The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System

Qingyu Meng
Scientific Computing and
Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
Email: qymeng@sci.utah.edu

Alan Humphrey
Scientific Computing and
Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
Email: ahumphrey@sci.utah.edu

Martin Berzins
Scientific Computing and
Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA
Email: mb@sci.utah.edu

Abstract—**The development of a new unified, multi-threaded runtime system for the execution of asynchronous tasks on heterogeneous systems is described in this work. These asynchronous tasks arise from the Uintah framework, which was developed to provide an environment for solving a broad class of fluid-structure interaction problems on structured adaptive grids. Uintah has a clear separation between its MPI-free user-coded tasks and its runtime system that ensures these tasks execute efficiently. This separation also allows for complete isolation of the application developer from the complexities involved with the parallelism Uintah provides. While we have designed scalable runtime systems for large CPU core counts, the emergence of heterogeneous systems, with additional on-node accelerators and co-processors presents additional design challenges in terms of effectively utilizing all computational resources on-node and managing multiple levels of parallelism. Our work addresses these challenges for Uintah by the development of new hybrid runtime system and Unified multi-threaded MPI task scheduler, enabling Uintah to fully exploit current and emerging architectures with support for asynchronous, out-of-order scheduling of both CPU and GPU computational tasks. This design coupled with an approach that uses MPI to communicate between nodes, a shared memory model on-node and the use of novel lock-free data structures, has made it possible for Uintah to achieve excellent scalability for challenging fluid-structure problems using adaptive mesh refinement on as many as 256K cores on the DoE Jaguar XK6 system. This design has also demonstrated an ability to run capability jobs on the heterogeneous systems, Keeneland and TitanDev. In this work, the evolution of Uintah and its runtime system is examined in the context of our new Unified multi-threaded scheduler design. The performance of the Unified scheduler is also tested against previous Uintah scheduler and runtime designs over a range of processor core and GPU counts.**

I. Introduction

An important trend in high performance computing is the planning and design of software framework architectures for emerging and future systems with multi-petaflop and eventually exaflop performance [1]. Such frameworks must address the formidable scalability and performance challenges associated with running on these systems, and must also insulate application developers from the inherent complexity of the parallelism involved. Traditional systems are now commonly augmented with graphics processing units (GPUs), and will soon be subject to the availability of other co-processor designs such as the Intel Xeon Phi [2]. Software framework designs must consider these heterogeneous architectures and additionally plan for future many-core designs.

In its near 15 year history, the Uintah Computational Framework (UCF) has continually addressed and overcome these challenges. Originally Uintah used an MPI-only approach to scheduling of computational tasks using an out-of-order execution model [3] that worked well for many cases, recently scaling to near 100K [4] cores on challenging fluid-mechanics problems. One limitation of its pure MPI scheduling is that tasks which are created and executed on different cores on the same node cannot share data.

A multi-threaded MPI scheduler was then used [1] 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 and worker threads communicated through Pthread conditional variables. Experimental results on challenging simulations using Adaptive Mesh Refinement (AMR) showed a 50% to 90% savings on memory usage by using the multi-threaded scheduler. This scheduler was then extended to heterogeneous systems [5], able to dispatch work to both CPU cores and GPUs on-node. This approach still did not result in the scalable execution of the target fluid-structure problem beyond 200K cores however.

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 numbers of cores, as there was not enough work for the control thread. When running with larger numbers of cores per node there is an increase in master control thread CPU usage with increasing numbers of worker threads. This made it difficult to balance the control thread workload with the worker threads and it has been observed that the master-slave model hits a control thread bottle neck as the number of cores per node increases. We has addressed this issue with the design of the Unified scheduler, presented here. This new scheduler allows all threads to process MPI sends and receives or to execute tasks concurrently without a control thread. In contrast to the master slave model, the decentralized scheduler is able to fully utilize all available cores on-node, regardless of the number of cores and can also fully utilize one or more GPUs available on-node. This design seeks to not only maximize node-level parallelism in current architectures, but also in future, many-core architectures with the looming prospect of less memory per core. This design remains broad enough to support other co-processor designs, such as the Intel Xeon Phi [2].

In this work, we examine the evolution of Uintah's hybrid multi-threaded MPI runtime system [1] to support, schedule and execute both CPU and GPU tasks simultaneously, without a central control thread. In what follows Section 2 provides an overview of the Uintah software, while Section 3 covers the history of Uintah task schedulers and their respective designs and ultimate scalability and performance barriers. Section 4 details the current Unified multi-threaded runtime system and how this design is well-suited for both current and emerging HPC architectures. In Section 5, we describe computational experiments that illustrate the effectiveness and performance of the Unified scheduler. This paper concludes by describing future work in this area.

