SCALABLE ASYNCHRONOUS MANY-TASK RUNTIME SOLUTIONS TO GLOBALLY COUPLED PROBLEMS

by

Alan Parker Humphrey

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

May 2019

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Alan Parker Humphrey**
has been approved by the following supervisory committee members:

Martin Berzins ,	Chair(s)	**02/08/2019** Date Approved
Robert M. Kirby ,	Member	**02/08/2019** Date Approved
Ganesh Gopalakrishnan ,	Member	**02/08/2019** Date Approved
Mary Hall ,	Member	**02/08/2019** Date Approved
James Sutherland ,	Member	**02/08/2019** Date Approved

by **Ross Whitaker** , Chair/Dean of
the Department/College/School of **Computer Science**
and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

Thermal radiation is an important physical process and a key mechanism in a class of challenging engineering and research problems. The principal exascale-candidate application motivating this research is a large eddy simulation (LES) aimed at predicting the performance of a commercial, 1200 MWe ultra-super critical (USC) coal boiler, with radiation as the dominant mode of heat transfer. Scalable modeling of radiation is currently one of the most challenging problems in large-scale simulations, due to the global, all-to-all physical and resulting computational connectivity. Fundamentally, radiation models impose *global data dependencies*, requiring each compute node in a distributed memory system to send data to, and receive data from, potentially every other node. This process can be prohibitively expensive on large distributed memory systems due to pervasive all-to-all message passing interface (MPI) communication. Correctness is also difficult to achieve when coordinating global communication of this kind. Asynchronous many-task (AMT) runtime systems are a possible leading alternative to mitigate programming challenges at the runtime system-level, sheltering the application developer from the complexities introduced by future architectures. However, large-scale parallel applications with complex global data dependencies, such as in radiation modeling, pose significant scalability challenges themselves, even for a highly tuned AMT runtime. The principal aims of this research are to demonstrate how the Uintah AMT runtime can be adapted, making it possible for complex multiphysics applications with radiation to scale on current petascale and emerging exascale architectures. For Uintah, which uses a directed acyclic graph to represent the computation and associated data dependencies, these aims are achieved through: 1) the use of an AMT runtime; 2) adapting and leveraging Uintah's adaptive mesh refinement support to dramatically reduce computation, communication volume, and nodal memory footprint for radiation calculations; and 3) *automating* the all-to-all communication at the runtime level through a task graph *dependency analysis phase* designed to efficiently manage data dependencies inherent in globally coupled problems.

For my wife Iris, our children Noah and Quinn, and my brother Robert.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

Adaptive Mesh Refinement	(AMR)
Argonne Leadership Computing Facility	(ALCF)
Application Programming Interface	(API)
Asynchronous Many-Task	(AMT)
Bounding Volume Hierarchy	(BVH)
Bulk Synchronous Parallel	(BSP)
Carbon Capture Multidisciplinary Simulation Center	(CCMSC)
Computational Fluid Dynamics	(CFD)
Compute Unified Device Architecture	(CUDA)
Central Processing Unit	(CPU)
Department of Energy	(DOE)
Directed Acyclic Graph	(DAG)
Discrete Ordinates Method	(DOM)
Exascale Computing Project	(ECP)
Graphics Processing Unit	(GPU)
High Performance Computing	(HPC)
Implicit Continuous-fluid Eulerian	(ICE)
Large-Eddy Simulation	(LES)
Material Point Method	(MPM)
Message Passing Interface	(MPI)
Monte Carlo Ray Tracing	(MCRT)
National Nuclear Security Administration	(NNSA)
National Science Foundation	(NSF)
Oak Ridge Leadership Computing Facility	(OLCF)
Open Multi-Processing	(OpenMP)
Partial Differential Equation	(PDE)
Photon Monte Carlo	(PMC)
Portable Operating System Interface	(POSIX)
POSIX Thread	(Pthread)
Predictive Science Academic Alliance Program	(PSAAP)
Radiative Transfer Equation	(RTE)
Refinement Ratio	(RR)
Reverse Monte Carlo Ray-Tracing	(RMCRT)
Single Program Multiple Data	(SPMD)
Texas Advanced Computing Center	(TACC)
Two-Dimensional	(2D)
Three-Dimensional	(3D)
Ultra-Super Critical	(USC)

ACKNOWLEDGEMENTS

I would like to thank all those involved with Uintah, past and present. In particular, I want to thank John Schmidt, Dan Sunderland, Todd Harman, Brad Peterson, Derek Harris, Qingyu Meng, and J. Davison de St. Germain. I would also like to thank my committee for their advice and input, and especially my advisor Martin Berzins, for his guidance, continual involvement, and thought-provoking discussions throughout my research. Finally, I would like to thank my wife, Iris, and our children, Noah and Quinn, for their patience and continual support. Without their love and presence in my life, this would not have been possible.

Funding Acknowledgements

- This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375.
- This work was supported by the National Science Foundation under subcontracts No. OCI0721659, the NSF OCI PetaApps program, through award OCI 0905068, and DOE NETL for funding under NET DE-EE0004449.
- 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 used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.
- This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under contract DE-AC02-06CH11357.
- Awards of computer time were provided for time on the DOE Titan and Mira systems by 1.) the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program and 2.) the Advanced Scientific Computing Research (ASCR) Leadership Computing Challenge (ALCC) program.

CHAPTER 1

INTRODUCTION

The exponential growth in high-performance computing (HPC) over the past 20 years has been fueled by a wave of scientific insights and discoveries, many of which would not be possible without the integration of HPC capabilities. This trend is continuing, with the United States Department of Energy (DOE) Exascale Computing Project (ECP) [2] listing 25 major applications focus areas [3] in energy, science, and national security missions. Many of these applications have significant economic and scientific impact, with solutions that can be advanced only by applying HPC techniques and resources. A broad class of such problems includes large-scale multiphysics applications requiring long-range interactions, such as molecular dynamics [4], cosmology [5], neutron transport [6], and radiative heat transfer calculations [7]. Thermal radiation in particular is an important physical process and a key mechanism in a class of challenging engineering and research problems. A principal challenge in modeling radiative heat transfer, however, is the strong nonlocal nature of radiation, with potential propagation of radiation across the entire domain from any point.

1.1 Target Problem

This research is motivated primarily by the target problem of University of Utah Carbon Capture Multidisciplinary Simulation Center (CCMSC), funded by the National Nuclear Security Administration (NNSA) Predictive Science Academic Alliance Program (PSAAP) II. The principal aim of the Utah CCMSC is, through petascale and eventually exascale simulations, to predict the performance of a commercial, 1200 MWe ultra-supercritical (USC) coal boiler being developed by General Electric (GE) Alstom Power, with radiation as the dominant mode of heat transfer. GE Power is currently building new coal-fired power plants throughout the world. Many of these units may potentially be 1200 MWe, twin-fireball (no dividing wall), USC units, each providing

power for nearly 1 million individuals. An example of such a power plant (˜90 m tall) is shown in Figure 1.1. Historically, twin-fireball (or eight-corner) units became part of the GE Power product offering because of the design uncertainty in scaling four-corner units from a lower Megawatt (MW) rating to a much higher MW rating. In order to decrease risk, both from GE Power's viewpoint, as well as from the customer's viewpoint, two smaller units were joined together to form a larger unit. Both the eight-corner units and the four-corner units have different mixing and wall absorption characteristics that must be fully understood in order to mitigate risk and have confidence in their respective

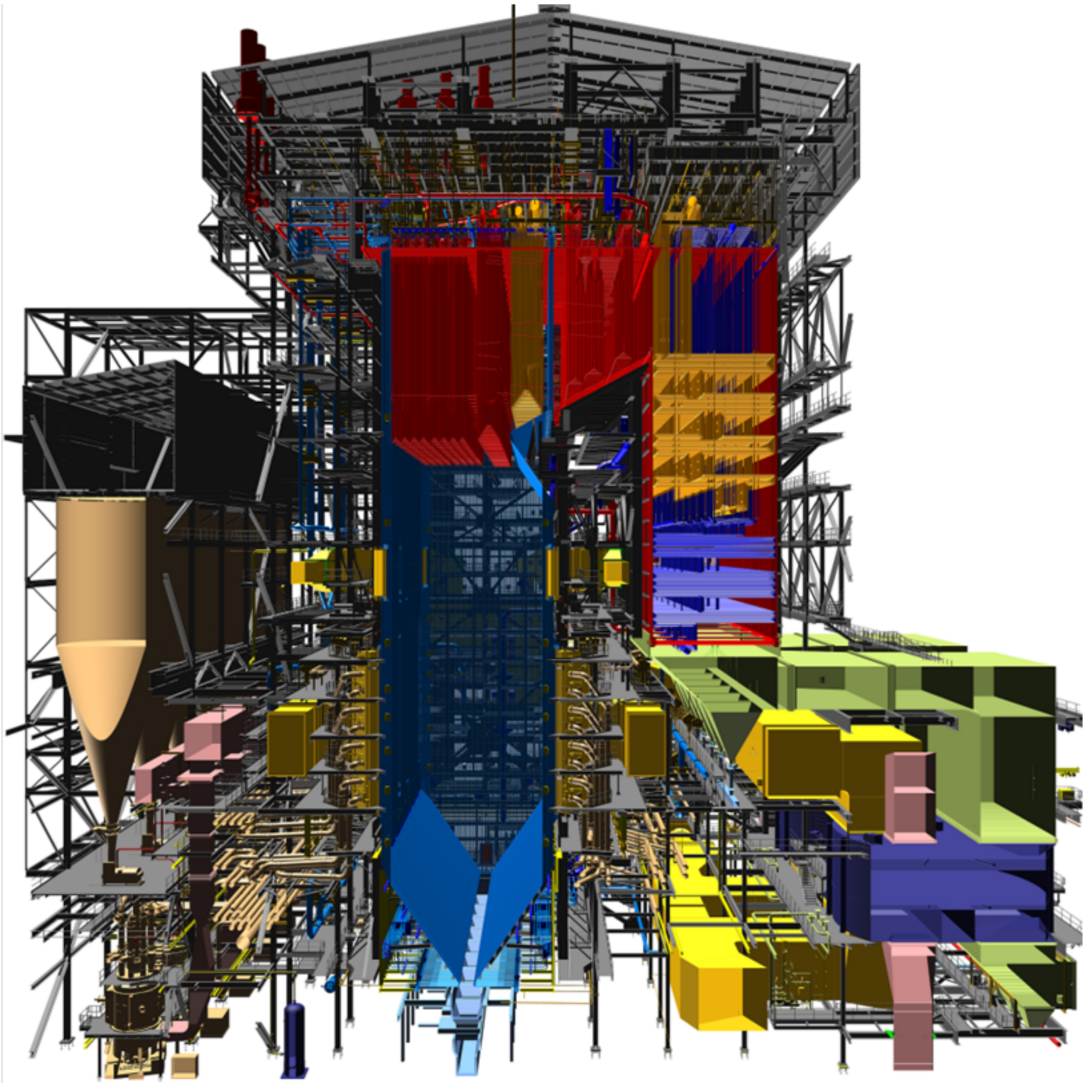

Figure 1.1. CAD rendering of GE Power's 1200 MWe USC two-cell pulverized.

designs. One key to this understanding is how the separated over-fired air (SOFA) inlets (Figure 1.2) that inject pulverized coal and oxygen into the combustion chamber should be positioned and oriented and what effect these positions have on the heat flux distribution throughout the boiler. The position of the SOFAs, as well as the firing angle and penetration of the SOFA jets, determines the stoichiometry, swirl strength, and temperature profiles in both the core of the furnace and in the near-wall region at the SOFA and upper furnace elevations. As a consequence, the SOFA configuration has a significant impact upon the furnace characteristics being studied

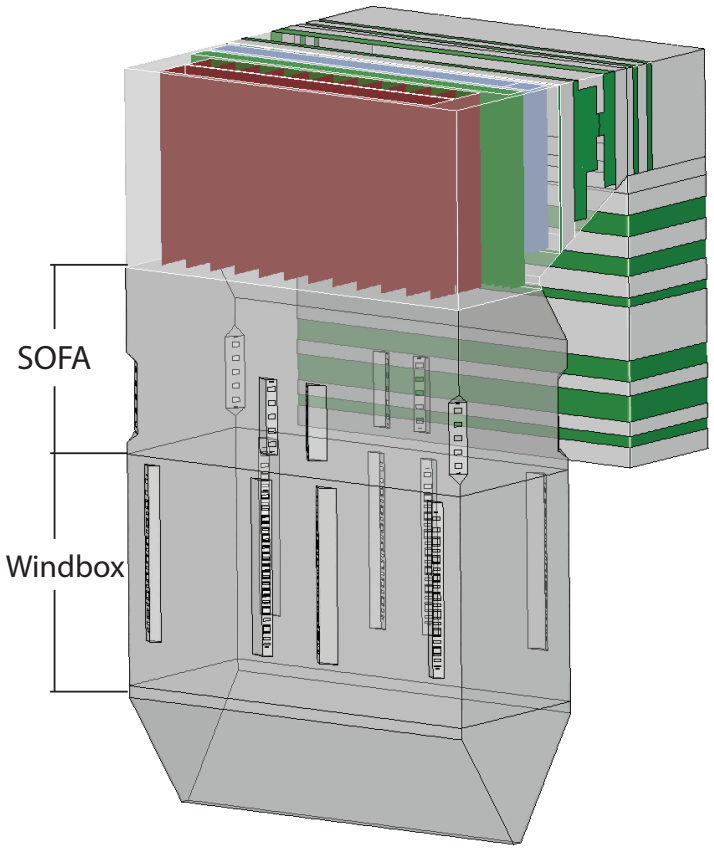

Figure 1.2. Primary wind-box and separated over fired air (SOFA) inlets.

From a computational perspective, a key focus of the Utah CCMSC is on using extreme-scale computing for reacting, large eddy simulation (LES)-based codes within the Uintah open-source framework, using current machines such as Titan and Mira, as well as the upcoming Summit and A21 systems, in a scalable manner. The CCMSC target simulation has been considered as an ideal exascale candidate given that the spatial and temporal resolution requirements on physical grounds give rise to problems between 50 and 1000 times larger than those we can solve today. Hence, the physical size of the CCMSC target boiler simulations necessitates the use of these large machines at near-capacity to adequately resolve the computational domain in a tractable amount of time. Figure 1.3 shows the CCMSC target boiler problem with a lower resolution, structured mesh.

Combustion systems such as furnaces and pool fires are radiation dominated [8]. Radiation is the dominant mode of heat transfer in the CCMSC boiler simulations, and because radiative heat transfer rates are generally proportional to the fourth power of the temperature, applications such as the CCMSC boiler simulations that simulate a turbulent combustion process are highly influenced by the accuracy of the radiation models used [9]. Radiation is fundamental to the CCMSC target problem, where the entire computational domain needs to be resolved to adequately model the radiative heat flux. Within the boiler, the hot combustion gases radiate energy to the boiler walls and to tubes carrying water and steam that is superheated to a supercritical fluid. This steam acts as the working fluid to drive the turbine for power generation. The residual energy in the mixture passes through a convective heat exchange system to extract as much of the remaining energy as possible into the working fluid. This radiative flux depends on the radiative properties of the participating media and temperature. The mixture of particles and gases emits, absorbs, and scatters radiation, the modeling of which is a key computational element in these simulations.

Scalable modeling of radiation, however, is currently one of the most challenging problems in large-scale simulations, due to the global, all-to-all nature of radiation, potentially affecting all regions of the domain simultaneously at a single instance in time [10]. Fundamentally, radiation models impose *global data dependencies*, requiring each compute node to send data to and receive data from potentially every other node. The radiation calculation, in which the radiative-flux divergence at each cell of the discretized domain is calculated,

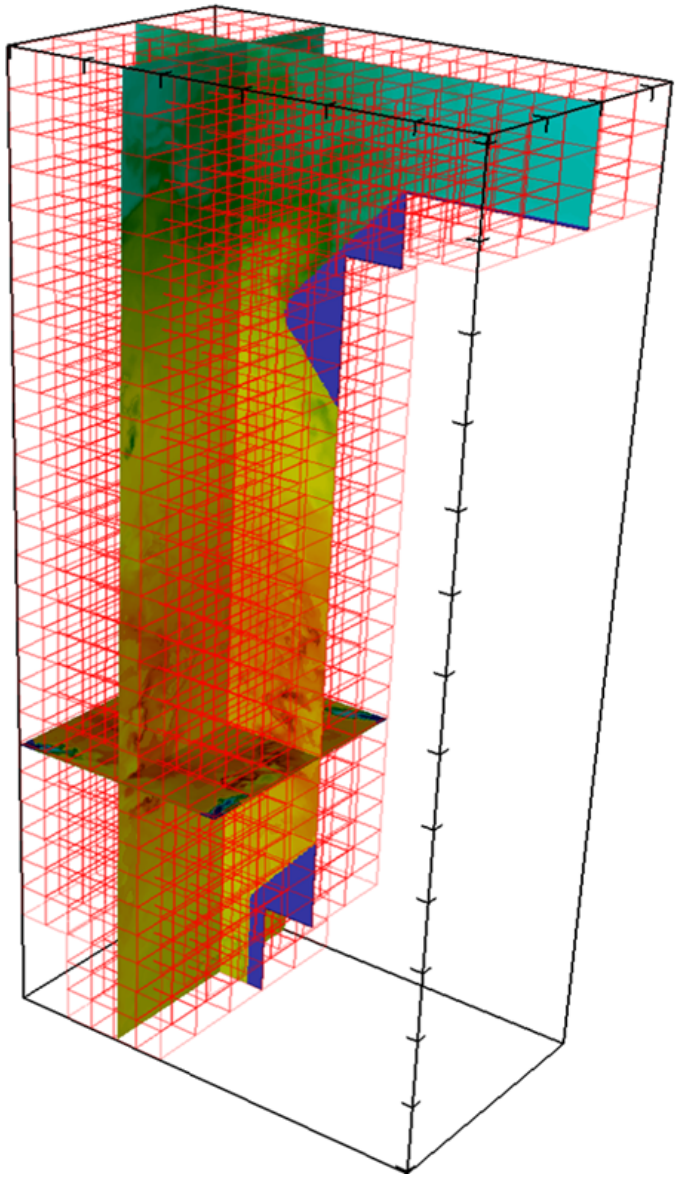

Figure 1.3. Boiler chamber with overlain structured mesh.

can be prohibitively expensive in terms of computational analysis, communication volume, and nodal memory footprint. Depending on the optical properties of the gas mixture and particles, radiation may take as much as 80% of the total computation time in a reactive flow simulation [8] when using approaches such as the discrete ordinates method (DOM, Section 2.2), one of the standard approaches to computing radiative heat transfer.

To simulate the Utah CCMSC target boiler problem requires expertise from many domains, including computer science, chemical engineering, fluid and material dynamics,

radiative heat transfer, coal combustion, LES, fire simulations, and multiscale modeling physics. This broad requirement has necessitated a framework that facilitates the cooperation between experts from different domains [11] to solve such complex multiscale multiphysics problems. Uintah [12], an open-source (MIT License) computational framework, has been employed by the Utah CCMSC to achieve this goal. Other industrial codes, such as Fluent, incorporate straightforward radiation models such as the discrete ordinates method in Fluent and Airpack [13], but scale to relatively small numbers of cores. At the national labs, many cutting-edge codes have been developed, such as Fuego, CFDLIB, Kiva, and radiation codes ATTILA, DANTE, WEDGEHOG, PARTISN [14, 15], but many of these are not generally available, are unsupported, or are targeted at other problems such as neutron transport.

The Uintah runtime then becomes a central focus of this dissertation. Uintah has been widely used for many different types of problems involving fluids, solids, and fluid-structure interaction problems, and it 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. 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 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 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 application developer.

At its core, Uintah is an asynchronous many-task (AMT) runtime system, and at the scales at which the CCMSC target boiler simulations are run, this becomes a necessity as the current focus in the design of large HPC codes is to make them as asynchronous as possible as a mechanism to improve application performance. AMT runtime systems are a possible leading alternative to mitigate exascale challenges at the runtime system-level, sheltering the application developer from the complexities introduced by future architectures [16]. Tasks scheduled by the AMT runtime may be executed in an adaptive manner,

effectively overlapping communication and computation. However, large-scale parallel applications with complex global data dependencies, such as in radiation modeling, pose significant scalability challenges themselves, even for a highly tuned AMT runtime.

Ultimately, through this research, the addition of a scalable, hierarchical radiation solver within Uintah will also benefit the general computational science engineering community in applications areas such as turbulent combustion simulation and other energy-related problems. The broader impact of this work may ultimately include algorithmic developments for related problems with pervasive all-to-all type communications in general, such as long-range electrostatics in molecular dynamics, and will be of importance to a broad class of users, developers, scientists, and students for whom such problems are presently a bottleneck.

1.2 Parallelism

It can be said that the objective of a parallel system is the fast execution of applications, faster than can be executed on a single processor system [17]. The need to accelerate scientific discovery has necessitated the use of parallelism and parallel design patterns in general. Moore's law, the observation that the number of transistors in a dense integrated circuit doubles about every two years, can also be thought of as leading inevitably to parallel computing, due primarily to power consumption and the inability to dissipate heat at current transistor levels. Throughout this dissertation, the terms processor, processing element, and core will refer to a single processing unit, which may be a single-core CPU or a single core of a multicore CPU.

Parallelism, using extra processing elements to do more work per unit time, differs subtly from concurrency, which is managing access to *shared* resources. In other words, concurrency is the composition of independently executing processes *sharing resources*, whereas parallelism is the simultaneous execution of *potentially related computations*. In the context of the Utah CCMSC, applications utilize concurrent threads of execution on multiple processing cores on large distributed systems, with and without accelerators, to solve existing problems faster as well as solve larger problems in a tractable amount of time. To simulate the CCMSC target boiler problem without extensive parallelism and distributed memory would not be possible, as these simulations take weeks (wall time)

to complete, even at 128,000 CPU cores. The memory requirements alone for a problem this size must be distributed among many compute nodes to even fit into available nodal memory. Massive parallelism is also required for the CCMSC boiler simulations due to the sheer magnitude of the computational requirements, without which a solution could simply not be obtained. To gain the requisite levels of parallelism, CCMSC applications must be mapped efficiently onto a parallel architecture, typically through the Uintah framework. These simulations rely on multiple levels of parallelism provided by Uintah, including: 1) domain decomposition (application specified); 2) task-level parallelism (automatic); and 3) data parallelism within a task, via OpenMP or CUDA (automatic). Thus, using more processing elements (processors or CPU cores), the solution to CCMSC target boiler simulations can (ideally) be computed faster. Additionally, larger problems can be simulated in a tractable amount of time using more CPU cores. For achieving faster times to solution and computing larger problems, this work focuses on two common scaling methods for assessing scalability.

1.2.1 Scaling and Efficiency

Martin et al. present an analytical tool with which to study the scalability of parallel algorithm-architecture combinations [18], from which we derive a summary and metrics for use with Uintah. These metrics are used throughout this dissertation to understand the parallel performance of Uintah as a whole by predicting and verifying the performance of individual components. Ultimately, this analytical tool will help developers better understand what changes are needed to increase the scalability of Uintah, specifically regarding task graphs with global dependencies.

Strong scaling refers to the change in execution time for a fixed problem size as the number of processors varies. Let $T(N,P)$ represent the time to solve a problem size N on P processing elements. Ideal strong scaling then occurs when the problem is solved P times faster than in serial which is described by

$$T(N,P) = \frac{T(N,1)}{P}. \tag{1.1}$$

Thus, when $T(N,P)$ for a fixed N is plotted on a log-log graph, ideal strong scaling appears as a straight line with a slope of -1. The strong scalability limit of a code is reached when

an increase in the number of parallel processors used on a fixed problem size does not result in a decrease in computational wall time [19].

Weak scaling refers to the change in execution time as the number of processors and the problem size vary proportionally to each other and is described by

$$T(N, P) = T(cN, cP), \tag{1.2}$$

where c is the factor by which both the problem size and solution time are proportionally increased. In other words, ideal weak scaling occurs when a problem that is c times larger is solved on c times as many processing elements in the same time as $T(N, P)$. Thus, when $T(N, P)$ for a fixed $\frac{N}{P}$ is plotted on a log-log graph, ideal scaling appears as a flat (horizontal) line with a slope of 0. For an algorithm to exhibit both ideal weak and strong scalability, $T(N, P)$ must be completely linear [11], [18].

We define ***parallel efficiency (strong)***, SE, as

$$SE(N, P, P_0) = \frac{T(N, P_0)P_0}{T(N, P)P}, \tag{1.3}$$

where P_0 is the reference point for which the efficiency at P is computed.

We define ***parallel efficiency (weak)***, WE, as

$$WE(N_0, P_0, P) = \frac{T(N_0P_0, P_0)}{T(N_0P, P)}, \tag{1.4}$$

where N_0 is the problem size per processing element at P_0 processing elements.

1.2.2 Parallel Systems

Shared memory parallelism on shared memory systems offers a single memory space used by all processing elements and is typically achieved through multithreading. This type of parallelism is traditionally accomplished explicitly through the use of threading libraries such as Pthreads [20] or implicitly through libraries such as OpenMP [21], a threading library that automatically parallelizes portions of an application through the use of directives. Pthreads are lower level and need to be created and managed explicitly by the programmer, whereas OpenMP directives instruct a compiler to generate and manage threads. The directives basically form an abstraction layer on top of the threading mechanism that the compiler chooses to use. OpenMP and Pthreads are also implemented at different levels, with Pthreads needing support only from the operating system and

OpenMP only from the compiler. Parallelism can be exploited with little effort through OpenMP, but the performance may not be as efficient as an explicit approach such as with Pthreads [11].

