Using Hybrid Parallelism to Improve Memory Use in the Uintah Framework

Qingyu Meng
Scientific Computing and Imaging Institute
University of Utah
Salt lake City, UT 84112 USA
qymeng@cs.utah.edu

Martin Berzins
Scientific Computing and Imaging Institute
University of Utah
Salt lake City, UT 84112 USA
mb@cs.utah.edu

John Schmidt
Scientific Computing and Imaging Institute
University of Utah
Salt lake City, UT 84112 USA
John.Schmidt@utah.edu

ABSTRACT

The Uintah Software framework was developed to provide an environment for solving fluid-structure interaction problems on structured adaptive grids on large-scale, long-running, data-intensive problems. Uintah uses a combination of fluid-flow solvers and particle-based methods for solids together with a novel asynchronous task-based approach with fully automated load balancing. Uintah's memory use associated with ghost cells and global meta-data has become a barrier to scalability beyond O(100K) cores. A hybrid memory approach that addresses this issue is described and evaluated. The new approach based on a combination of Pthreads and MPI is shown to greatly reduce memory usage as predicted by a simple theoretical model, with comparable CPU performance.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programing; G.1.8 [**Mathematics of Computing**]: Partial Differential Equations; G.4 [**Mathematics of Computing**]: Mathematical Software; J.2 [**Computer Applications**]: Physical Sciences and Engineering

Keywords

Uintah, hybrid parallelism, scalability, parallel, adaptive, memory

1. INTRODUCTION

An important trend in high performance computing is the planning and design of architectures for future computers with multi-petaflop and eventually exaflop performance. A recent DARPA report [5] discusses many of the challenges that face those trying to program such architectures. One of these is expected to be significantly less memory per core than is available today. At the same time today as the numbers of cores grow, the associated memory per core is already being reduced in present-day architectures. In this context a node will be assumed to consist of one or more sockets, each of which has multiple cores.

In the case of large parallel problem solving environments for computational science and engineering problems this trend is problematic with regard to the traditional approach of using one MPI process per core. There are two main reasons for this. The first is that any global meta-data must be replicated on the process associated with each core. The second is that in traditional domain decomposition approaches for solving partial differential equations, each core has a portion of the spatial mesh and must read the ghost cell data it needs for the stencil being used in the computation, as well as sending ghost-cell information to its neighbors. Thus each cores must potentially assign data storage for multiple copies of halo information that is already being stored on that node. Overall this approach results in replication of variables in already limited storage.

This problem has been recognized by a number of authors, e.g. [4,20] and as a result a number of software frameworks have moved from a model that only uses MPI to one that employs MPI to communicate between nodes and a shared memory model, often by using OpenMP, to map the work onto the cores in a node.

In this paper we consider the memory model and usage issues associated with Uintah [6, 9, 21] an open-source software framework (`www.uintah.utah.edu`). Uintah is novel in its use of a task-based paradigm, with complete isolation of the user from parallelism. The individual tasks are viewed as part of a directed acyclic graph (DAG) and are executed adaptively, asynchronously and now often out of order [18]. Uintah uses a novel adaptive meshing approach [16] as well as a variety of fixed mesh and particle solution methods. In recent previous work we showed that Uintah scales well to about 98K cores for some applications [6] including a sympathetic explosion modeling problem, funded by the NSF PetaApps Program, that is one of our main applications driving examples. As we approach problem sizes requiring greater than 100K cores on machines such as Jaguar, [1] and Kraken [2] the memory requirements of these problems requires a close examination of the overall memory usage within the Uintah framework. The typical message passing paradigm that Uintah operated under was that any data that needed to be shared to a neighboring processor must be passed via MPI. For multi-core architectures the process of passing data that is local to a node is both wasteful in terms of latency from MPI sends and receives and in the duplication of identical data that is shared between cores.

The threading model that is described in this paper demonstrates

TeraGrid'11 TeraGrid '11, July 18-21, 2011, Salt Lake City, Utah, USA.

[1] Jaguar is a DOE supercomputer located at the Oak Ridge National Laboratory with 18,688 compute nodes each of which contains dual hex-core AMD Opteron 2435 (Istanbul 2.6GHz) processors, 16GB memory, and a SeaStar 2+ router, giving 224,256 processing cores, 300TB of memory, and a peak performance of 2.3 petaflop/s

[2] Kraken is an NSF supercomputer located at the University of Tennessee/ Oak Ridge National Laboratory with 112,896 cores, and a similar architecture to Jaguar.

the memory savings that we have observed by eliminating the duplication of data within a node. The memory savings allows us to to expand the scope and range of problems that we have been unable to explore up until now. For the architectures of Kraken and Jaguar where the memory per node is limited to 16GB per node, the increase in memory savings is significant and potentially opens up the range of problems and core counts that have been up until now out of reach.

In this paper we look at how to extend the novel approach of Uintah to the use of this hybrid model. In contrast to many other approaches the Uintah task-based model lends itself better to the use of Pthreads, see [2], rather than OpenMP. In what follows Section 2 provides an overview of the Uintah software, while Section 3 describes examples of related efforts within other similar software frameworks. Section 4 describes the main uses of memory in Uintah related to ghost cells and global meta-data and provides a simple model that makes it possible to predict the potential reduction in memory use. Uintah's recent task execution algorithm and its extension to a thread-based model at node level is described in Section 5. Finally in Section 6 we describe experiments that illustrate over a range in scales of processor numbers what the improvements in memory are and shows that they match the predictions in Section 4. The paper concludes by describing future work in this area.

