224K cores and a peak performance of 2.33 petaflops

Section 9 provides some conclusions and areas for future work.

2 Uintah's Fluid-Structure Interaction Methodology

The design of the The Uintah Computational Framework was intended to make it possible to solve complex fluid-structure interaction problems on parallel computers. In particular Uintah is designed for *full physics* simulations of fluid-structure interactions involving large deformations and phase change. The term *full physics* refers to problems involving strong coupling between the fluid and solid phases with a full Navier-Stokes representation of fluid phase materials and the transient, nonlinear response of solid phase materials which may include chemical or phase transformation between the solid and fluid phases.

Uintah uses a full *multi-material* approach in which each material is given a continuum description and is defined over the complete computational domain. Although at any point in space the material composition is uniquely defined, the multi-material approach adopts a statistical viewpoint whereby the material (either fluid or solid) resides with some finite probability. To determine the probability of finding a particular material at a specified point in space, together with its current state (i.e., mass, momentum, energy), multi-material equations are used. The algorithm that uses a common framework to treat the coupled response of a collection of arbitrary materials is described below. This methodology follows the ideas of Kashiwa et al. [26, 27]. Individual equations of state are needed for each material to determine relationships between pressure, density, temperature and internal energy. Constitutive models are also required to describe the stress for each material based on appropriate input parameters (deformation, strain rate, history variables, etc.). In addition to those parameters, the multi-material nature of the equations also requires closure for the volume fraction of each material.

2.1 The ICE Multi-Material CFD Approach

In order to represent fluids in its multi-material CFD formulation, Uintah uses the ICE (Implicit, Continuous-fluid, Eulerian) method [23], further developed by Kashiwa and others at Los Alamos National Laboratory [28]. The use of a cell centered, finite volume approach is convenient for multi-material simulations in that a single control volume is used for all materials. This is particularly important in regions where a material volume goes to zero, as by using the same control volume for mass and momentum, if the material volume tends to zero, then the associated mass and momentum also similarly tend to zero. The technique is fully incompressible and compressible, allowing wide generality in the types of problems that can be simulated.

The Uintah implementation of the ICE technique uses operator splitting in which the solution consists of a separate Lagrangian phase where the physics of the conservation laws are computed and an Eulerian fluids phase in which the material state is transported via advection to the surrounding cells. The general solution approach is well-developed and described in [18, 20, 24, 42].

2.2 The Material Point Method

Solids in Uintah are represented by the particle method known as the Material Point Method (MPM) [39, 40]. MPM is a powerful technique for computational solid mechanics and has found favor in applications such as those involving complex geometries, large deformations and fractures, see [4] for these and many other examples. MPM is an extension to solid mechanics of FLIP [11], which is a particle-in-cell (PIC) method for fluid flow simulation [12]. Uintah also uses an implicit formulation of MPM [19]. In MPM Lagrangian particles or material points are used to discretize the volume of a solid material. Each particle carries state information (e.g. mass, volume, velocity, and stress) about the portion of the volume that it represents. The MPM method typically uses a cartesian grid as a computational scratchpad for computing spatial gradients. This grid may be arbitrary, and in Uintah it is the same grid used by the accompanying multi-material ICE CFD component. Particles are usually created on the finest level of the mesh and are always mapped back to the background grid according to their coordinates. The initial physical state of the solid is projected from the computational nodes to the cell centers collocating the solid material state with that of the fluid. This common reference frame is used for all physics that involve mass, momentum, or energy exchange among the materials. This results in a tight coupling between the fluid and solid phases. This coupling occurs through terms in the conservation equations, rather than explicitly through specified boundary conditions at interfaces between materials. Since a common multi-field reference frame is used for interactions among materials, typical problems with convergence and stability of solutions for separate domains communicating only through boundary conditions are alleviated. Considerable improvements in MPM and its analysis [36, 38, 41, 43] have resulted from work connected to the Uintah code.

3 Uintah Fluid-Structure Algorithm

The combination of MPM and ICE, MPMICE, is Uintah's fluid-structure interaction component. One of the challenges in multi-physics simulations is that the coupling algorithms involve calculations with each of the methods used in the domains that are coupled. Such calculations impose extra work in the coupling domain and also involve calls to the functions that implement the individual methods being coupled. In the Uintah coupling algorithm there are twelve steps. Some of these steps apply only to the fluid (as labeled by ICE), others apply only to the solid (as labeled by MPM), while other steps apply to both the fluid and the solid, (as labeled by MPMICE). If each phase of this algorithm is used as a synchronization point, then the parts of the domain not containing a multi-physics interface will be locked out while the interface calculation proceeds. The twelve steps used in the Uintah coupling algorithm are [24]:

1. Interpolate particle state to grid, MPM.
2. Compute the equilibrium pressure, ICE.
3. Compute face centered velocities for the Eulerian advection, ICE.
4. Compute sources of mass, momentum and energy as a result of phase changing chemical reactions, MPMICE.
5. Compute an estimate of the time advanced pressure, ICE.
6. Calculation of face centered pressure using a density weighted approach, ICE.
7. Material stresses calculation, MPM.
8. Compute Lagrangian phase quantities at cell centers, ICE.
9. Calculate momentum and heat exchange between materials, MPMICE.
10. Compute the evolution in specific volume due to the changes in temperature and pressure during the foregoing Lagrangian phase of the calculation, ICE.
11. Advect fluids for the fluid phase, ICE.
12. Advect solids for the solid phase, interpolate the time advanced grid velocity and the corresponding velocity increment back to the particles, and use these to advance the particle's position and velocity, respectively, MPM.

The difficulties associated with this algorithm from a parallel scalability point of view are twofold. The first difficulty is that the MPM work per patch depends on the number of particles per patch. This value constantly changes as particles enter or leave patches. The second difficulty is that as particles are not distributed throughout the domain, the work associated with MPM only takes place in an irregular and transient manner throughout the patch set.

4 Existing Uintah Scalability

In describing the existing scalability of Uintah there three topics that need to be covered.

4.1 Adaptive Mesh Refinement Algorithms, AMR

Before addressing fluid-structure interaction with AMR it is important to be sure that the adaptive meshing part of the algorithm scales independently. In the last 4 years, improvements within Uintah to the regridding and load balancing algorithms have led to a 40x increase in the scalability of AMR [29, 30]. Previously, Uintah used the well-known Berger-Rigoutsos algorithm [8] to perform regridding. However, at large core counts this algorithm performed poorly. The new regridder [30] defines a set of fixed-sized tiles throughout the domain. A search of each tile is then made for refinement flags without the need for communication and all tiles that contain refinement flags become patches. This regridder scales well at large core counts because cores only communicate once at the end of regridding when the patches are combined.