II. Overview of Uintah Software [5]

The Uintah Software was originally written as part of the University of Utah Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [6]. C-SAFE, a Department of Energy ASC center, focused on providing science-based tools for the numerical simulation of accidental fires and explosions. The aim of Uintah was to be able to solve complex multi-scale multi-physics problems. Uintah is regularly released as open source software [7]. Uintah is novel in its use of a asynchronous, task-based paradigm, with complete isolation of the application developer from parallelism. The individual tasks are viewed as part of a directed acyclic graph (DAG) and are executed adaptively, asynchronously and often out of order [3]. Uintah uses a novel adaptive meshing approach [4] as well as a variety of fixed mesh and particle solution methods.

In order to solve complex multi-scale multi-physics problems, Uintah makes use of a component design that enforces separation between large entities of software that 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 [8]. The Uintah component approach allows the application developer 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, GPU memory operations or notions of parallelization and load balancing. This approach also allows the developers of the underlying parallel infrastructure to focus on scalability concerns including load balancing, task scheduling, component switching 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 application developer.

Uintah currently contains four main simulation algorithms, or components: the ICE compressible multi-material Computational Fluid Dynamics (CFD formulation, the particle-based Material Point Method (MPM) for structural mechanics the combined fluid-structure interaction algorithm MPMICE [9], and the ARCHES combustion simulation component.

III. History and Survey of Uintah Task Schedulers and Runtime Systems

Uintah components define abstract but connected tasks on a generic grid patch. The connections between these tasks are defined by the variables that the tasks require and compute on the patch and its ghost cells. During the simulation, these tasks are created on the patches of a continually adapting grid and then mapped by Uintah onto the parallel machine. These tasks are the fundamental unit of work within Uintah and associated with each task is a C++ method which is used to perform the actual computation. This C++ method is implemented by the component developer and represents the serial algorithm to run on each patch. User tasks are shielded from the underlying parallelism being managed by Uintah itself, and in the context our new Unified scheduler, there are multiple levels of heterogeneous parallelism Uintah must manage (MPI, Pthreads and Nvidia CUDA). Without any knowledge of the underlying parallelism, the component developer need only register a particular task with the Uintah infrastructure and specify what variables the task will require and compute. The separation of these tasks from the runtime system is managed by the data warehouse along with a task scheduler and its associated data structures. The data warehouse is a dictionary based data structure, that manages all Uintah variables. A task can use a variable name and a patch ID key to load and save variables into the data warehouse. The data warehouse also manages MPI message buffers and automatically garbage collects variables when they are no longer needed. The Uintah 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 [10]. It also ensures that no input or output variable conflicts will exist in any two simultaneously running tasks. Uintah originally used a simple static MPI scheduler in which tasks were executed from a pre-determined list that was solely computed from a task graph's critical path. A limitation of this scheduler is that a single task waiting for messages caused the calculation on a particular core to sit idle.

The simulation grid in Uintah is partitioned into patches by a highly scalable regridder and assigned to nodes by a measurement-based load-balancer [10]. In each MPI process, the Uintah runtime system will schedule the tasks on local patches by using a local task graph and the data warehouse.