2. OVERVIEW OF UINTAH SOFTWARE

The Uintah Software was originally written as part of the University of Utah Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [9]. C-SAFE, a Department of Energy ASC center, focused on providing science-based tools for the numerical simulation of accidental fires and explosions. Uintah was originally capable of running on 4K processors and has now also been released as software[3] and extended to run on 98K processors through additional DOE and NSF funding.

The aim of Uintah was to be able to solve complex multiscale multiphysics problems, such as the benchmark C-SAFE problem. This is a multi-physics, large deformation, fluid-structure problem consisting of a small cylindrical steel container filled with a plastic bonded explosive (PBX9501) subjected to convective and radiative heat fluxes from a fire [12].

In order to solve such complex multi-scale multi-physics problems, Uintah makes use of a component design that enforces separation between large entities of software and can be swapped in and out, allowing them to be independently developed and tested within the entire framework. This has led to a very flexible simulation package that has been able to simulate a wide variety of problems including shape charges, stage-separation in rockets, the biomechanics of microvessels , the properties of foam under large deformation, and the evolution of large pool fires caused by transportation accidents [15], in addition to the exploding container described above. The application of Uintah to a petascale problem in hazard analysis arising from "sympathetic" explosions in which the collective interactions of a large ensemble of explosives results in dramatically increased explosion violence, was described in [6].

Uintah currently contains three main simulation algorithms, or components, that are capable of using Adaptive Mesh Refinement (AMR): the ICE compressible multi-material CFD formulation, the particle-based Material Point Method (MPM) for structural mechanics, the combined fluid-structure interaction algorithm MPMICE [12].

ICE is a "multi-material" CFD algorithm that was originally developed by Kashiwa and others at LANL [14] for incompressible and compressible flow regimes. This method conserves mass, momentum, energy, and the exchange of these quantities between materials and is used here on adaptive structured meshes consisting of hexahedral patches, often of 8^3 or 16^3 cells [16]. The Material Point Method is a particle method that is used to evolve the equations of motion for the solid materials applications involving complex geometries, large deformations and fracture. Originally described by Sulsky, et al. [24], MPM is an extension to solid mechanics of the well-known particle-in-cell (PIC) method for fluid flow simulation, that uses the ICE mesh as a computational scratchpad. The fluid-structure methodology is a combination of the MPM and ICE [12].

In addition the fixed mesh Arches component was designed for simulation of turbulent reacting flows with participating media radiation. It is a three-dimensional, Large Eddy Simulation (LES) code that uses a low-Mach number (Ma<0.3), variable density formulation to simulate heat, mass, and momentum transport in reacting flows. The LES algorithm solves the filtered, density-weighted, time-dependent coupled conservation equations for mass, momentum, energy, and particle moment equations in a Cartesian coordinate system [15]. The Arches code exhibits parallel scaling through its integration in the Uintah framework [22].

The Uintah component approach allows the application developers to only be concerned with solving the partial differential equations on a local set of block-structured adaptive meshes, without worrying about explicit message passing calls or notions of parallelization or load balancing. This approach also allows the developers of the underlying parallel infrastructure to focus on scalability concerns including load balancing, task (component) scheduling and communications. This component based approach to solving complex problems allows improvements in scalability to be immediately applied to applications without any additional work by the applications developer. Uintah's load balancer utilizes space-filling curves in order to cluster patches together [17].

An important feature of Uintah is its adaptive meshing capability. As reported in previous work [6, 16], Uintah's adaptive mesh regridder defines a set of fixed-sized tiles throughout the domain. Each tile is then searched, in parallel, for refinement flags without the need for communication. All tiles that contain refinement flags become patches. This regridder is advantageous at large scales because cores only communicate once at the end of regridding when the patch sets are combined. Testing of this new regridder showed good scaling up to 98K cores at which point there are only one or two patches with 4096 cells per core; at this point scalability begins to break down. However, the key problem was that we were not able to run a larger problem with more patches per core as there was insufficient memory to accommodate the larger run. The main challenge in moving Uintah up to the next size of parallel architecture is thus to reduce the memory footprint of the code per core. One approach that has been adopted by others to do this is that of hybrid parallelism in which OpenMP or Pthreads are used to obtain parallel performance at the level of a node and MPI is used between nodes.

3. RELATED PARALLEL FRAMEWORKS

At present there is much work on hybrid approaches often using MPI and OpenMP and more recently extending sometimes to the use of GPUs. These approaches have been used in several codes which are similar to some parts of Uintah and have been run on large parallel architectures. In the case of adaptive mesh codes there are many such solvers and frameworks such as the Chicago ASCI code FLASH [7] based on adaptive oct-tree meshes and the

[3] see `http://www.uintah.utah.edu`

physics AMR codes Enzo, [19] Cactus [11,20] and Castro [1]. The highly adaptive mesh refinement (AMR) scalable codes of Ghattas et al. [8] also use an oct-tree based approach for very different problems using only MPI. The recent AMR Gamer hydrodynamics codes of Schive [23] not only uses MPI with OpenMP, but also extends the model to accommodate GPUs.

In a similar manner to the recent work on Uintah, the Enzo framework [19] is also currently being extended with the Enzo-P and Cello AMR frameworks being designed with new AMR features designed for extreme parallel scalability, including new techniques to address basic data type issues related to data structure scalability.