4.2 Load Balancing and Task Execution

When the simulation grid changes as a result of physics or load imbalance, a new grid and mesh patches are created and the Uintah framework automatically migrates the mesh patches to their new homes on the cores. This automation is possible as the framework manages all the simulation variable allocations. The load balancing algorithm used accurately predicts the amount of work a patch requires by using data assimilation and measurement techniques with feedback as this approach out-performs traditional cost models, [29]. In this way Uintah's load balancer determines a reasonable allocation of patches to nodes using measurement and geometric information. The load balancer attempts to guarantee that an equal amount of work is distributed to each core so as to allow for an optimal scaling of the simulation on multiple cores. Once the AMR regridder generates the hexahedral mesh patches of regular mesh cells a Bounding Volume Hierarchy (BVH) tree is built or updated to hold all the patches for fast lookup on each independent mesh level. The patches are uniquely assigned to cores by the load balancer. Once a local set of patches is known, each core can run computational tasks on its own patches and communicate with other cores through MPI messages automatically generated by Uintah. A significant amount of development has also been done on out-of-order execution of tasks, which has produced a significant performance benefit in lowering both the MPI wait time and the overall runtime [9, 32] on 98K cores and beyond.

4.3 Data Warehouse

Uintah's data warehouse is a hashed-map-based dictionary which maps variable names and mesh patch or level id to the memory address of a variable. Each task can get its read and write variables by querying the data warehouse with a variable name and a patch id. After each timestep, the old data warehouse is frozen and is set to be read-only. A new data warehouse to store newly computed variables in this timestep is then initialized. Variables can also be carried over from the old to the new data warehouse so as to avoid memory reallocation. Besides acting as a central dictionary for all variables, the data warehouse also manages MPI communications buffers. When a MPI message needs to be received (say for a variable coming from another core) a foreign variable will be allocated by the data warehouse and marked as invalid. An associated MPI buffer for this foreign variable will be provided to hold the incoming message. When the variable is received, the foreign variable will be marked as valid and ready for use by computational tasks. As all Uintah variables are allocated and accessed through the data warehouse, this warehouse must also keep track of the life of any variable. All the intermediate variables are de-allocated through a scrubbing process if no longer needed.

The message passing paradigm that Uintah initially operated under was that any data that needed to be communicated to a neighboring core was passed via MPI. For multi-core architectures the process of passing data that is local to a node by using MPI is both time consuming and duplicates identical data that can be shared between cores.

Uintah now stores only one copy of global data per multi-core node in its data warehouse. The task scheduler now spins off tasks to be executed on, say, nc cores using a threaded model, [33] which results in the global memory used in a single shared data warehouse being a fraction of nc^{-1} of what is required for multiple MPI tasks, one per each of the nc cores. This memory saving expands the scope and range of problems beyond those that it has been possible to explore until now [10, 33]. This approach is also being extended to spin-off tasks to be executed on other types of processors, such as GPUs. A working prototype that uses as many as 1000 GPUs has recently been tested on the TitanDev project at Oak Ridge as well as full capability jobs being run on the Keeneland GPU system at NICS, which has multiple GPUs per node, [22].

Through this research Uintah has been able to show strong and weak scalability up to 196K cores on DOE's Jaguar for ICE with AMR as seen in [9, 29, 30, 33]. Finally the use of Uintah with scalable linear solvers such as hypre has led to good weak scalability up to about 200K cores [37].

5 Challenges of scaling on Fluid-Structure Problems

The motivating fluid-structure interaction problem used here arises from the simulation of explosion of a small steel container filled with a solid explosive (PBX-9501). The explosive ignites and begins to burn, converting the solid into a high temperature gas which in turn causes the container to pressurize. As the pressure increases the container expands and eventually ruptures violently. The benchmark problem used for this scalability study is the transport of a small cube of steel container piece inside of the PBX product gas at an initial velocity of Mach two. The simulation used an explicit formulation with the lock-step time stepping algorithm which advances all levels simultaneously. This problem exercises all of the main features of ICE, MPM and AMR and amounts to solving eight partial differential equations, along with two point-wise solves, and one iterative solve [9, 29]. This benchmark also included a model for the deflagration of the explosive and the material damage model ViscoSCRAM [7]. The simulation utilized three refinement levels with each level being a factor of four more refined than the previous level. The solution to this problem is shown in Figure 1. This problem was originally

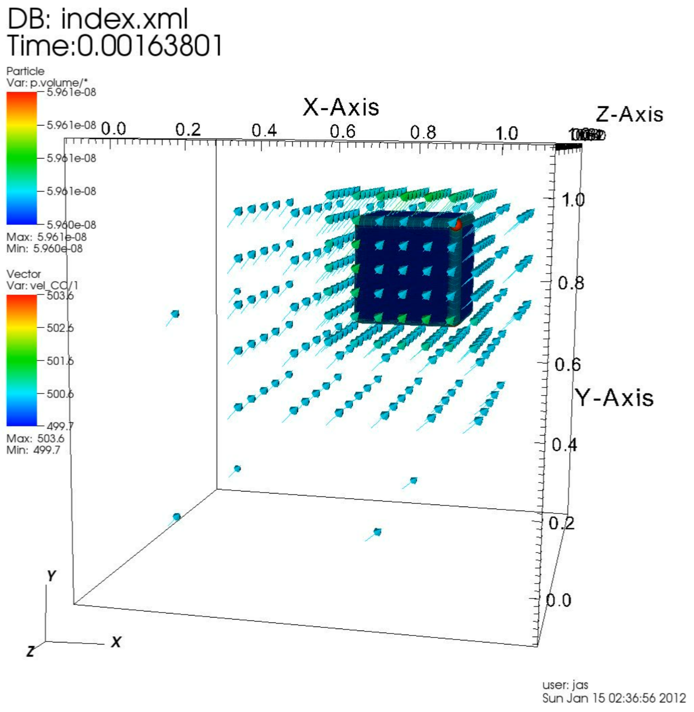

Figure 1: Solution of Fluid-Structure Interaction Problem.