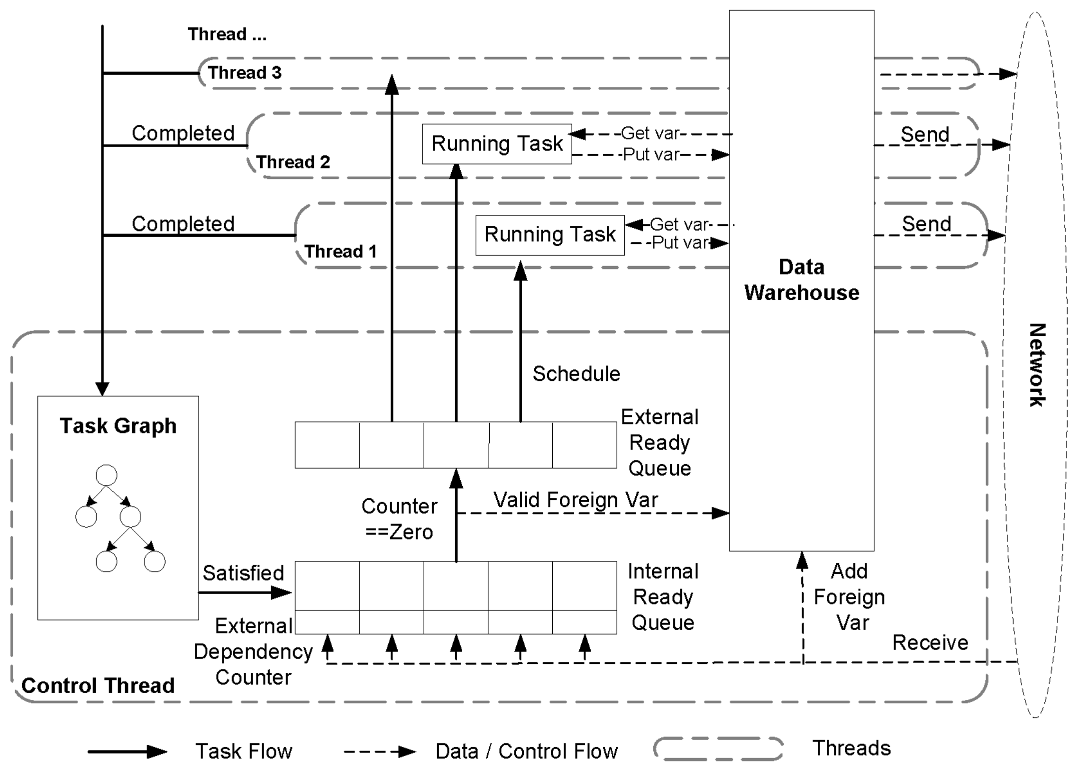

Fig. 1. Uintah CPU Task Scheduler: Master-Slave Model [1]

The task graph is a directed acyclic graph (DAG) [11] which is compiled by making connections on task's required and computed variables.

To address the scalability and performance challenges presented with each successive generation of machine and new architecture, many Uintah schedulers have been developed. The following subsections explore this design history in detail.

A. Dynamic MPI Scheduler [3]

In order to reduce the CPU idle time due to the wait for MPI communications, Uintah needed to move away from its decade-old static scheduler. Although the overlapping of communications had been used to compute this task list, the uncertainty of MPI message arrival had begun to increase on larger machines. In order to address this issue, a new Uintah dynamic scheduler was developed in [3] to better overlap communication and computation by using out-of-order task execution. A task was then allowed to execute once its required(input) variables were available. The static task list was replaced by multi-stage task queues to allow the scheduler to keep track of the different input availability states of tasks. An internal ready queue stored tasks whose required variables from local tasks were available. An external ready queue stored tasks whose foreign required variables from MPI messages were also available. As long as the external ready queue was not empty, the CPU could always execute tasks to overlap communications. When a task was running out-of-order, multiple versions of a variable may exist at the same time. A data warehouse variable versioning system was implemented to ensure correct memory access. Several runtime task priority algorithms [3] were also designed and tested to further improve scheduler performance. This scheduler significantly reduced the MPI wait time by 40% to 60%, allowing Uintah to scale to 98K cores on the NSF Kraken system and also improved the over all performance of Uintah [3].

B. Multi-threaded CPU Scheduler (Master-Slave Model) [1]

While the out-of-order execution model worked well for many cases, one limitation of its pure MPI scheduling is that variables have to be passed through MPI messages and copied to another process' memory even if the source and destination tasks were on the same multi-core node. A new multi-threaded MPI scheduler (Figure 1, *from [1]*) was designed in [1] to eliminate intra-node MPI messages and memory copies by adopting a shared memory model on-node. This was realized by creating multiple worker threads on the same multi-core node. In this way, tasks running on different cores could directly access all variables on the same node.

This multi-threaded MPI scheduler had one control thread and several worker threads per MPI process, which communicated through Pthread conditional variables. The control thread processed MPI receives, managed tasks queues and assigned ready tasks to worker threads. The worker thread simply executed the task that the control thread assigned to it. As Uintah variables can be accessed freely by any thread, many shared data structures, such as data warehouse and task queues, were redesigned to guarantee thread-safety. Experimental results [1] on typical fluid AMR simulations showed 50% to 90% savings on memory usage. This new multi-threaded MPI scheduler enabled Uintah to scale up to

