Addressing Global Data Dependencies in Heterogeneous Asynchronous Runtime Systems on GPUs

Brad Peterson
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, Utah
bpeterson@sci.utah.edu

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

John Schmidt
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, Utah
jas@sci.utah.edu

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

ABSTRACT

Large-scale parallel applications with complex global data dependencies beyond those of reductions pose significant scalability challenges in an asynchronous runtime system. Internodal challenges include identifying the all-to-all communication of data dependencies among the nodes. Intranodal challenges include gathering together these data dependencies into usable data objects while avoiding data duplication. This paper addresses these challenges within the context of a large-scale, industrial coal boiler simulation using the Uintah asynchronous many-task runtime system on GPU architectures. We show significant reduction in time spent analyzing data dependencies through refinements in our dependency search algorithm. Multiple task graphs are used to eliminate subsequent analysis when task graphs change in predictable and repeatable ways. Using a combined data store and task scheduler redesign reduces data dependency duplication ensuring that problems fit within host and GPU memory. These modifications did not require any changes to application code or sweeping changes to the Uintah runtime system. We report results running on the DOE Titan system on 119K CPU cores and 7.5K GPUs simultaneously. Our solutions can be generalized to other task dependency problems with global dependencies among thousands of nodes which must be processed efficiently at large scale.

KEYWORDS

Data dependencies, Asynchronous Many-Task, Programming Models, Runtime Systems, Scalability, GPU, Uintah, Coal Boiler, Radiative Heat Transfer

1 INTRODUCTION

A broad class of large-scale multiphysics applications requiring long-range interactions, such as molecular dynamics [7], cosmology [18], neutron transport [4], and radiative heat transfer [10] calculations use algorithms requiring global data dependencies. Such dependencies require each node to first send data to potentially every other node, and then prepare itself to receive data from most or all nodes. Once a node has received all data from other nodes, data dependencies must be gathered together into usable data objects. This sending, receiving, and gathering process can be prohibitively expensive both in terms of computational analysis and memory storage if the amount of data to be sent is large in contrast, say, to an MPI reduction.

This paper's motivation comes from our experience running massively parallel simulations aimed at predicting the performance of a commercial, 1000 MWe Ultra-Super Critical coal boiler. The size and complexity of these boiler simulations required a 351 million CPU hour INCITE award, 280 million and 71 million on the DOE Mira and Titan systems respectively. The Titan boiler case utilized the Uintah asynchronous many-task (AMT) runtime system [13, 14] which managed the scheduling and execution of over 8 million computational tasks on 119K CPU cores and 7.5K GPUs simultaneously.

In these boiler simulations the dominant mode of heat transfer is radiation, which presents significant challenges for AMT runtime systems [1] due to the all-to-all nature of radiation. The first of these challenges was that each Titan node was assigned ~1400 Uintah computational tasks, generating hundreds of thousands of global data dependencies introduced by the radiation solve. These dependencies become potential MPI messages for which Uintah must generate correct message tags [15]. Within Uintah, analyzing tasks for data dependencies is referred to as **dependency analysis**, part of the task graph compilation process. For standard stencil calculations, where each compute node only needs to search surrounding nodes containing neighboring simulation data, dependency analysis completes in milliseconds, even at scale. However, with the introduction of global dependencies, initial boiler runs on Titan required 4.5 hours for this dependency analysis at production scale. Additionally, the simulation required alternating between task execution patterns for timesteps involving either 1.) the standard computational fluid dynamics (CFD) calculation or 2.) CFD plus a radiation calculation to recompute the radiative source term (on Titan's GPUs) for the ongoing CFD calculation. Alternating between these separate task execution patterns occurred every 20 timesteps and required reanalysis of all global dependencies for the radiation solve, incurring potentially another 4.5 hour dependency analysis.

The second challenge was for each compute node to efficiently gather and combine together the thousands of dependencies sent from all other nodes into usable data objects ready for task execution. An initial attempt to address this challenge afforded each task its own copy of a data object, however this exhausted all available memory on-node. A follow-up attempt created only one shared data object

by utilizing locks, but contention for these locks was prohibitively slow. A new mechanism was needed.

The principal contribution of this work is to address these challenges through 1.) an improved search algorithm to reduce dependency analysis processing time by avoiding unnecessary searches, combined with multiple task graphs and 2.) a heterogeneous task scheduler and data store design which concurrently prepares tasks with simulation variables composed of shared dependencies gathered from potentially thousands of other nodes. We demonstrate how these changes do not require a large rewrite of key portions of Uintah, and how these improvements can be applied in a heterogeneous AMT environment with a mixture of CPU and GPU tasks providing speedups over a homogeneous set of CPU-only tasks. The solutions presented here can be generalized to other problems where each node has large numbers of data dependencies involving most or all of the domain. In addition, the solutions are also pertinent to task scheduler coordination schemes for preparation of simulation variables with global dependencies.

Section 2 describes the background of the data dependency challenges in the context of the target coal boiler problem, and provides detail on computational methods used, including radiation. Section 3 overviews the Uintah AMT runtime including its data stores, task graph, and task scheduler design. Section 4 describes a reduction in data dependency analysis times through an improved search algorithm and implementation of multiple task graphs to avoid redundant analysis with detailed complexity analysis. Section 5 provides novel mechanisms for concurrency and storage of simulation variables with large halos. Section 6 reports results on Titan using 119K cores and 7.5K GPUs using our improvements. Related work is given in Section 7 and conclusions in Section 8.