Distributed memory parallelism refers to a multiprocessor computer system in which each processing element has its own private memory, and computational tasks can operate only on local data. If remote data are required, a computational task must communicate with one or more remote processors. This internodal communication has historically been achieved through message passing across the communication network of a distributed memory system via the message passing interface (MPI), an API that defines a communication interface that explicitly communicates using sends and receives [22] and is the most commonly used method for achieving parallelism though domain decomposition.

Until more recently, applications have utilized either an MPI-only ("pure MPI") approach or a fully threaded, shared memory approach to parallelism, depending on the system architecture. To better utilize internode communication, most MPI implementations optimize internode communication through shared memory, although the MPI specification does not require such features [11]. More recently, applications have begun using ***hybrid memory parallelism*** approaches where MPI is utilized between nodes and threading is utilized within a node. This hybrid approach is often referred to as "*MPI+X*", where "*X*" is commonly Pthreads or OpenMP. The need to solve larger and more complex simulation problems while at the same time not incurring higher and higher power costs has led to an increasing focus on GPU and Intel Xeon Phi-based architectures. Many existing and most emerging high-performance computing (HPC) systems rely on such architectures.

1.3 The Uintah Software

The Uintah open-source (MIT License) software has been widely ported and used for many different types of problems involving fluids, solids, and fluid-structure interaction problems. The present status of Uintah, including applications, is described by [23]. The first documented release of Uintah was in July 2009 and the latest in September 2017 [12]. Uintah consists of a set of parallel software components and libraries that facilitate the solution of partial differential equations on structured adaptive mesh refinement (AMR)

grids. Uintah presently contains four main simulation components: 1) the multimaterial ICE [24] code for both low- and high-speed compressible flows; 2) the multimaterial, particle-based code MPM for structural mechanics; 3) the combined fluid-structure interaction (FSI) algorithm MPM-ICE [25]; and 4) the Arches turbulent reacting CFD component [26] that was designed for simulating turbulent reacting flows with participating media radiation. Uintah is highly scalable [27], [28]; runs on many National Science Foundation (NSF), Department of Energy (DOE), and Department of Defense (DOD) parallel computers; and is also used by many NSF, DOE, and DOD projects in areas such as angiogenesis, tissue engineering, green urban modeling, blast-wave simulation, semiconductor design, and multiscale materials research [23].

Uintah is unique in its combination of the MPM-ICE fluid-structure-interaction solver, Arches heat transfer solver, AMR methods, and AMT runtime system. Uintah also provides automated, large-scale parallelism through a design that maintains a clear partition between applications code and its parallel infrastructure, making it possible to achieve great increases in scalability through changes to the runtime system that executes the task graph, *without changes to the task graph specifications themselves*. The combination of the broad applications class and separation of the applications problems from a highly scalable runtime system has enabled engineers and computer scientists to focus on what each does best, significantly lowering the entry barriers to those who want to compute a parallel solution to an engineering problem. Uintah is open source and freely available and is the only widely available MPM code. The broad international user-base and rigorous testing ensure that the code may be used on a broad class of applications.

Particular advances made in Uintah are scalable adaptive mesh refinement [29] coupled with challenging multiphysics problems [30]. A key factor in improving performance has been the reduction in MPI wait time through the dynamic and even out-of-order execution of task graphs [31]. The need to reduce memory use in Uintah led to the adoption of a nodal shared-memory model in which there is only one MPI process per multicore node, and execution on individual cores is through Pthreads [32]. This shared-memory approach has made it possible to reduce memory use by a factor of 10 and to increase the scalability of Uintah to 768,000 CPU cores on complex fluid-structure interactions with adaptive mesh refinement. Uintah's thread-based runtime system [32], [33] uses decentralized execu-

tion [31] of the task graph, implemented by each CPU core requesting work itself and performing its own MPI. A shared-memory abstraction through Uintah's data warehouse hides message passing from the user but at the cost of multiple cores accessing the warehouse originally. A shared memory-approach that is lock-free [33] was implemented by making use of atomic operations (supported by modern CPUs) and thus allows efficient access by all cores to the shared data on a node. The nodal architecture of Uintah has been extended to run tasks on one or more on-node accelerators [34]. This unified, heterogeneous runtime system [35] makes use of a multistage queue architecture to organize work for CPU cores and accelerators in a dynamic way. Finally, performance portability has been achieved within Uintah [36], [37] for manycore- and GPU-based systems via the Kokkos library [38].

1.3.1 Arches Combustion Simulation Component

Arches is the primary CCMSC simulation component within Uintah, and was designed for the simulation of turbulent reacting flows with participating media radiation. It is a 3D, large eddy simulation (LES) code described in [39]. Arches uses a low-Mach number (M < 0.3), variable density formulation to model heat, mass, and momentum transport in reacting flows.

Arches solves the coupled mass, momentum, and energy conservation equations on a staggered finite-volume mesh for the gas and solid phase with combustion [26], [40]. The discretized equations are integrated in time using an explicit, strong stability-preserving second- or third-order Runge-Kutta method [41]. Spatial discretization is handled with central differencing where appropriate for energy conservation or flux limiters to maintain numerical accuracy. The low-Mach, pressure-projection formulation requires a solution of sparse linear systems at each timestep using the *Hypre* linear solver package [42]. The turbulent subgrid velocity and species fluctuations [43] are modeled with the dynamic Smagorinsky closure model. The solution procedure solves the intensity equation over a discrete set of ordinates and, like the pressure equation, is formulated as a linear system that is solved using *Hypre*. Arches is second-order accurate in space and time and is highly scalable through Uintah to 256K cores [44] and its coupled solvers, such as hypre [42]. Research using Arches has been done on radiative heat transfer using the parallel discrete

ordinates method [45] (DOM, a modeling method developed at Los Alamos National Laboratory for neutron transport) and the P1 approximation to the radiative transport equation [46]. Work done by Sun [47] and Hunsaker [48] has shown that Monte Carlo ray tracing methods are potentially more efficient and offer an alternative to DOM.

1.4 AMT Runtime System and Task Graph Motivation

The trend toward larger and more diverse computer architectures, with or without co-processors and GPU accelerators, and differing communications networks poses considerable challenges for achieving both performance and scalability when using general purpose parallel software for solving multiscale and multiphysics applications. One approach that is suggested as a suitable candidate for exascale problems is to use directed acyclic graph (DAG)-based codes, [49]. Software frameworks that execute machine-independent applications code using a runtime system that shields users from architectural complexities offer a possible solution. Such codes have the advantage that the tasks in the task graph may be executed in an adaptive manner, and if enough tasks are available, choosing an alternate task may avoid communications delays. At the same time, if the applications software is written in an abstract task graph manner so as to be executed by a runtime system that adaptively varies task graph execution, then it may be possible to combine a modest degree of portability across a number of large-scale parallel architectures with scalability at large core counts [50].

Asynchronous many-task programming models and runtime systems are becoming more widely considered as a way to address the scalability and performance challenges the exascale machine model poses to current-generation HPC codes [16]. These challenges primarily involve increased concurrency, nodal heterogeneity, deep memory hierarchies, and variable performance due to thermal throttling, all requiring adaptivity with respect to the order of execution of computational tasks. Traditional bulk synchronous parallel (BSP) approaches tend to overspecify a computation by imposing particular parallel execution order, limiting the possibilities for exploiting additional levels of parallelism.

As an alternative to the single program multiple data (SPMD) model in which a communicating sequential task in each process is executed, the AMT model views a program as a flow of data control and data relationships (constraints) between tasks. The runtime

system then extracts the appropriate level of parallelism, preserving these constraints. During program execution, these tasks are launched in any order based on the availability of their input data, enabling multiple concurrent task execution on available computational resources [16]. *A significant portion of the problem for AMT approaches then becomes one of scheduling tasks to hide latency.* Sinnen [17] identified the task-based parallelization process as:

1) subtask decomposition;
2) dependence analysis;
3) task scheduling.

Using explicit task dependencies, this scheduling operation can be deferred to the AMT runtime system. When every task is explicitly annotated with the set of tasks on which it depends, the order in which tasks are launched by the application does not limit the possible schedules, a key point to note. A single version of the source code describes all valid schedules, and the runtime has the freedom to generate a schedule to run most efficiently on a given system [51].

A natural way to consider an application's collection of tasks and their dependencies is a graph, with nodes in the graph representing tasks and edges between nodes representing explicit dependencies between tasks. Dependency analysis is considered as an important foundation for scheduling, and is a precedent relationship in which one task must be completed before another task begins to run [52]. During this process, dependency needs to be built between tasks to form a directed acyclic graph (DAG) This graph is often called a task graph. Figure 1.4 shows an example of a Uintah task graph. This structure captures both data and dependence flow, offering opportunities for asynchrony. The principal issue for this graph, outside of the scheduling and execution of the tasks it contains, is how the graph is constructed, e.g., whether the graph is constructed and analyzed at compile time, at runtime, etc., and the complexity of this operation. The basic properties AMT runtimes must consider for a graph with explicit dependencies are: 1) the runtime overhead must be low; 2) the costs involved in creating, storing, and analyzing the graph must be minimal; and 3) a graph-based approach must also scale well to very large task graphs.

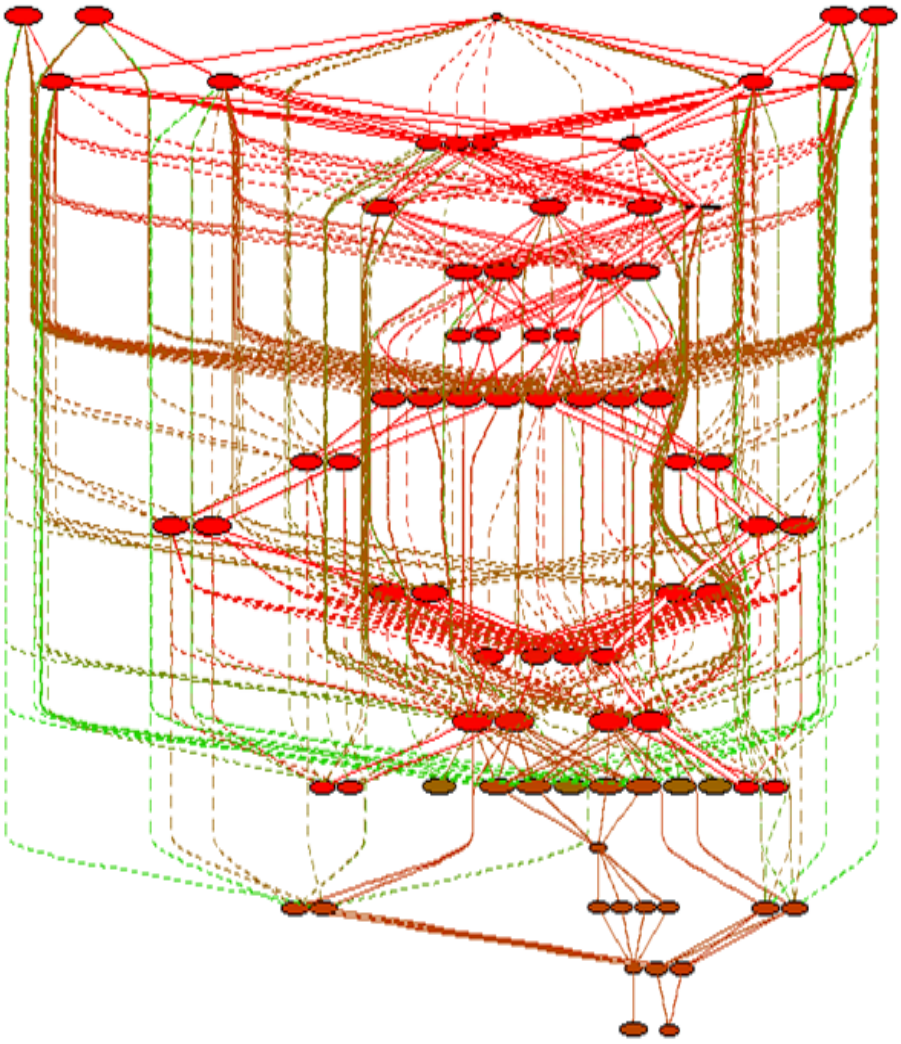

Figure 1.4. An example of a Uintah task graph.

1.4.1 Uintah Task Graph Generation

Uintah uses a structured grid of hexahedral cells defined in the Uintah input file. This Uintah grid can contain one or more adaptive mesh refinement (AMR) levels with different resolutions, and each AMR level is further divided into smaller hexahedral patches (Figure 1.5). Since finer AMR grid levels may not be continuous across the domain, Uintah uses a binary bounding volume hierarchy (BVH) tree to save patches on a particular grid level. This BVH tree is much like a k-dimensional (KD) tree, a data structure used for organizing some number of points in a space with k dimensions, effectively a binary search tree where data in each node are a k-dimensional point in space. In other words, it is a

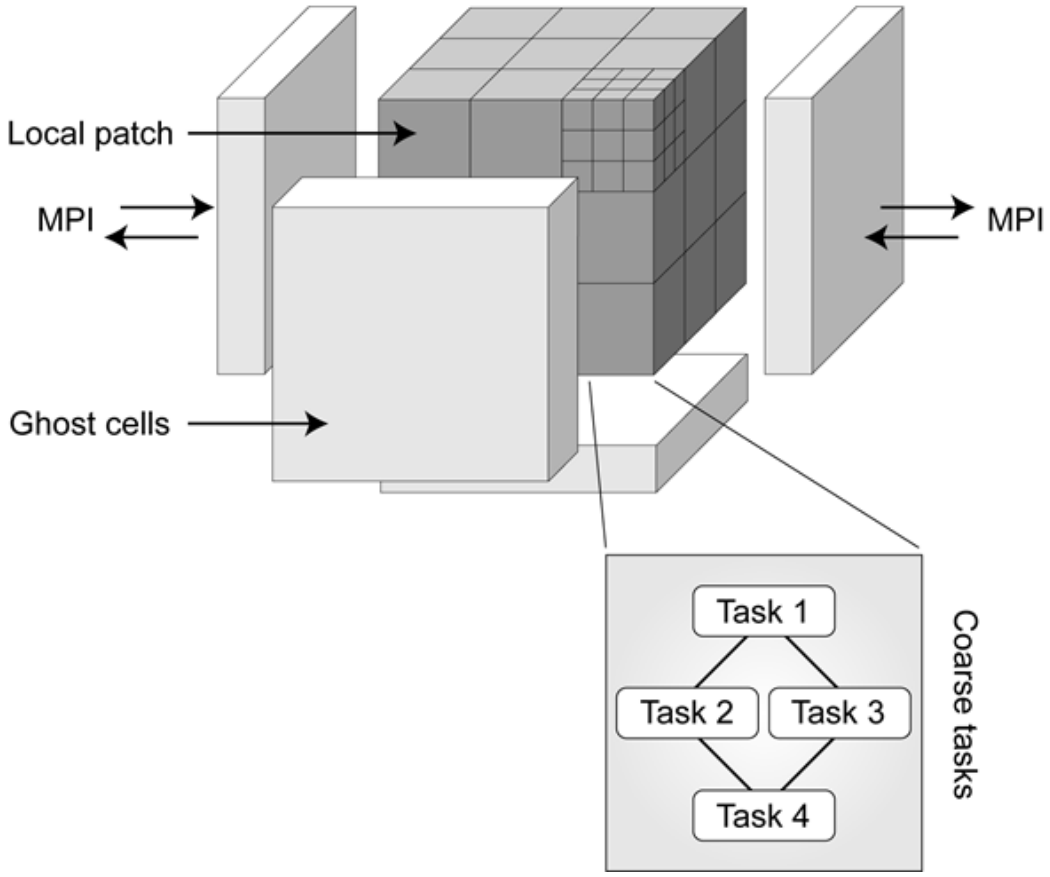

Figure 1.5. Uintah mesh patch showing four residing local tasks. Ghost cells arrive for local tasks via MPI. MPI messages are automatically generated by infrastructure through a dependency analysis phase, which uses the task specification of its *"requires."*

space-partitioning data structure useful for range and nearest neighbor searches. For quick patch queries within a given index range (used in processor neighborhood construction, among other Uintah infrastructure tasks), patches are stored in a tree structure where nodes in each level are divided in the maximum range dimension. To facilitate efficient searches, patches are sorted on each grid level with complexity

$$\mathcal{O}(n \cdot log(n)), \tag{1.5}$$

and are equally divided these patches into two sets. With $log(n)$ levels in total, the complexity of the BVH tree constructor becomes

$$\mathcal{O}(n \cdot log(n)^2). \tag{1.6}$$

Each patch set query has a time complexity of

$$\mathcal{O}(k \cdot log(n))), \tag{1.7}$$

where k represents the number of patches returned.

After patches on each grid level are created, the Uintah load balancer is used to partition and assign patches to individual compute nodes that communicate via MPI. Once each multicore node has its patches assigned, tasks are then created on the patches. A Uintah task is defined by three major attributes: 1) the "require" variables it needs to start computing with the number of ghost cells; 2) the "compute" variables that it will calculate; and 3) a serial call back function that can run on any patch. In this way, the user needs to write serial code only on a generic patch. When the call back happens, the actual patch is then fed into the task.

Uintah's task graph (Figure 1.4) is based on a distributed directed acyclic graph (DAG) of task dependencies. Uintah uses a process of checking the overlap of the input variables and the output variables for each task that makes it possible to create this directed acyclic graph by connecting the input and output of tasks. 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. The task DAG is created after a particular compute 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 occurs only once at initialization and when regridding takes place if AMR is employed. At the end of task graph compilation, an MPI message tag is assigned to each external dependency (a dependency that connects two tasks associated with patches on different multicore nodes). As Uintah creates tasks only on local and neighboring patches, task graph compilation can be done concurrently among nodes without any internodal communication.

1.5 Thesis Statement

The principal aims of this research are to demonstrate how an AMT runtime (e.g., Uintah) can be adapted, making it possible for complex multiphysics applications with global coupling to scale on current petascale and emerging exascale architectures. In this

dissertation, these aims are achieved through: 1) the use of an AMT framework itself; 2) adapting and leveraging Uintah's adaptive mesh refinement (AMR) support to dramatically reduce computational analysis, communication volume, and nodal memory footprint for radiation calculations; and 3) efficiently orchestrating the all-to-all communication required of radiation through a task graph *dependency analysis phase*, designed to efficiently handle global dependencies. This dissertation shows that it is the combination of these approaches that makes it possible to scale large industrial simulations such as the USC boiler problem with radiation.

1.6 Unique Contributions

The unique contributions of this dissertation are as follows:

1) The first study to support task scheduling and execution on GPUs within an AMT framework. [34], [35];
2) Developing a scalable approach to radiation modeling, making it available through the Uintah open-source framework [50], [53], [54];
3) Modifying the Uintah infrastructure to support these algorithms at large scale [34], [35] [50], [53], [54];
4) Achieving excellent **strong** scalability for RMCRT up to 262,144 CPU cores and 16,384 GPUs, through a unique and novel application of Uintahs AMR support [53], [54];
5) Achieving excellent **weak** strong scalability for RMCRT up to 262,144 CPU cores through a unique and novel application of Uintahs AMR support, specifically, aggressive coarsening of global, radiation mesh as problem size increases;
6) Achieving efficient, automated halo management within an AMT runtime for globally coupled problems [53], [55], [10];
7) Showing these ideas work for an industrial-size production boiler problem, running on the largest petascale architectures available [10].

These contributions have been achieved by:

- Using a reverse Monte Carlo ray tracing (RMCRT) approach for radiation modeling. This idea originally comes from the heat transfer and oncology communities, **but has never been modified or adapted for use at large scale** [34], [35] [50], [53], [54];

- Reducing the cost of RMCRT all-to-all communication phase and nodal memory footprint by using an adaptive mesh refinement (AMR) approach in order to achieve scaling; fine mesh is used only locally, and a coarse mesh is used elsewhere for the RMCRT ray marching algorithm. *Although the AMR methods used in this work are not new, the application of these methods to radiative heat transfer algorithms and their scalability is novel* [50], [53], [54];
- The introduction of novel nonblocking, thread-scalable data structures for managing asynchronous MPI communication requests, replacing previously problematic Mutex-protected vectors of MPI communication records [54];
- Addressing complexity involved with Uintah task graphs at scale with RMCRT, task graph compilation, and automated MPI message generation with global halos [53], [55];
- Temporal scheduling, a feature that adds support within Uintah for multiple primary task graphs. Through provided interfaces, an application can directly place tasks into a task graph of choice, and provide a computed task graph index for the Uintah infrastructure to execute for a given timestep [55];
- Spatial scheduling, a feature that allows application to schedule tasks on a subset of the entire domain. Previously, computational tasks were scheduled uniformly across the entire domain by Uintah [10].

These contributions have now made Uintah the only AMT runtime offering auto MPI message generation and scalable task graphs when considering problems with global data dependencies. Apart from Uintah, most AMT runtimes require the application developer explicitly supply far more characteristics of data dependencies. A global dependency problem such as radiative heat transfer would require explicitly writing tasks for all data dependency pairs across the computational domain.

Finally, the broader impact of this work may ultimately include algorithmic developments for related problems with pervasive all-to-all type communications in general, such as long-range electrostatics in molecular dynamics, and will be of importance to a broad class of users, developers, scientists, and students for whom such problems are presently a bottleneck.

1.7 Document Organization

The contributions outlined in Section 1.6 are presented in detail throughout the remainder of this dissertation, which is organized as follows: a detailed summary of radiation modeling is provided in Chapter 2, followed by a review of related work with an overview of existing AMT runtime systems and programming models in Chapter 3. The introduction of radiation as the driver for the Utah CCMSC as well as the origins of Uintah's task graph scalability challenges are provided in Chapter 4. Chapter 5 and Chapter 6 present strong and weak scaling results for the RMCRT radiation model, to 262,144 CPU cores and 16,384 GPUs, respectively. Addressing Uintah task graph scalability in the context of a large-scale industrial boiler problem is covered in Chapter 7. Addressing final task graph complexity for fully scalable task graphs with radiation within Uintah is covered in Chapter 8, with future directions provided. Finally, a summary of work accomplished for this dissertation, conclusions, and lessons learned are given in Chapter 9.

CHAPTER 2

RADIATION MODELING

The heat transfer problem arising from the clean coal boilers being modeled by the Utah CCMSC with the Uintah framework has thermal radiation as a dominant heat transfer mode and involves solving the conservation of energy equation (2.1) and radiative transfer equation (2.2) simultaneously. Scalable modeling of radiation is currently one of the most challenging problems in large-scale simulations, due to the global, all-to-all nature of radiation [17], potentially affecting all regions of the domain simultaneously at a single instance in time. Treatment of thermal radiation is different from neutron diffusion because radiation travels at light speed, so transit times are usually negligible, whereas neutron speeds are much slower and time-of-flight must be considered [56]. To simulate thermal transport, two fundamental approaches exist: random walk simulations and finite element/finite volume simulations, e.g., discrete ordinates method (DOM) [57], [58], which involves solving many large systems of equations. Accurate radiative-heat transfer algorithms that handle complex physics are inherently computationally expensive [59], particularly when high accuracy is desired in cases where spectral or geometric complexity is involved. These algorithms also have limitations with respect to scalability, bias, and accuracy.

Arches (Section 1.3.1) was designed to solve the mass, momentum, and thermal energy governing equations inherent to coupled turbulent reacting flows. Arches has relied primarily on a legacy DOM solver to compute the radiative source term in the energy equation [26]. Monte Carlo ray tracing (MCRT) methods for solving the radiative transport equation offer higher accuracy in two key areas where DOM suffers: geometric fidelity and spectral resolution. In applications where such high accuracy is important, MCRT can become more efficient than DOM approaches. In particular, MCRT can potentially reduce the cost significantly by taking advantage of modern hardware on large distributed

shared memory machines [60], and now on distributed memory systems with on-node accelerators such as graphics processing units (GPUs).

Uintah currently supports three fundamentally different approaches to solving the radiative transfer equation (RTE) to predict heat flux and its divergence (operator) in these domains: 1) discrete ordinates method (Section 2.2); 2) reverse Monte Carlo ray tracing (Section 2.3); and 3) spatial transport sweeps (Section 2.4). A more comprehensive discussion on the relative strengths and weaknesses of these three approaches is provided by Tencer et al. [56], with initial discussion of the feasibility of these methods for emerging architectures.

2.1 Solving the Radiative Transport Equation

In combustion simulations, three different physical processes govern the dynamics of the problem: fluid flow, chemical reactions, and heat transfer, with radiative heat transfer being a dominant mode [61]. Thermal radiation in the target boiler simulations is loosely coupled to the computational fluid dynamics (CFD) due to time-scale separation. For example, when considering nonsteady fluid flow in a typical application, timesteps are of order 10^3 to 10^6 , whereas chemical reactions cover a much wider range of time scales, from 1 to 10^{10} seconds. With radiative heat transfer occurring at the speed of light, the resulting time scales are on the order of 10^9. To enable accurate modeling of many combustion applications, the inclusion of the interaction between these three processes is essential [61]. Thermal radiation is the rightmost source term ($\nabla \cdot q_r$) in the conservation of energy equation shown by

$$\rho c_v \frac{DT}{Dt} = -\nabla \cdot (\kappa \nabla T) - p\nabla \cdot v + \Phi + Q''' - \nabla \cdot q_r, \tag{2.1}$$

where ρ is density, c_v is the specific heat, T is the temperature field, p is the pressure, k is the thermal conductivity, c is the velocity vector, Φ is the dissipation function, Q''' is the heat generated within the medium (e.g., chemical reaction), and $\nabla \cdot q_r$ is the net radiative source term. The energy equation is then conventionally solved by Arches (finite volume), and the temperature field, T, is used to compute the net radiative source term. This net radiative source term is then fed back into the energy equation (for the ongoing CFD calculation), which is solved to update the temperature field, T.