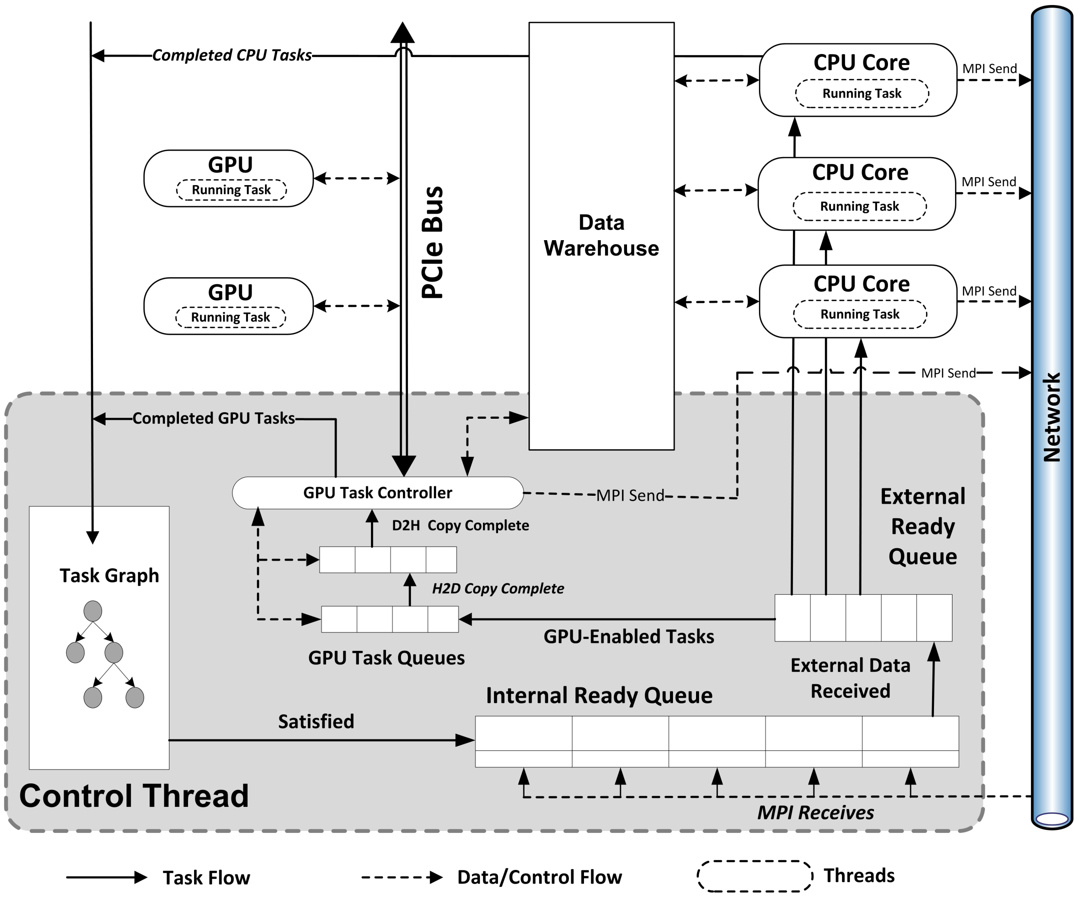

Fig. 2. Uintah CPU-GPU Task Scheduler: Master-Slave Model [5]

196K cores on the DoE Jaguar XT5 system and became the basis for the heterogeneous multi-threaded MPI scheduler [5] which allowed Uintah to dispatch tasks to GPUs as well as CPU cores on a node.

C. Multi-threaded CPU-GPU Scheduler (Master-Slave Model) [5]

In the same fashion that Uintah insulates the application developer from the parallelism its infrastructure provides via the multi-threaded CPU scheduler, the hybrid CPU-GPU version also hides and carefully manages details related to GPU memory allocation and transfer. Associated with each Uintah task is a C++ method which is used to perform the actual computation. In the context of the hybrid CPU-GPU scheduler, a GPU task is represented by an additional C++ method that is used for GPU kernel setup and invocation. This design uses Nvidia CUDA C/C++ exclusively for both the Uintah infrastructure and user GPU tasks.

Central to the master-slave design of the hybrid multi-threaded CPU-GPU scheduler (Figure 2, *from [5]*) is the multi-stage queuing architecture for efficient scheduling of CPU and GPU tasks. The CPU-GPU scheduler utilized four task queues: an internal ready and external ready queue for CPU tasks and an additional pair of queues for the GPU; one for initially ready GPU tasks; those that have their requisite simulation variable data copies from host-to-device pending, and a second for the corresponding device-to-host data copies pending completion. It should be noted that both GPU task queues are priority queues and thus preserve a given task priority algorithm established by the scheduler itself.

The hybrid CPU-GPU scheduler also maintained a set of queues for CUDA *stream* and *event* handles (one per device representing separate CUDA contexts for each), and assigned them to each simulation variable per time step to overlap with other host-to-device memory copies as well as kernel execution [5]. These *stream* and *event* handles provide a mechanism to detect completion of asynchronous memory copies without a busy wait, using `cudaEventQuery(event)`. This allows querying the status of all device work preceding the most recent CUDA 4.0 API call to `cudaEventRecord()` [12]. On systems with multiple on-node GPUs, the hybrid CPU-GPU scheduler must additionally manage a CUDA calling context for each device.

First, if a task's internal dependencies were satisfied, then that task was placed in the CPU internal ready queue where it waited until all required MPI communication had finished. In this same step, if the task was GPU-enabled, the task was then put into the host-to-device copy queue for advancement toward execution. As long as the CPU external queue was not empty, there were always tasks to run. Execution of a task took place on the first available CPU core or GPU and the scheduler resided on a single, dedicated core per node.

CPU tasks were dispatched by the control thread to available CPU cores when they signaled the need for work. GPU tasks were assigned in a round-robin fashion to available GPUs on-node once their asynchronous host-to-device data copies had completed. This design helped to overlap MPI communication and asynchronous GPU data transfers with CPU and GPU task execution, significantly reducing MPI wait times [5].

Ultimately, the GPU task went to the pending device-to-host copies queue. A GPU-enabled task in most cases has several computed Uintah variables to return from the device to the host. The device-to-host copies queue was where tasks resided while waiting for these operations to complete. Upon completion of these data transfers, the task was marked as completed and its MPI sends were posted. Finally the GPU task was removed from the pending device-to-host copies queue, allowing other dependent tasks to proceed.