2 TARGET PROBLEM BACKGROUND

The global dependencies in our target 1000 MWe coal boiler problem arise from solving the radiative heat transfer equation (RTE) [8, 10]. Both the DOE Titan and Mira systems were used to simulate coal boiler designs using different methods for computing the RTE. On Mira, the global radiation dependencies required numerous sparse, global linear solves for the discrete ordinates method [12]. For every data dependency sent out from one compute node to another, the source node could expect to receive a corresponding data dependency in return. On the Titan platform, radiative heat transfer was computed using a reverse Monte Carlo Ray Tracing (RMCRT) technique [8], which requires replication of radiative properties to facilitate local ray tracing. Data dependencies here required more analysis as dependencies were not symmetrical. A compute node sending out a data dependency to another node did not always receive a similar data dependency in return.

The production problem computed on Titan used a uniform Cartesian mesh subdivided into ~497 million cells and ran for 220K timesteps over 5.5 days of simulation (wall) during which various physics parameters and runtime optimizations were tested and analyzed. Within Uintah, a group of **cells** is organized into a fundamental unit termed a **patch**, with the production problem having ~121 thousand patches. Simulation variables residing in Uintah's patches are termed **patch variables**. A patch variable needed by one compute node but found on another is considered to be a single data dependency. The superset of patches with the same mesh spacing is termed a **level**. Uintah provides support for Adaptive Mesh Refinement (AMR), viewing the computational grid as a sequence of nested, successively finer levels $1, ..., l_{max}$, such that $G = \cup\, G_l$ where G_l is a collection of a patches with the same mesh spacing.

2.1 RMCRT Radiation Model

The scalability of the RMCRT model has been demonstrated to 256K CPU cores [8] and 16K GPUs [9] on a benchmark problem [5]. The challenge was then in using this model in the production boiler case.

The RMCRT model creates between dozens to hundreds of rays per cell, each moving in a random direction. Local ray tracing is facilitated by using a local fine mesh on which the solution is calculated and a coarse version of the entire mesh replicated on each node. This replication was made possible by a global, but scalable communication phase [8], which generates thousands of data dependencies on each compute node. Without local ray tracing, there would be significantly more communication due to the need to transfer billions of rays via MPI throughout a timestep.

In the Titan boiler case, a two-level mesh refinement approach was used to reduce these data dependencies [8]. This two-level computational grid is described by 1.) a highly resolved fine mesh level for the CFD calculation and 2.) the coarse mesh level, replicated on each compute node for the radiative properties. As shown in Figure 1, a ray begins from the fine mesh level partition present on the node and will eventually transition onto the global coarser mesh level stored on that node as it moves outward, thus giving massively parallel ray tracing on a node. Radiation data at the ray's starting point is updated when the ray terminates.

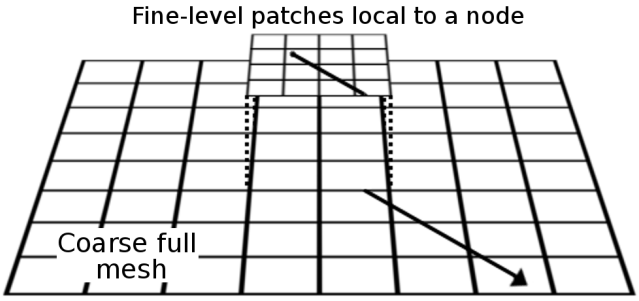

Figure 1: A 2D representation of an RMCRT ray as it moves across a domain with two levels of refinement. A ray begins on the fine mesh level and transitions to the coarse mesh level, terminating after its intensity falls below a specified threshold.

2.2 The Arches Combustion Component

The Uintah Arches turbulent combustion component is used to compute coal multiphysics for the target boiler problem. Arches is a three-dimensional, large eddy simulation (LES) code that uses a low-Mach number variable density formulation to simulate heat, mass, and momentum transport in reacting flows [8]. Additional fine-grained CPU tasks are responsible for simulating coal combustion, kinetics, and deposition. Arches is second-order accurate in space and time and is highly scalable through Uintah to 256K cores [17] and its coupled solvers, such as hypre [6].

The combination of Arches and RMCRT tasks dramatically complicated data dependency analysis. The Arches component generated close to one thousand tasks per compute node, with many containing direct dependencies on the global radiation data. Arches's CPU tasks required concurrency and data sharing mechanisms to move data in and out of GPU memory for RMCRT tasks. Every 20 timesteps a radiation solve takes place to update the radiative source term. The additional tasks generated during the radiation solve result in a repeated analysis of data dependencies.

3 UINTAH OVERVIEW

The Uintah framework [3, 13] is an asynchronous many-task runtime system designed to support multiphysics simulations for a broad range of problems involving fluids, solids, and fluid-structure interaction problems. Uintah is unique in its approach for providing a clear separation of duties between the application developer and the underlying runtime system. An application developer defines tasks by supplying key parameters for each task and includes such things as the simulation variables a task requires or will compute, halo extents for simulation variables, whether it is a GPU task, and a function pointer to the task code. Once all tasks are defined and listed in algorithmic order of priority, task code is written in C++ or CUDA. The Uintah runtime manages all underlying details of dependency analysis, task graph generation, MPI messages generation, halo scattering and gathering, task preparation and execution, data store concurrency, checkpointing, etc.

Uintah over-decomposes the computational domain into a structured grid of rectangular cuboid nodes or cells with support for adaptive mesh refinement (AMR). All Uintah simulation data is stored in patch variables maintained within a data store known as a **data warehouse** [13] which are shared among all threads in a compute node, allowing Uintah to launch one MPI rank per compute node rather than one MPI rank per CPU core. Uintah currently maintains a host (CPU) memory and GPU memory data warehouse, known as the **Host Data Warehouse** and the **GPU Data Warehouse**.