A radiatively participating medium can emit, absorb, and scatter thermal radiation. The RTE (2.2) is the equation describing the interaction of absorption, emission, and scattering for radiative heat transfer and is an integro-differential equation with three spatial variables and two angles that determine the direction of $\hat{s}$ [62].

$$\begin{aligned} \frac{dI_\eta(\hat{s})}{ds} &= \hat{s}\nabla I_{\eta(\hat{s})} \\ &= k_\eta I_\eta - \beta_\eta I_\eta(\hat{s}) \\ &+ \frac{\sigma_{s\eta}}{4\pi}\int_{4\pi} I_\eta(\hat{s})\Phi_\eta(\hat{s_i},\hat{s})d\Omega_i, \end{aligned} \tag{2.2}$$

In (2.2), k_η is the absorption coefficient, $\sigma_{s\eta}$ is the scattering coefficient (dependent on the incoming direction s and wave number η), β_η is the extinction coefficient that describes total loss in radiative intensity, and I_η is the change in intensity of incoming radiation from point s to point $s + ds$ and is determined by summing the contributions from emission, absorption, and scattering from direction $\hat{s}$ and scattering into the same direction $\hat{s}$ at wave number η. $\Phi_\eta(\hat{s_i},\hat{s})$ is the phase function that describes the probability that a ray coming from direction s_i will scatter into direction $\hat{s}$, and integration is performed over the entire solid angle Ω_i [63], [62].

In general, the heat flux divergence can be computed using

$$\nabla \cdot q = 4 * \pi * S - \int_{4\pi} I_\Omega d\Omega, \tag{2.3}$$

where S is the local source term for radiative intensity, and I_Ω is computed using the RTE for gray nonscattering media requiring a global solve via

$$\frac{dI_\Omega}{ds} = k * (S - I_\Omega). \tag{2.4}$$

Here, s is the one-dimensional (1D) spatial coordinate oriented in the direction in which intensity I_Ω is being followed, and k is the absorption or attenuation coefficient. The lack of time in the RTE implies instantaneous transport of the intensity, appropriate for most applications. The methods for solving the RTE discussed here aim to solve for I_Ω using (2.4), which can then be integrated to compute the radiative flux and divergence.

2.1.1 Modeling Spectral Effects

A common approximation employed in solving the RTE in simulations consists of neglecting the spectral dependency of the absorption coefficient by assuming a gray gas

(meaning they behave the same radiatively across the entire spectrum). This assumption enables the efficient use of finite difference schemes, such as the discrete ordinates method (Section 2.2), which require a low computational effort and provide a high level of accuracy [64]. Modeling spectral radiation within Arches radiation models was initially avoided due to the computational cost. Ultimately, it was argued that modeling spectral affects was unnecessary for the CCMSC target boiler simulations in particular, because the particles are known to be effectively gray, and are the dominant emitters/absorbers in these systems. Shortly after the development of transport sweeps (Section 2.4), a spectral solver built off of the spatial/temporal scheduling (Chapter 7) was developed and tested. Simulations showed that using five weighted sum of gray gases only slightly changed the solution (less than 5%). Although this study was limited, it did somewhat validate the longstanding persuasion that the gray particles washed out the gaseous spectral affects. Based on these results, as well as the examples given by those who have studied the influence of radiative heat transfer in turbulent flows using the gray gas approximation [65], [66], [67], [64], the radiation models used by the Utah CCMSC, and discussed here, do not model spectral effects and use a gray gas approximation.

2.2 Discrete Ordinates Method

The radiation calculation in the CCMSC boiler simulations, in which the radiative-flux divergence at each cell of the discretized domain is calculated, can take up to 50% of the overall CPU time per timestep using the discrete ordinates method (DOM) [57], one of the standard approaches to computing radiative heat transfer. DOM transforms the RTE into a set of simultaneous partial differential equations and solves the RTE by discretizing the left-hand side of (2.4), which results in a four- or seven-point stencil, depending on the order of the first derivative. This transformation is accomplished by discretizing the angular domain into a well-defined set of ordinate directions and integrating along the path lengths. Instabilities arise when using the higher order method, so often the seven-point stencil is avoided, or a combination of the two stencils is used. The four-point stencil results in numerical diffusion that impacts the fidelity of the solve, but for low ordinate counts it can improve solution accuracy.

DOM was first used in stellar radiation and then later adopted by the neutron transport

community [59]. This method is computationally expensive; involves multiple global, sparse linear solves; and presents challenges both with the incorporation of radiation physics such as scattering and the use of parallel computers at very large scales. DOM has two major shortcomings: 1) the ray effect, a consequence of angular discretization; and 2) false scattering, a consequence of spatial discretization errors [8]. The ray effect is well documented in [61], and can be reduced by increasing the size of control volumes and surface zones. Veljkovic [61] mentions that a primary consequence of the shortcomings inherent to DOM is that with the finer mesh, a finer angular discretization must be used. These requirements can lead to a high-performance penalty, as the cost of this method grows as $\mathcal{O}(h^5)$, where h is a characteristic length of the mesh. As shown in [10], DOM has been demonstrated to scale, but it is computationally expensive, due to the numerous global sparse linear solves. These solves are handled by Uintah, which is fully equipped to use the Hypre solver package [68]. In the case shown in [10], as many as 30-40 backsolves were required per radiation step, with up to an order of magnitude more solves required in other cases. It should be noted that, due to their computational cost, the radiation solves are computed roughly once every 10 timesteps, because the radiation solution does not change quickly enough to warrant a more frequent radiation calculation.

DOM poses no particular challenge to Uintah regarding task graph compilation, since the global, sparse linear solves are carried out by the Hypre solver package, with Uintah having no knowledge of the underlying task and data dependencies.

2.3 Reverse Monte Carlo Ray Tracing

The CCMSC has been actively pursuing the use of photon Monte Carlo (PMC) methods, originally considered in [9], and more specifically reverse Monte Carlo ray tracing (RMCRT) [63], initially developed in [47] and [48] to compute radiative heat flux and its divergence when such high accuracy is important. Monte Carlo ray tracing (MCRT) methods for solving the RTE offer higher accuracy in two key areas where DOM suffers: geometric fidelity and spectral resolution.

DOM, Sweeps, MCRT, and RMCRT all aim to approximate the radiative transfer equation. In the case of RMCRT, a statistically significant number of rays (photon bundles) are traced from a computational cell to the point of extinction. This method is then able to cal-

culate energy gains and losses for every element in the computational domain. The process is considered "reverse" through the Helmholtz Reciprocity Principle, e.g., incoming and outgoing intensity can be considered as reversals of each other [69]. Through this process, the divergence of the heat flux for every subvolume in the domain (and radiative heat flux for surfaces, e.g., boiler walls) is computed by (2.5), as rays accumulate and attenuate intensity (measured in watts per square meter, SI units based on the Stefan-Boltzmann constant) according to the RTE for an absorbing, emitting, and scattering medium

$$\nabla \cdot q \;=\; \kappa(4\pi I_{emmited} - \int_{4\pi} I_{absorbed} \mathrm{d}\Omega), \tag{2.5}$$

where the rightmost term, $\int_{4\pi} I_{absorbed} \mathrm{d}\Omega$ is represented by the sum $\sum_{r=1}^{N} I_r \frac{4\pi}{N}$ for each ray r up to N rays. The integration is performed over the entire solid angle Ω. Ray origins are randomly distributed throughout a given computational cell. In our implementation, the Mersenne Twister random number generator [70] is used to generate ray origins. The ray marching algorithm proceeds in a similar fashion to the voxel traversal algorithm shown in [71]. Following the detailed description of the RMCRT-specific ray marching algorithm provided by Hunsaker in [59], the psuedocode in Algorithm 1 and Algorithm 2 shows how (2.5) is implemented within the Uintah RMCRT model, computing the flux divergence for each computational cell in the domain.

Within the Uintah RMCRT module, rays are traced backwards from the detector, thus eliminating the need to track ray bundles that never reach the detector [63]. Rather than integrating the energy lost as a ray traverses the domain, RMCRT integrates the incoming intensity absorbed at the origin, where the ray was emitted. RMCRT is more amenable to domain decomposition, and thus, so is Uintah's parallelization scheme due to the backward nature of the process [8] and the mutual exclusivity of the rays themselves. Figure 2.1 shows the back path of a ray from **S** to the emitter **E**, on a nine-cell structured mesh patch. Each i^{th} cell has its own temperature T_i, absorption coefficient κ_i, scattering coefficient σ_i, and appropriate pathlengths $l_{i,j}$ [34]. In each case, the incoming intensity is calculated in each cell and then traced back through the other cells. The Uintah RMCRT module computes how much of the outgoing intensity has been attenuated along the path. When a ray hits a boundary, as on surface 17 in Figure 2.1, the incoming intensities will be partially absorbed by the surface. When a ray hits a hot boundary surface, its emitted surface

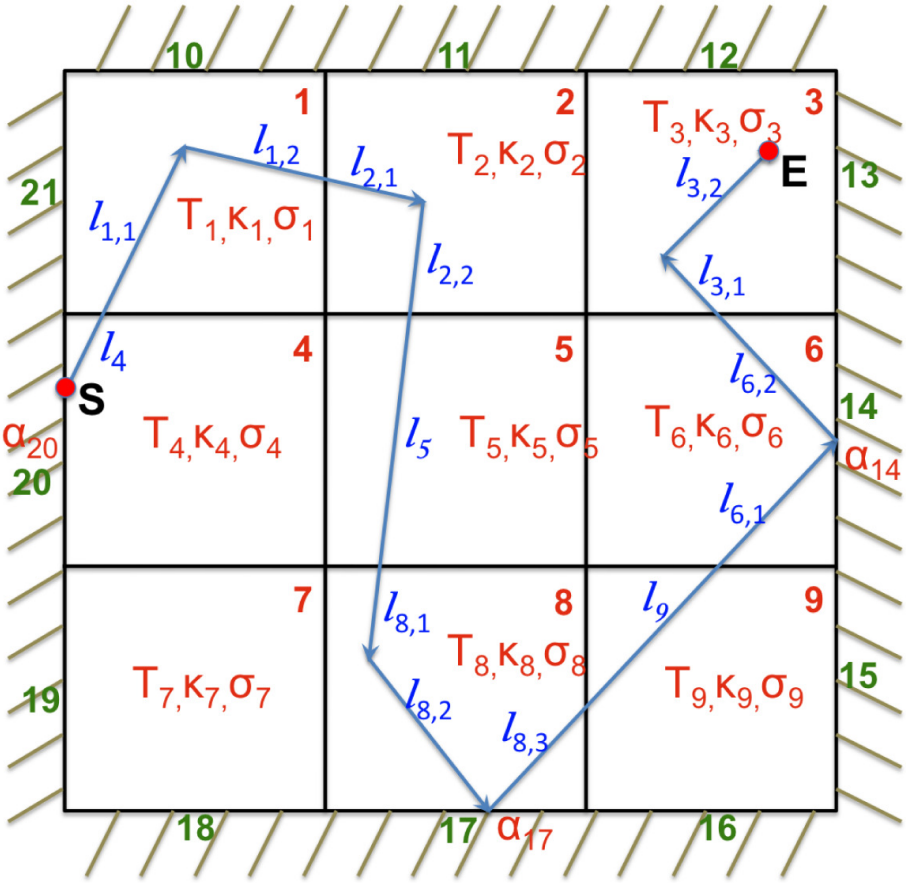

Figure 2.1. 2D Outline of reverse Monte Carlo ray tracing.

intensity contributes to point **S**. Rays are terminated when their intensity is sufficiently small [34].

RMCRT uses rays more efficiently than forward MCRT, but it is still an *all-to-all* method, for which all of the geometric information and radiative properties (temperature T, absorption coefficient κ, and cellType (boundary or flow cell)) for the entire computational domain must be accessible by every ray [8]. When using a ray tracing approach, two approaches for parallelizing the computation are considered for structured grids:

1) Parallelize by patch-based domain decomposition with local information only and pass ray information at patch boundaries via MPI;
2) Parallelize by patch-based domain decomposition with global information and reconstruct the domain for the quantities of interest on each node by passing domain information via MPI.

Algorithm 1 Ray marching pseudocode for Uintah RMCRT radiation model.

```
1:  procedure ray_trace( patches, materials, oldDataWarehouse, newDataWarehouse)
2:      for all (cells in a mesh patch) do                ▷ patch-based region of interest
3:          intensity_sum gets 0)                          ▷ initialize intensity to 0
4:          for all (rays in a cell) do
5:              find_ray_direction()
6:              find_ray_location()
7:              update_intensity_sum()
8:          end for
9:          compute divergence of heat flux
10:     end for
11: end procedure
```

Algorithm 2 Pseudocode for updating sum of radiative intensities.

```
1:  procedure UPDATE_INTENSITY_SUM(ray_origin, ray_direction,num_steps,lo_idx,hi_idx )
2:      initialize all ray marching variables
3:      while (intensity > threshold) do                                  ▷ threshold loop
4:          while (current ray is within computational domain) do
5:              obtain per-cell coefficients
6:              advance ray to next cell          ▷ accounts for moving between mesh levels
7:              update ray marching variables
8:              update ray location
9:              in_domain ← cell_type[curr]              ▷ terminate ray if not a flow cell
10:             compute optical thickness
11:             compute contribution of current cell to intensity_sum
12:         end while
13:     end while
14:     compute wall emissivity
15:     compute intensity
16:     update intensity sum
17: end procedure
```

The first approach becomes untenable due to potentially billions of rays whose information would need to be communicated as they traverse the domain. In the second approach, the primary difficulty is efficiently constructing the global information for millions of cells in a spatially decomposed (patch-based) domain. The second approach is the one used in this work. Although reconstruction of all geometry on each node has been shown to limit the size of the problem that can be computed [48], we will show that the multilevel mechanisms in Uintah allow representing a portion of the domain at a coarser resolution, thus lowering the memory usage and message volume, ultimately scaling to over 256,000 CPU cores. The hybrid memory approach of Uintah also helps as only one copy of geometry

and radiative properties is needed per multicore node [33]. RMCRT will be invoked largely on coarser mesh levels, and the CFD calculation will be performed on the highest resolved mesh.

2.4 Spatial Transport Sweeps

A third approach allows for solving for the intensities with a four-point stencil known as *spatial transport sweeps*, a *sweeping method*, or simply *sweeps* [72], [73], [74], [75], [76]. Sweeps is a lightweight spatially serial algorithm in which spatial dependencies dictate the speed of the algorithm. These dependencies impose serialized internodal communication requirements and account for the bulk of the algorithm's cost. Although the sweeping method is inherently serial, it can be parallelized over many ordinate directions and spectral frequencies. Within Uintah, this spatial dependency challenge is addressed using spatial task-scheduling techniques detailed in Section 7.5. These techniques are central to this dissertation.

Sweeps is formulated in a way that it can be solved as a sparse linear system or as a sweeping algorithm with one-sided spatial dependencies. Because sweeps is mathematically identical to the linear solve, the A-matrix constructed for the first iteration of the linear solve is modified and used for sweeps. Sweeps, mathematically, is a back substitution problem of this linear system, or a single Gauss-Seidel iteration, which for an upwind stencil and down-wind sweeping direction, converges exactly on the solution after a single iteration. In essence, a sweep is a single Gauss-Seidel-linear solve iteration without approximation. This process is done in stages for each intensity and phase, both of which are defined below. Although this staging process is serialized by the reliance of corner-to-corner dependencies, it can show good performance when sweeping a large quantity of independent solves. For a nonscattering medium, the angular and spectral intensities are all independent of each other, allowing for parallelization of the solve.

On large, distributed memory systems, the intensities are stored on multiple compute nodes, making communication between them expensive and inefficient. To address this problem, one processor (or node) needs to operate *only on intensities that have satisfied their spatial dependency*. The method shown here is based on the algorithm for a simple rectangular domain; however, it further supports identification of these dependencies for

complex domains with nonrectangular shapes. To most easily convey the methodology used, we start by describing the algorithm on a rectangular domain.

Consider a domain with 3×3 subunits. Within Uintah, these subunits are referred to as *patches*. A diagram showing how these patches are divided is shown in Figure 2.2 and Figure 2.3. The number labeling each patch (Figure 2.2) designates the phase in which a sweep is relevant for a single intensity, from the $x + y + z + octant$, with a single wave number. Note that these phases are defined as

$$P = x_i + y_i + z_i, \tag{2.6}$$

where x_i, y_i, and z_i are the patch indices in the x, y, and z directions. The patch indices are defined as the number of patches away from the origin patch. Hence, the total number of phases required to complete a single complete full-domain sweep is

$$P_{max} = x_{max} + y_{max} + z_{max}, \tag{2.7}$$

where $x_{max}, y_{max}, z_{max}$ are the maximum. Numbers designate the designated phase indices of the patches within the domain. We determine the patch indices using the subdomain with the patch ID provided by Uintah.

Uintah, by default, numbers its patches in the order of z, y, x (Figure 2.3). From this numbering, we can determine the point in space in which the sweep is currently located using modulo operators, the patch dimensions, and the patch ID. The patch index is then converted to the patch indices x_i, y_i, z_i for each patch. Using the Uintah task scheduler, we can indicate to a task what this phase is. This process is more complicated when conducting sweeps with multiple intensity directions. First, consider additional intensities that are in the x^+, y^+, z^+ directions. To keep as many processors busy as possible in the computation, we create stages.

A stage S is defined as $S = I + P$, where I is the intensity index relevant to a single octant. The maximum number of stages is known via the equation $S_{max} = I_{max} + P_{max}$. The phase equation for the x^-, y^-, z^- octant results in

$$P = x_{max} - x_i + y_{max} - y_i + z_{max} - z_i. \tag{2.8}$$

Hence, eight phase equations are possible, depending on the combination of directions. The task designates the stage and intensity and then computes a function mapping its

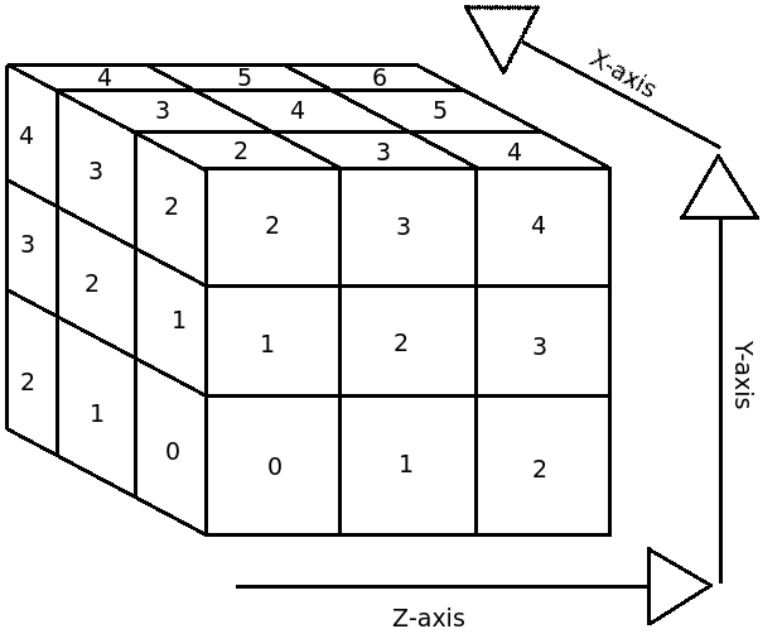

Figure 2.2. A rectangular domain divided into 27 subdomains, labeled by the designated phase.

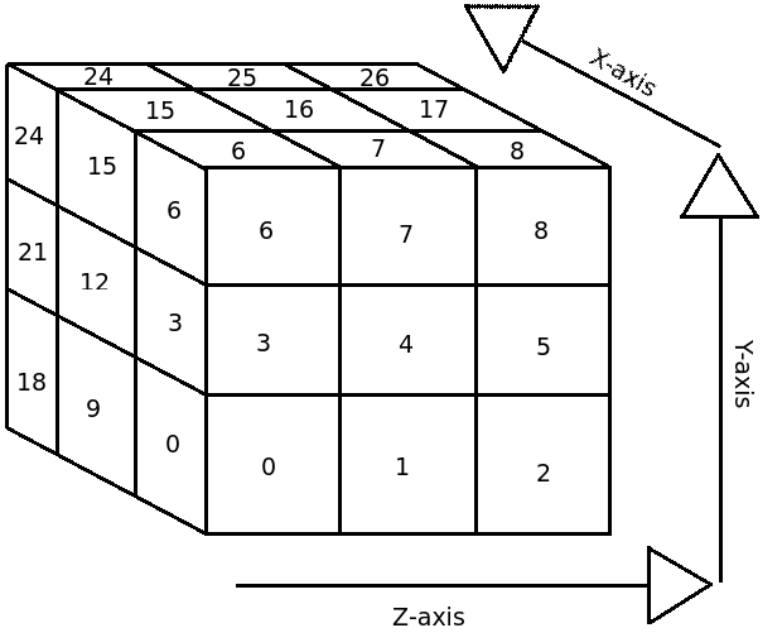

Figure 2.3. A rectangular domain divided into 27 subdomains, labeled by Uintah patch ID.

patch ID to its spatial patch index using a series of modulos. If the patch and intensity are relevant to the local processor, then the task executes; otherwise, it exits the task.

Algorithm 3 provides pseudocode responsible for creating `Uintah::PatchSubsets` for the spatial tasks used in transport sweeps. For clarity, the `Uintah::PatchSet` is a a larger set of `Uintah::Patch` (introduced with Figure 1.5), typically assigned to an entire owning MPI rank, whereas the `Uintah::PatchSubset` is the subset of these Patches, provided to a computational task at runtime by Uintah. Starting at Line 22 in Algorithm 3, different `Uintah::PatchSubsets` are created because, depending on direction of the sweep, the relevant `Uintah::PatchSubset` changes the corner in which the sweep begins, hence the std::vector of `Uintah::PatchSubsets`. In Algorithm 3, "p" denotes moving in the positive direction (e.g., $X_p \Rightarrow$ x+), and "m" denotes moving in the negative direction (e.g., $X_m \Rightarrow$ x-). With the addition of spatial scheduling (Section 7.5), a key contribution of this dissertation, tasks are scheduled only on the Uintah patches in the specified Uintah::PatchSets, and not uniformly across the entire computational domain as was done historically before this work. Spatial scheduling is central to the Uintah formulation of transport sweeps.

Algorithm 4 provides the implementation detailing how the requires→modifies chaining is formulated to facilitate inter-patch communication. Line 33 in Algorithm 4 shows the usage of the `Uintah::PatchSubsets` created starting in Line 22 of Algorithm 3. Both Algorithm 3 and Algorithm 4 are currently part of the `schedule_computeSourceSweep` method, responsible for creating and scheduling the spatial tasks involved with transport sweeps within the Arches component.

Algorithm 3 Creating a `Uintah::PatchSubset` for each spatial task.

```
1: // sweeping algorithm uses spatial parallelism to improve efficiency
2: // total number of tasks = num_sweeping_phases + num_ordinates-1
3: const PatchSet* all_patches ← lb.getPerProcPatchSet(level)
4: std::vector<const Patch*> local_patches ← all_patches.getSubset()
5: for (int i=0; i<2; ++i) do
6:     for (int j=0; j<2; ++j) do
7:         for (int k=0; k<2; ++k) do
8:             // basic concept i + j + k must always equal phase
9:             int phase ← dir_phase_adj[i][j][k]
10:            for (; phase < total_phases-dir_phase_adj[1-i][1-j][1-k]; ++phase) do
11:                std::vector<const Patch*> relevant_patches(0)
12:                ...                ▷ perform checks that i,j,k are in the domain
13:                ...                        ▷ adjust for non- x+,y+,z+ directions
14:                Point patch_center(iadj,jadj,kadj)              ▷ using adjusted i,j,k
15:                relevant_patches.push_back(level.getPatchFromPoint(patch_center))
16:                PatchSubset* sweeping_patches ← new PatchSubset(relevant_patches)
17:
18:                sweeping_patches.sort()
19:
20:                // p denotes in the positive direction, e.g., Xp -> x+
21:                // m denotes in the negative direction, e.g., Xm -> x-
22:                if (i==0 && j==0 && k==0) then
23:                    patches_XpYpZp.push_back(sweeping_patches)
24:                end if
25:                if (i==0 && j==0 && k==1) then
26:                    patches_XpYpZm.push_back(sweeping_patches)
27:                end if
28:                if (i==0 && j==1 && k==0) then
29:                    patches_XpYmZp.push_back(sweeping_patches)
30:                end if
31:                if (i==0 && j==1 && k==1) then
32:                    patches_XpYmZm.push_back(sweeping_patches)
33:                end if
34:                if (i==1 && j==0 && k==0) then
35:                    patches_XmYpZp.push_back(sweeping_patches)
36:                end if
37:                if (i==1 && j==1 && k==0) then
38:                    patches_XmYmZp.push_back(sweeping_patches)
39:                end if
40:                if (i==1 && j==1 && k==1) then
41:                    patches_XmYmZm.push_back(sweeping_patches)
42:                end if
43:            end for                                                    ▷ phase
44:        end for                                                ▷ z+ z- direction
45:    end for                                                    ▷ y+ y- direction
46: end for                                                       ▷ x+ x- direction
```

Algorithm 4 Scheduling subsweeps as needed. This scheduling is achieved via Uintah dependency (requires → modifies) chaining to facilitate interpatch communication.

```
1:
2: //-----------------------------------------------------------------------
3: // These tasks compute the intensities, on a per patch basis.
4: // The spatial tasks were developed by looking at a single direction,
5: // then re-used for other directions by re-mapping the processor IDs.
6: //-----------------------------------------------------------------------
7: int num_octants ← 8
8: for (int istage=0; istage<num_stages; ++istage) do
9:     for idir=0; idir<num_octants; ++idir do
10:        int first_intensity ← idir*num_dir/num_octants
11:        // do calculation for non-cubic domain adjustment
12:        // assumes that ordinates are stored in octants (8 bins)
13:        // with similar directional properties in each bin
14:        ...
15:        // loop over per-octant-intensities
16:        for all (per-octant intensities) do
17:            int intensity_iter ← (curr_oct_intensity + idir * (num_dir/num_octants))
18:            std::stringstream task_name ← DO_Radiation_sweep
19:            task_name << istage << _ << intensity_iter
20:            Task task ← new Task(task_name.str(),istage, intensity_iter)
21:            task.requires(cell_type_label,0);
22:            task.requires(abskt_label,0);
23:            // requires->modifies chaining for inter-patch communication
24:            for (int iband=0; iband< num_bands;++iband) do
25:                VarLabel curr_label ← labels[intensity_iter+iband*DO_Model::getIntOrdinates()
26:                task.modifies(curr_label)
27:                // Toggle comm depending on phase/intensity using equation:
28:                // iStage = iPhase + intensity_within_octant_x
29:                // 8 different patch subsets, due to 8 octants
30:                // also using temporal scheduling -- radiation task graph
31:                if (DO_model.xDir/ydir/zdir(first_intensity) == 1 then
32:                    task.requires(labels[curr_label],patches_XmYmZm[istage-int_x],1
33:                    sched.addTask(task,patches_XmYmZm[istage-int_x],Radiation_TG);
34:                end if
35:                ...
36:                // do this same chaining for all combos x,y,z
37:                // 111,110,101,100,011,010,001,000
38:                ...
39:                if (DO_model.xDir/ydir/zdir(first_intensity) == 0 then
40:                    task.requires(labels[curr_label],patches_XmYmZm[istage-int_x],1
41:                    sched.addTask(task,patches_XpYpZp[istage-int_x],Radiation_TG);
42:                end if
43:            end for                                                ▷ iband
44:        end for                                  ▷ intensity_x (per-octant intensities)
45:    end for                                                        ▷ octants
46: end for                                                         ▷ istage
```

CHAPTER 3

OVERVIEW OF EXISTING AMT RUNTIMES

3.1 Legion

Legion [77] is a data-centric task-based programming model with higher level constructs, moving away from the procedural style of MPI and Charm++ to a highly declarative program expression [16]. The principal Legion runtime component is Realm, the "low-level" runtime that manages the execution of a mapped Legion application [51]. Legion uses GASNet as the underlying communication layer. The Legion "task graph" naturally exposes the available parallelism in the application as well as presents opportunities for hiding the latency of any required communication, as is done in Uintah. Although asynchronous task launches and nonblocking data movement are common in existing programming models, Realm makes all runtime operations asynchronous, which includes resource management, performance feedback, and even synchronization primitives [51].