D. Multi-threaded CPU Scheduler (Decentralized Model) [13]

A potential bottleneck in the centralized, master-slave model was that a worker thread might become idle if the control thread could not respond to its next ready task request quickly enough. In the presence of additional GPU tasks, this could potentially be even more pronounced as the control thread must also coordinate data transfers to and from the GPU and also manage the additional GPU task queues. To better facilitate a quick response, the control thread was assigned to a dedicated core. However, this ultimately led to the control thread core being under-utilized. The solution adopted in [13] was the design of a new decentralized multi-threaded scheduler, eliminating the central control thread, thus allowing all threads to process MPI sends and receives and also to execute tasks freely and concurrently without using a control thread [13]. Instead of requesting a ready task from the control thread, all threads in the decentralized multi-threaded scheduler can directly pull tasks from the one of two ready queues. When a thread pulls a task from the internal ready queue, non-blocking MPI receives are then posted. When a thread pulls a task from the external ready queue, the call-back function for that task is executed after which its MPI sends are posted. The decentralized CPU scheduler was then able to fully utilize all available cores on-node, regardless of the number of cores and outperformed the previous master-slave model. It was also confirmed that the de-centralized model solved the issue of under utilization of the control thread core.

The multi-threaded scheduler originally used locking to protect shared data structures. This overhead increases with the number of cores per node as contention for acquiring locks also increases. Based on our timing results on Uintah read-write locks, the data warehouse lock was seen to be the largest single source of overhead. With this overhead in mind, novel lock-free data structures and algorithms using atomic instructions were designed to replace the use of high-level, heavy-weight Pthread locks on frequently accessed data structures [13].

IV. A New Unified Runtime System (Decentralized CPU-GPU Model)

The natural design progression given the success of the original CPU-GPU scheduler and the generally superior performance and potential of the decentralized model, was to extend the decentralized CPU design to heterogeneous systems, allowing all threads to process MPI sends and receives and to execute both CPU and GPU tasks concurrently without a control thread. Through this design extension, a unified multi-threaded runtime system and approach to scheduling Uintah computational tasks has been developed. The Unified Scheduler and runtime system is the principal contribution in this work and allows Uintah to not only exploit current heterogeneous architectures, but also plans for emerging and future many-core designs. Much of this design path has been motivated by machines such as NSF Keeneland and the upcoming DoE Titan and NSF Stampede systems. As mentioned in [5], to adapt the Uintah Computational Framework for hybrid CPU-GPU architectures, we elected to use Nvidia CUDA C/C++ for numerous reasons, namely looking at the upgrade path of the DoE Jaguar XK6 system to Titan [14] and also the Keeneland Initial Delivery System (KIDS) [15], we see a trend in the use or planned use of Nvidia GPUs.

Adding GPU capability to a decentralized multi-threaded model presents several notable challenges. As stated earlier, all threads in the decentralized multi-threaded model can directly pull tasks from task queues, not solely the control thread, thus creating potential race conditions on all shared data structures in general, but specifically in the task queues. Within the Unified scheduler and runtime system there are now four total task queues; two queues for staging CPU tasks and a corresponding pair for GPU tasks, all of which must be thread-safe. Individual access to the GPU queues is relatively infrequent, and more often a read than a write, hence multiple reader, single writer synchronization primitives are used to protect access and minimize lock overhead.

In the same way that access to CPU-only task data in the *data warehouse* must be guaranteed to be thread-safe, access to the current data structures that track corresponding GPU data must be similarly protected. As described in [5], before a GPU task is placed into the GPU host-to-device copy queue the Unified scheduler initiates the device memory allocations and asynchronous host-to-device data copies for the task's simulation variables. To carry out these operations, the data warehouse must be queried by the Unified scheduler for the location and size of the data required for computation on the GPU. It is here that space in the data warehouse for the result of the GPU computation is also allocated on the host. These operations produce sets of pointers to device and host memory for both a task's requires(input) and computes(output) variables that must be managed. Additionally, host memory pointers are registered by the Unified scheduler to be copied to the GPU via DMA using a call to `cudaHostRegister()` combined with the `cudaHostRegisterPortable` flag from the CUDA 4.1 API. This creates page-locked memory from pre-allocated host memory that is considered page-locked by all CUDA contexts and ultimately accelerates PCIe transfers and eliminates resetting of CUDA contexts when referencing the registered host memory [5]. This information must also be tracked in order to cleanly unregister the page-locked host memory when a task has completed. All of this pointer information is kept in a set of maps maintained by the Unified scheduler. Access to each of these maps must also be guaranteed thread safe. Here, access to these data structures is currently infrequent as the overall number of GPU-enabled Uintah tasks is relatively low. Hence access to the maps can be regulated by the same read-write locks used in the task queues without significant overhead. However, as more Uintah tasks are ported to the GPU, this could become a potential bottleneck. This issue, should it arise, will be addressed through the creation of a GPU data warehouse that encapsulates these maps and uses the same novel lock-free data structures and algorithms used in the current Uintah data warehouse to eliminate the heavier-weight Pthread locks.