The Cactus framework [11, 20] allows the composition of individually developed components (called thorns) to full applications. Cactus parallelizes its data structures on distributed memory architectures via spatial domain decomposition, with ghost cells added to each MPI processes part of the grid. The fourth order methods used in Cactus require three ghost cells and consequently impose a significant potential memory overhead for each MPI process. This potential memory overhead is partially overcome by Cactus using OpenMP within a multi-core node to avoid memory replication of ghost cells and also to increase performance [20]. Similarly the astrophysics Castro code [1] uses an OpenMP/MPI approach to achieve weak scaling to 196K cores, for compressible flow problems.

However, there are overheads associated with OpenMP commands, see [2, 10], and the natural task-based structure of Uintah makes it more natural to consider Pthreads. Performance of the two approaches seems broadly similar [3]. While Pthreads programming is arguably more complex and potentially error-prone that the relative, if beguiling, simplicity of OpenMP, within Uintah threading is only used at a systems level and is not visible to the user. It is also perhaps somewhat easier to incorporate Uintah tasks within threads at present, even though OpenMP from version 3.0 onwards does support tasking.

4. UINTAH GLOBAL DATA STRUCTURES

The global memory usage of Uintah when using a straight MPI model for communication and computation is broken down into three main areas: shared ghost/halo data from the main computational data from the solution of partial differential equations,global meta data for the underlying computational grid and load balancing, and finally the external library requirements. This last case is most easily dealt with in that, based on experiments run on a single node of Ranger, roughly a third of the memory use was devoted to external third party libraries such as MPI and other operating system dependencies, compared to the internal memory usage within the Uintah framework,

4.1 Ghost cell data in Uintah

The ICE fluid-flow algorithm is a multi-material computational fluid dynamics approach that solves the compressible Navier Stokes representation of fluid materials. The state of a single material is described by eight quantities and include mass, velocity, internal energy, temperature, specific volume, volume fraction, stress, and equilibration pressure. For N materials, there are N*8 state variables that are solved for during a single time step. During the individual steps of the ICE algorithm, ghost cell data from one patch must be transferred to neighboring patches. For a typical step, a single layer of ghost cell data is required, however, there are some steps of the algorithm that require two layers of ghost cell data. In cases in which turbulence modeling is included, three layers of ghost cell data are included for several of the state variables. In the computational experiments described in Section 6, two materials were used in an AMR calculation, so there were 16 state variables with their associated ghost cell data that needed to be transferred during each timestep of the solution phase.

In the typical Uintah MPI model, ghost cell data is copied to a buffer on the sending processor and then sent to the receiving processor where it is stored in a buffer before being copied to the Data Warehouse. This buffer holds a message consisting of variables whose destination is the same. Although during the sending and receiving stage, there are potentially four copies of the data that are resident in memory, once the ghost cell data has been copied to the Data Warehouse, the buffers holding ghost cell data are deallocated requiring only two copies of the ghost cell data at any given time. In Uintah the Data Warehouse is the repository of solution variables that exists inside each process. The applications code typically reads the variables it needs from the Data Warehouse, updates these variables and then writes back the updated variables, see [18].

It is straightforward to articulate this overhead in a framework like Uintah, as the following example illustrates. Consider the case when each core has n_{el}^3 cubic mesh patches each of which has n_p^3 points in it. The number of mesh points native to that core is then given by N_{pc} where

$$N_{pc} = n_{el}^3 n_p^3$$

Suppose that the computational stencil has a halo of n_h ghost cells, then the storage needed per core for the halo information, as denoted by N_h is

$$N_h = 2\ n_h 6 n_{el}^2 n_p^2$$

where the facor of two corresponds to a doubling of storage in connection with MPI. The memory overhead percentage associated with the halo is then given by M_{over}, where

$$M_{over} = \frac{N_h}{N_{pc}} \times 100\% = \frac{12 n_h}{n_{el} n_p} \times 100\%$$

In Uintah n_{el} is often in the range 2-4, $n_p = 12$ [18] and $n_h = 2$ thus giving a halo overhead of 100% if $n_{el} = 2$ and a halo overhead of 50% if $n_{el} = 4$. Of course this is a considerable simplification and for a mesh partition that is not cubic when stored on a core the halo may be even larger. These numbers correspond to a similar overhead identified in Cactus [25].

4.2 Global meta-data in Uintah

The Uintah framework currently requires that certain data must be replicated across the entire domain. This meta-data includes the underlying grid layout and load balancing information. The current implementation requires that every processor must know the extents of every patch (currently just a high and low 3-vector in index space) as well as which processor owns which patches. Although this lightweight data structure is relatively small at present (60 bytes or 7.5 doubles per patch) and can easily be communicated, the growing demand for larger number of processors and patches and the requirements of AMR can approach the point where this data structure may, as we will see below, dominate the memory per core in an MPI approach.

On a machine with NT_c cores in total the size of this meta-data structure is N_{md} where

$$N_{md} = 7.5 N T_c\ n_{el}^3.$$

For a small number of partial differential equations each of which will need storage of $O(N_{pc})$ the mesh storage, N_{md} will quite easily exceed the core storage if say, 100K cores are used.

Although only having one copy of the mesh data per node will help to reduce the memory requirements, this may not completely

solve the problem when the number of cores and nodes approaches those predicted for exascale machines [5].