Although there are many commonalities in the graph approach between the AMT runtimes, perhaps the most obvious difference between Legion and Uintah is the graph creation and execution strategy, *which for Legion is to execute the task graph as it is being constructed*, keeping only a frontier of the graph available and containing only ready tasks that have not yet been executed. Without the whole graph available, any scheduling or mapping decisions made by the runtime are local. The frontier of the Legion task graph must include enough parallelism that good local scheduling or mapping decisions are even possible. The key goal is to make sure the application is able to "run ahead" of the actual execution and is achieved by a depth-first traversal of one components' tasks before moving on to the next. Explicit dependencies allow the enumeration and execution of tasks to be asynchronous, but the maximal benefit is obtained when all dependencies are explicit. Any application dependency that cannot be expressed explicitly requires some other task to wait before launching it, and this wait brings back all the drawbacks of the implicit dependency model.

Legion distinguishes between a `SingleTask` that is similar to a single function call and an `IndexSpaceTask` that is similar to a potentially nested `for loop` around a function call with the restriction that each invocation be independent. If the `IndexSpace` is explicitly declared as disjoint, the runtime can reduce the associated dynamic runtime analysis cost compared to the alternative of expressing each index space task as a single task. Every Legion program executes as a tree of tasks with a top-level task spawning subtasks that can recursively spawn further subtasks [51]. A root task is initially executed on some processor in the machine, and this task can launch an arbitrary number of subtasks. Each subtask can in turn launch its own subtasks. The task tree has no depth limit in a Legion program execution. A subtask can access only regions (or subregions) that its parent task could access; furthermore, the subtask can have permissions only on a region compatible with the parent's permissions. As mentioned above, tasks must a priori declare all data they will operate on. Because of the tree structure, parent tasks must a priori declare any data that their children will operate on [16].

Effectively, the graph is a `RegionTreeForest`, which provides good high-level details about functionality. This graph is made up of `IndexTreeNodes`, which are inserted into the tree dynamically at runtime. This approach again allows Legion to execute the task graph as it is being constructed. In short, the complexity of this operation is amortized over the entire execution of the task graph, with average complexity being just that of insertion and the subsequent analysis that is triggered upon task insertion. This operation is fully overlapped with task execution (e.g., thread-safe insertion). The RegionTreeForest defines the interface between all RegionTreeForest data structures and the rest of the Legion runtime. The RegionTreeForest is the Uintah equivalent of the DataWarehouse, TaskGraph, and BVHTree combined.

In contrast to Uintah, automatic dependency management within the Legion runtime system requires the application developer explicitly supply far more characteristics of data dependencies. The implication is that for a globally couple problem like the CCMSC target boiler simulation with radiation, this approach would require explicit task creation from the developer for globally communicating the radiative properties for computation. This scenario effectively becomes an intractable programming problem, explicitly writing tasks for all pairs across the computational domain, making correctness difficult to achieve.

3.2 Charm++

Charm++ [78, 79, 80] is an AMT runtime system designed around the migratable-objects programming model and the actor execution model. The actor execution model differs subtly from the communicating sequential task (CSP) model of MPI: with CSP, a worker is typically considered to be active until it reaches a synchronization or communication point, whereas in the actor model, workers ("actors") are considered inactive until they receive a message. Programming in Charm++ resembles that of MPI more than Legion or Uintah [16]. A Charm++ program is essentially a C++ program where some components describe its parallel structure. Sequential code can be written using any programming technologies that cooperate with the C++ toolchain, which includes C and Fortran. Parallel entities in the user's code are written in C++. These entities interact with the Charm++ framework via inherited classes and function calls [80].

The basic unit of parallel computation in Charm++ is the *chare*. According to the Charm++ manual [81], "A Charm++ computation consists of a large number of chares distributed on available processors of the machine, and interacting with each other via asynchronous method invocations." These chare object methods, which may be invoked remotely, are known as entry methods, where invocation is performed asynchronously, in keeping with the non-preemptible nature of work units in Charm++. Asynchronous entry method invocation on a remote chare is equivalent to remote procedure calls (RPC) or active message passing. The parameters to a remote method invocation are automatically marshalled on the sender side (serialized into a packed buffer) and unmarshalled by the recipient [16].

The specification by the programmer of parallel workflow in Charm++ is primarily done using a specialized charm interface minilanguage written in files with the extension `.ci`. The runtime system crosscompiles these ci files into C++ code, which a standard C++ compiler can then compile into the main executable. Beyond this basic usage, the ci file syntax allows the programmer to write event-driven code, in which execution flow proceeds based on the satisfaction of certain preconditions, expressed using the when construct [80].

Surprisingly, the baseline Charm++ runtime system has no explicit representation of the task graph that will execute. Task graphs have never played a substantial role in any

Charm++ application that the Charm++ developers are aware of. Rather, objects have been designed to represent elements of the problem decomposition, and individual tasks they need to perform over their lifetime get encoded into their compiled control flow, driven by messages as they arrive. Put another way, each processor sees the objects residing locally and the messages in its queue. Individual objects generate and respond to messages based on the code of their methods. Objects with simple life cycles can be purely reactive or use simple flags and counters to manage their behavior.

Objects with more complex life cycles can use an internal DAG notation Charm++ provides to generate structured control flow. The Charm++ team has recently released a TaskGraph library that is designed to make it easier to solve "directed information flow" problems. More experimental features do form explicit task dependency representations to manage data movement in various places. The declaration of data usage (in/out/inout, etc.) is used to inform strategies for GPUs, out-of-core execution, high bandwidth memories, etc.

3.3 Others - HPX, PaRSEC, STAPL, StarPU, DARMA, and OCR

HPX [82] is a parallel runtime system that extends the C++11/14 standard to facilitate distributed operations, enable fine-grained constraint-based parallelism, and support runtime adaptive resource management [82]. The HPX runtime design strategy provides a general asynchronous many-task runtime solution that is highly dependent on existing and emerging C++ standards. A principal focus seems to be at the intranode level, as an intranodal runtime system. HPX uses task scheduling and message passing to enable asynchrony and proper ordering of tasks and extends the notion of a partitioned global address space (PGAS), resulting in an active component referred to as an active global address space. In the same way Charm++ supports migratable objects, the HPX active global address space allows for transparent internodal migration of objects to achieve dynamic communication, synchronization, scheduling, task placement, and data migration [82]. HPX currently has no support for automatic data dependency analysis, halo scattering and gathering, etc.

PaRSEC [83] is a generic framework for architecture-aware scheduling and manage-

ment of fine-grained tasks on distributed many-core heterogeneous architectures and includes libraries, a runtime system, and various development tools to aid in porting. Applications using PaRSEC are expressed as a DAG of tasks with labeled edges designating data dependencies. PaRSEC's DAGs are represented in a compact format that can be queried to discover data dependencies. Similar to Uintah, and due to the dataflow representation, communications are implicit, handled automatically by the runtime. Specifically, in the PaRSEC model, data exchange is not explicitly coded by the developers into their application, as in MPI, but implied in the application's dataflow representation [83]. Like Uintah, this internodal communication automation allows the runtime to employ nonblocking communication to efficiently overlap communication and computation. PaRSEC is the only other AMT runtime for which the literature shows an implementation of radiation transport sweeps [84], discussed above in Section 2.4. How sweeps is supported within Uintah, which dramatically advances radiation modeling capabilities for the CCMSC target boiler problem, will be discussed in more detail in Section 7.5.

STAPL [85], or Standard Template Adaptive Parallel Library, resembles HPX in that it is a parallel programming framework that extends C++ and STL to provide unified support for shared and distributed memory parallelism. A primary aim of STAPL is that it plays a similar role in facilitating parallel program development as the ISO C++ standard library plays for sequential C++ programming [85]. STAPL provides varying levels of abstraction, appropriately ranging from application developer to a runtime system developer. Parallel programs can be composed by nonexpert parallel programmers using building blocks from the core STAPL library (much like Uintah). Users can be aware of the distributed nature of the machine, but such awareness is not required.

StarPU [86] is a runtime system providing a high-level, unified execution model tightly coupled with an expressive data management library. The main goal of StarPU is to provide a way to generate parallel tasks over heterogeneous hardware and also develop and tune scheduling algorithms. The StarPU task structure includes a high-level description of task data and how that data are accessed (i.e., R, W, R/W). It is also possible to express tasks dependencies in a way that allows programmers to express complex task graphs with very little effort. Since tasks are launched asynchronously, this approach to dependency expression allows StarPU to reorder tasks to improve performance. Because

some application developer interaction is required to aid StarPU in MPI transfers among nodes, halo transfers must be accomplished through user-defined tasks, making radiation or any global dependency problem virtually impossible to handle using StarPU.

The **Distributed Asynchronous Resilient Models and Applications (DARMA)** co-design programming model is not a runtime itself, but rather a translation layer between a front end that provides a straightforward API for application developers in asynchronous multitasking (AMT) runtime systems and a back end that is an existing AMT runtime system [87], [88]. Consequently, an application developer can write a DARMA code and run it using several different runtime system back ends, e.g., Charm++, Pthreads, HPX, and OCR with more back ends under development [88]. DARMA could then potentially provide a back end for Uintah.

The **Open Community Runtime (OCR)** [89] is an AMT runtime system for extreme scale computing, developed through collaboration among Rice University, Intel Corporation, UIUC, UCSD, Reservoir Labs, Eqware, ET International, University of Delaware, and several national laboratories. OCR is a community-driven project defined by a formal specification. Although OCR is a runtime system, its low-level API suggests a natural programming model, but most application programmers will prefer higher level programming models that run on top of OCR [90]. Two examples include HabaneroC++ [91] and Intel CnC [92] with a port of Legion currently underway.

3.4 AMT Summary

This chapter has provided a survey of current leading AMT runtimes and programming models and a cross section of those under initial or active development. The current AMT community clearly represents a broad range of different design points and philosophies within the design space of AMT models. The AMT runtimes and models covered here comprise the majority of those available today as well as those under initial development; however, not one offers support for *automated global halos*. Legion and Charm++ remain the only two models that offer the closest conceivable approach; however, they require the application developer explicitly supply far more characteristics of data dependencies. For the CCMSC target boiler problem, global dependencies within OCR would result in millions of hand coded tasks, an untenable solution.

CHAPTER 4

RADIATION AS A DESIGN DRIVER

This chapter details how Uintah's hybrid multithreaded MPI runtime system [32] was extended to support, schedule, and execute both GPU and CPU tasks simultaneously. This work was initially motivated by the Utah CCMSC need for an efficient, parallel radiation transport algorithms for use in the CCMSC target boiler simulations, *which are amenable to both domain decomposition strategies and GPU parallelization.* Creating an efficient, GPU-accelerated radiation model based on RMCRT methods (Section 2.3) is the focus of this chapter. *Due to introduction of the RMCRT radiation model, the principal vehicle for this research, the challenge of globally coupled dependency analysis, was discovered within Uintah.*

Additionally, this chapter introduces the benchmark radiation problem described by Burns and Christon [1], which is used and referenced throughout this dissertation and provides an analytical solution to compare numerical results against. Figure 4.1 depicts the Burns and Christon benchmark problem, an isothermal unit cube, centered on the origin and oriented so that the sides of cube are orthogonal to the principal Cartesian axes [1]. The walls of the cube are cold and black, and the interior of the cube consists of gray, nonscattering, absorbing/emitting material with a constant temperature and a tri-linearly varying absorption coefficient. Initial conditions consist of a uniform temperature field and varying absorption coefficient.

4.1 Original GPU Engine

Humphrey et al. mention in [34] that 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. With ever-imposed demands on system architects for increased density and power efficiency, traditional systems are now being augmented with an increasing number of graphics processing units (GPUs) [93]. This design is most notable in systems such as the Keeneland Initial Delivery

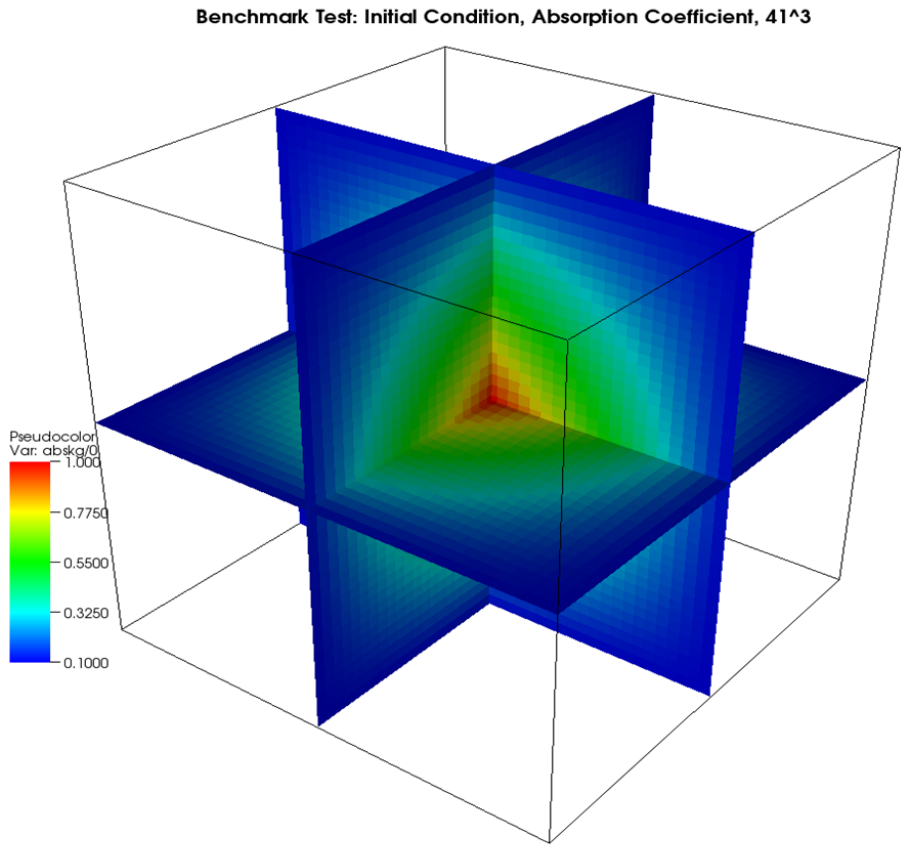

Figure 4.1. Burns and Christon benchmark radiation problem.

System (KIDS)[1] [94]. This architectural trend is also evidenced in the upgrade path of the DOE Jaguar[2] system to Titan [95].

Significant challenges face those trying to program for such architectures. The first of these challenges is the prospect of significantly less memory per core as the number of cores per socket continues to grow. In order to address this challenge, as recognized by a number of authors [96], [97], Uintah [30], an open-source software framework, has moved

[1]KIDS is an experimental HP-Nvidia GPU cluster located at the National Institute for Computational Sciences with 120 compute nodes, each with two Intel Xeon X5660 (Westmere 6-core @2.8GHz) processors, 24GB memory, InfiniBand QDR (single rail) interconnect and 3 Nvidia Tesla M2090 GPUs.

[2]Jaguar is a DOE supercomputer located at the Oak Ridge National Laboratory with 18,688 compute nodes each of which contains a single 16-core AMD Opteron 6200 Series (Interlagos cores @2.6GHz) processor on one of its two sockets, 32GB memory and Gemini interconnect, giving 299,008 processing cores. Currently on 960 nodes, the second socket contains a single Nvidia Tesla 20-series GPU. This 960-node partition is known as TitanDev.

from a model that uses only MPI to one that employs MPI to communicate between nodes and a shared-memory model using Pthreads to map the work onto available cores in a node [32]. At the time, the Uintah task-based model lent itself better to the use of Pthreads rather than OpenMP. This approach has led to the development of a multithreaded MPI runtime system, including a threaded task scheduler that has enabled Uintah to show excellent strong and weak scaling up to 196,000 cores on the DOE Jaguar XT5 system and good initial scaling to 262,144 cores on the upgraded DOE Jaguar XK6 system [44]. Using this approach has reduced Uintah's on-node memory usage by up to 80% [32].

A second challenge posed by such architectures is the design of runtime systems that maximize system utilization by fully exploiting all available processing resources on-node. Central to this goal is overcoming the inherent bandwidth bottleneck of PCI-express (PCIe) transfers to and from the GPU, as discrete GPUs are typically hosted in PCIe slots. Data copy across the PCIe bus, which has a maximum theoretical bandwidth of 8.0GB/s (PCI Express Gen2 x16 for the Nvidia Tesla C20 series cards). In practice, this rate is closer to 3.3GB/s when using paged memory, and 5.3GB/s using pinned (page-locked) memory. For memory bandwidth bound tasks, this bottleneck requires more advanced techniques to harness the computational power offered by GPUs. Many current approaches to this problem leave CPU cores idle during GPU-based computation, and others simply do not extend their focus beyond a single GPU. These approaches waste substantial available computational power. 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 [31].

This chapter examines the design of a CPU-GPU scheduler in the context of a developing scalable hierarchical ray-tracing radiation transport model to provide Uintah with additional capabilities for heat transfer and electromagnetic wave propagation. This work directly addresses the second major challenge introduced by heterogeneous systems, specifically utilizing all processing resources available on-node.

4.2 Developing a Uintah Radiation Model

In reacting flow simulations, the main computational cost is the solution of the large number of systems of linear equations required by the discrete ordinates method. Although the solution of these systems can be made to scale [44], it is important to reduce this cost. With this cost reduction in mind, more recent work has been based upon the use of more efficient reverse Monte Carlo ray tracing (RMCRT) methods, e.g., [63], [8], [48]. RMCRT lends itself to scalable parallelism because the intensities of each ray are mutually exclusive. Therefore, multiple rays can be traced simultaneously at any given cell and timestep [34].

Humphrey et al. demonstrated in [34] how to extend the Uintah framework so that problems involving radiation can *also* be directly supported within Uintah. Some kinds of radiation transport problems already use CFD codes and AMR techniques [7], [98]; however, other problems require the concept of tracing rays or particles, such as the simulation of light transport, heat, radiation, or electromagnetic waves.

The approach adopted in Uintah is the use of RMCRT methods, as described by [63]. This approach has the important advantage that by using the principle of reciprocity in radiative transfer, rays are traced backwards from the computational cell, thus eliminating the need to track ray bundles that never reach that cell [63]. In RMCRT, rather than following a ray forward and calculating the energy it has lost, the amount of incoming intensity from its path absorbed by the origin where the ray was emitted is calculated. Sun et al. [8] point out that RMCRT is more amenable to domain decomposition and thus a parallel implementation due to the backward nature of the process. Figure 2.1 shows the back path of a ray from S to the emitter E, on a nine-cell structured mesh patch. Each i^{th} cell has its own temperature T_i, absorption coefficient κ_i, scattering coefficient σ_i, and appropriate pathlengths $l_{i,j}$. In each case, the incoming intensity is calculated, say in cell 4, and then traced back through the other cells. The intensity is integrated along the ray path to compute a divergence of the heat flux or a surface flux. When a ray hits a boundary (as on surface 17 in the figure), it can be either reflected or absorbed depending on the surface properties. Rays are terminated when their intensity is sufficiently small.

Despite the improved efficiency over forward MCRT, there are considerable challenges in the efficient implementation of RMCRT as it is an all-to-all method, where all of the ge-

ometry information and property model information for the entire computational domain must be present on each processor [8]. This characteristic severely limits the size of the problem that can be computed due to memory constraints, especially with large highly resolved physical domains [34]. This challenge is being addressed by using the multilevel mechanisms within Uintah to represent a portion of the domain at a coarser resolution, thus lowering the memory usage [48]. The hybrid memory approach of Uintah [32] also helps as only one copy of geometry is needed per multicore node. In general, the data required by the RMCRT algorithm are projected to all of the coarser levels, with each level spanning the entire domain. For each fine-level patch, data from the coarser levels are retrieved from the Uintah `DataWarehouse`, encompassing the patch in a stair-step fashion.

CPU-only scalability studies of the RMCRT for the benchmark problem, as described by Burns and Christon [1], were run by Humphrey [34] on on a single-level grid [48] with 256^3 cells, using 25 and 100 rays per cell. Each scaling run was run for ten timesteps, one patch per processor, and the mean time per timestep was computed. These preliminary results, generated by Humphrey [34], show reasonable scaling up to 768 cores. Above this core count the loss of scalability is perhaps due to increased communication costs and/or a load imbalance. Nevertheless, these results provide a good proof of concept and an excellent starting point for this work.

4.3 Scheduler Architecture

As noted by Meng et al. [32], Uintah is a sophisticated computational framework that can integrate multiple simulation components, analyze the dependencies and communication patterns between them, and execute the resulting multiphysics simulation. These operations are accomplished 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 grid.

Associated with each task is a `C++` method that is used to perform the actual computation. In the context of the new hybrid CPU-GPU scheduler, a GPU task is represented by an additional `C++` method that is used for GPU kernel setup and invocation [34]. 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++ [78], [79], [80] (Section 3.2). In order to increase efficiency, the task graph is created and stored locally [30]. Uintah's CPU-GPU task scheduler is responsible for computing the dependencies of tasks, determining the order of execution, and ensuring that the correct interprocess communication is performed [30]. It also ensures that no input or output variable conflicts will exist in any two simultaneously running tasks. In the migration of the Uintah Computational Framework to hybrid CPU-GPU architectures, we elected to use Nvidia CUDA `C/C++` for numerous reasons. Looking at the upgrade path of the DOE Jaguar XK6 system to Titan [95] and also the Keeneland Initial Delivery System (KIDS) [94], we see a trend in the use or planned use of Nvidia GPUs [34]. These are the target machines on which we are already running both CPU and mixed CPU-GPU simulations. Initial runs using ported portions of the CFD component ICE have demonstrated the ability of our CPU-GPU scheduler to run capability jobs on both KIDS and TitanDev, utilizing all CPU cores and all GPUs simultaneously on each machine. KIDS currently has 1440 CPU cores and 360 Nvidia Tesla 20-series GPUs and TitanDev, 15360 CPU cores and 960 Nvidia Tesla 20-series GPUs.

The principal additions made by this new CPU-GPU scheduler are: significant leveraging of the Nvidia CUDA Asynchronous API [99] to best overlap PCIe transfers and MPI communication with GPU and CPU computation; insulating the component developer from the complexities and details involved with device memory management and asynchronous operations, by automatically managing these operations; and using knowledge of the task graph and task dependencies to pre-fetch data needed for simulation variables prior to task execution. Hence, when a GPU task is ready to run, data needed for the task are already resident in GPU main memory. The GPU task need merely query the scheduler for device pointers and invoke the kernel.

The existing Uintah code base is nearly 800K lines of code, a significant challenge to port in terms of infrastructure and existing simulation components. Although OpenCL [100] has the potential to support more than just GPUs and will be a consideration for use in the future, Nvidia CUDA currently offers far greater support in terms of performance and analysis tools as well as an API allowing for easier performance gains

and portability for existing codes. Below we describe the design of our CPU-GPU scheduler and its use of the Nvidia CUDA Asynchronous API [99] in detail.

4.3.1 Multithreaded Runtime System Design

The overall design of the multithreaded MPI runtime system is explained in great detail in [32], but to provide context, we review its design briefly here. We then describe in detail how this architecture has been extended by our recent work, adapting Uintah to run on current and emerging heterogeneous systems.