In addition to computational tasks, the Uintah task-graph also consists of global tasks that require the result of MPI collective operations. Third party library tasks that ”hijack” the Uintah framework to do their own MPI communication are also global tasks. As the current MPI standard does not provide non-blocking collective operations, these global tasks need to be scheduled at the same time to proceed without a load imbalance. This load imbalance occurs when nodes choose different paths before executing a global synchronization task, as they need to synchronize at that particular global task. So if a particular node has completed more tasks than another, the thread running the global task in the node with fewer completed tasks stays idle, hence a load imbalance is observed. To solve this problem, tasks are divided into different phases. Each phase contains only one global task and this task is only scheduled if all other tasks in its phase have completed. In this way, we can minimize the blocking time in global tasks and reduce synchronization load imbalance. The addition of GPU tasks and the associated logic involved with processing GPU tasks and task queues has introduced additional challenges with regard to global Uintah tasks. Existing logic has been reorganized and further logic has been added in the Unified scheduler to ensure scheduling of a given global task remains delayed until both CPU and GPU tasks in its phase have completed.

The run method for each thread also exposes a potential performance bottleneck in that the Unified scheduler contains a critical section that is protected by Mutex, a Pthread mutual exclusion primitive called the *scheduler lock*. This critical section contains numerous choices for work that a particular thread may choose from. Thus for any given thread, the duration between acquiring and releasing the *scheduler lock* must be as short as possible or risk a serialization point. With the addition of the GPU task queues, the number of places to poll for work in this section has now increased. The Unified scheduler addresses this issue with the simple use of a set of flags, one of which will be set for a thread that holds the *scheduler lock*, after which the lock is promptly released. The set flag dictates what work the thread will do concurrently with other threads beyond the critical section.

Preliminary results have confirmed that Uintah's new Unified scheduler and runtime system demonstrate an ability to effectively and efficiently utilize all available computational resources on-node, even on heterogeneous systems and also outperforms the previous master-slave model. This design also proves a promising direction for future many-core architectures with high core counts per node and the prospect of diminishing amounts of memory per core.

V. Improvement and Results

In evaluating the relative performance improvements of the Unified scheduler, several initial tests were performed. The first test looks at CPU only data from a single 32-core Cray XE6 node and compares execution times of the CPU-only master-slave model [1] to the new Unified scheduler. The second test looks at data from a single 12-core, 3-GPU heterogeneous node, comparing execution times of the hybrid CPU-GPU master-slave model [5] to the new Unified scheduler. Lastly we plot scaling data from runs on the DoE Jaguar system (CPU-only) and compare Uintah's MPI-only scheduler to its multi-threaded schedulers (master-slave model). These plots also include data from TitanDev[1], comparing GPU and CPU implementations of the RMCRT problem from [5].

Number of Cores	2	4	8	16	32
Master-Slave	57.28	20.72	9.40	4.81	2.95
Unified	29.79	15.70	8.23	4.54	2.78

TABLE I
EXECUTION TIME: CPU-ONLY MASTER-SLAVE VS UNIFIED

Table I shows a CPU only performance comparison between the master-slave and Unified models on a single Cray XE6 node (two 16-core AMD Opteron 6200 Series processors each with Interlagos cores @2.6GHz) for a combined MPMICE problem using AMR. In this case the Unified model outperforms the master-slave model on all runs up to 32 cores. By monitoring the CPU utilization of each core, it was confirmed that the unified model solved the issue of load imbalance of the core that runs the control thread and the cores running the worker threads. Furthermore, when one master control thread with 1, 3, 7, 15 and 31 worker threads are used, the CPU loads on the control thread increases linearly and are about 0.3%, 0.7% 1.7% 3.0% and 6.9% respectively. There is an increase in master control thread CPU usage with increasing numbers of worker threads.

Table II shows a hybrid CPU-GPU performance comparison between the master-slave and unified models on a 12-core heterogeneous node (two Intel Xeon X5650 processors each

[1]TitanDev is a 960 node partition on the DoE supercomputer Jaguar, available during its upgrade to Titan in late 2012. Each node contains a single 16-core AMD Opteron 6200 Series (Interlagos cores @2.6GHz) processor on one of its two sockets, the second socket contains a single Nvidia Tesla 20-series GPU, for a total of 15,360 CPU cores and 960 GPUs.

with Westmere 6-core @2.67GHz, 2 Nvidia Tesla C2070 GPUs and 1 Nvidia GeForce 570 GTX GPU) for the GPU-enabled Reverse Monte Carlo Ray Tracer (RMCRT) presented in [5] with 25 rays per cell and a problem size of 41^3. This is the benchmark problem from [16]. In this case the Unified model outperforms the master-slave model on all runs up to 12 cores. These results also confirm that the performance bottleneck found in the de-centralized model is even more pronounced in the presence of additional GPU tasks, with performance when using 2 and 12 threads respectively being 16% to 37% faster for the Unified model for this problem.