run on NSF's Kraken system [31] with the scaling results shown in Figure 2. The benchmark involved four runs of varying size. Each run uses an initial coarse mesh with doubled resolution with respect to the previous run. The refinement criteria used in MPMICE refines the mesh anywhere that particles exist. This refinement criteria led to each problem being about eight times as large as the previous one. In the four cases shown the number of mesh cells were 619K, 3.8M, 21.3M and 152M respectively, while the number of MPM points used to represent the solid were 2.1M, 16.8M, 134M, and 1.1B in each of the strong scaling cases associated with the four solid lines. In the case of the run with the largest number of points, the rightmost solid line, the strong scaling clearly breaks down as the line turns up. In this case the code has only only 16% relative efficiency [31] at the final point with respect to the run in the top leftmost corner. In this example, as the solid

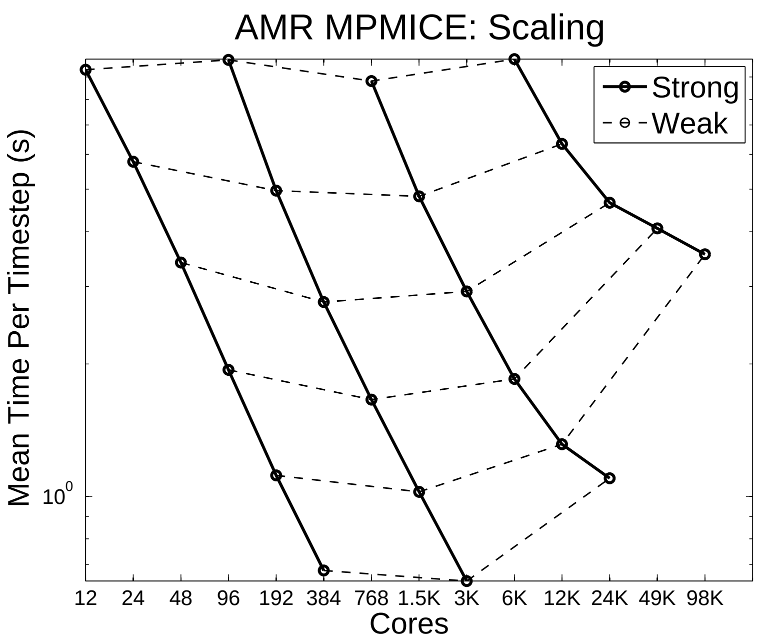

Figure 2: Initial poor scaling of Uintah on the fluid-structure interaction problem

moves through the domain, the number of particles in a certain area changes significantly. Figure 3 shows how the number of particles and execution time changes during the simulation at two different locations. Location A is at the front of the object while location B is at one of the edges of the object. Figure 3 shows that the computing cost of a patch with particles is about six times as great as the cost of a patch without particles. this causes a serious load imbalance issue which leads to a poor scaling results as shown in Figure 2 above. Even with the measurement-based load balancer, the work load of a region is hard to precisely predict. When running with pure MPI scheduler, if one core has finished its assigned work faster than the other cores, it has to stay idle and wait for them to finish even if they are in the same node. With the same code, adaptive mesh refinement scaled well to 98K cores, [33]. The challenges that arose in the scaling of this fluid-structure interaction problem directly influenced the work presented here.

6 Uintah Task Graph and Schedulers

In order to explain how Uintah schedules tasks in executing the task graph, an explanation is needed of how the task-graph is generated as well as a brief description of the approaches used to execute it. Uintah uses both data parallelism and task parallelism to achieve high scalability when running with the hybrid MPI/threads scheduler in [33]. After patches are assigned to nodes and detailed tasks are created on local and neighboring mesh patches, Uintah computes dependencies between tasks by comparing each task's computed (output) and required (input) variables. If a task's computed and required variables are read from and sent to tasks on local patch, then an internal dependency is detected and marked as such on the task graph. If a task requires variables from (or computes variables for) a task on a patch held by another core or node then an external dependency is detected. In this case a unique MPI message tag is assigned on both the "computing" node and the "requiring" node. Once all dependencies are detected, a directed acyclic graph (DAG) will be compiled. The Uintah scheduler uses the task graph to determine the order of execution, assigns tasks to local computing resources (CPU/GPU) and ensures that the correct inter-process communication is performed.

6.1 Dynamic Scheduler

Originally, Uintah used a static scheduler in which tasks were executed in a predetermined order. A limitation of this approach is that a single task waiting for messages causes the calculation on a core to sit idle. In order to address this issue, a new Uintah dynamic scheduler was developed so as to better overlap communication and computation by using out-of-order task execution, [32]. Task queues were added to allow the scheduler to keep track of the different states of all tasks. An internal ready queue stores tasks whose local dependencies have been satisfied and are waiting for external MPI messages to arrive. An external ready queue stores tasks whose internal and external dependencies have all been satisfied and are now ready for execution. As long as the external ready queue is not empty, a