As mentioned in [32], the core scheduler component that stores simulation variables is the `DataWarehouse`. The `DataWarehouse` is a hashed-map-based dictionary that maps a variable name and patch ID to a memory address. 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 the `DataWarehouse`. Uintah's innovative load balancer utilizes space-filling curves in order to cluster patches together [101]. 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's infrastructure components were carefully designed to be stored in a distributed manner, it was 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 was that tasks that were created and executed on the same node could not share data. Uintah's multithreaded MPI scheduler [32] solves this problem by dynamically assigning tasks to worker threads during execution and shares the same infrastructure components between threads. This design uses 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 `DataWarehouse` and has read and write access to them.

As Humphrey et al. point out in [34], central to the design of the dynamic CPU-GPU scheduler (Figure 4.2) is the multistage queuing architecture for efficient scheduling of CPU and GPU tasks. The CPU-GPU scheduler utilizes four task queues: an internal ready queue and an external ready queue for CPU tasks and two queues for the GPU; one for initially ready GPU tasks; those that have requisite simulation variable data copies from

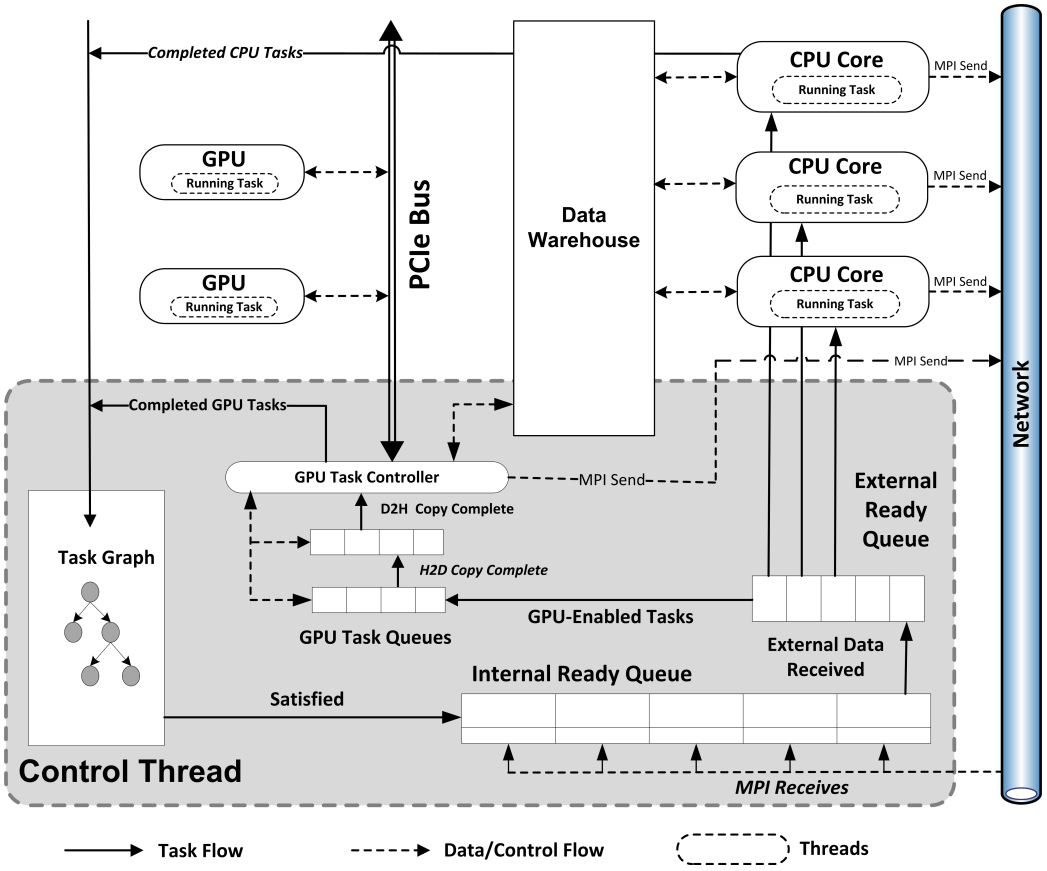

Figure 4.2. Original Uintah CPU-GPU task scheduler architecture.

host-to-device pending; and a second for the corresponding device-to-host data copies pending completion. First, if a task's internal dependencies are satisfied, then that task will be put in the CPU internal ready queue where it will wait until all required MPI communication has finished. In this same step, if the task is GPU-enabled, the task is then put into the host-to-device copy queue for advancement toward execution. Ultimately, the task goes to the pending device-to-host copies queue. As long as the CPU external queue is not empty, there are always tasks to run. Execution of a task takes place on the first available CPU core or GPU and the scheduler resides on a single, dedicated core per node [34]. CPU tasks are dispatched by the control thread to available CPU cores when they signal the need for work. GPU tasks are assigned in a round-robin fashion to available GPUs on-node once their asynchronous host-to-device data copies have completed. This design helps to overlap MPI communication and asynchronous GPU data transfers with CPU and GPU task execution, significantly reducing MPI wait times [34].

4.3.2 Asynchronous GPU Techniques

Significant difficulties arise when mixing concurrency APIs, most notably race conditions, deadlock, and general synchronization complexities. Within Uintah's CPU-GPU scheduler is a combination of MPI, Pthreads, and Nvidia CUDA, a combination that must be managed with care to avoid such difficulties. Multiple GPUs per node further complicate this situation in the presence of asynchronous memory copies and multiple device contexts (one CUDA calling context per device per process). In the same fashion that Uintah insulates the application developer from the parallelism its infrastructure provides, it also hides and carefully manages details related to GPU memory allocation and transfer. The Fermi-based GPUs found on the target machines mentioned at the beginning of this section offer additional ways to achieve asynchronous concurrent execution of kernels. These GPUs have two copy engines and support multiple kernels running concurrently. Using these features, GPU tasks can be copying data to and from the device as well as running multiple kernels simultaneously. In order to exploit these features, the CPU-GPU scheduler creates and manages queues of CUDA *Streams* [99], one for each device on-node. *Streams* provide a means to perform multiple operations simultaneously in that operations from different *streams* can be interleaved and also run concurrently. Our implementation also uses CUDA *Events* [99], which are used for timing and in checking completion of operations such as asynchronous memory copies to and from the GPU.

4.3.3 Extending the Uintah Task Class

Previously, the portion of the Uintah Task class responsible for actual execution of the `C++` method representing the computation to perform was comprised of a single instance of an `Action` class, which contains a single function pointer to the `C++` method to run. With the addition of GPU tasks, we have modified the Uintah Task class to include an additional `Action` instance with an associated pointer to the function containing the GPU kernel setup and invocation. This modification was accomplished without altering any existing interface or simulation component.

The design decision to support registration of multiple function pointers was to ultimately add the ability for the scheduler to chose between execution of the CPU or GPU version of the task at runtime. It may be the case that if all on-node GPUs are currently

busy or unavailable and there exists an idle CPU core, then it is best to execute a particular task on that CPU core. Currently, if a GPU task has a GPU implementation, it is executed on the GPU.

4.3.4 Prefetching GPU Task Data

When the CPU-GPU scheduler begins dispatching ready tasks from the CPU external ready queue, it diverts GPU-enabled tasks to the initially ready GPU task queue. Just prior to this step, the CPU-GPU scheduler initiates the device memory allocations and asynchronous host-to-device data copies for the requisite simulation variables. These asynchronous data copies are accomplished by querying the `DataWarehouse` for the location and size of the data required for computation and also requesting that the `DataWarehouse` allocate space for the result of the computation. We have exposed a flat representation of the underlying 3D data structure representing each simulation variable on a patch. This linear array maps relatively easily onto the GPU. To fully exploit the aforementioned levels of concurrency, the host memory to be copied to device must be page-locked. This page-locked host memory guarantees the memory will not be paged to disk. The CPU-GPU scheduler then registers for direct memory access (DMA) the host memory to be copied to the GPU using a call to +cudaHostRegister()+ with the `cudaHostRegisterPortable` flag. This call and flag pair creates page-locked (often referred to as pinned) memory from preallocated host memory that is considered page-locked by all CUDA contexts. This step avoids a bounce buffer, accelerates PCIe transfers, and also eliminates resetting of CUDA contexts when referencing the registered host memory. A call to `cudaHostRegister()` can be cleanly performed from the host without setting a context.

The new scheduler infrastructure maintains a set of queues for *stream* and *event* handles (one per device representing separate contexts for each), and assigns them to each simulation variable per timestep to overlap with other host-to-device memory copies as well as kernel execution. These *stream* and *event* handles are stored by the associated task itself and effectively provide a mechanism to detect completion of asynchronous memory copies without a busy wait, using `cudaEventQuery(event)`. This mechanism, developed by Humphrey in [34], allows querying the status of all device work preceding the most recent CUDA API call to `cudaEventRecord()` [99].

On systems with multiple on-node GPUs such as KIDS, the CPU-GPU scheduler must also manage a CUDA calling context for each device. The CUDA calling context is set per device prior to subsequent CUDA API calls on that device. In general, the CPU-GPU scheduler assigns a device to the task itself (round-robin), allocates space on the device, marks the task as initiated, and then starts the asynchronous host-to-device memory copies. The entire GPU task processing algorithm is shown in Algorithm 5, where it should be noted that CPU task processing, as shown in [32], is interleaved with the GPU task processing.

A call to `cudaEventRecord()` is then made after a call to `cudaMemcpyAsync()`, and these *event* pointers are stored with the task itself. The task is then placed into the initially ready GPU task queue. The priority of GPU tasks is based on the same prioritization algorithm used in the CPU external-ready queue, and thus the overall task priority is preserved. Task placement is accomplished asynchronously with respect to the CPU, which is continually responding to requests from idle CPU cores for work. This series

Algorithm 5 GPU task controller logic.

```
1:  while (completedTasks < totalTasks) do
2:      if (numExternalReadyTasks() > 0) then
3:          if (highest priority task isGPUEnabled()) then
4:              initiateH2DCopies (task, iteration)
5:              task.markInitiated ()
6:              addInitiallyReadyGPUTask (task)
7:          end if
8:      end if
9:      if (numInitiallyReadyGPUTasks() > 0) then
10:         if (task.checkH2DCopyDependencies ()) then
11:             runGPUTask (task, iteration)
12:             addCompletionPendingGPUTask (task)
13:         end if
14:     end if
15:     if (numCompletionPendingGPUTasks() > 0) then
16:         if (task.checkD2HCopyDependencies ()) then
17:             postMPISends (task, iteration)
18:             reclaimStreams (task)
19:             reclaimEvents (task)
20:             task.completed ()
21:         end if
22:     end if
23: end while
```

of steps essentially prepares the GPU memory needed by the task and is completed prior to task execution. All data related to each task's host and device pointers are kept in a set of maps maintained by the CPU-GPU scheduler. These maps ultimately become a separate GPU `DataWarehouse` [35].

4.3.5 GPU Tasks: Execution, Completion, and MPI Sends

During successive iterations of the CPU-GPU scheduler's task controller algorithm, the scheduler checks for existing tasks in the initially ready GPU task queue and determines if its host-to-device memory copies have completed. Determining GPU task availability is accomplished by performing `cudaEventQuery(event)` on each of a task's recorded *events*. The scan is essentially linear in the size of the list of *events* to query, but this size is never greater than 10 elements, and is constant time, e.g., $O(1)$. If all *event* queries return with `cudaSuccess`, the GPU task is ready to run. The C++ method associated with the kernel setup and invocation can then be executed. The component queries the scheduler for device pointers and a *stream* to associate with the kernel launch. The component then passes these pointers to the kernel routine that performs the computation on the device. To transfer the results of the computation back to the host, the component code requests a device-to-host copy via the infrastructure API. The scheduler in turn initiates the asynchronous memory copy from device to host destination and records the *events* associated with the task. Afterward, the task is placed in the completion-pending GPU task queue.

Within the CPU-GPU scheduler's task processing loop (Algorithm 5), the *events* in the *stream* associated with the device-to-host memory copy (and kernel used to compute results) of the highest priority GPU task can be queried for completion. Success returned on each of a task's *events* indicates the task has completed execution. The results are then guaranteed to be in the host-side `DataWarehouse`. At this point, the task can be marked as completed and the CPU-GPU scheduler then reclaims all of the *events* and *streams* used by the task. MPI sends from the GPU task can then be posted. The GPU task is finally removed from the completion pending task queue, allowing other dependent tasks to proceed.

4.4 Computational Experiments

In this section we examine the performance of Uintah's new hybrid CPU-GPU scheduler and runtime system by running the RMCRT benchmark problem described by Burns and Christon in [1], which has a constant initial temperature field and the absorption coefficient specified by a analytical function. This problem is run on a single level using both 41^3 and 128^3 cells. In both cases, the CPU-only version of the *RayTrace()* method consumes more than 90% of the total compute time. Significant speed-ups in this portion of the code yield significant speed-ups in overall time to solution.

We chose to use 41^3 initially so the computed divergence of the heat flux could be compared to the data published in [1]. For these runs, 25, 50, and 100 rays per cell were used. The testbed RMCRT component was run for ten timesteps with one patch per core for the CPU implementation and one patch per GPU for the GPU implementation, with the mean time per timestep computed and compared. We then describe the approach taken in the GPU implementation of the ray tracer, observing the raw speed-ups obtained, and compare a single Nvidia M2090 GPU against first a single core and then all cores on a node. These cores were Intel Xeon X5660 (Westmere) @2.8GHz and AMD Opteron 6200 Series (Interlagos) @2.6GHz for KIDS and TitanDev, respectively. We also examined the scaling behavior of the CPU and GPU implementations.

As mentioned in Section 2.3, RMCRT lends itself to scalable parallelism because the intensities of each ray are mutually exclusive. Therefore, multiple rays can be traced simultaneously at any given timestep in each cell in every Uintah patch. This exclusivity between ray intensities leads us to the approach taken with the GPU implementation, where 2D slices of the 3D patch are tiled with 2D `threadblocks`. These slices are in the two fastest moving dimensions (as the patch cells are traversed), X and Y. A single CUDA thread is assigned to each computational cell. Each thread (within a `threadblock`) is then responsible for tracing the set of rays associated with its respective cell for each slice. Each thread calculates the sum of the intensities from its set of rays, and the divergence of the heat flux for the cell, completely independent of other threads. Again, this exclusivity between rays avoids potentially costly atomic operations and synchronization. This approach also allows for a single kernel launch per timestep, avoiding the overhead associated with multiple kernel launches.

Table 4.1 shows the relative time to solution for both CPU and GPU implementations, and the speed-ups obtained on the single-level RMCRT testbed component using a grid size of 41^3. These timings were a direct comparison on a single node of KIDS and TitanDev for 25, 50, and 100 rays per cell. The first set of timings compares a single CPU core against a single Nvidia M2090 GPU on-node. The second set compares all CPU cores (12 on KIDS and 16 on TitanDev) with the same single GPU. These results show significant speed-ups on both machines.

As would be expected, the times to solution using the GPU implementation for each run are roughly equal for both machines; however, the CPU version of the ray tracer runs considerably faster on Keeneland than on TitanDev. An interesting additional result, not shown in Table 4.1, is that when using all three on-node GPUs on Keeneland and comparing against the CPU implementation, the speed-ups were not as significant. The slowdown can likely be attributed to the NUMA and contention effects within the multi-GPU HP SL390 nodes described in [93]. Currently, the CPU-GPU scheduler has no notion of GPU affinity. Addressing this issue to maximize utilization of the additional on-node computational resources in multi-GPU systems will be a focal point in future work.

Table 4.1. GPU speed-ups relative to CPU implementation on a single node of Keeneland and TitanDev.

Single CPU core vs. single GPU				
Machine	Rays	CPU (s)	GPU (s)	Speedup
Keeneland 1 Core Intel	25	28.32	1.16	24.41
	50	56.22	1.86	30.23
	100	112.73	3.16	35.67
TitanDev 1 core AMD	25	57.82	1.00	57.82
	50	116.71	1.66	70.31
	100	230.63	3.00	76.88

All CPU cores vs. single GPU				
Machine	Rays	CPU (s)	GPU (s)	Speedup
Keeneland 12 Cores Intel	25	4.89	1.16	4.22
	50	9.08	1.86	4.88
	100	18.56	3.16	5.87
TitanDev 16 Cores AMD	25	6.67	1.00	6.67
	50	13.98	1.66	8.42
	100	25.63	3.00	8.54

Using the CPU-GPU scheduler, we were able to run capability jobs on both machines, using all CPU cores and GPUs on-node, but we saw diminishing returns at larger scale. The all-to-all nature of this problem severely limits the size of the problem that can be computed, and hence single-level RMCRT does not yet scale well due to memory constraints with large highly resolved physical domains. Figure 4.3 shows strong scaling results (generated by Humphrey in [34]) for both CPU and GPU implementation on TitanDev. Similar CPU-only scalability studies of the same single-level RMCRT benchmark problem are described in [1]. Figure 4.3 illustrates that the GPU implementation quickly runs out of work, and strong scaling begins breaking down around eight GPUs. 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 both implementations suffer from the same exorbitant communication costs that are the central difficulty in this problem [34]. This scalability issue becomes a principal focus of this dissertation, and is initially addressed in Chapter 5.

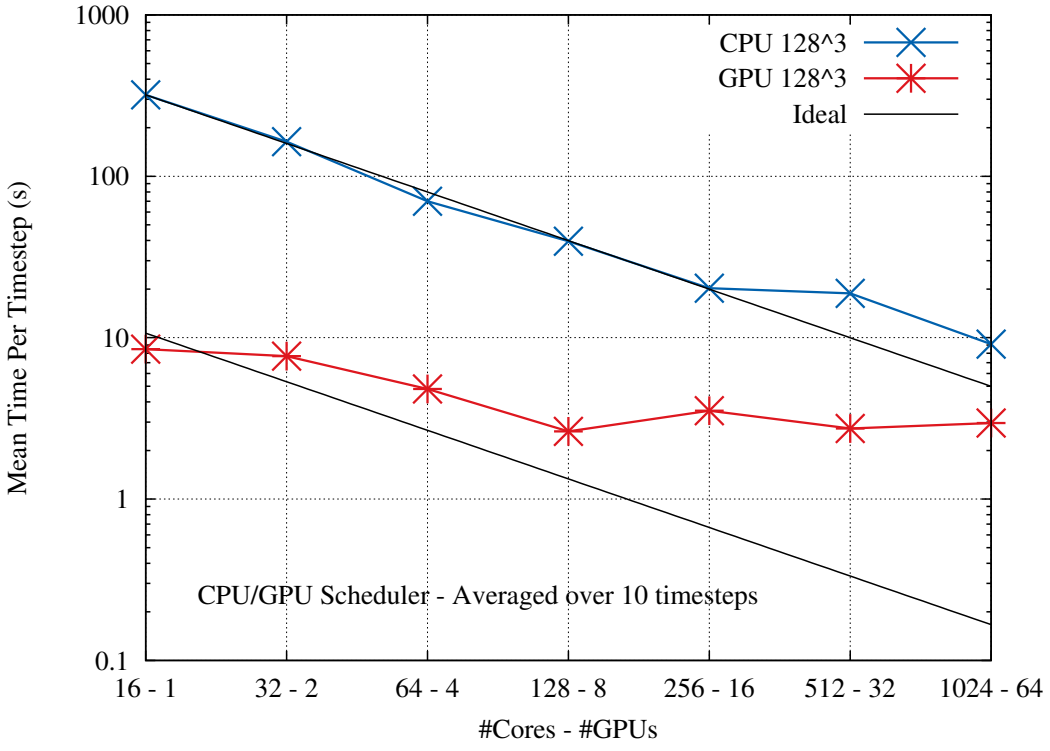

Figure 4.3. Single-level RMCRT strong scaling comparison on TitanDev.

4.5 Task Graph Scalability

Meng, Humphrey, and Berzins proposed in [50] that if the applications software is written in an abstract task graph manner so as to be executed by a runtime system that adaptively varies task graph execution, then it may be possible to combine a modest degree of portability across a number of large-scale parallel architectures with scalability at large core counts. High efficiency is achieved when the runtime environment is allowed to flexibly schedule the execution order of the various computational tasks [52]. This methodology was tested on three leading Top500 machines as of November, 2012 [102] – OLCF Titan[3], TACC Stampede[4], and ALCF Mira[5] – using three diverse and challenging applications problems. These machines make use of three very different processors and networks. Two of the machines, Titan and Stampede, have GPU accelerators and Intel Xeon Phi co-processors, respectively. Of particular interest was the application that used a combination of the Arches [103] turbulent combustion simulation component coupled with RMCRT for radiation.

The primary discovery in [50] related to this dissertation was Uintah's inability to complete RMCRT simulations beyond 16,000 CPU cores due to intractable task graph compilation times, e.g., nearly one hour at that scale on the DOE Titan and Mira systems. This road block would become the predominant scalability concern with this challenging

[3]Titan is a Cray KX7 system located at Oak Ridge National Laboratory, where each node hosts a 16-core AMD Opteron 6274 processor running at 2.2 GHz, 32 GB DDR3 memory and 1 NVIDIA Tesla K20x GPU with 6 GB GDDR5 ECC memory. The entire machine offers 299,008 CPU cores and 18,688 GPUs (1 per node) and over 710 TB of RAM. Titan uses a Cray Gemini 3D Torus network, 1.4 μs latency, 20 GB/s peak injection bandwidth, and 52 GB/s peak memory bandwidth per node.

[4]Stampeded is an XSEDE resource at Texas Advanced Computing Center, with Intel's new co-processor technology, the Xeon Phi. The host processors are eight-core PowerEdge C8220, Xeon E5-2680 operating at 2.7GHz with an Intel Xeon Phi co-processor operating at 1.0GHz. Each compute node has two eight-core sockets with 32 GBytes of memory. Stampede is outfitted with $6,400$ compute nodes and $102,400$ cores providing greater than 2 PFlops for the compute cluster and greater than 7 PFlops for the co-processors). The total system memory is 205 TB with over 14 PBytes of shared disk space using the Lustre file system. The system components are connected via a fat-tree FDR InfiniBand interconnect.

[5]Mira is a IBM Blue Gene/Q system located at Argonne National Laboratory that enables high-performance computing with low power consumption. The Mira system has 49,152 nodes, each having 16 1600 MHz PowerPC A2 cores per node, providing a total of 786,432 cores. Each node has 16 GB of RAM and the network topology is an integrated 5D torus with hardware assistance for collective and barrier functions and 2GB/sec bandwidth on all 10 links per node. The latency of the network varies between 80 nanoseconds and 3 microseconds at the farthest edges of the system. The interprocessor bandwidth per flop is close to 0.2, which is higher than many existing machines. There are two I/O nodes for every 128 compute nodes, with one 2 GB/s bandwidth link per I/O node. Mira uses the GPFS file system. Ranks are assigned with locality guarantees on the machine.

global dependency problem. Figure 4.4 shows strong scaling results for this problem and illustrates the 16,000 core simulation barrier, especially on Mira.

Although this problem surfaced again for Uintah's AMR regridding phase in [104], the underlying issue with task graph compilation in the context of global dependency problems was not revisited until RMCRT was in the critical path for the CCMSC mission, through the work done in [55] (Chapter 7) for a 351 million CPU hour INCITE award, 71 millions hours of which were on the DOE Titan system. Figure 4.5 shows weak scaling based on how individual components in the overall AMR regridding phase were improved. The task graph portion of the overall regridding phase Figure 4.5 within Uintah is a central focus of this dissertation.

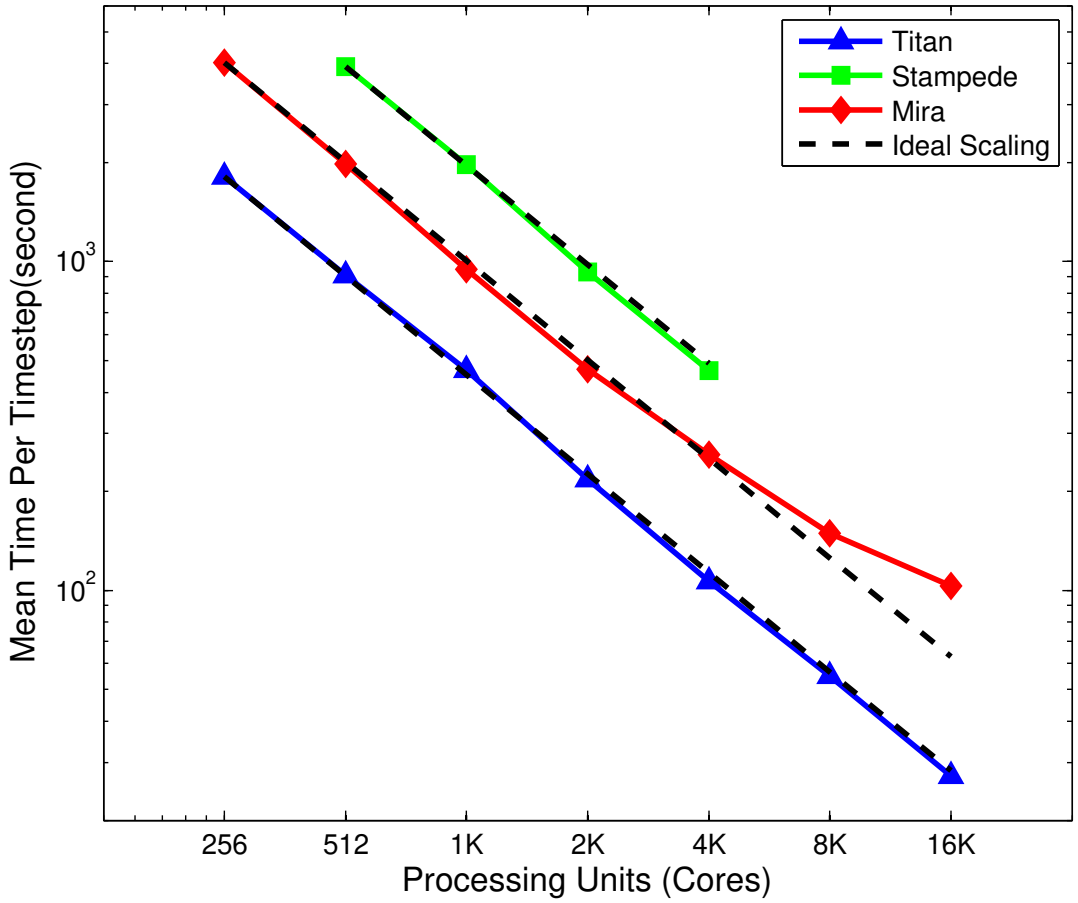

Figure 4.4. RMCRT strong scaling barrier at 16K cores due to task graph compilation.