4.3 A model for memory saving in Uintah

In order to assess the possible memory saving from the use of a hybrid approach, consider a node with n_c cores so that the total number of cores is given by .

$$NT_c = n_{node} \times n_c.$$

On each node there are a total of $6n_c\, n_{el}^3$ internal and external faces of all the patches. Of these only $6n_{el}^2 n_c^{\frac{2}{3}}$ patch faces are on external faces of the node in that they connect to patches on other nodes. The potential memory saving consists of the difference between these two terms as well as the saving due to there being only one copy of the global data structure. Hence the percentage potential memory saving, M_{sav} is given by

$$M_{sav} = \frac{6(n_c n_{el}^3 - n_{el}^2 n_c^{\frac{2}{3}})2n_h n_p^2 + 7.5NT_c n_{el}^3(n_c-1)}{6n_c n_{el}^3 2n_h n_p^2 + n_c n_{el}^3 n_p^3 + 7.5NT_c n_{el}^3 n_c)} \times 100\%$$

where the term $n_c n_{el}^3 n_p^3$ approximates the native variables stored per node. Dividing both sides by $n_c n_{el}^3$ gives

$$M_{sav} = \frac{(12\alpha)n_h n_p^2 + 7.5 n_{node}(n_c-1)}{12 n_h n_p^2 + n_p^3 + 7.5 n_{node} n_c)} \times 100\%$$

where $\alpha = 1 - n_{el}^{-1} n_c^{-1/3}$. Given that $n_p = 12$, $n_h = 2$, $n_c = 12$, $n_{el} \approx 2$ and $\alpha \approx 0.76 \approx 1 - 1/(2.3 n_e l)$ for Kraken, we get

$$Mem_{saved} \approx \frac{29 + 0.91 n_{node}}{58 + n_{node}} \times 100\%$$

Thus giving a potential memory saving of 90% as the number of nodes , n_{nodes}, becomes large, or, alternatively, a memory reduction to below 10% of the memory used with MPI alone.

5. UINTAH TASK-GRAPH ENGINE

As noted in the introduction, Uintah is a sophisticated computational framework that can integrate multiple simulation components, analyze the dependencies and communication patterns between them, and execute the resulting multi-physics simulation. This is done by utilizing an abstract task-graph representation of parallel computation and communication to express data dependencies between components. The task-graph is a directed acyclic graph of tasks. Each task consumes some input and produces some output (which is in turn the input of some future task). These inputs and outputs are specified for each patch in a structured AMR grid. Associated with each task is a C++ method which is used to perform the actual computation. Each component specifies a list of tasks to be performed and the data dependencies between them. The task-graph approach of Uintah shares many features with the migratable object philosophy of Charm++ [13]. In order to increase efficiency, the task graph is created and stored locally [6].

Uintah's task scheduler is responsible for computing the dependencies of tasks, determining the order of execution and ensuring that the correct inter-process communication is performed [6]. Originally, Uintah used a static scheduler in which tasks were executed in a pre-determined order. This caused delays when a single task was waiting for a message. The new Uintah dynamic scheduler changes the task order during the execution to overlap communication and computation [18]. This scheduler required a large amount of development to support the out-of-order execution, which produced a significant performance benefit in lowering both the MPI wait time and the overall runtime. The dynamic scheduler utilizes two task queues: an internal ready queue and an external ready queue. If a task's internal dependencies are satisfied, then that task will be put in the internal ready queue where they will wait until all required MPI communication has finished. As long as the external queue is not empty, the processor always has tasks to run. This can help to overlap the MPI communication time with task execution. This approach reduces MPI wait times significantly, as shown in [6, 18].

5.1 Multi-Threaded MPI Runtime System

In the Uintah framework, after the regridder changes the simulation grid and the load balancer generates the patch distribution, the scheduler will create new sets of detailed tasks, compile a new task graph and initialize data warehouses. Originally, Uintah used both dynamic and static schedulers, based solely on MPI, in which data structures were created on each MPI process. Although most of Uintah infrastructure components are carefully designed to be stored in a distributed manner, it is necessary for some data to be stored multiple times, e.g. neighboring patch sets, neighboring tasks and ghost variables. A limitation of pure MPI scheduling is that tasks which are created and executed on the same node cannot share data. The new multi-threaded MPI scheduler described below solves this problem by dynamically assigning tasks to worker threads during execution and share the same infrastructure components between threads. The architecture of the runtime system has been extended to support multi-threaded execution. Compared to Uintah's dynamic MPI scheduler, the new multi-threaded MPI scheduler has one control thread and several worker threads per MPI process. The control thread holds all infrastructure components such as the regridder, the load balancer, the task graph and the data warehouse and has read and write access to them.

As the control thread is responsible for sending ready tasks to all worker threads, its efficiency is crucial for the performance of the whole code. If bottlenecks exists in the control thread, the worker threads may not able to get tasks in time and so will stay idle. This will cause the whole simulation process to slow down. In Uintah's multi-threaded MPI scheduler, the control thread is designed to be lightweight in order to provide very quick responses to each worker thread. In the implementation considered here, the control thread gives priority to assigning tasks to worker threads. Only when all ready tasks have been assigned, will the control thread then start to process task queues and received MPI messages. Also, a separate core is allocated for the control thread. This allows control thread to manage task queues and process MPI receives without undue delay.

The worker threads are designed to be easily manageable and only to execute tasks assigned by the control thread. Each worker thread has read-only access to all infrastructure components and also has write access to the data warehouse and the scheduler queues.