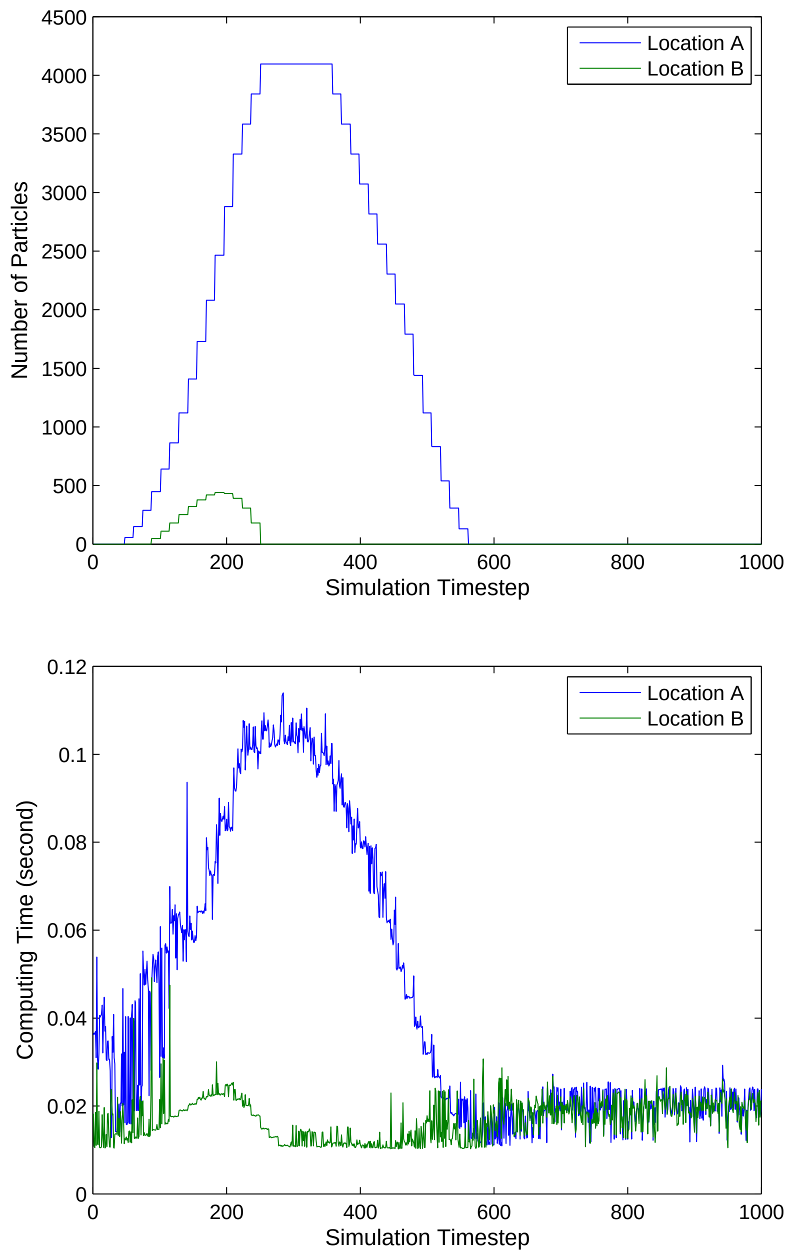

Figure 3: Number of particles and task execution time at two different locations

core can always be given tasks to execute. Once a task is completed, the scheduler will use the task graph to find the subsequent internally satisfied tasks and add them to internal ready queue. In order to support dynamic scheduling of tasks, the Uintah data warehouse was changed from a hashed map to a hashed multi-map to allow the saving of multiple variables under the same variable name and a patch id key. This change was necessary as when a task is running out-of-order, multiple versions of a variable may exist at the same time. This scheduler produced a significant performance benefit in lowering both the MPI wait time and the overall runtime [32].

6.2 Multi-thread Scheduler: Master-Slave Model

While the out-of-order execution model of [32] worked well for many cases, one limitation of its pure MPI scheduling is that tasks which are created and executed on different cores on the the same node cannot share data. A new multi-threaded MPI scheduler was used [33] to solve this problem by dynamically assigning tasks to worker threads during execution. This multi-threaded MPI scheduler used a master-slave model which had one control thread and several worker threads per MPI process. The control thread processed MPI receives, managed tasks queues and assigned ready tasks to worker threads. The worker threads simply executed their task and then asked the control thread to assign the next task. The control thread and worker threads communicated through pthread conditional variables. Uintah variables can be accessed freely by any thread as the task graph guarantees that there are no memory conflicts on any two ready tasks. However, many shared data structures, such as data warehouse and task queues, needed to be thread-safe and consistent across threads. As the Uintah framework manages tasks input and output, most tasks are already thread safe and can be supported by the multi-threaded scheduler without rewriting any task code. Experimental results on AMR ICE simulations showed 50% to 90% savings on memory usage by using the multi-threaded scheduler. This approach still did not result in the scalable execution of the fluid-structure problem described in Section 5, however.

6.3 Multi-thread Scheduler: De-centralized Model

A potential bottleneck in the master slave model is that a worker thread may become idle if the control thread cannot respond to its next ready task request quickly enough. In order to guarantee a short response time, the control thread was assigned to a dedicated core. This approach led to this core being under-utilized when running with small number of cores, as there was not enough work for the control thread. The solution to this was to design a new de-centralized multi-threaded scheduler to allow all threads to process MPI sends and receives or to execute tasks concurrently without a control thread. Instead of asking the control thread for a ready task, the threads in the de-centralized multi-threaded scheduler directly pull tasks from the two ready queues. When a thread pulls a task from the internal ready queue then MPI nonblocking receives will be posted. Also, when a thread pulls a task from the external ready queue then a task's call-back function will be executed and MPI sends will be posted after task execution. Furthermore each thread must keep checking the task queues and MPI receives when it is idle as there is no longer a central controller. As this could lead to busy-locking on those shared resources when multiple threads become idle and keep acquiring read locks, a two stage execution approach is used.

The first stage is to check if any work is available, either to process MPI receives or to execute a task for the current thread. When there is a ready task or a pending MPI receive, the scheduler will switch to the second stage to execute the task or to process MPI receives concurrently. If no work is available, a mutex needs to be acquired before checking all the task queues and MPI receives again. The scheduler will not release this mutex and so will prevent other idle threads from checking task queues and MPI receives until a new ready task is available or a new MPI receive is posted. In this way, when multiple threads become idle, checking for shared resources will be slowed down by this mutex and priority is given to threads that are able to update the task queue or post MPI receives.

Number of Cores	2	4	8	16	32
Master-Slave	57.28	20.72	9.4	4.81	2.95
De-centralized	29.8	15.84	8.2	4.59	2.78

Table 1: Execution Time: Master-Slave vs De-centralized

Table 1 shows a performance comparison between the de-centralized and master-slave models in a shared memory node. In this case the de-centralized model outperforms the master-slave model on all runs up to 32 cores per node. By monitoring CPU utilization of each core, it was confirmed that the de-centralized model solved the issue of under utilization of the core that runs the control thread. Furthermore, while the master-slave model will likely hit a control thread bottle neck when number of cores per node increases, this is less likely with the de-centralized model.

7 Uintah Hybrid Parallelism Improvement

This section will discuss how the MPI multi-threaded hybrid approach was used to overcome the challenges with regard to scaling fluid-structure problems. Unlike all the tasks in ICE components which were thread safe, several race conditions were found to exist on MPM components. Most of these race conditions were due to the use of global temporary data structures instead of local ones and were easily fixed.