Uintah's task graph is based on a distributed directed acyclic graph (DAG) of task dependencies. Uintah utilizes a static task graph of data dependencies for two purposes, 1.) automated MPI message generation among compute nodes, and 2.) scheduling the preparation and execution of tasks within a compute node. A directed acyclic graph (DAG) of tasks is created after a node analyzes all data dependencies on all tasks that node will execute. This task graph is cached and reused in subsequent timesteps if the data dependencies do not change (this scenario is common among most simulations using Uintah). Dependency analysis only occurs once at initialization and when regridding takes place if AMR is employed.

Uintah has a task scheduler responsible for both preparing each tasks' patch variables in memory and efficiently executing those tasks. Every CPU thread on a compute node is assigned to the scheduler with a work loop that checks various shared queues and pools and assigns work to each thread. For example, one CPU thread may initiate an asynchronous MPI send or process MPI receives, another CPU thread may be executing a CPU task, another may invoke an asynchronous GPU kernel, another may initiate a GPU-to-host memory transfer for a patch variable, while yet another may be writing simulation data to disk for checkpointing.

4 DEPENDENCY ANALYSIS OPTIMIZATIONS

Owing to the global dependencies, the sheer size and complexity of the resulting task graphs at a production scale presented two significant challenges, determining potential dependencies and repeated dependency checking before and after radiation solves. The following two subsections describe these challenges and our solutions. In this discussion we use the term **node** to mean **compute node**. During dependency analysis, each *node* then considers itself the **source node**. This analysis is distributed in that each *node* performs its own. We also use the term **task** to mean **Uintah computational task**.

4.1 Uintah Task Dependency Analysis

Uintah's runtime system uses a three step algorithm on each source node to discover internodal data dependencies for MPI message generation. In the first step a set of nodes are identified in which halo exchange *may* occur based on halo requirements specified by the application. This list is termed a **processor neighborhood** in Uintah, and refers to the total number of MPI ranks in the simulation owning patches that *may* interact with a particular source node. In the second step, **task objects** are created by assigning tasks to patches. Each source node creates a collection of task objects that execute within its processor neighborhood. In the final step, each source node analyzes its collection of task objects to identify the data dependencies with tasks that execute within the source node. After all of the identified dependencies are placed into a task graph, then MPI messages tags are created [15].

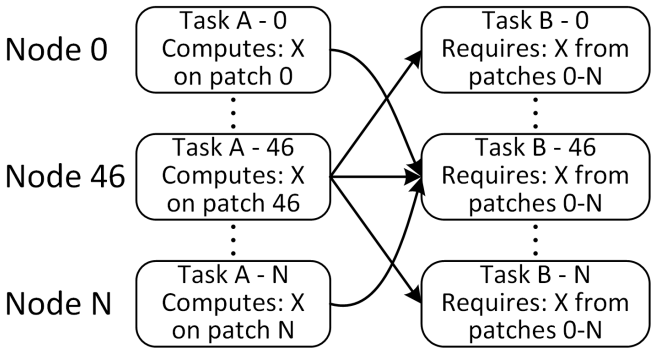

Figure 2: A visualization of data dependencies from the perspective of Node 46 in a simple N node problem with a global dependency on simulation variable X. After Node 46's Task A executes, the data dependencies must be sent out to $N-1$ other nodes. Similarly, before Node 46's Task B executes, it must receive data dependencies from $N-1$ other nodes.

4.1.1 Example Dependency Analysis. A simplified problem demonstrating global dependencies is given in Figure 2, where each node has a processor neighborhood consisting of all other nodes in the computational domain. Each source node searches all tasks within its processor neighborhood to determine which tasks it interacts with (via halo exchange). For simulations with only local communication (nearest neighbor), the resulting processor neighborhood is relatively small [15], being at most 26 nodes immediately

surrounding the source node (3 dimensions - x,y,z). For large-scale simulations with global communication and roughly 1000 tasks per node, a processor neighborhood may contain thousands of nodes and millions of potential dependencies.

4.2 Global and Local Neighborhoods

For the initial boiler simulations on Titan, the processor neighborhood included every node in the simulation, effectively the entire computational domain. A consequence of this approach is that tasks requiring only local communication still searched every task on every node for potential dependencies.

Using multiple processor neighborhoods to limit the search region for tasks reduced the time to identify task dependencies. Tasks requiring global dependencies search a global neighborhood and tasks with only local dependencies search a much smaller local neighborhood. The challenge in creating multiple neighborhoods is determining the appropriate neighborhood bounds for tasks which *only compute variables* and do not require halos. For example, in Figure 2, the set of "A" tasks do not specify that the simulation variable to be computed will be a globally dependent variable. These tasks on their own do not provide enough information to determine if they belong to a global or local neighborhood.

The key insight was to use the maximum halo extents on a per-variable-basis in contrast to the per-task basis to identify the boundary for either a local or global neighborhood. All tasks are searched to determine the maximum halo extent for each simulation variable. Each task is then assigned a maximum halo extent based on the largest halo extent of its simulation variables. From this per-variable maximum halo extent, a processor neighborhood can be correctly determined. When this algorithm is applied to the simple problem in Figure 2, the set of "A" tasks are assigned to a global neighborhood as the node shares a simulation variable with another "B" task that indicates the variable has global dependencies. If another set of tasks only used simulation variables requiring one cell of halo data, those tasks would be assigned a local neighborhood. If any sets of tasks share the same halo requirements, their dependencies are still analyzed as these task graph edges are vital to ensure proper ordering of task execution.