Number of Cores	2	4	6	8	10	12
Master-Slave	4.55	4.09	3.95	3.68	3.64	3.34
Unified	3.82	3.52	3.09	2.90	2.50	2.09

TABLE II
EXECUTION TIME: CPU-GPU MASTER-SLAVE VS UNIFIED

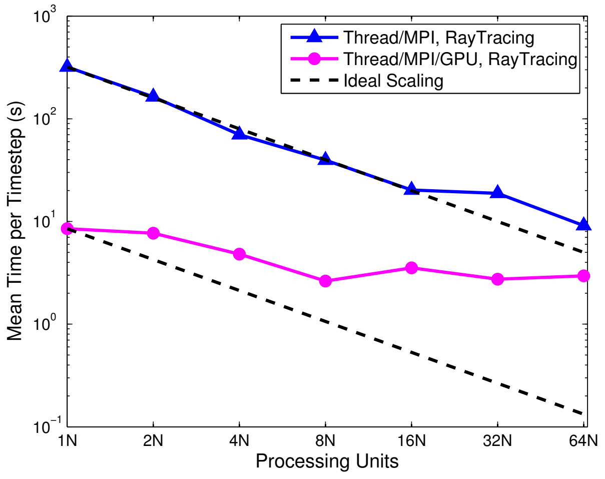

Fig. 4. Uintah GPU/CPU Execution Time (Thread/MPI RayTracing: N=16 CPU cores, Largest=1024 CPU cores; Thread/MPI/GPU RayTracing: N=16 CPU and 1 GPU, Largest=1024 CPU and 64 GPU)

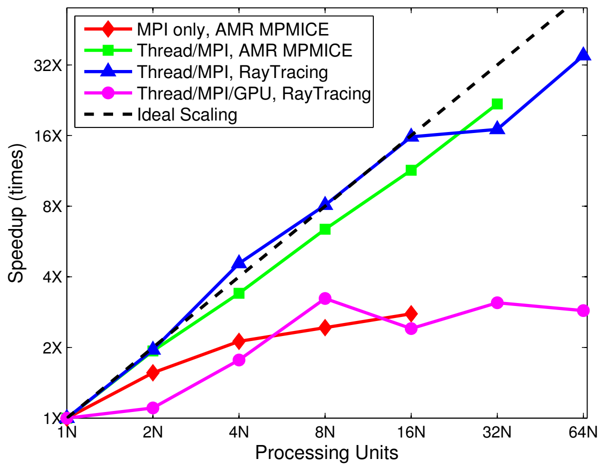

Fig. 3. Uintah Scaling Overview (MPI only AMR MPMICE: N=6144 CPU cores, Largest = 98K CPU cores; Thread/MPI AMR MPMICE: N=8192 CPU cores, Largest=256K CPU cores; Thread/MPI RayTracing: N=16 CPU cores, Largest=1024 CPU cores; Thread/MPI/GPU RayTracing: N=16 CPU and 1 GPU, Largest=1024 CPU and 64 GPU)

Figure 3 shows the Uintah strong scaling results when using MPI-only, multi-threaded MPI and multi-threaded MPI with GPU schedulers on two different problems: AMR MPMICE and RMCRT Raytracing. For a large-scale MPMICE AMR problem, Uintah originally scaled up to 96K CPU cores with MPI only on the DoE Jaguar XK6 system. By using the multi-threaded MPI scheduler (decentralized), Uintah can achieve significantly better scalability, up to 256K CPU cores on Jaguar. This simulation used 3.62 billion particles with three refinement grid levels. For the GPU-enabled Reverse Monte Carlo Ray Tracer (RMCRT) problem, 100 rays per cell were used with a problem size of 128^3.

Figure 4 isolates the CPU vs GPU scaling results in an effort to better clarify the scaling breakdown in the GPU implementation of the RMCRT problem. Although the mean time per timestep for the GPU implementation is still considerably lower than the CPU implementation at this point (up to 64 GPUs), ultimately there is insufficient work, and the GPU implementation is subject to the same communication costs as the CPU implementation [5] due to the all-to-all nature involved with radiation modeling.

CONCLUSIONS AND FUTURE WORK

In this paper, we have covered the history and evolution of Uintah in the context of its task schedulers and runtime systems, all leading up to the development of the Unified heterogeneous task scheduler and runtime system described in this work. We have shown that our Unified multi-threaded scheduler design is capable of utilizing all on-node computational resources on current and emerging multi-core and heterogeneous systems efficiently and automatically. This work has also illustrated how our Unified design keeps the application developer insulated from the multiple levels of parallelism inherent in heterogeneous systems by a separation of the user implemented tasks from the Uintah runtime system. We have also shown preliminary results that confirm the decentralized multi-threaded design used in the Unified scheduler not only outperforms previous designs, but is also well positioned to efficiently exploit emerging and future many-core architectures.