7.1 Reducing Particle Relocation Costs

As noted above, in Uintah, solid objects are represented by MPM particle variables. During a timestep, each particle's new location co-ordinates and other physical attributes will be computed by MPM tasks and saved in the data warehouse. If a particle's position moves from one patch to another, it is the infrastructure's responsibility to map those particle variables back into the background grid according to their new locations. This is done by inserting a relocation task to the task graph. During this process, each particle's new owner patch will be located

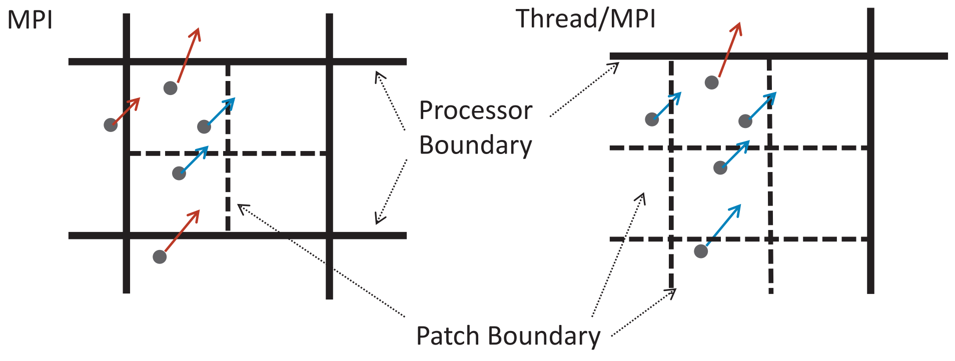

Figure 4: Particle relocation

in the patch BVH tree according to its new co-ordinates. If the new owner patch is located on the same node, only a simple re-indexing of the patch's particle variable array is required. However, if the new owner patch is located on another node, the particle information is transmitted using MPI. In particle relocation, only the sending node has knowledge of the new particle location and how many particles and their variables need to be transferred. The destination node must know the source node id first, as MPI nonblocking receives cannot be posted without a source rank id. To transfer the source node id to all destination nodes, Uintah uses an MPI_scatter message which can distribute a vector to all nodes concurrently.

In this way, for any particle that is moved using MPI, two messages are required. Thus the cost of moving a particle off a node is much more expensive than that of moving a particle inside a node or core. As illustrated in Figure 4, when using a hybrid MPI multi-threaded approach, the number of particles that need to be sent is significantly reduced when using one MPI process per node as opposed to one MPI process per core, as all transfers internal to a node no longer use MPI.

7.2 Load Balancing Improvements

By using the hybrid scheduling approach described above, all tasks in the same node can be executed by any idle cores on that node. Moreover, the load balancer now profiles and predicts workload per node instead of per core. The accuracy of prediction is improved as changes of workload over a larger region are generally more stable than those over a small region. The average core load imbalance value was reduced from about 60% to 25% when running with nearly one patch per core at 100K cores on the Jaguar XK6 by using this approach.

7.3 Using Lock-free Data Structures

A major overhead of the multi-threaded scheduler approach [33] is due to the use of locking to protect shared data structures. This overhead keeps increasing with the number of cores per node as contention for acquiring locks also increases. A significant reduction in tasks waiting was obtained by eliminating large amounts of locking overhead through a re-design of some of the shared data structures so as to make them lock-free. In particular by using hardware-based atomic operations [15], which are supported by modern CPU. These new data structures can be made to be more efficient than using the traditional pthread read write lock and mutex.

Uintah has three types of variables: 1) Grid variables exist everywhere in the simulation grid for flow simulations. Grid variables on the same node can share a combined 3D array with different memory windows. As both the memory window and the 3D array are reference counted, the associated memory will be deallocated when no longer referenced. 2) Particle variables exist only at a certain point for solid MPM simulations in Uintah. Particles in a Uintah patch are saved in a simple vector and indexed by a subset of that vector. By saving recent query ranges, particle sets are cached so as to speed up later queries. 3) Reduction variables are designed to combine multiple values provided by different tasks. The reduction variable operator must be associative, so that the order of computation will not alter the final value, apart from rounding errors. When the same reduction variable is computed multiple times on the same node, the locally reduced value will be updated immediately and stored in the data warehouse. After all local tasks have computed reduction variables, the global value of the reduction variable will be calculated through a call to MPI_AllReduce. As mentioned above, Uintah variables can share the same memory window and 3D array. In fact, most of Uintah objects such as grid level, variable label, MPI buffers are also reference counted so that they can be easily deallocated when no longer needed. When running in multi-threaded mode originally, reference counters were protected by a fixed size array of mutexes to ensure correctness. Once a new reference counted object was created, a mutex from this array was assigned to it in a round-robin fashion. This design allowed many objects to share a mutex instead of each object creating its own as there may be thousands of reference counted objects and dynamically creating thousands of mutexes is too expensive. However potential false-conflicts may exist when accessing two unrelated objects which happen to share the same mutex. This reference counting lends itself to the use of atomic operations, [15]. A new reference counting class was implemented in Uintah by using *add_and_fetch* and *sub_and_fetch* atomic operations to replace the pthread mutex vector.

As described above, all tasks depend on the hashed multi-map data warehouse to look up and save variables by using a patch id and variable name key. Each data warehouse has a pthread read/write lock to protect the hashed multi-map. A read-only lock needs to be acquired when a task looks up a variable's memory address from data warehouse. A write lock needs to be acquired when a new variable need to put result into the data warehouse either from a computational task or from MPI message. Based on our timing results on Uintah read-write locks, the data warehouse lock was seen to be the largest single source of overhead. For this reason the hashed multi-map data warehouse was redesigned to be lock-free by using a two-step look up strategy. During task-graph compilation, a hash map containing keys of all locally computed and required variables is created on-the-fly. This hash map will not change during task execution and can also be shared by multiple data warehouses until the task graph needs to be recompiled. The actual container of variables in each data warehouse will be a pre-allocated vector which has exactly the same size as this hash map. The value of this pre-computed hash map will be the index to the container vector. When accessing a variable, the data warehouse will first use the hash-map key to locate the variable from its private container vector. As the pre-computed hash map is read-only during the task execution, no lock is needed to protect it. When updating the container vector, atomic operations are used to achieve consistency among threads. A simplified version of variable add algorithm using *compare_and_swap* operation is shown in Algorithm 1. A variable reduce algorithm for updating reduction variables using *test_and_set* is shown in Algorithm 2.

Algorithm 1 Add a variable to the data warehouse

```
**function** ADD($key, var$)
    $idx \leftarrow hash\_map[key]$
    $di \leftarrow$ **new** $dataitem()$
    $di^\wedge.var \leftarrow var$
    **repeat**
        $di^\wedge.next \leftarrow vector[idx]$
    **until** $compare\_and\_swap(\&vector[idx], di^\wedge.next, di)$
**end function**
```

Algorithm 2 Reduction of a variable into the data warehouse

```
**function** REDUCE($key, var$)
    $idx \leftarrow hash\_map[key]$
    $di \leftarrow$ **new** $dataitem()$
    $di^\wedge.var \leftarrow var.clone()$
    **repeat**
        $old \leftarrow test\_and\_set(\&vector[idx], 0)$
        **if** $old = 0$ **then**
            $old \leftarrow di$
        **else**
            $old^\wedge.var^\wedge.reduce(di^\wedge.var)$
            **delete** $di$
        **end if**
        $di \leftarrow test\_and\_set(\&vector[idx], old)$
    **until** $di = 0$
**end function**
```

8 Experimental Results

This section will consider whether or not hybrid parallelism can sufficiently improve the performance and scalability of fluid-structure interaction problems in Uintah. We will also examine the performance difference between using a lock-free data structure and a traditional lock-protected data structure. The prototypical simulation study used in Section 5 was used to compare the hybrid multi-threaded/MPI approach, the multi-threaded/MPI with lock-free data warehouse approach and the MPI approach.