5.2 Control Thread

The control thread has two task queues (Figure 1): the internal ready queue and external ready queue. After the task graph is compiled, all the pre-satisfied tasks will be placed in the internal ready queue. The value of the counter for tracking outstanding MPI messages is set according to information provided by the task graph. When this counter reaches zero, the communication phase is complete and the task is ready to be executed. At that point it is placed in the external ready queue. When scheduling a task the scheduler chooses a task in the external ready queue based on a prioritization algorithm.

At the time when a task is being scheduled, the control thread will select an idle thread as a target thread and assign a task to it.

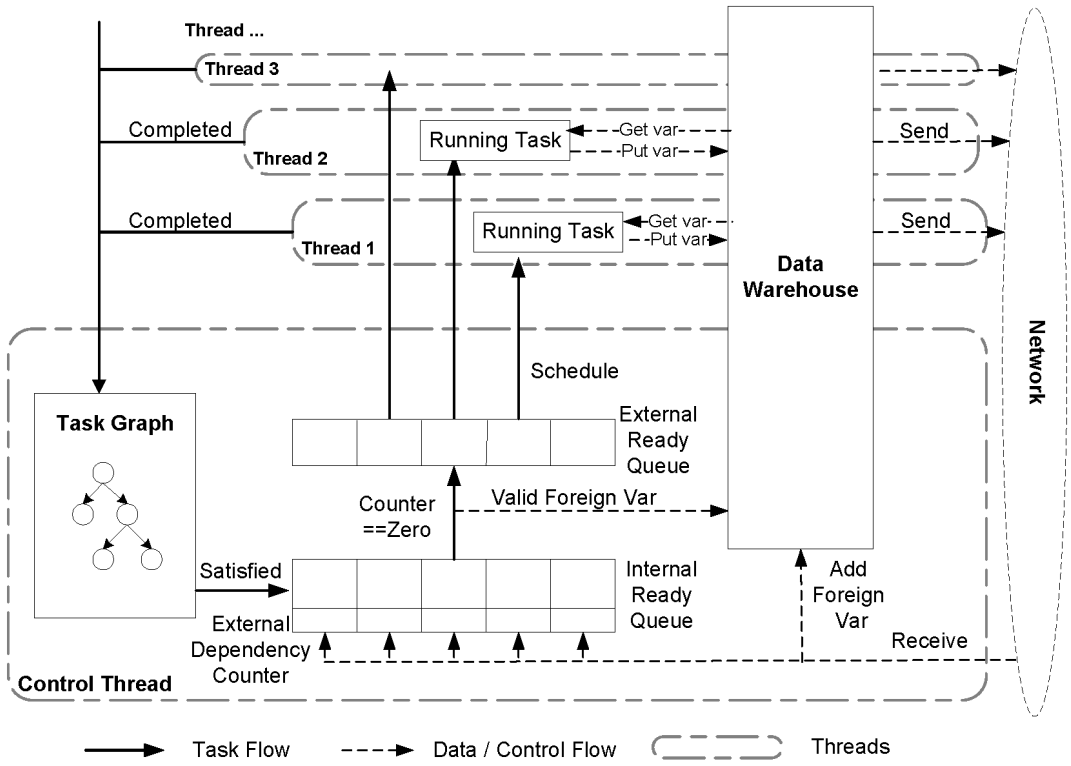

Figure 1: Architecture of Uintah hybrid task scheduling system

Algorithm 1 Control Thread

```
while doneTasks < totalTasks do
  if ReadyQ.Count () > 0 then
    if idleThreads.Count () = 0 then
      nextCondition.Wait()
    end if
    targetThread ← pickIdleThreadI()
    targetThread.task ← ReadyQ.pop()
    targetThread.runCondition.Signal()
    doneTasks++
  else
    if runnigThreads.Count () = 0 then
      receiveMPIs.WaitSome()
    else
      receiveMPIs.Test()
    end if
  end if
end while
```

After the assignment, the control thread will then wake up this target worker thread through the worker thread's condition variable signal. If all threads are busy, the control thread will block itself by waiting based on its own condition variable until a worker thread signals it. Since MPI receives and task dependencies are also processed by control thread, when the external ready queue is empty, the control thread will call MPI_Testsome or MPI_Waitsome to process or wait for incoming MPI messages. The control thread has three states: processing MPI receives and task dependencies, blocked while requesting idle thread and blocked in MPI wait. The algorithm for the control thread must be carefully designed to handle all these cases. A simplified version of control thread code is shown in Algorithm 1. The task external ready queue is referred to as ReadyQ here, and the internal ready queue is not shown.

5.3 Worker Thread

In Uintah's multi-threaded MPI scheduler, each worker thread has been made easily manageable in that its data structure only contains few variables for thread controlling and status recording. The scheduler will create a number of worker threads according to user's specification. When those threads are initialized, they will immediately block on their run conditions. Algorithm 2 shows the main loop of every worker thread. When a run signal is called by the control thread, the worker thread will be awoken and then start running tasks. After each task is executed, its MPI Sends will also be posted by worker thread. As the MPI send operation only requires read only access to the data warehouse, multiple messages can be sent out concurrently. The worker thread will then ask the control thread for its next task and block on a run condition again until the next task is assigned. If control thread is blocked, a signal from worker thread will wake the control thread up. When a task is completed, the worker thread will also check if any task's local dependencies are satisfied based on the task graph. Any newly satisfied task will be placed in the internal ready queue waiting for the

control thread's process.