Through the development of Unitah's Unified scheduler, the data warehouse lock was seen to be the largest single source of overhead based on timing results on Uintah read-write locks, and in the way the data warehouse has been made efficient with a lock-free implementation, we are also considering an efficient, lock-free GPU data warehouse. Additionally, we would like to pursue designing a mechanism for the Unified scheduler to decide at runtime whether to run a particular task on a CPU core or on a GPU.

In the near future, we will be using early access to Intel

Xeon Phi [2] with plans to extend Uintah's scheduler to support such co-processor designs as well. And, with the eminent arrival of the massive-scale heterogenous DoE Titan in late 2012, larger scaling runs to further test our Unified scheduler and runtime design will also be performed. Given that Titan will potentially have 10,000 or more Nvidia Kepler K20 GPUs, we will also be leveraging the advanced features available through CUDA 5.0 and Kepler, specifically Dynamic Parallelism to further improve GPU utilization by the Uintah framework.

Acknowledgments

This work was supported by the National Science Foundation under subcontracts No. OCI0721659, the NSF OCI PetaApps program, through award OCI 0905068, by DOE INCITE award CMB015 and Director's Discretionary Allocation CMB021 for time on Jaguar and DOE NETL for funding under NET DE-EE0004449. Uintah was written by the University of Utahs Center for the Simulation of Accidental Fires and Explosions (C-SAFE) and funded by the Department of Energy, subcontract No. B524196. This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735. This research also used resources from the TitanDev program, the 960-node partition of the Cray XK6 Jaguar system outfitted with Nvidia GPUs located at Oak Ridge National Laboratory (ORNL). We would like to thank the Resource Utilization Council at the Oak Ridge Leadership Computing Facility for granting our project early access to TitanDev resources. We would also like to thank all those previously involved with Uintah, Justin Luitjens in particular.

References

[1] Q. Meng, M. Berzins, and J. Schmidt, "Using Hybrid Parallelism to Improve Memory Use in the Uintah Framework," in *Proc. of the 2011 TeraGrid Conference (TG11)*, Salt Lake City, Utah, 2011.

[2] Intel Corporation, "Intel MIC Web Page," 2012, http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html.

[3] 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. [Online]. Available: http://www.sci.utah.edu/publications/meng10/Meng_TaskSchedulingUintah2010.pdf

[4] 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. [Online]. Available: http://www.sci.utah.edu/publications/luitjens10/Luitjens_ipdps2010.pdf

[5] A. Humphrey, Q. Meng, M. Berzins, and T. Harman, "Radiation modeling using the uintah heterogeneous cpu/gpu runtime system," in *Proceedings of the XSEDE 2012 Conference*. ACM, 2012.

[6] 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*. IEEE, Piscataway, NJ, nov. 2000, pp. 33–41. [Online]. Available: http://www.sci.utah.edu/publications/dav00/uintah-hpdc00.pdf

[7] The Center for the Simulation of Accidental Fires and Explosions, "Uintah Web Page," 2012, http://www.uintah.utah.edu/.

[8] M. Berzins, "Status of Release of the Uintah Computational Framework," Scientific Computing and Imaging Institute, Tech. Rep. UUSCI-2012-001, 2012.

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

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

[11] M. Berzins, Q. Meng, J. Schmidt, and J. Sutherland, "Dag-based software frameworks for pdes," in *Proceedings of HPSS 2011 (Europar Bordeaux August 2011)*, 2012.

[12] N. Corp., "Nvidia Developer Zone Web Page," 2012, http://developer.nvidia.com/nvidia-gpu-computing-documentation.

[13] Q. Meng and M. Berzins, "Scalable large-scale fluid-structure interaction solvers in the uintah framework via hybrid task-based parallelism algorithms," *Submitted to Concurrency and Computation: Practice and Experience*, 2012.

[14] U.S. Department of Energy, Oak Ridge Natioanl Laboratory and Oak Ridge Leadership Computing Facility, "Titan Web Page," 2011, http://www.olcf.ornl.gov/titan/.

[15] J. Vetter, R. Glassbrook, J. Dongarra, K. Schwan, B. Loftis, S. McNally, J. Meredith, J. Rogers, P. Roth, K. Spafford, and S. Yalamanchili, "Keeneland Web Page," 2009, http://keeneland.gatech.edu/.

[16] S. P. Burns and M. A. Christen, "Spatial domain-based parallelism in large-scale, participating-media, radiative transport applications," *Numerical Heat Transfer, Part B: Fundamentals*, vol. 31, no. 4, pp. 401–421, 1997.