8.1 Single Node Performance Improvement

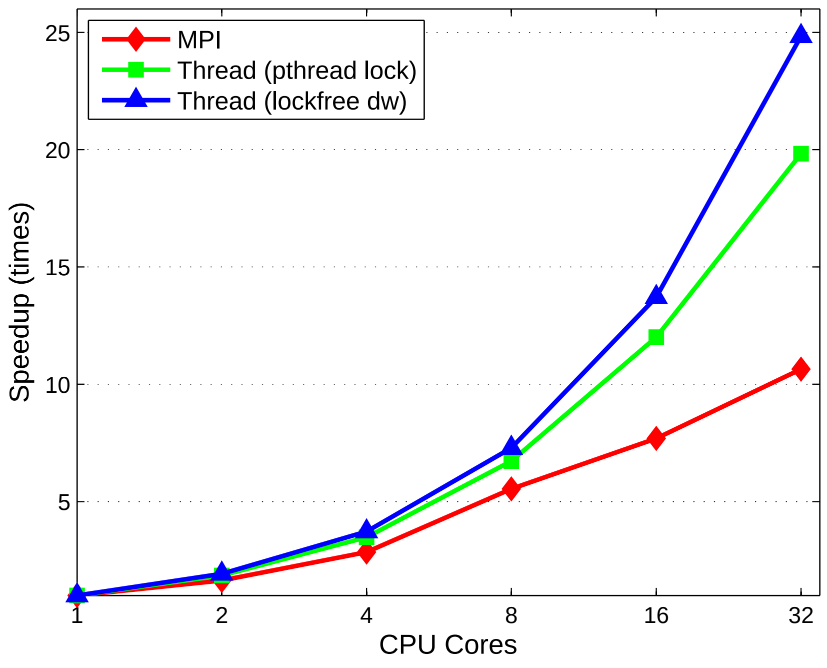

Figure 5: Performance comparison

The first comparison is between the hybrid multi-threaded MPI approach and the lock-free data warehouse on a single shared memory node. This test was run on a Jaguar Cray XK-6 external node with 32 AMD interlagos cores. Three sets of strong scaling benchmark results were gathered by using the dynamic MPI scheduler, the de-centralized multi-threaded MPI hybrid scheduler with the old pthread locking data warehouse and the de-centralized multi-threaded MPI hybrid scheduler with the lock-free data warehouse. The input files for all three runs are identical and generate 887K particles on an AMR grid. Figure 5 shows the speedups when running with 2, 4, 8, 16 and 32 cores. The multi-threaded scheduler with the lock-free data warehouse is 1.4X faster than with pthread locking data warehouse and 2.4X faster than using MPI only.

Table 2 shows the execution time comparison results when running with different numbers of MPI processes and with different numbers of threads per MPI process. CPU affinity was used to guarantee that each thread is assigned to a dedicated core. Threads in the same MPI process were assigned to nearby cores so as to limit any possible cache-coherence overheads. These benchmark results show that the optimized performance for this problem is achieved when running with the maximum number of available threads per MPI process.

nMPI	32	16	8	4	2	1
nThread	1	2	4	8	16	32
ExecTime(s)	5.18	3.77	3.05	2.79	2.62	2.22

Table 2: Mixed MPI and Threads Performance (total 32 cores)

8.2 Scalability Improvement

The scalability benchmark involved four runs with varying problem sizes using a similar approach to that in Section 5 and the same test problem. Each run uses a mesh that was refined by a factor of two in each dimension with respect to the previous run. As the mesh is refined where particles exist, eight times as many particles will be created than on the original coarse mesh. The number of particles created in the four runs are 7.1 million, 56.6 millon, 452.9 millon and 3.62 billion respectively. This leads to each run being approximately eight times as large as the previous run. As we will also generate weak scaling results at the same time, the problem size per node needs to be constant. Therefore, eight times as many cores were used from one run to the next. These tests were run on the Jaguar Cray XK-6 machine with up to 256K cores and with 16 cores per node. Figure 6 shows the scaling results for four benchmark tests. Weak scalability is represented by the almost-horizontal dashed lines and strong scalability by the almost-straight diagonal solid lines in the four runs. The strong scaling efficiency of the largest problem (the rightmost solid line) is 68% at 256K cores relative to the base case of 16K cores.

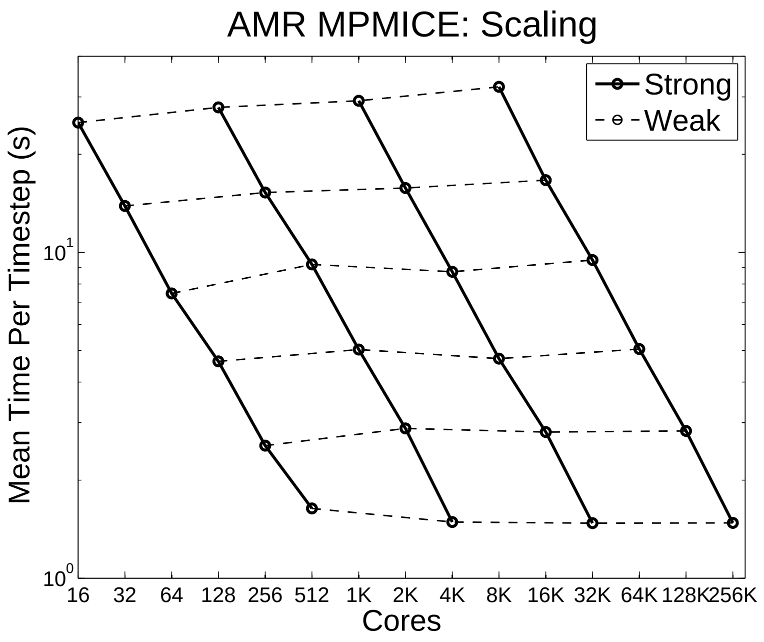

Figure 6: Strong and Weak Scaling on Jaguar XK-6

By using optimized hybrid multi-threaded MPI algorithms, the number of MPI ranks involved in communication is reduced without much performance overhead. This approach leads to better weak scaling. Figure 7 shows the weak scaling efficiency compared to the previous MPI only benchmark result from Section 5. The base cases to calculate efficiencies are 24 cores for MPI runs on Kraken and 64 cores for the hybrid multi-threaded MPI runs on Jaguar XK6. These two series are the same as the second from bottom dashed weak scaling line in Figure 2 and the bottom dashed weak scaling line in Figure 6. The results shows significant improvement of weak scaling efficiency when using hybrid multi-threaded MPI approach especially on runs with large core counts.