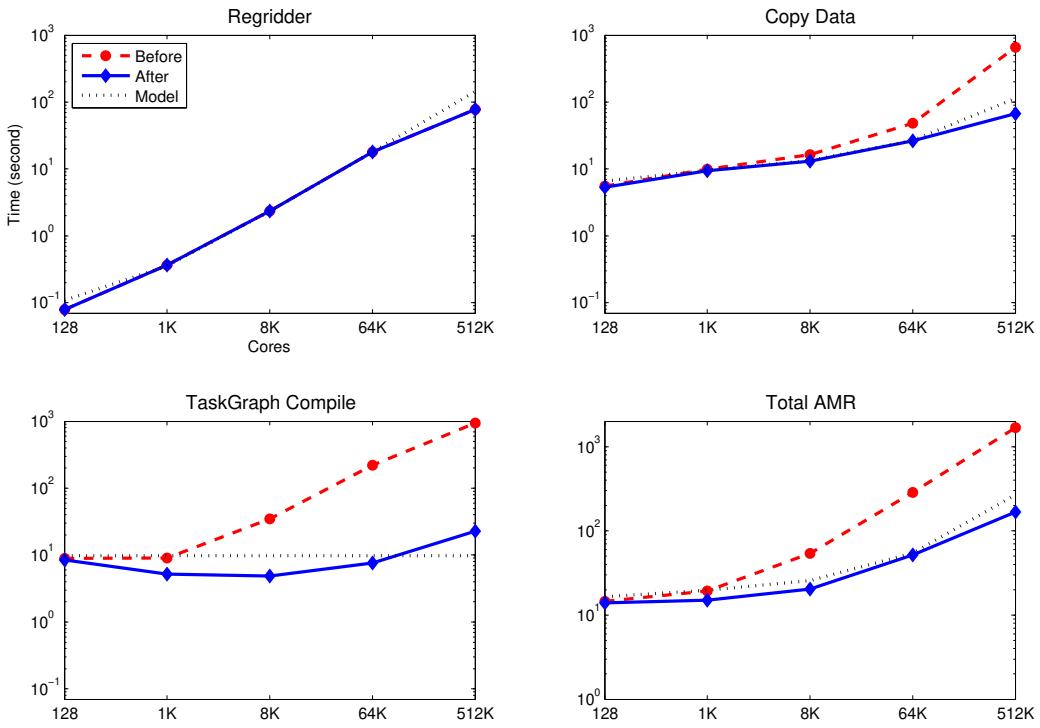

Figure 4.5. AMR improvement breakdown: weak scaling.

4.5.1 Summary and Conclusion

This chapter has introduced the central role radiation modeling plays in the CCMSC target boiler simulation, and has shown that the CPU-GPU scheduler design is capable of running Uintah simulations, specifically the RMCRT radiation model, on current and emerging heterogeneous systems, fully utilizing all on-node computational resources simultaneously. However, we face significant scalability challenges inherent in the RMCRT problem, as shown in our results. Developing a scalable approach to this problem is addressed in Chapter 5. Other aspects of the CPU-GPU scheduler will be improved upon as well. Most notably, the centralized control thread design will need to be revised by moving to a decentralized design [33]. The central control thread design will become a severe performance bottleneck as CPU core counts on-node continue to grow. This approach has already been taken in our multithreaded CPU task scheduler [33], and is planned for the CPU-GPU scheduler. This design will allow any thread to fetch and execute both CPU and GPU tasks and also to send and receive its own MPI messages. Implementing an efficient, lock-free GPU `DataWarehouse` is another consideration as is

implementing a mechanism for the CPU-GPU scheduler to decide at runtime whether to run a particular task on a CPU core or on a GPU.

Although the investigation of scalability with respect to the different processors and communications performance concluded that the adaptive DAG-based approach provides a very powerful abstraction for solving challenging multiscale, multiphysics engineering problems on some of the largest and most powerful computers available today [50], the key discovery related to the global dependency problem within Uintah, related to this dissertation, was Uintah's inability to complete RMCRT simulations beyond 16,384 CPU cores due to intractable task graph compilation times. Addressing this critical issue is covered in Chapter 5.

CHAPTER 5

SCALABLE RADIATION MODELING TO 262,144 CPU CORES

Radiative heat transfer is an important mechanism in a class of challenging engineering and research problems. A direct all-to-all treatment of these problems is prohibitively expensive on large core counts due to pervasive all-to-all MPI communication. The massive heat transfer problem arising from the next generation of clean coal boilers being modeled by the Uintah framework has radiation as a dominant heat transfer mode. Reverse Monte Carlo ray tracing (RMCRT) can be used to solve for the radiative-flux divergence while accounting for the effects of participating media. The ray tracing approach used here replicates the geometry of the boiler on a multicore node and then uses an all-to-all communication phase to distribute the results globally. The cost of this all-to-all is reduced by using an adaptive mesh approach in which a fine mesh is used only locally, and a coarse mesh is used elsewhere. A model for communication and computation complexity is used to predict performance of this new method. This chapter shows this model to be consistent with observed results and demonstrate excellent strong scaling to 262,144 cores on the DOE Titan system on problem sizes that were previously computationally intractable [53].

5.1 Uintah RMCRT Approaches

The Uintah RMCRT module includes numerous approaches, each designed for a specific use case, ranging from a single-level method to a full adaptive mesh refinement using an arbitrary number of grid levels with varying refinement ratios. This chapter focuses on the multilevel mesh refinement approach and its scalability to large core counts. CPU scaling results for this approach are shown in Section 5.3.

5.1.1 Single-Level

The single-level RMCRT approach was initially implemented as a proof of concept to begin comparisons against the legacy DOM solver within the Uintah Arches component. This approach focused on the benchmark problem described by Burns and Christon in [1]. In this approach, the quantity of interest, the divergence of the heat flux, ∇q is calculated for every cell in the computational domain. The entire domain is replicated on every node (with all-to-all communication) for the following quantities: κ, the absorption coefficient, a property of the medium the ray is traveling through; σT^4, a physical constant $\sigma \cdot$ temperature field, T^4 and; *cellType*, a property of each computational cell in the domain to determine if along a given path, a ray will reflect or stop on a given computational cell. These three properties are represented by 1 double, 1 double, and 1 integer value, respectively.

For N_{total} mesh cells, the amount of data communicated is $\mathcal{O}(N^2)$. Although accurate and effective at lower core counts, the volume of communication in this case overwhelms the system for large problems in our experience. Calculations on domains up to 512^3 cells are possible on machines with at least 2GB RAM per core and only when using Uintah's multithreaded runtime system, described in Section 1.3. Strong scaling breakdown for the single-level approach occurs around 8-10K CPU cores for a 384^3 domain. Currently, Uintah has a production-grade GPU implementation of this single-level approach that delivers a 4-6X speed-up [1] in mean time per timestep for this benchmark with a domain size of 128^3 cells. The work done to achieve accelerator task scheduling and execution is detailed in [34]. Initial scalability and accuracy studies of the single-level RMCRT algorithm are also shown in [48], which examines the accuracy of the computed divergence of the heat flux as compared to published data and reveals expected Monte-Carlo convergence.

5.1.2 Multilevel Adaptive Mesh Refinement

In this adaptive meshing approach, a fine mesh is used locally, and only coarser representations of the entire domain are replicated on every node (with all-to-all communication) for the radiative properties, T, κ *cellType*. The fine level consists of a collection of patches where each patch is considered a region of interest and individually processed

[1] 1-NVIDIA K20 GPU vs. 16-Intel Xeon E5-2660 CPU cores @2.20GHz

using a local fine mesh and underlying global coarse-mesh data. Figure 5.1 and Figure 5.2 both illustrate a three-level mesh coarsening scheme and how a ray might traverse this multilevel domain. Surrounding a patch is a halo region that effectively increases the size (at the finest resolution) of each patch in each direction, x, y, and z. This distance is user specified. An arbitrary number of successively coarser levels (received by each node during the all-to-all communication phase) reside beneath the fine level for the rays to travel across once they have left the fine level. Each ray first traverses a fine-level patch until it moves beyond the boundary and surrounding halo of this fine-level patch. At this point, the ray moves to a coarser level. Once outside this coarse level, the ray moves again to a coarser level. The rays move from level to level, similar to stair stepping, until the coarsest level is reached. Once on the coarsest level, a ray cannot move to a finer level. The key goal of this approach is to achieve a reduction in both communication and computation costs, as well as nodal memory footprint. This approach is fundamental to the CCMSC target problem, the 1200 MWe boiler predictive case where the entire computational domain needs to be resolved to adequately model the radiative heat flux.

Initial scalability results on a two-level methane jet problem using the Arches component are shown in [50]. This problem was run on the the DOE Titan, Mira, and NSF Stampede systems with ten rays per cell, two grid levels, a refinement ratio of four, and a problem size of 256^3 cells on the highest resolved mesh. These results provided an excellent starting point by showing scaling to 16K CPU cores. Scaling results beyond 16,000 cores at the time was not possible due to the intractable task graph compilation times noted in Section 4.5, which were largely resolved in Section 5.3.1, with scaling results shown in Section 5.3.

5.2 RMCRT Complexity Model

In this section, we generalize the initial analysis given in [50] of the two-level scheme of [48] to a detailed discussion of both the computational and communications costs of a multiple-mesh-level approach. Initially, our approach involves replicating the geometry of the target problem and constructing an adaptive mesh for the radiation calculations. The adaptive mesh used by the radiation calculation may be constructed directly from the efficient mesh data structure used to describe the whole mesh. This is a one-time procedure

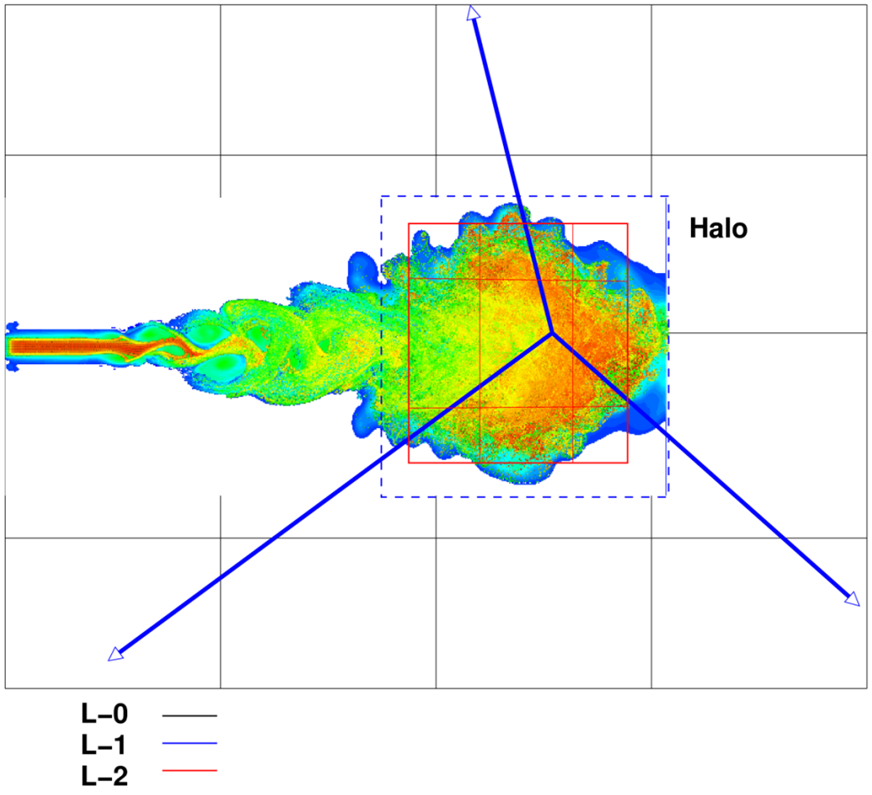

Figure 5.1. RMCRT - 2D diagram of three-level mesh refinement scheme. This scheme uses a coarser representation of computational domain with multiple mesh levels. L-2 corresponds to the highly resolved, CFD mesh, and L-1 and L-0 correspond to successively coarser meshes used for RMCRT ray marching.

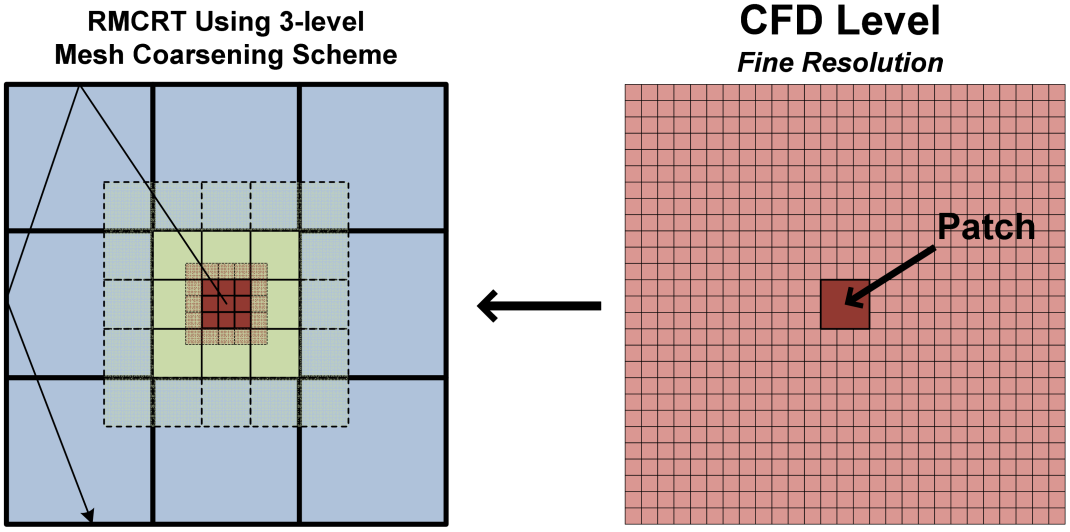

Figure 5.2. RMCRT - 2D diagram of three-level mesh refinement scheme, illustrating how a ray from a fine-level patch (right) might be traced across a coarsened domain (left).

and so is not analyzed further here.

We suppose that on N^3_{nodes} compute nodes there is a global fine mesh of n^3_{mesh} cells in n^3_{patch} mesh patches. Define

$$n_{local} = n_{mesh}/N_{nodes}, \tag{5.1}$$

and define

$$n_{plocal} = n_{patch}/N_{nodes}, \tag{5.2}$$

so that each node has n^3_{local} fine mesh cells in n^3_{plocal} patches. In the ray tracing algorithm, each compute node then has to compute the heat fluxes, ∇q on its local mesh by ray tracing **and** to export the temperatures T and and the absorption coefficient κ on the original mesh to neighboring "halo" nodes, or in a coarsened form (possibly at multiple levels) to other nodes. Finally, the coarsest mesh representations are distributed to all the other nodes. The amount of information per cell transmitted is two doubles $\kappa, \sigma T^4$, and one integer, *cell_type*.

To assess the complexity of RMCRT on a fixed fine mesh, computational experiments to measure the per cell and per ray cost of the RMCRT:CPU implementation were conducted on a single CPU with a single-level grid. Ray scattering and reflections were not included in these experiments. In both experiments, the absorption coefficient was initialized according to the benchmark of Burns and Christon [1] with a uniform temperature field. A grid with a single patch and 1 MPI process was used, thus eliminating any communication costs. The grid resolution varied from $16^3, 32^3, 64^3$ to 128^3 cells with each cell emitting 25 rays. The mean time per timestep (MTPTS) was computed using seven timesteps. The code was instrumented to sum the number of cells traversed during the computation, and it was shown that

$$MTPTS = (n^3_{mesh})^{1.4}. \tag{5.3}$$

In the second experiment, the number of grid cells in the domain was fixed at 41^3 and the number of rays, n_{ray} per cell varied. The MTPTS was computed over 47 timesteps. It was shown here that the MTPTS varies linearly with the number of rays per cell. Based on these experiments, the cost for a single patch, without any communication, is approximately given by

$$T_{rmcrt}^{global} = C^* n_{rays} {n_{mesh}^3}^{4/3}, \tag{5.4}$$

where C^* is a constant. This result may be interpreted as saying that the rays from each of the n_{mesh}^3 cells travel a distance of n_{mesh} cells on average. In the case of a fine mesh on a node and a coarse representation of the rest of the mesh,

$$T_{rmcrt}^{local} = C^* n_{rays} \left[{n_{local}^3}^{4/3} + (n_{mesh} 2^{-m})^3)^{4/3} \right], \tag{5.5}$$

where 2^m is the refinement ratio used to obtain the coarse mesh. It is possible to extend this analysis to more mesh refinement levels.

5.2.1 Communications Costs

The main step with regard to communication is to update the temperatures T and the absorption coefficients κ every timestep. On a uniform fine mesh, this temperature update is done by each node sending out the values of these quantities to all the other compute nodes, and in a multiple-mesh-level approach the update is accomplished by each node sending out the values of these quantities on the coarse- and fine-mesh values locally.

5.2.2 Fine Mesh Global Communications

Each node has to transmit $(N_{nodes}^3 - 1)$ messages of size $(n_{local})^3 \cdot 3$. This transmission is currently accomplished by a series of asynchronous sends but could be done with an `MPI_Allgather`. This transmission has a complexity of

$$\alpha 3 log(N_{nodes}) + \beta \frac{N_{nodes}^3 - 1}{N_{nodes}^3} (n_{local})^3, \tag{5.6}$$

for N_{nodes}^3 nodes with n_{local}^3 elements per mesh patch, where α is the latency and β is the transmission cost per element [105]. This result applies for both the recursive doubling and Bruck algorithms [105]. Other recursive doubling algorithms result in a complexity of

$$\alpha 3 log(N_{nodes}) + \beta (N_{nodes}^3 - 1)(n_{local})^3, \tag{5.7}$$

so the cost may be dependent on the MPI implementation used.

5.2.3 Coarse Mesh All-to-All

In the case of using a coarse mesh in which the mesh is refined by a factor of 2^m in each dimension, each node has to transmit $(N_{nodes}^3 - 1)$ messages of size $(n_{local} 2^{-m})^3 * 3$, which

reduces the communications volume, but not the number of messages, by a factor of 2^{3m} overall.

5.2.4 Multilevel Adaptive Mesh Refinement

This approach considers each fine-level patch (individually) in the domain as a region of interest (ROI), and for each fine-level patch, the highest resolved CFD mesh is used. Figure 5.2 illustrates one patch being such a region of interest. In the case of a region of interest consisting of P_{int} patches, the compute node must transmit the fine-mesh information to all the local nodes close to the ROI. In this context, let L_i be the nodes that are i levels of nodes removed from the node containing the region of interest. There will then be 26 level-1 nodes and 98 level-2 nodes. Of course, at the edges of a spatial simulation domain or in the case of a small domain of interest, each node will have to communicate fine-mesh values of $\kappa, \sigma T^4$ to only a fraction of the nodes. In this case, let $L^{i,j}_{active}$ be the number of active nodes (halo-level nodes) at level j, where $j < N_{levels}$, active for the ith level of interest, where active nodes are the local halos from the fine mesh. Furthermore, let the refinement factor be $\frac{1}{2^{m(i,j)}}$ active at this level. Then, the fine-mesh communication associated with this region of interest is given by

$$Com_{fhalo} = \sum_{j=1}^{N_{levels}} L^{i,j}_{active}(\alpha + \beta(n_{local}2^{-m(i,j)})^3 * 3). \tag{5.8}$$

This refinement factor means that the ratio of communications to computations R_{atio} is now given as

$$R_{atio} = \frac{((N^3_{nodes} - 1))(\alpha + \beta(n_{local}2^{-m})^3 * 3) + Com_{fhalo}}{T^{local}_{rmcrt}}, \tag{5.9}$$

where α and β are defined above and scaled by the cost of a FLOP. Overall, this expression allows us to analyze the relationship between computation and communications.

Strong scaling of RMCRT does not change the overall volume of data communicated. Increasing the number of N_{nodes} by a factor of two simply reduces n_{local} by two. This relationship does mean that the number of messages increases even with the total communications value being constant. Moving to `MPI_Allgather` also has the same problem, but the factor of $3logN_{nodes}$ also increased by adding 3. Thus, for enough rays n_{rays} with enough refinement by a factor of 2^m on the coarse radiation mesh, the computation will likely

dominate. A key challenge is that storage of $O(n_{mesh}2^{-m})^3)$ is required on a multicore node and that an AMR mesh representation is needed at very large core counts. Some aspects of this analysis are not dissimilar to earlier work by Berzins et al. [106] on PDE solvers with global coarse-mesh operations using algorithms related to those of [107]. The results of this analysis will make it possible to prioritize subsequent serial and parallel performance tuning and and also perhaps to make projections regarding performance on forthcoming petascale and exascale architectures.

5.3 CPU Scaling Results

In this section, we show strong scalability results on the DOE Titan XK7 [2] system for the Burns and Christon [1] benchmark problem using the multilevel mesh refinement approach. We define strong scaling as a decrease in execution time when a fixed size problem is solved on more cores. This work focuses on using all CPU cores available on Titan. Work described in Chapter 6 focuses on also using all of Titan's GPUs in addition to its CPUs, following our prototype work on a single mesh in [34], [50]. The scaling challenges faced in this work have become apparent only by running this challenging problem at such high core counts, stressing areas of infrastructure code in ways never before seen, specifically Uintah's task graph compilation phase.

5.3.1 Task Graph Compilation Algorithm Improvements

With Uintah's directed acyclic graph (DAG)-based design [50], during an initial simulation timestep, the initial timestep of a restart, or when the grid layout or its partition changes, a new task graph needs to be created and compiled. Task graph compilation is a complex operation with multiple phases, including creation and scheduling of tasks themselves on local and neighboring patches (for halo exchange), keeping a history of what these tasks require and compute, setting up connections between tasks (edges in the DAG), and finally assigning MPI message tags to dependencies.

As the RMCRT ray trace task requests ghost cells across the entire domain (a global

[2]Titan is a Cray KX7 system located at Oak Ridge National Laboratory, where each node hosts a 16-core AMD Opteron 6274 processor running at 2.2 GHz, 32 GB DDR3 memory and 1 NVIDIA Tesla K20x GPU with 6 GB GDDR5 ECC memory. The entire machine offers 299,008 CPU cores and 18,688 GPUs (1 per node) and over 710 TB of RAM. Titan uses a Cray Gemini 3D Torus network, 1.4 μs latency, 20 GB/s peak injection bandwidth, and 52 GB/s peak memory bandwidth per node.

halo) for ray marching, Uintah's task graph compilation algorithm was overcompensating when constructing lists of neighboring patches for local halo exchange *on the highly resolved, fine-mesh level*. The cost of this operation grew despite the number of patches per node remaining constant, resulting in task graph compilation times of over 4 hours at 32,000 cores with 32,000 total patches. These untenable compilation times necessitated extensive algorithmic improvements to the task graph compilation algorithm. The original complexity of this operation was

$$\mathcal{O}(n_1 \cdot log(n_1) + n_2 \cdot log(n_2)), \tag{5.10}$$

which was due to Uintah's task graph compilation algorithm overcompensating when constructing lists of neighboring patches (Section 7.2.2) for consideration in local halo exchange on the highly resolved fine mesh.

Uintah's load balancer was considering *all patches on the fine level as potential neighbors*. These algorithmic improvements entailed modifying Uintah's load balancer to short circuit unnecessary searches for patches across the fine mesh. The details of these modifications are provided in [53], with further elaboration found in [55]. The complexity in 5.10 after algorithmic improvements became

$$\mathcal{O}(n_1 \cdot log(n_1)) + \mathcal{O}\Big(\frac{n_2}{p} \cdot log(n_2)\Big), \tag{5.11}$$

where n_1 is the number of patches on the coarse level, n_2 is the number of patches on the fine level, and p is the number of processor cores. These improvements reduced the 4-hour task graph compilation time to under 1 minute at 32K cores, ***thus making possible the results presented here***.