Algorithm 2 Worker Thread

```
while !QUIT do
  runCondition.Wait()
  task.Run()
  task.SendMPIs()
  task ← EMPTY
  controlThread.nextCondition.Signal()
end while
```

5.4 Thread-safe Data Warehouse

As mentioned above, the core scheduler component that stores simulation variables is the data warehouse. The data warehouse is a hashed-map-based dictionary which maps variable name and patch id's to the memory address of a variable. Each task can get its read and write variable memory by querying the data warehouse with a variable name and a patch id. The task dependencies of the task graph guarantees that there are no memory conflicts on local variables access, while variable versioning guarantees that there are no memory conflicts on foreign variables access. These mechanisms have been implemented for supporting out-of-order task execution in our previous work using a dynamic MPI scheduler [18]. This means that a task's variable memory has already been isolated. Hence, no locks are needed for reads and writes on a task's variables memory.

However, the dictionary data itself still needs to be protected when a new variable is created or an old variable is no longer be needed by other tasks. As dictionary data must be consistent across the worker threads, the data warehouse has to be modified to be thread-safe by the addition of read-only and read-write locks. When a task needs to query the memory position of a variable, a read-only lock must be acquired before this operation is done. When a task needs the data warehouse to allocate a new variable, or to cleanup an old variable, a read-write lock must be acquired before this operation is done. while this increases the overhead of multi-thread scheduling, locking on dictionary data is still more efficient way than locking the all the variables.

5.5 Task Requirements

The Uintah scheduler ensures that no input and output variable conflicts will exist in any two simultaneously running tasks. This also greatly helps users to write thread-safe simulation components. In fact, all tasks in the ICE and AMRICE components are thread safe and can be supported by the multi-threaded scheduler without rewriting any task code. It is still possible, however, that some components are not thread safe even through all tasks' input and output are isolated. For example, when tasks reuse temporary static buffers which are allocated inside the task code, those tasks can not be executed concurrently. In order to enforce thgis will require a rewrite of some task code. We are still working to make more of Uintah's simulation components compatible with the new multi-threaded MPI scheduler.

5.6 Global Synchronization

In the approach proposed here, control threads may receive an MPI message and more than one worker thread may send MPI messages concurrently. The implication is that those MPI routines must be capable of being used by multiple threads. Many MPI libraries such as MPICH2 and OpenMPI already support thread-safety without the need for any user-provided thread locks. Once parallel environment setups up are done correctly, the point-to-point MPI communication interfaces do not require any changes. In the Uintah framework, this type of task, which only communicates with neighboring tasks, are called *Normal* tasks.

However, Uintah also support *Global* tasks that require the result of a global communication. Those global tasks are created when a task computes a global variable which needs to be updated through the whole grid, e.g., computation of the total mass of the system, or when a task calls a third party library which need the MPI communicator as an argument. e.g., calling PETSc. These global tasks will create one instance on each processor instead of one on each patch and need to be scheduled everywhere in the system at the same time. In the purely MPI scheduler, as no non-blocking reduction routines are provided, a synchronization phase is introduced to support scheduling global tasks. Tasks are divided into different phases in which each phase contains only one global task. The scheduler only executes the global task if all of the other tasks in its phase have completed and then moves to the next phase. In this way, global tasks will be execute in a fixed order.

When running in a multi-threaded environment, since many MPI collective communications can happen at the same time, the whole of the task schedule will not be blocked by a single global task. Hence the synchronization phase is removed in Uintah's multi-threaded MPI scheduler. However there is another problem that arises when scheduling this type of task in a multi-threaded MPI environment. At present there are no message tags in current MPI collective routines. One process may not able to process multiple MPI collective calls correctly as there are no message tags to distinguish them. In order to solve this problem, multiple copies of communicators are created. When scheduling, one communicator is assigned to each global task. This allows multiple global tasks to run at the same time safely without blocking or interfering with each other.

6. COMPUTATIONAL EXPERIMENTS

The aim of this section is to examine whether the hybrid memory version of Uintah reduces the memory requirement sufficiently for us to consider running larger problems. In what follows, measurements of the memory usage were obtained by the MallocTrace memory profiling library described in [16]. Two prototypical simulation studies were used to compare the hybrid multi-threaded/MPI approach versus the MPI approach. The two metrics that we looked at were memory usage and run time. The ICE algorithm was tested in both the single level and AMR case using a simulation of the transport of two fluids with a prescribed initial velocity of Mach two. For this problem, the conservation of mass, momentum, and energy equations were solved for two inviscid fluids. The fluids exchange momentum and heat through the exchange terms in the compressible Navier Stokes governing equations. This simulation is an explicit formulation and utilized the w-cycle execution model for time stepping in which proportionally smaller timesteps were used on adaptively refined mesh patches. This problem exercises all of the main features of ICE and amounts to solving eight partial differential equations, along with two point-wise solves, and one iterative solve [6,16]. This AMR ICE benchmark [16] involved three runs of varying sizes denoted by A,B and C. The refinement algorithm used tracked the interface between the two materials, causing the simulation to regrid often while maintaining a fairly constant sized grid, which allows the scalability to be more accurately measured. This criteria led to each problem being about four times as large as the previous one. All three runs are based on a same 3 level adaptive mesh problem but with different resolutions. Run A uses a 64x64x64 coarse level resolution and contains in total 26.8 million cells on all 3 levels of the refined mesh. Run B uses a 128x128x128