9 Conclusions and Future Work

These results presented here show great performance improvements, and also show good scalability so far on 256K cores on Jaguar Cray XK-6 by using the hybrid multi-threaded MPI scheduler and the new lock-free data structures. However alongside the lock-free data warehouse, many other shared data structures such as task queues, particle subset caches and patch BVH trees still need to be redesigned

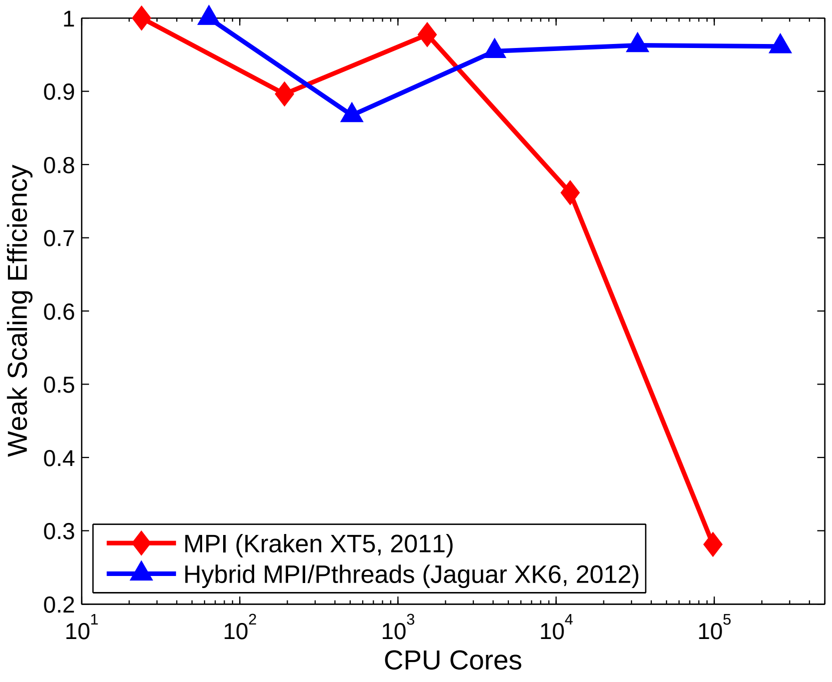

Figure 7: Weak Scaling Efficiency Comparison

to be lock-free. Even though these data structures are accessed less frequently than the data warehouse, the waiting time for acquiring locks related to these structures is going to grow as the number of core per node increases. The future removal of all these locks and making Uintah fully lock-free will improve the scaling here still further on present and future many-core and multi-core machines. The present decentralized scheduler will also be used instead of the current master-slave model used in the Uintah GPU scheduler [22]. This will require the current GPU controller thread routines to be rewritten to guarantee thread-safety, but will allow these scalability results to be extended to heterogeneous CPU-GPU architectures.

Acknowledgment

This work was supported by the National Science Foundation under subcontracts No. OCI0721659, the NSF OCI PetaApps program, through award OCI0905068 and by DOE INCITE award CMB015 for time on Jaguar. Uintah was written by the University of Utah's Center for the Simulation of Accidental Fires and Explosions (C-SAFE) and funded by the Department of Energy, subcontract No. B524196. We would also like to thank Alan Humphrey, Todd Harman and all those presently and previously involved with Uintah, Justin Luitjens in particular.

References

[1] Scientific Grand Challenges: Crosscutting Technologies for Computing at the Exascale Report from the Workshop Held February 2-4, 2010 `http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Crosscutting_grand_challenges.pdf`

[2] S.W. Attaway, M.W. Heinstein, J.W. Swegle Coupling of smooth particle hydrodynamics with the finite element method Nuclear Eng. Design, 150 (1994), pp. 199–205

[3] Attaway, S.A.; Barragy, E.J.; Brown, K.H.; Gardner, D.R.; Hendrickson, B.A.; Plimpton, S.J.; Vaughan, C.T.; , "Transient Solid Dynamics Simulations on the Sandia/Intel Teraflop Computer," Supercomputing, ACM/IEEE 1997 Conference , vol., no., pp. 58, 15-21 Nov. 1997

[4] M. Berzins. Status of Release of the Uintah Computational Framework, SCI Technical Report, No. UUSCI-2012-001, SCI Institute, Univ. of Utah, 2012.

[5] K. Brown, S. Attaway, S.J.Plimpton, B. Hendrickson, Parallel Strategies for Crash and Impact Simulations, Computer Methods in Applied Mechanics and Engineering, 184, 375-390 (2000).

[6] A. T. Barker and X.-C. Cai. Scalable parallel methods for monolithic coupling in fluid-structure interaction with application to blood flow modeling. J. Comput. Phys. 229, 3 (February 2010), 642-659.

[7] J.G. Bennett, K.S. Haberman, J.N. Johnson, B.W. Asay, B.F. Henson. A Constitutive Model for the Shock Ignition and Mechanical Response of High Explosives" J. Mech. Phys. Solids 46:2303, 1998.

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

[9] M. Berzins, J. Luitjens, Q.Meng, T.Harman, C.A.Wight and J. Petersonr. Uintah a scalable framework for hazard analysis Proc. of 2010 Teragrid Conf. July 2010, ACM.

[10] M. Berzins, Q.Meng, J.Schmidt and J. Sutherland. DAG-Based Software Frameworks for PDEs Proc. of HPSS 2011 (Europar Bordeaux August 2011), Springer 2012 (to appear)

[11] J.U. Brackbill and H.M. Ruppel FLIP: A Method for Adaptively Zoned, Particle-in-Cell Calculations of Fluid Flow in two Dimensions. Journal of Computational Physics 65, 314-343 (1986).

[12] J.U. Brackbill Particle Methods. International Jour. Numer. Meths. in Fluids, 2005:47:693-705.