5.3.2 CPU Strong Scaling of Multilevel Adaptive Mesh Refinement RMCRT

This scaling study focuses on a two-level AMR problem based on benchmark described in [1], which exercises all of the main features of the AMR support within Uintah in addition to the radiation physics required by our target problem. A fine-level halo region of four cells in each direction, x, y, z, was used. The AMR grid consisted of two levels with a refinement ratio of four, the CFD mesh being four times more resolved than the radiation mesh. For three separate cases, the total number of cells on the highest resolved

level was 128^3, 256^3, and 512^3 (green, red, and blue lines, respectively in Figure 5.3), with 100 rays per cell in each case. The total number of cells on the coarse level was 32^3, 64^3, and 128^3. In all cases, each compute core was assigned at least one fine-mesh patch from the CFD level. Figure 5.3 shows excellent strong scaling characteristics for our prototype, two-level benchmark problem [1]. The eventual breakdown in scaling in each problem size is due to diminishing work, when a patch's MPI messages begin to exceed the cost of its computation, and hence the runtime system cannot overlap computation with communication. Figure 5.4 additionally shows the MPI wait associated with the global and local communications for this calculation alongside the execution times for the three cases above. Although the actual communication patterns for this problem are perhaps more complicated than our predictive model, due to MPI message combining and packing done by Uintah, both Table 5.1 and Figure 5.4 illustrate points made in Section 5.2, that global communications dominate and that the local communications do not have a significant

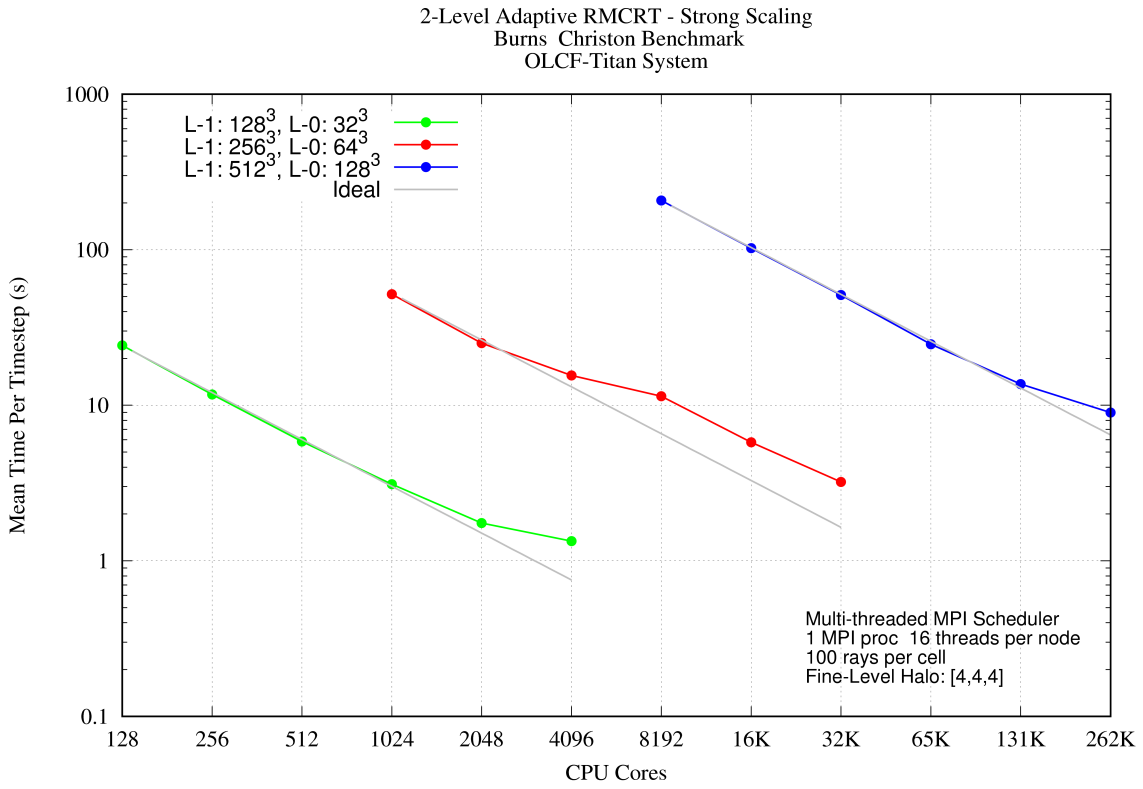

Figure 5.3. Strong scaling of the two-level benchmark RMCRT problem on the DOE Titan system. L-1 (Level-1) is the fine CFD mesh and L-0 (Level-0) is the coarse radiation mesh.

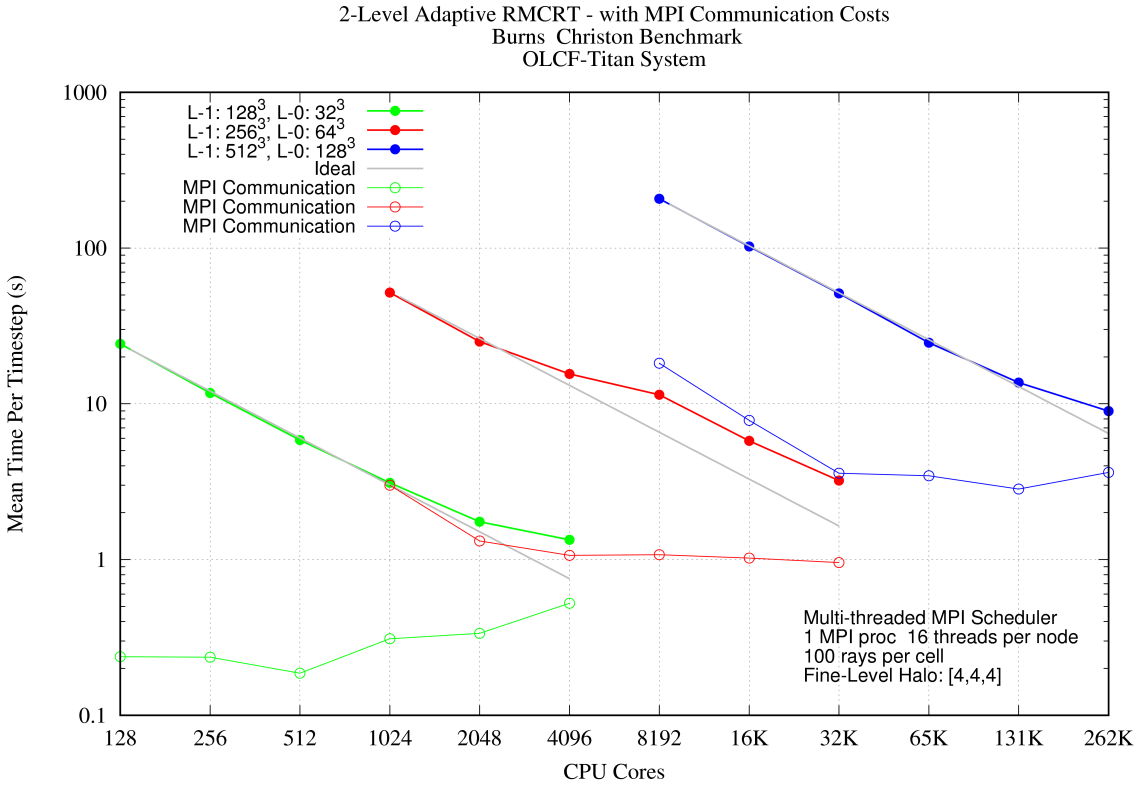

Figure 5.4. Strong scaling with communication costs of the two-level benchmark. L-1 (Level-1) is the fine CFD mesh and L-0 (Level-0) is the coarse radiation mesh.

Table 5.1. Total number of MPI messages and average number of messages per MPI rank (a single rank per node) for each problem size, 128^3, 256^3, and 512^3 (fine mesh).

	Cores								
	256	1k	4k	8K	16K	32K	64K	128K	256k
128^3									
total	1001	5860	36304						
avg	62.5	91.6	141.8						
256^3									
total		9843	52.1K	105.2K	212.1K	437.7K			
avg		153.8	203.3	205.6	207.0	213.7			
512^3									
total				338.2K	673.8K	1.36M	2.71M	5.42M	10.88M
avg				660.5	658.0	663.65	662.6	661.36	662.83

impact, and for enough rays and enough refinement on the coarse radiation mesh, the computation does in fact dominate (5.9 of our predictive model). These results also show how the number of MPI messages grows with the number of cores. A key point to note, as is evidenced by the dominating global communications, is that the refinement ratio of four reduces the global communication phase by a factor of 64 (ignoring communications latency for large messages) over a fine mesh all-to-all. ***If this communications phase took 8-64 times as long, it would destroy scalability.***

5.3.3 Weak Scaling Results

Figure 5.5 shows a log-log plot representing the weak and strong scalability for three problems sizes using the Burns and Christon radiation benchmark problem [1]. Table 5.2 shows the exact timings for the same benchmark radiation problem.

The refinement ratio is the ratio between the number of grid cells in the highly resolved CDF mesh relative to the underlying coarse mesh used in the radiation calculation. For example, a refinement ration of two simply means the coarse radiation mesh is coarsened by a factor of 2 in each direction, x,y,z, for a total reduction in cells by a factor of 8 (compared to the highly resolved CFD mesh). The timings shown in Table 5.2 show excellent weak and strong scaling up to 262,144 CPU cores (16,384 compute nodes). To achieve the results shown, each problem used a radiation mesh coarsened by a factor of two from the previous problem size, e.g., the small problem (128^3 total cells) had a refinement ratio of two, the medium-sized problem (256^3 total cells) used a refinement ratio of four, and the largest problem (512^3 total cells) used a refinement ratio of eight. This result shows that through novel use of AMR, the Utah CCMSC radiation calculations using RMCRT can be made to both strong and weak scale, even to large core counts.

Another key point regarding the $\mathcal{O}(p^2)$ communications growth with RMCRT, where p is the number of communicating processes, is that without leveraging AMR, when a fixed mesh grows by a factor of 8 (ignoring communications latency for large messages), both communications and local computation grow by this same factor. Coarsening by factor of 8 then leaves the communication and local computation workload the same as before the factor of 8 growth, allowing the weak scaling shown here. More generally, with $'M'$ coarse levels on a node, adding $'N'$ more levels for weak scaling at most only multiplies

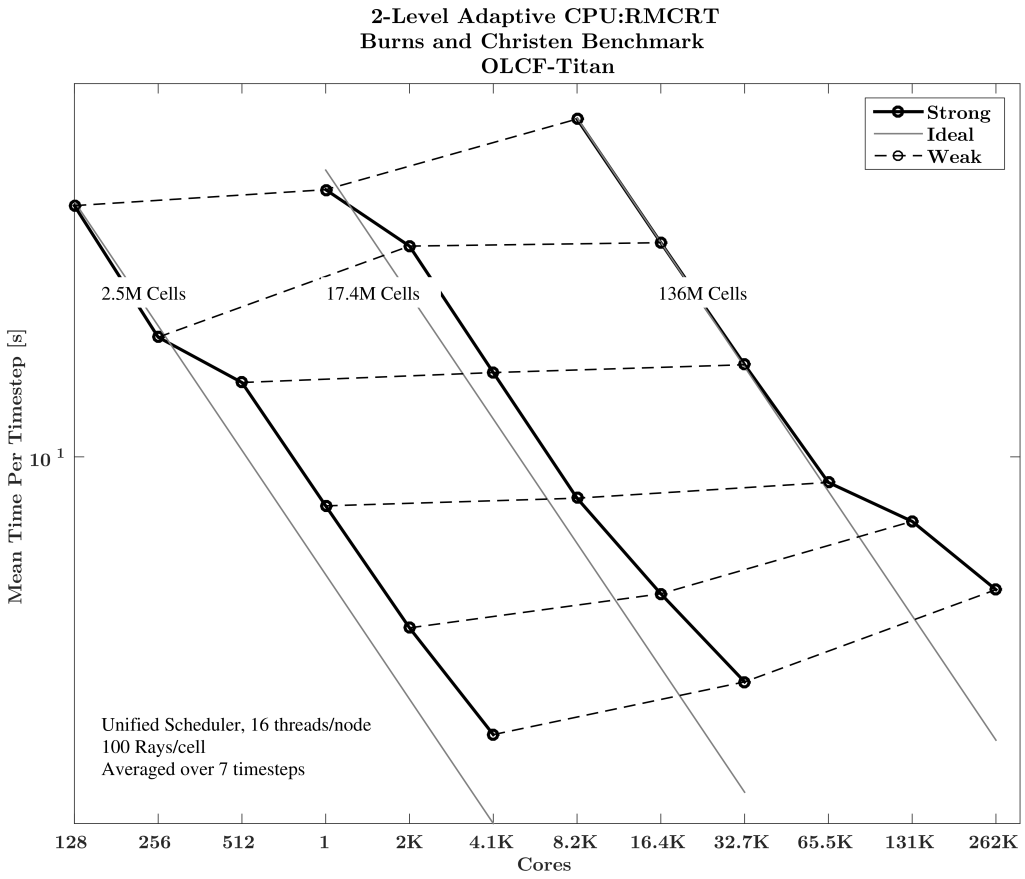

Figure 5.5. RMCRT weak and strong scalability for the Burns and Christon benchmark.

the computational and communications work by a factor of $(N + M)/M$.

Table 5.2. Scaling results from 128 to 256K CPU cores (8 to 16K nodes, respectively), for three separate problem sizes, with 128^3, 256^3, and 512^3 total cells in the computational domain, respectively. Times are the mean time per timestep (seconds) for each run.

Small 128^3 cells Refinement Ratio: 2		Medium 256^3 cells Refinement Ratio: 4		Large 512^3 cells Refinement Ratio: 8	
#cores	time (sec)	#cores	time (sec)	#cores	time (sec)
128	40.53	**1k**	44.31	**8k**	65.68
256	19.48	**2k**	32.36	**16k**	32.98
512	15.13	**4k**	15.99	**32k**	16.71
1k	7.61	**8k**	7.94	**64k**	8.67
2k	3.86	**16k**	4.66	**128k**	6.98
4k	2.13	**32k**	2.85	**256k**	4.77

5.3.4 Multilevel Accuracy Considerations

To quantify the error associated with coarsening the radiative properties (temperature T, absorption coefficient κ, and cell type (boundary or flow cell)), an error analysis was performed using a simplified version of the adaptive-meshing approach described in Section 5.2.4. The grid consisted of a fine and coarse mesh, and during a radiation timestep the quantities necessary to compute ∇q were interpolated to the coarser grid level. The radiation calculation was performed on the coarse level including all ray tracing. The ∇q was then compared using the computed solution of the Burns and Christon [1] benchmark problem at the prescribed 41 locations. One hundred rays per cell were used in the computation, and the refinement ratio between the coarse and fine grids was varied from 1 to 8. Figure 5.6 shows the L2 norm error of ∇q versus refinement ratio. This error represents a worse case scenario, as only coarsened quantities are used in the computation.

In addressing the issue of accuracy, our approach will be to continue sending the coarse mesh in the *all-to-all* communication phase of each simulation timestep, but to recover

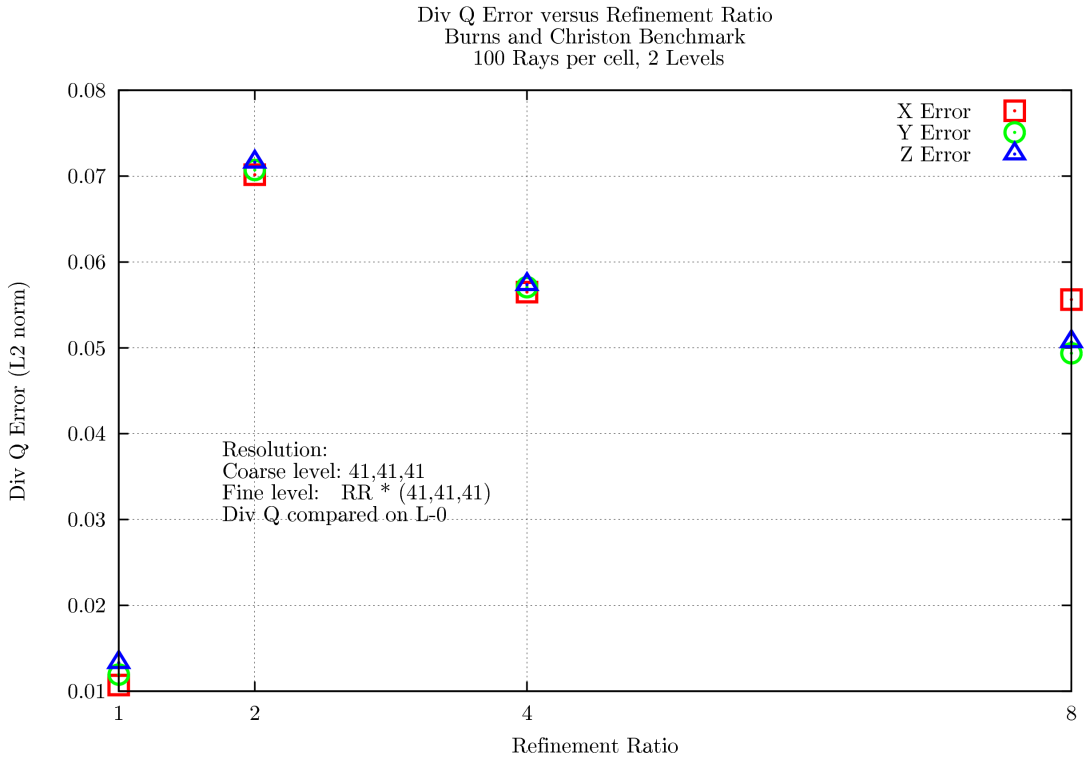

Figure 5.6. L2 norm error of ∇q vs refinement ratio. The error in each direction (x,y,z) is shown.

the fine-mesh values of the radiative properties through interpolation. This approach is well suited for GPU accelerators such as those on the Titan system, where FLOPS are inexpensive relative to the cost of data movement. Further compression of the coarse-mesh information will also be investigated.

5.4 Summary and Conclusion

The work in [53] demonstrated that through leveraging the multilevel AMR infrastructure provided by the Uintah framework, a scalable approach to radiative heat transfer using reverse Monte Carlo ray tracing was possible. *Although the AMR methods used in this work are not new necessarily, the application of these methods to radiative heat transfer algorithms and their scalability is novel.* These scaling results provide a promising alternative to approaches to radiation modeling, such as discrete ordinates. Using the cost model for communication and computation [53], we can predict how our approach to radiation modeling may scale and perform on current, emerging, and future architectures.

The primary focus in moving beyond this study would be to continue development of RMCRT capabilities and to provide support for several additional energy-related problems within the scope of the Utah CCMSC. The calculations demonstrated in this work would become ideal candidates for large-scale accelerator use, employing large numbers of rays for every cell in the computational domain, specifically, using the whole of machines such as Titan with accelerators. Adapting this work to leverage the on-node GPUs of Titan [54] is detailed in Chapter 6.

CHAPTER 6

SCALABLE RADIATION MODELING TO 16,384 GPUS

The need to solve larger and more complex simulation problems while at the same time not incurring higher and higher power costs has led to an increasing focus on GPU and Intel Xeon Phi-based architectures. Many existing and most emerging high-performance computing (HPC) systems rely on such architectures. In the case of the DOE Titan system, with a theoretical peak performance of 27 petaflops, over 90% of the computational power comes from its 18,688 GPUs. These heterogeneous systems pose significant challenges in terms of programmability due to deep memory hierarchies, vendor-specific language extensions, and memory constraints, e.g., less device-side memory compared to host memory per node. To preserve current capabilities on upcoming machines and to solve larger and more complex simulations on existing machines, HPC codes must effectively leverage manycore architectures. This chapter details the significant changes to Uintah needed to leverage GPU-based architectures, such as DOE Titan and the proposed DOE Summit, for large-scale calculations. To achieve good performance on these architectures, it is important that algorithms and codes effectively leverage an arbitrary number of GPUS on-node while simultaneously utilizing all GPUs available in an allocation, potentially thousands.

This chapter demonstrates that radiative heat transfer problems can be made to scale within Uintah on heterogeneous systems through a combination of reverse Monte Carlo ray tracing (RMCRT) techniques and adaptive mesh refinement (AMR) to reduce the amount of global communication. In particular, significant Uintah infrastructure changes, including a novel lock and contention-free, thread-scalable data structure for managing MPI communication requests, and improved memory allocation strategies, were necessary to achieve excellent strong scaling results to 16,384 GPUs on Titan [54].

A principal challenge in modeling radiative heat transfer is the strong nonlocal nature of radiation, with potential propagation of radiation across the entire domain from any point. For our RMCRT model, this challenge translates to an all-to-all communication requirement that replicates the boiler geometry on each node to facilitate local ray tracing. This challenge is addressed by leveraging Uintah's AMR capabilities in a novel way, using Cartesian mesh patches to generate a fine-mesh that is used only locally (close to each grid point) and a successively coarser mesh that is used further away, via a level-upon-level approach. This approach is fundamental to the CCMSC target problem, where the entire computational domain needs to be resolved to adequately model the radiative heat flux. Using this approach, we previously showed excellent strong scaling to over 262,144 CPU cores on the DOE Titan system for problem sizes that were previously intractable with a single fine-mesh RMCRT approach due to on-node memory constraints [53]. This scaling was consistent with the communication and computation model in [53].

The challenges in moving from a CPU to a GPU-based multilevel RMCRT algorithm using this mesh refinement approach have extended well beyond what a typical GPU port of a CPU code might entail. In the case of the Uintah open-source framework, additional complexities are posed by these architectures based on a core Uintah design that focuses on insulating the application developer from the underlying architecture. Thus, in the context of heterogeneous systems, Uintah's asynchronous task-based paradigm requires that all *host-to-device* and *device-to-host* data copies for computational task dependencies (inputs and outputs), as well as device context management, must be handled automatically in the same way MPI messages are generated by the Uintah runtime system, as shown in [34], [35].

The current Uintah model has departed from an MPI-only approach and now employs a shared memory model on-node [32], [35]. This combination of MPI + Pthreads, and in the presence of GPUs, also Nvidia CUDA, all coupled with shared data structures and the use of MPI_THREAD_MULTIPLE (where all CPU threads perform their own MPI sends and receives), creates an environment for potential race conditions and deadlock scenarios, some of which manifest only at larger scale in our experience and are routinely difficult to debug.

This chapter focuses on how the challenges involved with this mixed concurrency

environment were addressed, enabling this difficult globally coupled, all-to-all problem to scale to 16,384 GPUs on the DOE Titan system, showing this approach to be feasible in production boiler calculations on current and future GPU-based heterogeneous architectures. This result has been achieved through a focused progression, starting first with the work presented in [34] and Section 4.1 to achieve basic GPU task scheduling and execution. This initial GPU-realted work within Uintah implemented a proof-of-concept, single-level GPU RMCRT algorithm and heterogeneous task scheduler and runtime system within Uintah, and was the origin of this work. Second in this progression was the preliminary multilevel RMCRT work, focusing on CPU scaling, where excellent strong scaling to over 262,144 CPU cores was demonstrated on the DOE Titan system [53] and presented in Chapter 5. The specific contributions made by this work in moving from a CPU to a GPU-based multilevel RMCRT algorithm are the extensive modifications to the Uintah infrastructure necessary to achieve the GPU scaling results shown in Section 6.4.

These contributions are, specifically:

1) Leveraging Uintah's AMR infrastructure, applied to radiation in a novel way to reduce the volume of communication sufficiently so as to allow scalability. Uintah's AMR capabilities are introduced in Section 1.3, along with an overview of Uintah. The details of this novel application of AMR are covered in Section 5.1.2;
2) Changing the way that AMR meshes are stored on the GPU to overcome the limited available GPU global memory. This change has entailed a significant extension of the Uintah `GPU DataWarehouse` system [50] to support a mesh-level database, a repository for shared, per-mesh-level variables such as global radiative properties. This result has allowed multiple mesh patches, each with associated GPU tasks, to run concurrently on the GPU while sharing coarse, radiation mesh data. This extension of the `GPU DataWarehouse` is discussed in Section 6.2, which also gives an overview on radiation transport and describes our GPU-based multilevel RMCRT model.
3) The introduction of novel nonblocking, thread-scalable data structures for managing asynchronous MPI communication requests, replacing previously problematic Mutex-protected vectors of MPI communication records. *To be nonblocking, a wait, failure, or resource allocation by one thread cannot block progress on any other thread.* Nonblocking data-structures are lock-free if at all steps at least one thread is guaranteed

to make progress, and are wait-free if at any step all threads are guaranteed to make progress [108]. Section 6.3 describes these changes and their motivation, and also shows speed-ups in local MPI communication times made possible through these infrastructure improvements.

4) A vastly improved memory allocation strategy to reduce heap fragmentation is covered in Section 6.3. This strategy allows running simulations at the edge of the available nodal memory on machines like Titan.
5) Determining optimal fine-mesh patch sizes to yield GPU performance while maintaining over-decomposition of the computational domain to hide latency. This patch size optimization is covered in Section 6.4, where we provide strong scaling results over a wide range of GPU counts up to 16,384 GPUs, and also show the results of differing patch configurations across this range of GPUs

6.1 RMCRT and Ray Tracing Overview

The principal motivation for the development of a GPU-based RMCRT radiation calculation arises from the computational intensity of the radiation solve in the CCMSC production runs, consuming as much as 50% of the overall CPU time per timestep when using DOM. Additionally, this work is motivated by access to large-scale GPU-based machines like DOE Titan, where over 90% of the available FLOPS are on the GPUs. Many of the CCSMSC target simulations will target Titan over its life span. Beyond this, utilization of the DOE Summit system is planned.

Reviewing the RMCRT model from Section 2.3, RMCRT uses rays more efficiently than forward MCRT, but it is still an *all-to-all* method, for which all of the geometric information and radiative properties for the entire computational domain must be accessible by every ray [47]. These radiative properties consist of κ, the absorption coefficient, a property of the medium the ray is traveling through; σT^4, a physical constant $\sigma \cdot$ temperature field, T^4; and *cellType* (boundary or flow cell), a property of each computational cell in the domain. In our approach, the boiler geometry is replicated on each node and ray tracing takes place without the need to pass ray information across nodal boundaries (via MPI) as rays traverse the computational domain. Our RMCRT approach is afforded the choice of replication due to the relative simplicity of the boiler geometry.

To address these communication challenges, we have developed a multilevel AMR approach for both CPU [53] and now GPU, in which a fine-mesh is only used close to each grid point, and a successively coarser mesh is used further away, significantly reducing MPI message volume and nodal memory footprint. This algorithm allows for the radiation computation to be performed with an appropriate mesh resolution while still being coupled with other physics components. The LES CFD, particle transport, and particle reactions are solved on a different mesh resolution appropriate to their physics and models. This balanced approach to coupling multiphysics is made possible by Uintah's AMR design. The amount of data stored on every computational patch is significantly reduced, and the computational overhead for successively finer computation is eliminated when not needed.