An additional complexity with the production coal boiler problem comes from tasks that execute on multiple mesh levels with non-uniform halo extents across mesh levels. Our solution was to further extend our improvements to define the maximum halo extents not just on a per-variable basis, but on the basis of a *per-variable and per-level tuple*. Although this solution was motivated by the boiler problem, it applies to any Uintah problem with a mixture of local and global dependencies.

4.2.1 Complexity Analysis. Consider a global dependency analysis problem with one neighborhood on a single mesh level. Uintah assigns tasks to patches uniformly across a given mesh refinement level, e.g. for 10 tasks over 100 patches, there would be exactly 1000 task objects for the entire computational grid. These tasks objects are later analyzed to find external (MPI) dependencies with a source node. The number of task objects T is then given by ng, where n is the number of patches owned by nodes (MPI ranks) in the global processor neighborhood and g is the number of generalized tasks. When only local communication is considered, n would be small with a maximum of 26 surrounding patches (3 dimensions - x,y,z) and the total number of task objects is manageable. However, when considering global dependencies, the dependency analysis becomes $O((ng)^2)$, a search between every task/patch tuple, ng in the computational domain.

In the Titan boiler simulation, the 2 mesh refinement levels used to achieve scalability [8] further increased the number of global dependencies due to the inter-level dependencies using the same global neighborhood for every task on every mesh refinement level. The number of task objects T is then given by

$$T = \sum_{l=1}^{l_{tot}} \sum_{t=1}^{t_{tot}} \sum_{n=1}^{n_{tot}} P_{l,t,n} \tag{1}$$

where l_{tot} is the total number of mesh levels, t_{tot} is the total number of tasks assigned to mesh level l, n_{tot} is the total number of neighborhoods for mesh level l, and $P_{l,t,n}$ is the number of patches in a particular mesh level's neighborhood. It is this total task object count, T that we seek to reduce, much like a partial order reduction algorithm prunes an exponential search space.

The generalized complexity of the task object count, T shown by Equation 1 in the target boiler case prior to our improvements was

$$O(n_f g_f) + O(n_c g_c) \tag{2}$$

where n_f and n_c are the number of fine- and coarse-level patches respectively, and g_f and g_c are the number of fine and coarse mesh level generalized tasks that would be created. Prior to our improvements, the total number of task objects created in the target boiler case was $T = \mathbf{10,044,888}$, with $n_f = 119,462$, $n_c = 1,440$, $g_f = 84$ and $g_c = 7$, and so $(n_f g_f = 10,034,808) + (n_c g_c = 10,080) = \mathbf{10,044,888}$. Introduction of global and local processor neighborhoods tailored for a task's variables and mesh level reduced the total number of task objects by 81x. This reduced the complexity in Expression 2 to

$$O(\frac{n_f}{p} t_{fl}) + O(\frac{n_c}{p} t_{cl}) + O(n_c t_{cg}) + O(n_f t_{fg}) \tag{3}$$

where n_f and n_c are the number of fine- and coarse-level patches respectively, p is the number of nodes, t_{fl} is the number of fine-level tasks with a local neighborhood, t_{cl} is the number of coarse-level tasks with local neighborhood, t_{cg} is the number of coarse-level tasks with a global neighborhood (tasks which globally distribute coarse-level simulation variables among nodes), and t_{fg} is the number of fine-level tasks with a global neighborhood (tasks which compute on the fine-level and require a global coarse mesh).

Table 1 illustrates the improvements from this reduction in complexity for our standard RMCRT benchmark problem [5]. Results were computed on a single node with an Intel Xeon CPU E5-2660 @ 2.20GHz. In the full coal boiler simulation at 119K cores, the task graph processing was reduced 93% from 4.5 hours to roughly 20 minutes. This was deemed acceptable for the Titan boiler case as it was a one-time cost and could be amortized over the entire simulation, running for 220K timesteps over 5.5 days of simulation (wall) time.

The remaining 20 minutes is largely due to one production task associated with the dominant fourth term of Expression 3, $O(n_f t_{fg})$. This particular task computes on every fine-level patch and also requires a global coarse mesh. Therefore, for every fine-level patch

Initial Dependency Analysis Improvements			
# of Fine Level Patches	Original time (s)	Improved time (s)	Speedup
64	~0.00	~0.00	1x
512	0.03	0.02	1.5x
4K	0.51	0.25	2.04x
32K	20.90	1.41	14.82x
128K	468.98	6.80	68.97x
256K	2331.66	15.02	155.24x

Table 1: Task graph compilation improvements combined with multiple task graphs (Section 4.3) enabled scaling to 122K patches for the target boiler problem.

there exists data dependencies with every coarse-level patch. The runtime stores these dependencies for each task in linked lists and most of the 20 minutes was spent in list traversal, searching for matching dependencies. From a nodal perspective most of these are duplicate dependencies. Work is underway to address this remaining cost by analyzing dependencies on a per-node basis to automatically eliminate all duplicates. When complete, the fourth term in Expression 3 will change to $O(d_f t_{fg})$, where d_f is the number of *nodes* containing fine level patches. This change will retain the use of multiple processor neighborhoods while greatly reducing the number of dependencies stored and analyzed by the runtime.