[13] H.-J. Bungartz, J. Benk, B. Gatzhammer, M. Mehl and T. Neckel: Partitioned Simulation of Fluid-Structure Interaction on Cartesian Grids. In H.-J. Bungartz, M. Mehl and M. Schfer (ed.), Fluid-Structure Interaction - Modelling, Simulation, Optimisation, Part II of LNCSE. Springer, Berlin, Heidelberg, 2010

[14] Z. Dostl, V. Vondrk, D. Hork, C. Farhat and P. Avery, "Scalable FETI Algorithms for Frictionless Contact Problems", in: Domain Decomposition Methods in Sciences and Engineering XVII, Lecture Notes in Computational Science and Engineering (LNCSE), eds. U. Langer et al., Springer, Vol. 60, Berlin, pp. 263–270 (2008)

[15] K. Fraser. Practical lock-freedom. Technical Report UCAMCL- TR-579, University of Cambridge Computer Laboratory, 2004.

[16] J.D. Germain, J. McCorquodale, S.G. Parker, C.R. Johnson. Uintah: A massively parallel problem solving environment. In *HPDC'00: Ninth IEEE International Symposium on High Performance and Distributed Computing*, IEEE Computer Society: Washington, DC, USA 2000; 33.

[17] J. Gotz, K. Iglberger, M. Sturmer, U. Rude. Direct Numerical Simulation of Particulate Flows on 294912 Processor Cores, High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for, pp.1-11, 13-19 Nov. 2010

[18] J.E. Guilkey, T.B. Harman, and B. Banerjee. An Eulerian-Lagrangian approach for simulating explosions of energetic devices. Computers and Structures, 85:660–674, 2007.

[19] J.E. Guilkey and J.A. Weiss. Implicit time integration for the material point method: Quantitative and algorithmic comparisons with the finite element method. Int. J. Num. Meth. Eng., 57:1323–1338, 2003.

[20] J.E. Guilkey, T.B. Harman, A. Xia, B.A Kashiwa, and P.A. McMurtry. An eulerian-lagrangian approach for large deformation fluid-structure interaction problems, part 1: Algorithm development. In Fluid Structure Interaction II, Cadiz, Spain, 2003. WIT Press.

[21] Grinberg, Ilan; Wiseman, Yair; , "Scalable parallel simulator for vehicular collision detection," Vehicular Electronics and Safety (ICVES), 2010 IEEE Int. Conf.on , pp.116-121, 15-17 July 2010

[22] A. Humphrey, Q.Meng, M.Berzins and Todd Harman Radiation Modeling Using the Uintah Heterogeneous CPU/GPU Runtime System SCI Technical Report, No. UUSCI-2012-003, SCI Institute, University of Utah, 2012.

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

[24] T.B. Harman, J.E. Guilkey, B.A Kashiwa, J. Schmidt, and P.A. McMurtry. An eulerian-lagrangian approach for large deformation fluid-structure interaction problems, part 2: Multi-physics simulations within a modern computational framework. In Fluid Structure Interaction II, Cadiz, Spain, 2003. WIT Press.

[25] L. V. Kale, E. Bohm, C. L. Mendes, T. Wilmarth, and G. Zheng. Programming petascale applications with Charm++ and AMPI. Petascale Computing: Algorithms and Applications, 1:421–441, 2007.

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

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

[28] B.A. Kashiwa and R.M. Rauenzahn. A cell-centered ice method for multi-phase flow simulations. Technical Report LA-UR-93-3922, Los Alamos National Laboratory, Los Alamos, 1994.

[29] J. Luitjens and M. Berzins. Improving the performance of Uintah: A large-scale adaptive meshing computational framework. In Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS10), 2010.

[30] J. Luitjens and M. Berzins. Scalable parallel regridding algorithms for block-structured adaptive mesh refinement, Concurrency And Computation: Practice And Experience, Vol. 23, No. 13, pp. 1522–1537. 2011.

[31] J. Luitjens. The Scalability of Parallel Adaptive Mesh Refinement Within Uintah, University of Utah Doctoral Thesis, 2010.

[32] Q. Meng, J. Luitjens, and M. Berzins. Dynamic task scheduling for the Uintah framework. In Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10), 2010.

[33] Q. Meng, M. Berzins and J. Schmidt. Using hybrid parallelism to improve memory use in the Uintah framework. In Proceedings of the Teragrid 2011 Conference, ACM (2011).

[34] S.G.Parker. A component-based architecture for parallel multi-physics PDE simulation. *Future Generation Comput. Sys.* 2006; 22 (1):204-216.

[35] S.G. Parker, J. Guilkey, T. Harman, A component-based parallel infrastructure for the simulation of fluid-structure interaction, Engineering with Computers, pp.277-292, Vol. 22, Iss. 3, 2006.

[36] A. Sadeghirad, R. M. Brannon, and J. Burghardt, A convected particle domain interpolation technique to extend applicability of the material point method for problems involving massive deformations, Int. J. for Numerical Methods in Engineering, vol. 86, iss. 12, pp. 1435-1456, 2011.

[37] J.Schmidt Large Scale Parallel Solution of Incompressible Flow Problems using Uintah and hypre SCI Technical Report, No. UUSCI-2012-002, SCI Institute, University of Utah, 2012.

[38] M. Steffen, R.M. Kirby, M. Berzins. Decoupling and Balancing of Space and Time Errors in the Material Point Method (MPM), Int. J. for Numerical Methods in Engineering, Vol. 82, No. 10, pp. 1207–1243. 2010.

[39] D. Sulsky, Z. Chen, and H. L. Schreyer. A particle method for history-dependent materials. Comp. Methods Appl. Mech. Engrg., 118:179–196, 1994.

[40] D. Sulsky, S.J. Zhou, and H.L. Schreyer. Application of a particle-in-cell method to solid mechanics. Computer Physics Communications, 87:236–252, 1995.

[41] L.T. Tran, M Berzins. IMPICE Method for Compressible Flow Problems in Uintah, International Journal for Numerical Methods in Fluids, Published online 20 July, 2011.

[42] L.T. Tran, J. Kim, M. Berzins. Solving Time-Dependent PDEs using the Material Point Method, A Case Study from Gas Dynamics, In International Journal for Numerical Methods in Fluids, Vol. 62, No. 7, pp. 709–732. 2009.

[43] P.C. Wallstedt and J.E. Guilkey, An Evaluation of Explicit Time Integration Schemes for use with the Generalized Interpolation Material Point Method, Journal of Computational Physics, 227, 9628-9642, 2008.