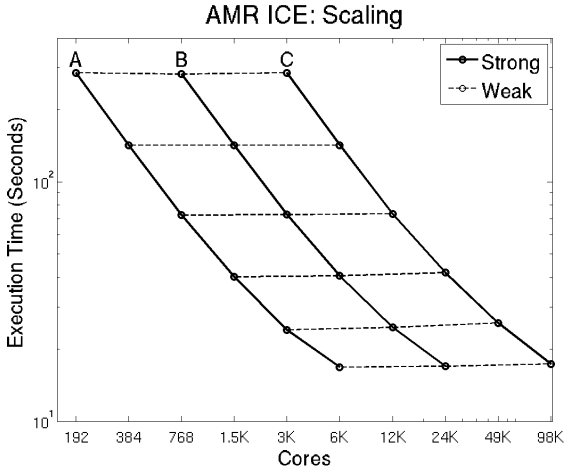

Figure 2: Scalability Results for Cases A,B and C.

coarse level resolution and contains 108 million cells. Run C uses a 256x256x256 coarse level resolution and contains 435 million cells. Thus the problem size of run C is thus about four times than run B and the problem size of run B is also about four times that of run A. The refinement ratio of all three runs is 4 to 1. Three sets of experiments were run on the 12 cores per node of Kraken: a pure MPI case, a case where the 12 cores were split into two times 6 threads and a third case of 12 threads per node. The results with 12 threads were slightly better, but not significantly different and so are reported here.

The scalability results for cases A,B, and C are shown in Figure 2. Weak scalability is represented by the almost horizontal lines and strong scalability by the almost straight diagonal lines in the three cases. These runs are similar to those reported in [6].

ICE Strong/Weak Runs Memory % Used Relative to MPI						
Weak	Strong	Run A	Strong	Run B	Strong	Run C
Run	Cores	%	Cores	%	Cores	%
1	192	50	768	60	3072	61
2	384	46	1536	53	6144	47
3	768	43	3072	44	12288	36
4	1536	34	6144	33	24576	27
5	3072	25	12288	24	49152	18
6	6144	19	24576	17	98304	11
ICE Strong/Weak Runs CPU % Used Relative to MPI						
Weak	Strong	Run A	Strong	Run B	Strong	Run C
Run	Cores	CPU%	Cores	CPU%	Cores	CPU%
1	192	85	768	85	3072	88
2	384	84	1536	85	6144	91
3	768	90	3072	90	12288	95
4	1536	86	6144	90	24576	97
5	3072	98	12288	99	49152	100
6	6144	107	24576	104	98304	101

Table 1: AMR ICE Relative Memory and CPU time with Hybrid Approach Compared to MPI.

Table 1 show the reductions in memory and the relative CPU times when using the hybrid approach. In the strong scaling cases, the memory saving increases when running with more cores. This follows from the analysis in the previous section, because for the same grid, the ghost cell data increases as a proportion of the total data when running with more cores, but a fixed size problem size, hence the saving is larger when the number cores increases. However the saving of CPU time decreases and some times slightly exceeds the pure MPI case when running with more cores. The reason is that most of CPU savings come from eliminating in-socket MPI communications. When the number of cores increases, the amount of in-socket MPI communication decreases. Hence the saving of CPU time decreases when running with more cores. The overhead due to the use of the threaded approach, principally that of locking on the data warehouse and other non-threaded components such as the load balancer will offset the savings from eliminating in-socket MPI communications.

For the weak scaling cases, the memory saving shows a slight increase when running with more cores. This is because the global meta-data increases as the number of core increases. Even through most of memory savings that come from reducing ghost cell data copies stays constant, the memory saving from reducing the number of copies of global meta-data increases, as shown in Section 4. Hence we can see more memory savings on large number of cores in weak scaling tests even though the ghost cell data per node is constant. In terms of CPU usage, as the in-socket MPI communication per node stays the same, the effect of switching to the hybrid approach is also roughly constant for these weak scaling cases.

Using the hybrid multi-threaded MPI scheduler, we have also been able to successfully run both the AMR problem and the non-AMR fluid structure interaction problem described in Sections 6 and 7 of [6] on as many as 196K cores on Jaguar, with good scaling results, due to the reduced memory requirement. These reductions in memory are illustrated by the two material CFD test problem from [16] used on Jaguar using 110K cores that could not have been previously run using due to memory constraints. This problem had a resolution of 2048^3 cells and 128^3 patches distributed amongst 110,592 cores on Jaguar. The overall memory use per node was reduced from 13.5 GB per node to 1GB per node (12 cores) when running the same size problem using the non-threaded MPI scheduler with 98K cores. Attempts were made to run this same problem on 110K cores with the MPI scheduler, but the problem size was too large and we ran out of memory on each node. The hybrid MPI/threaded approach thus allows us to consider problems that were previously out of our scope due to memory constraints.

7. FUTURE WORK

These preliminary results show great memory savings, and show great promise so far on 200K cores on Jaguar. However alongside these memory improvements, it is the case that further algorithmic improvements will be needed, particularly for fluid-structure interaction problems, for Uintah to be used routinely used with 200-300K cores. Achieving this level of fidelity and scalability is the next challenge of this PetaApps project. Although the present approach has introduced considerable memory savings, the global mesh meta-data will probably have to be revised in the future as the number of nodes, sockets, cores and hence mesh patches used keeps growing. Instead of every process currently knowing the extents of every patch and which processors own which patches, as the number of patches grow the size of this global meta-data will also grow and a hierarchical or local algorithm and data structure will probably need to be devised. The present model also has the potential advantage in that it may be possible to adapt it to make use of threads executing on GPUs for example. This will probably require considering the possible decomposition of individual tasks to take advantage of the GPUs architecture.

8. ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under subcontracts No. OCI0721659, the NSF OCI PetaApps program, through award OCI 0905068 and by DOE INCITE award CMB015 for time on Jaguar and DOE NETL for funding under NET DE-EE0004449, 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 all those previously involved with Uintah and Justin Luitjens in particular.

9. REFERENCES

[1] A.Almgren, J.Bell, D.Kasen, M.Lijewski, A.Nonaka, P.Nugent, C. Rendlemano, R. Thomas, and M. Zingale. MAESTRO, CASTRO and SEDONA - petascale codes for astrophysical applications. submitted for publication also arXiv:1008:2801v1, July 2010.

[2] Shameen Akhter and Jason Roberts. *Multi-core Programming*. Intel Press, 2006.

[3] A. Ali, L.Johnsson, and J.Subhlok. Scheduling fft computations on smp and multicore systems. In *ICS 07 Conference*, New York, NY, USA, 2007. ACM.

[4] P. Balaji, A. Chan, and E. Lusk W. Gropp, R. Thakur. Non-data-communication overheads in MPI: analysis on Blue Gene/P. In *Proc. of the 15th Euro. PVM/MPI Users' Group Meeting on Recent Advances in PVM and MPI*, pages 13–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, P. Kogge (Editor, Study Lead) nd S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report TR-2008-13, Department of Computer Science, Notre Dame University, 2008.

[6] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C.A. Wight, and J.R. Peterson. Uintah - a scalable framework for hazard analysis. In *TG '10: Proc. of 2010 TeraGrid Conference*, New York, NY, USA, 2010. ACM.

[7] B.Fryxell, K.Olson, P.Ricker, F.X.Timmes, M.Zingale, D.Q.Lamb, P.Macneice, R.Rosner, J.W. Rosner, J.W. Truran, and H.Tufo. FLASH an adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *The Astrophysical Journal Supplement Series*, 131:273–334, November 2000.

[8] C. Burstedde, L. C. Wilcox, and O. Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[9] J. D. de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson. Uintah: A massively parallel problem solving environment. In *Ninth IEEE International Symposium on High Performance and Distributed Computing*, pages 33–41. IEEE, Piscataway, NJ, November 2000.

[10] G.Bronevetsky, J.Gyllenhaal, and B. de Supinski. CLOMP: Accurately characterizing openmp application overheads. In R. Eigenmann and B. de Supinski, editors, *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 13–25. Springer Berlin / Heidelberg, 2008.

[11] T. Goodale, G.Allen, G.Lanfermann, J.Masso, T. Radke, E.Seidel, and J.Shalf. The Cactus framework and toolkit: Design and applications. In *Vector and Parallel Processing VECPAR 2002*, Lecture Notes in Computer Science, Berlin, 2003. Springer.

[12] 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.

[13] 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.

[14] B. A. Kashiwa. A multifield model and method for fluid-structure interaction dynamics. Technical Report LA-UR-01-1136, Los Alamos National Laboratory, Los Alamos, 2001.

[15] G. Krishnamoorthy, S. Borodai, R. Rawat, J. P. Spinti, and P. J. Smith. Numerical modeling of radiative heat transfer in pool fire simulations. In *ASME 2005 International Mechanical Engineering Congress (IMECE2005)*, November 2005.

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

[17] J. Luitjens, M. Berzins, and T. Henderson. Parallel space-filling curve generation through sorting. *Concurr. Comput. : Pract. Exper.*, 19(10):1387–1402, 2007.

[18] 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.

[19] B. O'Shea, G. Bryan, J. Bordner, M. Norman, T. Abel, R. Harkness, and A. Kritsuk. Introducing Enzo, an amr cosmology application. In *Adaptive Mesh Refinement - Theory and Applications*, volume 41 of *Lecture Notes in Computational Science and Engineering*, pages 341–350, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] C.D. Ott, E. Schnetter, G. Allen, E. Seidel, J. Tao, and B. Zink. A case study for petascale applications in astrophysics: simulating gamma-ray bursts. In *Proc. of the 15th ACM Mardi Gras conference: From lightweight mash-ups to lambda grids.*, MG '08, pages 18:1–18:9, New York, NY, USA, 2008. ACM.

[21] S. G. Parker. A component-based architecture for parallel multi-physics PDE simulation. *Future Generation Comput. Sys.*, 22:204–216, 2006.

[22] R.Rawat, J. Spinti, W.Yee, and P.J. Smith. Parallelization of a large scale hydrocarbon pool fire in the Uintah PSE. In *ASME 2002 International Mechanical Engineering Congress and Exposition (IMECE2002)*, pages 49–55, November 2002.

[23] H-Y. Schive, U.-H. Zhang, and T.Chiueh. Directionally unsplit hydrodynamic schemes with hybrid MPI/OpenMP/GPU parallelization in amr. 2011.

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

[25] J. Tao, G. Allen, I. Hinder, E. Schnetter, and Y. Zlochower. XIREL:standard benchmarks for numericla relativity codes using Cactus and Carpet. Technical Report CCT-TR-2008-5, Center for Computationa and Technology, Louisiana State University, Baton Rouge, Louisiana, 2008.