4.3 Multiple Task Graphs

Previously, whenever Uintah detected a different set of dependencies compared to a previous timestep's dependency set, a new data dependency analysis was triggered. If no change in dependencies occurred between timesteps, the previous task graph was reused. The key goal was to avoid the all-to-all communication (required for radiation) on regular CFD timesteps. As noted in Section 1, this approach was suitable for typical, stencil-based Uintah-based simulations, as the number of dependencies was much smaller with dependency analysis completing in milliseconds. However, as noted in Section 4.2.1, this dependency analysis still required 20 minutes. Recomputing this every 20 timesteps was still untenable.

Our solution added support within Uintah for temporal scheduling based on using multiple task graphs. With this approach, the type of timestep being executed determines which task graph is used, and consequently which tasks are executed for that timestep. The application developer is responsible for defining how many task graphs are needed and in which task graph a particular task executes. The Uintah runtime creates each of these task graphs upfront only once during the initial timestep, handling the dependency analysis for each task graph separately. The task graphs are cached and reused throughout the remainder of the simulation, so that no further data dependency analysis phase is required.

Analysis of a task graph does not require knowledge of exactly what task graph existed in the prior timestep. In every task graph Uintah creates a special runtime task called *send_old_data* and associates it with every data warehouse simulation variable. A current task graph requiring a dependency from a prior timestep can create a task graph edge with this *send_old_data* task, as this task is always guaranteed to exist no matter what prior task graph was used.

As an example, suppose a task developer required one task run in a radiation timestep, and a second task run in a CFD timestep. Previously the application developer would inform Uintah's runtime of both tasks with:

```
sched->addTask(taskRadiation);
sched->addTask(taskCFD);
```

Then when either task is executed, the application developer placed conditional statements within that task to short circuit task execution if the current timestep did not match the task's purpose. For example, on a regular CFD timestep, the ray tracing task used to update the radiative source term would execute, however the conditional placed by the application developer would simply have the task method immediately return, avoiding the execution of the ray tracing code. Unfortunately the Uintah task scheduler's data preparation phases had no knowledge of these conditionals, resulting in unnecessary dependency analysis and subsequent global communication.

With multiple task graphs, this process is greatly simplified for the application developer. A simple enum of task graphs is supplied in a header file and tasks are associated with an enum element:

```
sched->addTask(taskRadiation, RADIATION);
sched->addTask(taskCFD, CFD);
```

Any tasks not assigned to a specific task graph are placed into all task graphs, which ensures backward compatibility of existing tasks used within other simulations.

5 TASK SCHEDULER AND DATA WAREHOUSE OPTIMIZATIONS

Uintah's task scheduler is responsible for gathering together halo data from other nodes into usable data objects. In the target problem, multiple tasks on a node routinely required different nodal data (patch variables) while using the same data dependencies (halo data). Prior task scheduler logic could share halo data at the cost of locking mechanisms [9], or the task scheduler could retain asynchrony at the cost of duplicating halo data [16]. The target problem required sharing both halo data and retaining asynchrony as the prior task schedulers either executed inefficiently due to contention caused by locks or the problem could not fit into GPU memory due to duplicated halo data. This section describes the problem in detail and presents our current novel solution.

Properly sharing data and its dependencies evolved through three distinct phases. The first approach [9], termed **Phase I**, created one shared data object that is composed of all patch variables and all halo data for a simulation variable. It used a third data warehouse to manage simulation variables that encompassed an entire mesh level. Phase I used CPU locks and GPU barriers that affected performance by preventing GPU kernel overlap, resulting in GPU tasks executing serially. The second approach, **Phase II** [16], allowed sharing of patch variables among tasks, but did not allow sharing of halo data. Scheduler threads coordinated among themselves through atomic bitsets which represent lifetime states of a simulation variable. One bitset was assigned to each patch variable to ensure only one scheduler thread can allocate, prepare halo data, and copy a simulation variable. The overall goal was asynchrony and overlapping of GPU

tasks, as can be seen in Figure 3. Phase II avoided the third data warehouse in Phase I, however the the duplication of halo data drastically increased memory usage overhead in the target problem.

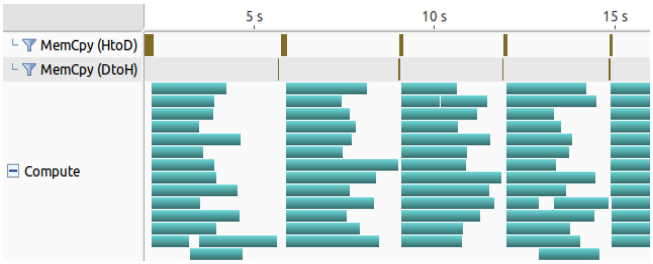

Figure 3: Visual profiling of Phase II [16] showing four timesteps with asynchrony and overlapping of GPU tasks. The gaps between timesteps illustrates lack of full GPU occupancy.

Phase III combines the best attributes of Phase I and Phase II. Phase III introduces new task scheduler and data warehouse changes to share both patch variables and halo data in shared data objects. The task scheduler is asynchronous and lock-free. Phase III avoids a third data warehouse, a large data warehouse code rewrite, and overly complicated logic.

Our solution 1.) decouples components of a data warehouse item, 2.) allows multiple data warehouse items to share decoupled objects, 3.) introduces additional asynchronous task scheduler logic for shared decoupled objects, and 4.) fits into existing data warehouse logic throughout Uintah. A simplified visual representation of the new data warehouse layout is given in Figure 4. A **data warehouse entry** now only contains identifying metadata (e.g. simulation variable name and patch ID). A **data description object** contains metadata for simulation variable layout and usage status. Multiple data warehouse entries may share a data description object using shared pointers. If data sharing is not needed (which is the case for most existing simulations using Uintah), existing data warehouse functionality is retained by having each data warehouse entry point to its own data description object.

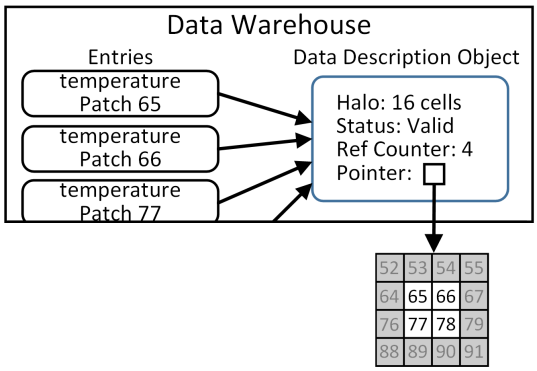

Figure 4: Simplified Data Warehouse design - Phase III.

Creation of data description objects is managed by task scheduler logic. When a scheduler thread analyzes an upcoming task for simulation variable preparation, it computes a spatial box necessary to encompass all halo data for a group of simulation variables assigned to various patches on that node. If the halos are small (such as one cell of halo data), the box only spatially encompasses one simulation variable, and requires no additional work. If the halos are large and multiple simulation variables are spatially contained within a box, then the scheduler thread begins a process to share a data description object among all the corresponding data warehouse entries.

As an example, suppose a simulation is laid out in a 2D grid exactly as shown in Figure 4. Each node is assigned its own square block of four patches for task computation. Further, suppose the halo for a simulation variable requires a full patch of 16 cells in every direction. One possible box, shown in Figure 4, encompasses a region of 16 patches numbered as shown. The four patches assigned to the node would then share a data description object, and the four data warehouse entries for this simulation variable would all refer to the same shared object.

Multiple scheduler threads preparing different tasks may simultaneously attempt to form the same shared data description object for a contiguous group of simulation variables. An atomic bitset is employed for coordination among scheduler threads. As no shared object has yet been created, the bitset used is the one associated with the smallest unique integer patch ID among the boxed entries. Once the separate patch variables have been merged into one shared data description object, the atomic bitset is copied into the shared object and a bit updated to indicate the simulation variable and its associated halo data are all valid and ready for use. Any other scheduler threads processing a task requiring any of these data warehouse entries in this box must either wait or seek new work from a work queue.

Previously, seven bits of the bitset described these different states of simulation variables: `allocating`, `allocated`, `copyingIn`, `validForUse`, `gatheringHaloCells`, `validWithHaloCells`. This work added bits for `mergingDataObjects` and `mergedDataObject`. Additionally, the status bitset now uses an additional 16 bits to hold a reference-counting short integer. This count represents how many data warehouse entries are sharing the data description object.

Overall, this new data warehouse design has many desirable qualities. There is no need for a third data warehouse as described in Phase I. Non-cubic domains are supported by allowing multiple data description objects to exclude void regions where no patches exist. Task scheduler thread asynchrony is retained. Memory overhead is kept low since duplication is avoided. All variables are managed in current data warehouses without requiring a large refactoring of the infrastructure code. All logic for this data warehouse can co-exist for other projects using Uintah which have dramatically different data dependency requirements.

The most important improvement is a reduction of memory usage on a node. Table 2 shows with this approach, the target problem fits within GPU memory, whereas before it could not fit within GPU memory in Phase II. Results were computed on a single node with an Intel Xeon CPU E5-2660 @ 2.20GHz with 32 GB RAM running Uintah on 16 CPU threads and a NVIDIA GeForce GTX TITAN X with 12 GB of RAM.

Improvements in Overhead			
Type of Mesh		Timestep (s)	Host Mem (MB)
Coarse: 32^3 cells, 4^3 patches Fine: 64^3 cells, 4^3 patches	Phase I	1.79	57
	Phase II	0.21	3073
	Phase III	0.15	65
Coarse: 32^3 cells, 4^3 patches Fine: 128^3 cells, 4^3 patches	Phase I	4.95	213
	Phase II	1.41	23229
	Phase III	0.82	279
Coarse: 64^3 cells, 4^3 patches Fine: 128^3 cells, 4^3 patches	Phase I	6.44	218
	Phase II	Exceeded memory	
	Phase III	1.08	311

Table 2: Results of running only GPU RMCRT benchmark tests for the three phases detailed in this section. Phase I has low memory usage but high wall time overhead due to frequent GPU blocking calls. Phase II improves wall times, but memory usage is unacceptably large. Phase III's low overhead results in both faster wall times and low memory usage.

5.1 Improving GPU Occupancy

Section 5 assumes concurrently executing scheduler threads for preparation and execution of GPU tasks. This subsection describes an optimization to this scheduler model enabling wall time speedups through better GPU occupancy. The target problem assigned 15 or 16 patches per node. However, as seen in Figure 3, this patch count inefficiently occupied a GPU throughout a timestep. Titan's NVIDIA K20X GPUs contained 14 streaming multiprocessors (SMs), not enough for the 15 or 16 GPU tasks to compute simultaneously, slightly oversubscribing the total SMs. Instead of assigning only 14 patches per node (and thus using ~10% more compute hours), we desired a solution which retained a low patch count per node for faster dependency analysis while also providing good GPU occupancy.

We first attempted to split a kernel into multiple blocks, but a kernel would not vacate SMs until all of its blocks computed, leaving some SMs idle. We then split each GPU task into multiple kernels and launched them on a shared task GPU stream, but we observed serialization among kernels when multiple CPU threads asynchronously launched multiple kernels intermixed with host-to-device transfers. Our adopted solution assigns each GPU task multiple GPU streams and splits a task into multiple kernels. The task scheduler does not consider a GPU task complete until each GPU stream assigned to that task completes all of its operations. Figure 5 demonstrates significantly better SM occupancy with smaller kernels on multiple streams. We measured speedups of 1.2x compared to the baseline of supplying only one kernel and one stream per task. This approach of launching higher numbers of small kernels to a GPU best allows Uintah to target GPUs with differing number of SMs. The only requirement is that a GPU's compute capability version must support more concurrent kernels than SMs.

6 SUMMARY OF RESULTS

A before and after summary of all optimizations at full scale (119K CPU cores and 7.5K GPUs) is shown in Table 3. Both CPU and

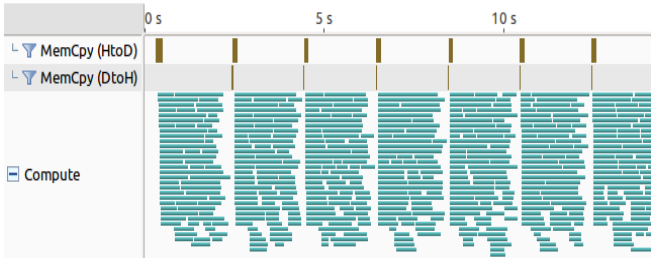

Figure 5: Visual profiling - Phase III showing six successive timesteps. Uintah's scheduler supplies each GPU task multiple streams so that task can be split into multiple kernels, executed concurrently and achieving better GPU occupancy.

Operation	Before	After
Dependency analysis	4.5 hours	20 minutes
Dependency re-analysis	Before and after each radiation calc	No longer required
Nodal memory footprint	21 GB	3.5 GB
RMCRT radiation avg timestep	77.4 sec (CPU) 97.1 sec (GPU)	53.1 sec (GPU)
Nonradiative avg timestep	8.34 sec	8.38 sec

Table 3: Cumulative results of improvements from Sections 4 and 5. Full boiler simulation, 129K CPU cores and 7.5K GPUs.

GPU production boiler runs used a fine mesh level patch size of 16^3 cells, 15 to 16 fine mesh level patches per node, and 150 RMCRT rays per cell. The production problem size was ~497 million cells distributed among ~121 thousand total mesh patches.

The first two data rows of Table 3 come from work in Section 4. We measured 93% reduction in task graph dependency analysis times on the coal boiler simulation's initial timestep. The implementation of multiple task graphs eliminated subsequent re-analysis when alternating between radiation and non-radiation timesteps. Both of these improvements required a one-time, 20 minute dependency analysis phase, but it was amortized over several hundred thousand timesteps and was a negligible cost overall.

The table's last three rows come from work in Section 5. Initially the 21 GB of host memory usage was too large for the 6 GB of memory in Titan's GPUs. Our work reduced this to 3.5 GB per node, allowing the problem to run in GPUs, as well as reducing wall times by requiring less data to be transfered across the PCIe bus.

The simulation was tested in both a homogneous mode of CPU only tasks and a heterogenous mode where GPU and CPU tasks intermixed. In the heterogenous mode, the Uintah runtime reached a state where data dependencies for RMCRT tasks were prepared quickly and kernels spread among a GPU with the vast majority of the timestep being GPU memory-bounded. For radiation timesteps, the runtime is no longer a bottleneck for efficiency, and any further speedups would need to come from algorithmic changes.

The non-radiative timestep wall times demonstrate that the overhead of adding task scheduler heterogeneity is minimal. In these timesteps, the hundreds of tasks were processed by more work pools and queues in a heterogeneous environment than in a homogeneous environment. As shown in the last row of Table 3, the average timesteps in both environments are relatively similar.

7 RELATED WORK

Other AMT runtimes use different approaches for handling data dependencies. Charm++ [11] requires the user to manage data dependencies. Users create a set of interacting data objects called *chares* (roughly analogous to Uintah tasks). A dependency is created through an event of one *chare* sending a message to another, which could be on the same node or on another node. Charm++ does not have an explicit task graph, and concurrency of shared data resources in data structures happens through user programmed code and proper message structuring. Global dependencies would likewise require *chares* to manually send numerous messages to facilitate an all-to-all communication structure. The Legion [19] runtime system automates dependency analysis and concurrency by first requiring the application developer to supply many more characteristics of a data structure's data dependencies. Global dependencies would likewise require using the Legion API to manually specify every dependency to allow Legion to automate all communication. Legion also leverages a parallel global address space approach using GASNet for all internode communication. The DARMA project [2] functions at lower level on the software stack to provide a generalized tool to facilitate AMT functionality. For example, Uintah could conceivably be built on top of DARMA. DARMA's focus is on nodal management of tasks, and data dependencies across many nodes are again left up to the application developer.

8 CONCLUSIONS AND FUTURE WORK

We have demonstrated optimizations for the sending, receiving, and gathering process for global dependencies in an asynchronous runtime system using GPUs. We have shown that globally coupled problems such as radiation heat transfer create overhead challenges for identifying data dependencies, storing dependencies into a static task graph, and avoiding data store duplication. We have demonstrated these challenges in the context of a full multiphysics simulation of a 1000MWe coal boiler on Titan using 119K CPU cores and 7.5K GPUs using the Uintah asynchronous many-task runtime.

We reduced data dependency analysis processing through two means. First, eliminating unneeded dependency checks by considering dependency neighborhoods on a per-variable and per-level basis, rather than on a simulation-wide basis. Second, elimination of redundant processing of dependencies mid-simulation by enabling caching of multiple task graphs that are processed only on the initial timestep. We reduced data duplication overhead by sharing simulation variable data on a compute node in a data object. Enabling these shared data objects in an asynchronous and concurrent environment required novel modifications to our task scheduler and data stores. At full scale, these modifications demonstrated that tasks could be processed faster in a heterogeneous task environment than a homogeneous CPU-only environment.

Our future work will extend Uintah's task scheduler and data stores to a more generalized and portable manner. We are using Kokkos for portability and ensuring Uintah's runtime is performant for both Xeon Phis and GPUs.

9 ACKNOWLEDGEMENTS

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

REFERENCES

[1] J. Bennett et al. 2015. *ASC ATDM level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms*. Technical Report. Sandia National Laboratories. http://www.sci.utah.edu/publications/Ben2015c/ATDM-AMT-L2-Final-SAND2015-8312.pdf

[2] Janine C. Bennett, Jeremiah Wilke, et al. 2016. The DARMA Approach to Asynchronous Many-Task Programming. In *Presented at ECP Review 2016, Sandia National Laboratories*.

[3] M. Berzins. 2012. *Status of Release of the Uintah Computational Framework*. Technical Report UUSCI-2012-001. Scientific Computing and Imaging Institute.

[4] J. Briesmeister. 2000. *MCNP – A General Monte Carlo N-Particle Transport Code, Version 4C*. Technical Report LA-13709-M. Los Alamos National Lab.

[5] S. P. Burns and M. A. Christen. 1997. SPATIAL DOMAIN-BASED PARALLELISM IN LARGE-SCALE, PARTICIPATING-MEDIA, RADIATIVE TRANSPORT APPLICATIONS. *Numerical Heat Transfer, Part B: Fundamentals* 31, 4 (1997), 401–421.

[6] Robert D. Falgout, Jim E. Jones, and Ulrike Meier Yang. 2006. The Design and Implementation of hypre, a Library of Parallel High Performance Preconditioners. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, AreMagnus Bruaset and Aslak Tveito (Eds.). Lecture Notes in Computational Science and Engineering, Vol. 51. Springer Berlin Heidelberg, 267–294. https://doi.org/10.1007/3-540-31619-1_8

[7] Hans Fangohr, Andrew R. Price, Simon J. Cox, Peter A.J. de Groot, Geoffrey J. Daniell, and Ken S. Thomas. 2000. Efficient Methods for Handling Long-Range Forces in Particle-Particle Simulations. *J. Comput. Phys.* 162, 2 (Aug. 2000), 372–384. https://doi.org/10.1006/jcph.2000.6541

[8] A. Humphrey, T. Harman, M. Berzins, and P. Smith. 2015. A Scalable Algorithm for Radiative Heat Transfer Using Reverse Monte Carlo Ray Tracing. In *High Performance Computing*, Julian M. Kunkel and Thomas Ludwig (Eds.). Lecture Notes in Computer Science, Vol. 9137. Springer International Publishing, 212–230. https://doi.org/10.1007/978-3-319-20119-1_16

[9] A. Humphrey, D. Sunderland, T. Harman, and M. Berzins. 2016. Radiative Heat Transfer Calculation on 16384 GPUs Using a Reverse Monte Carlo Ray Tracing Approach with Adaptive Mesh Refinement. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1222–1231. https://doi.org/10.1109/IPDPSW.2016.93

[10] J Patrick Jessee, Woodrow A Fiveland, Louis H Howell, Phillip Colella, and Richard B Pember. 1998. An adaptive mesh refinement algorithm for the radiative transport equation. *Journal of computational Physics* 139, 2 (1998), 380–398.

[11] Laxmikant V. Kale and Sanjeev Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. *SIGPLAN Not.* 28, 10 (Oct. 1993), 91–108. https://doi.org/10.1145/167962.165874

[12] G. Krishnamoorthy, R. Rawat, and P.J. Smith. 2006. Parallelization of the P-1 Radiation Model. (2006). Numerical Heat Transfer, Part B: Fundamentals, 49 (1), 1-17.

[13] Q. Meng, A. Humphrey, and M. Berzins. 2012. The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*. 2441–2448. https://doi.org/10.1109/SCC.2012.6674233

[14] Qingyu Meng, Alan Humphrey, John Schmidt, and Martin Berzins. 2013. Investigating Applications Portability with the Uintah DAG-based Runtime System on PetaScale Supercomputers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 96, 12 pages. https://doi.org/10.1145/2503210.2503250

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

[16] B. Peterson, H. Dasari, A. Humphrey, D. Sunderland, J. Sutherland, T. Saad, and M. Berzins. 2016. Reducing Overhead in the Uintah Framework to Support

Short-Lived Tasks on GPU-Heterogeneous Architectures. Submitted *International Journal of Parallel Programming, 2016* (2016).

[17] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland. 2013. Large Scale Parallel Solution of Incompressible Flow Problems using Uintah and hypre. In *Proceedings of CCGrid 2013*. IEEE/ACM.

[18] Volker Springel. 2005. The cosmological simulation code GADGET-2. *Monthly notices of the royal astronomical society* 364, 4 (2005), 1105–1134.

[19] Stanford University. 2017. Legion Web Page. (2017). http://legion.stanford.edu.