6.2 Multilevel GPU Implementation

Following our original proof-of-concept GPU task scheduler introduced in [34], a single-level CPU and GPU RMCRT approach was initially considered. This approach was to begin comparisons against the current DOM solver within the Uintah ARCHES component, using the benchmark problem described by Burns and Christon in [1]. Accuracy studies of this single-level RMCRT approach are shown in [48] for this benchmark, which examines the accuracy of the computed divergence of the heat flux and shows expected Monte Carlo convergence when compared to the published data in [1]. In this approach, the quantity of interest, the divergence of the heat flux, ∇q, is calculated for every cell in the computational domain. The entire domain was replicated on every node (with all-to-all communication) for the radiative properties. This replication occurred on the single fine mesh, which for N_{total} mesh cells, the amount of data communicated is $\mathcal{O}(N_{total}^2)$.

Although this single, fine-mesh approach was highly accurate and effective at lower core and GPU counts, problem sizes beyond 256^3 were intractable for highly resolved domains, especially on machines with less than 2GB of memory per core. GPU scalability results were shown up to 64 GPUs through the work done in [34] to achieve basic accelerator task scheduling and execution. Using a problem size of 128^3, the volume of communication coupled with the PCIe transfers begins to dominate, and the GPUs were

starved for work with only a single patch per GPU. These difficulties led to the use of an AMR approach that uses a mesh hierarchy to limit the amount of communication on CPU architectures [53].

Figure 5.2 best illustrates this approach with a 2D diagram of three-level mesh refinement scheme, illustrating how a ray from a fine-level patch (right) might be traced across a coarsened domain (left). In general, the data required by our multilevel RMCRT algorithm from the fine CFD mesh are projected to all coarse levels subject to a user-defined refinement ratio (typically 2 or 4), where each coarse level spans the entire domain. Our general multilevel RMCRT ray marching process is described in detail in[53], which includes a precise model of communication and computation.

A significant challenge in moving to a GPU-based, multilevel RMCRT algorithm is the limited amount of global memory available on the current generation of GPUs found on Titan. These Nvidia K20X models have 6GB compared to 32GB CPU host-side. The Uintah `DataWarehouse` design automatically generates MPI messages and keeps multiple versions of variables for out-of-order scheduling and execution [31], as different tasks may require the same variable on the same neighboring patch multiple times for differing ghost cell requirements. Tasks may also need input variables prior to modification. In order to support these and other scenarios, the "*on-demand*" `DataWarehouse` provides the application the illusion it has access to memory it does not actually own (via the task input specification, where the ghost cell requirement is specified). In the context of our multilevel RMCRT radiation model, this is a global halo, or "infinite ghost cell" requirement on all coarse levels. Because of this design, data from the coarser levels are retrieved from the Uintah `DataWarehouse` for each fine-level patch on a node. This situation presents problems for a limited memory footprint as on Titan's K20X GPUs.

The solution to this problem has been to effectively short-circuit the creation of these redundant global copies of the radiative properties on the host and their subsequent transfer across the PCIe bus to the GPU. This solution has been achieved by a significant extension of the Uintah `GPU DataWarehouse` system [50] to support a level database that stores a single copy of shared global radiative properties (per-mesh level based on Uintah's level-upon-level approach to AMR). Our solution has effectively minimized PCIe transfers and ultimately allowed multiple mesh patches, each with GPU tasks, to run concurrently on

the GPU while sharing data from the coarse radiation mesh. This design leverages the two copy engines available on the K20X GPUs and also makes use of support for running multiple, concurrent kernels. Using these features, Uintah can copy data for multiple fine-mesh patches to the GPU, each sharing a global copy of the coarsened radiative properties [54].

Data for these GPU tasks can be simultaneously copied to-and-from the device as multiple RMCRT kernels run simultaneously. `CUDA Streams`, managed by the Uintah infrastructure, provide additional concurrency, as operations from different streams can be interleaved [34], [35].

6.3 Uintah Infrastructure Improvements

Uintah uses an "MPI + *X*" approach, a combination of MPI + (Pthreads + Nvidia CUDA). This mixed concurrency model has the potential for problematic race conditions and deadlock scenarios, some of which manifest only at larger scale in our experience. Significant infrastructures changes were necessary to improve nodal throughput and to expose more concurrency while maintaining correctness within this complex environment. In particular, it was necessary to choose optimal data structures and algorithms to efficiently expose concurrency as well as to maintain critical sections around legacy serial data structures. Furthermore, it was vital for Uintah to manage limited resources such as nodal memory through the use of custom allocators that allow frameworks like Uintah to choose more optimal allocation policies for different objects to better utilize available resources and improve nodal throughput.

6.3.1 Multithreaded Processing of Asynchronous MPI

Uintah uses MPI_THREAD_MULTIPLE currently (which to the best of our knowledge is rarely adopted by MPI users), which allows individual threads to perform their own MPI sends and receives. Initial attempts to run at large scale with accelerators in this environment exposed a subtle race condition in the shared vector used to process outstanding MPI_Requests via `MPI_Testsome()`, which was protected by a Pthread write-lock. This race scenario involved multiple threads simultaneously processing the same received message, with all threads allocating a buffer for the same MPI message, and only one thread actually processing the message and invoking the callback to deallocate its buffer. Other threads

may have allocated buffers that were never released, resulting in a severe memory leak in the Uintah infrastructure, which caused the application to quickly fail at large scale due to *out-of-memory* errors on the compute nodes.

Although this scenario was present in other simulations, it was evident only at large scale and only significant within our RMCRT radiation model due to the high volume and size of MPI messages. Despite this scenario, the approach to multithreaded processing of asynchronous MPI within Uintah had worked seemingly well for all cases until now.

A more coarse-grained critical section was not feasible because it would have serialized a substantial portion of the algorithm. The solution ultimately required a fundamental redesign in the data structure and algorithm used to manage MPI communication records in a multithreaded environment [54]. The new algorithm leverages a novel wait-free pool, which is thread scalable and contention free, to store individual MPI requests. The wait-free pool iterator is implemented as a unique, move-only object that toggles an atomic flag to protect access to the referenced value to prevent data races, i.e., multiple threads modifying the same value. Using `C++11` features (*atomics, move constructor, move assignment, and disabling copy construction and copy assignment*) to implement a unique protected iterator that guarantees no two threads can have iterators that de-reference the same object. `MPI_Test()` is then used on each request individually in contrast to the prior design, which used `MPI_Testsome()` to test a collection of requests. This solution, outlined in Algorithm 6, results in much simpler code with fewer allocations and eliminates the complexity of managing the previously used locked vectors of `MPI_Request` objects and their related critical sections.

Algorithm 6 Wait-free MPI_Request pool

```
1: RecvCommList& recv_list = m_recv_lists[id];
2: auto ready_request = [](CommNode const& n)→bool{return n.test();};
3: iterator = recv_list.find_any(ready_request);
4: if (iterator) then
5:     MPI_Status status;
6:     iterator→finishCommunication(m_comm, status);
7:     recv_list.erase(iterator);
8: end if
```

Achieving multithreaded correctness and performance requires different algorithm and data structure choices. These choices are illustrated in our example by moving to the use of `MPI_Test()` from multiple/many threads and away from the complexity of managing data structures designed for the use of `MPI_Testsome()`.

6.3.2 Memory Allocation and Management Strategy

After identifying and addressing the race condition described above, our RMCRT benchmark problem [1] still failed at scale due to memory-related issues, although it ran longer before failure. Further investigation revealed that extreme heap fragmentation was occurring when running our RMCRT benchmark problem. Persistent small allocations mixed with transient large allocations fragmented the heap such that it grew continually, acting as though a significant memory leak still existed. Using Google's tcmalloc [109], a highly scalable memory allocator for multithreaded applications reduced heap fragmentation, but the mixture of persistent and transient allocations still resulted in unacceptable fragmentation. Furthermore, frequent small allocations from multiple threads also caused a performance degradation due to contention of shared resources. The performance of the infrequent large allocations was not a factor in the overall performance.

6.3.3 Custom Allocators to Reduce Fragmentation

Developing custom allocator classes for Uintah's MPI buffers and `GridVariables` (simulation variables that reside on Uintah's Cartesian mesh patches at cell centers, nodes, or faces x,y,z) allowed us to leverage our knowledge of how a data structure would be used to distinguish between large/small and transient/persistent allocations, which greatly improved memory utilization and reduced fragmentation.

To eliminate the observed heap fragmentation, this work developed allocators to address the range of allocation sizes that were causing the fragmentation. For large allocations, this work completely avoids the heap by implementing a specialized allocator that uses `mmap` to allocate anonymous virtual memory. Although `mmap` is a system call and can be slower than a standard `malloc`, it was more important to avoid fragmenting the heap than to optimize the throughput of large allocations. Throughput was not a concern for the performance of large allocations, but it is critical for frequent small allocations. To

manage our small transient objects, i.e., objects that are frequently created and destroyed, a lock-free memory pool was developed on top of the `mmap` allocator to avoid the heap and to maximize throughput. All other infrequent allocations are still managed using the heap.

With these custom allocators, Uintah is now better able to manage memory requirements by designing for specific needs and requirements to reduce fragmentation and increase throughput when necessary. As the scope of problem sizes increase and simulations are pushed to the full capacity of available resources, specific management strategies and algorithms must be developed to better use available resources. Generic algorithms are no longer sufficient to accommodate the bleeding edge that high-performance computing lives on.

Using these techniques, portions of Uintah infrastructure code related to communication were significantly simplified, and nodal throughput was improved by a factor of 2-4X in processing local MPI communication (the time spent posting MPI messages for individual threads). Figure 6.1 shows the time spent doing local communication, before and after our infrastructure improvements for our CPU implementation of the Burns and Christon [1] RMCRT benchmark on Titan. These runs were from 512 to 16,384 nodes, with a two-level problem with 136.31M cells, 512^3 on the fine CFD mesh, and 128^3 on the coarse radiation mesh. This problem had 262,144 total mesh patches. Per Uintah's automatic MPI message generation mechanism to meet the halo requirements of tasks residing on patches, communication, both local and global, is done for each patch in the computational domain. The red line represents times for stock code before changes, and the green line shows times after our improvements.

Table 6.1 lists data from Figure 6.1 with times before and after infrastructure improvements and the associated speed-ups. More detailed information on communication frequency, as well as average message volume and latency for this radiative heat transfer calculation, is shown in [53]. The speed-ups shown in Table 6.1 are a direct result of removing only a single Mutex and related critical sections. It is expected that refactoring other sections of infrastructure code will yield similar improvements.

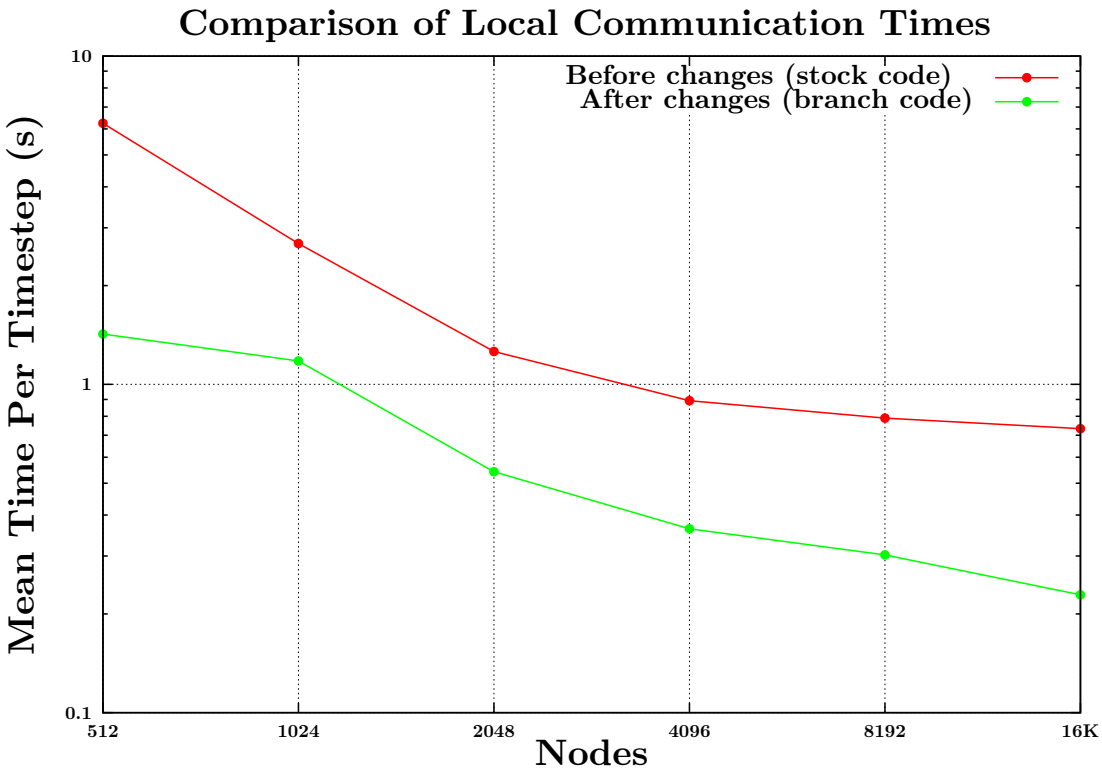

Figure 6.1. Comparison of the local communication time (sec) before and after infrastructure improvements.

Table 6.1. Local communication data shown in Figure 6.1 with speed-ups.

Comparison in Local Communication Times						
#Nodes	512	1k	2k	4k	8k	16k
Time before (sec)	6.25	2.68	1.26	0.89	0.79	0.73
Time after (sec)	1.42	1.18	0.54	0.36	0.30	0.23
Speedup (X)	4.40	2.27	2.33	2.47	2.63	3.17

6.4 GPU Scaling Results

In this section, we show strong scalability results on the DOE Titan XK7 [1] system for the Burns and Christon [1] benchmark problem using the GPU implementation of the multilevel mesh refinement approach. We define *strong scaling* as a decrease in execution time when a fixed size problem is solved on more cores and *weak scaling* as the change in execution time as the number of processors and the problem size vary proportionally to each other. We define parallel efficiency, E as

$$E = \frac{T_{serial}}{N * T_{parallel}(N)}, \tag{6.1}$$

where T_{serial} is the time to solution using 1 processing unit, N is the number of processing units, and $T_{parallel}(N)$ is the time to solve the same problem with N processing units. Strong scaling is important in our case because the CCMSC seeks to solve a fixed target problem in a tractable amount of time using more compute resources. To achieve this goal, the Utah CCMSC needs the whole of machines like Titan.

In previous work [53], [54], weak scaling results were not shown due to the general nature of the growth in communication for this problem, specifically that radiation or any globally coupled algorithm grows quadratically as $\mathcal{O}(N^2)$ (N is the number of communicating MPI ranks) with respect to the problem size. Although this is true for a single, fine-mesh problem, weak scaling could be achieved within Uintah by using varying degrees of adaptive mesh refinement for the radiation mesh as the core count grows for a fixed problem size. Weak scaling of RMCRT using the methods described is explored in conjunction with AMR and varying degrees of refinement in Chapter 5.

Figure 6.2 and Figure 6.3 show the performance and scalability of the multilevel RMCRT:GPU algorithm for three patch sizes. In each fine-level cell in both problems, 100 rays were used to compute the divergence of the heat flux. The number of cells in a patch was varied, 16^3 (red), 32^3 (green), and 64^3 (blue). Each simulation consisted of a grid with two AMR levels and used a refinement ratio of four (RR:4) between the levels. All simulations

[1] Titan is a Cray KX7 system located at Oak Ridge National Laboratory, where each node hosts a 16-core AMD Opteron 6274 processor running at 2.2 GHz, 32 GB DDR3 memory, and 1 NVIDIA Tesla K20x GPU with 6 GB GDDR5 ECC memory. The entire machine offers 299,008 CPU cores and 18,688 GPUs (1 per node) and over 710 TB of RAM. Titan uses a Cray Gemini 3D Torus network, 1.4 μs latency, 20 GB/s peak injection bandwidth, and 52 GB/s peak memory bandwidth per node.

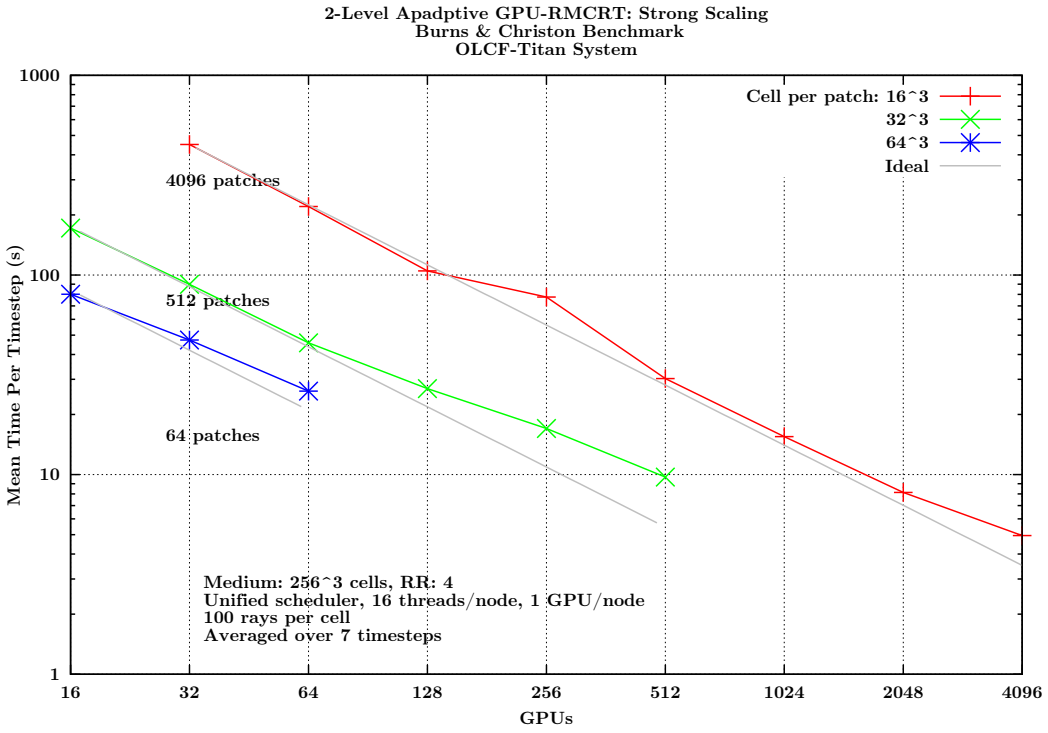

Figure 6.2. GPU strong scaling of the MEDIUM two-level benchmark RMCRT problem for three patch sizes on the DOE Titan system.

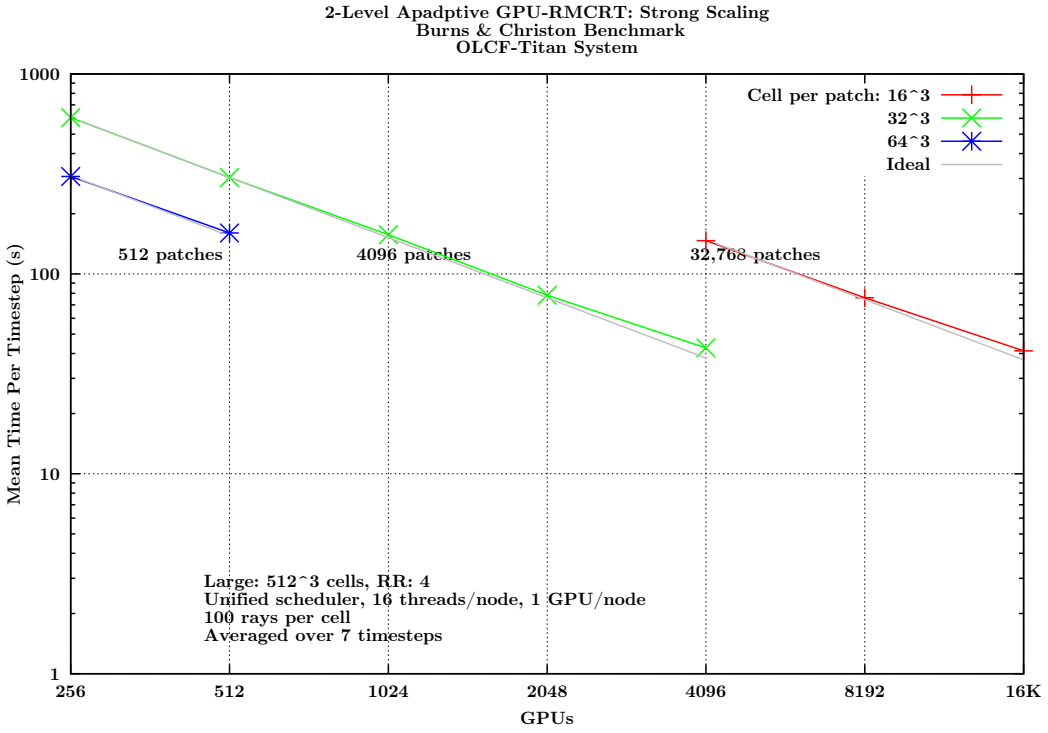

Figure 6.3. GPU strong scaling of the LARGE two-level benchmark RMCRT problem for three patch sizes on the DOE Titan system.

were run on the DOE Titan system, leveraging the single GPU per node with Uintah's hybrid, multithreaded task scheduler and runtime system originally designed and tested in [35], [50] using 16 threads and one GPU per node. This scheduler and runtime system has been heavily modified, as outlined in Section 6.3, to achieve the results shown here.

For the simulation results shown in Figure 6.2, the total number of cells in the domain was 17.04 million. The fine level contained 256^3 cells and the coarse level contained 64^3 cells. For the larger simulation results shown in Figure 6.3, the total number of cells in the domain was 136.31 million. The fine level contained 512^3 cells and the coarse level contained 128^3 cells. Using 6.1, the strong scaling efficiency of the large benchmark problem (Figure 6.3) is 96%, going from 4,096 to 8,192 GPUs, and 89%, going from 4,096 to 16,384 GPUs.

These results show, in general, that using larger patches provides more work per GPU and yields a more significant speed-up. With the improvements to the Uintah infrastructure outlined in this work, we observe excellent scaling through processing multiple patches per GPU, and the algorithm and implementation scale well to 16,384 GPUs as a result of these improvements to Uintah. These results also offer a promising and scalable approach to radiative heat transfer calculations for the CCMSC target boiler problem on current and emerging heterogeneous architectures.

6.5 Related Work

Much of the work done toward developing scalable radiation transport models can be found in computational astrophysics and cosmology, involving problems such as neutron star merger, supernova, and high-energy density plasma. This work is in the context of codes like ARWIN, the AZEuS adaptive mesh refinement, magnetohydrodynamics fluid code, the more general AMR-based FLASH code [110], based on oct-tree meshes, and the physics AMR code Enzo [111]. Humphrey et al. [53] demonstrated a scalable approach to radiation modeling to 262,144 CPU cores using a reverse Monte Carlo ray tracing approach with AMR. At the national labs, radiation codes such as RAMSES and PARTISN [14] exist, but are not generally available or target problems like neutron transport, as found in CRASH [112], a block adaptive mesh code for multimaterial radiation hydrodynamics. Some radiation transport problems use CFD codes and AMR techniques [7], [98], [113];

however, a broad range of problems exist that require the concept of tracing rays or particles, such as the simulation of light transport and electromagnetic waves [53]. Much of the available literature on GPU-based Monte Carlo ray tracing approaches to radiation can be found in the oncology community, where GPUs are used for radiation dose calculation [114].

Overall, there are very few cases of GPU usage at the scale reported here. Gaburov et al. [115] have published results on the evolution of the Milky Way galaxy, a calculation done using 18,600 GPUs on DOE Titan. Gray et al. [116] were among the first to take advantage of at least 8,192 GPUs in parallel with their Ludwig soft matter physics application. Many of these reports involve stories concerning code-related issues scaling on Titan and offer detailed discussion on how software teams overcame significant code and algorithmic challenges in porting their applications to GPU-based architectures.

6.6 Summary and Conclusion

This work has demonstrated that radiative heat transfer problems can be made to scale within Uintah on current petascale heterogeneous systems through a combination of reverse Monte Carlo ray tracing (RMCRT) techniques and AMR to reduce the amount of global communication. This approach aims to enable the Utah CCMSC to run the target 1200 MWe boiler problem on current and emerging GPU-based architectures at large scale.

This work has also demonstrated the necessity of choosing optimal data structures and algorithms to efficiently expose concurrency. We have illustrated how maintaining critical sections around serial data structures in legacy code increases code complexity and the likelihood of the introduction of difficult race conditions and deadlock scenarios, especially when using mixed concurrency models, namely *MPI + (Pthreads + Nvidia CUDA)*. Furthermore, this work has shown the necessity for frameworks like Uintah to better manage limited memory through the use of custom allocators to choose better allocation policies and to better utilize available resources, improving nodal throughput. The specific contribution of this work is the development of a scalable radiation model for current and emerging heterogeneous architectures, made widely available through the Uintah open-source framework.

CHAPTER 7

AN INDUSTRIAL BOILER PROBLEM WITH RADIATION

A broad class of large-scale multiphysics applications requiring long-range interactions, such as molecular dynamics [4], cosmology [5], neutron transport [6], and radiative heat transfer [7] calculations, each 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 in terms of both computational analysis and memory storage if the amount of data to be sent is large in contrast, for example, to an MPI reduction.

The motivation for this chapter comes from our experience running massively parallel simulations aimed at predicting the performance of a commercial, 1200 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 [35], [50], which managed the scheduling and execution of over 8 million computational tasks on 119,000 CPU cores and 7,500 GPUs simultaneously.

In these boiler simulations, the dominant mode of heat transfer is radiation, which presents significant challenges for AMT runtime systems [16] due to the all-to-all nature of radiation. The principal challenge related to this dissertation 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 [31]. 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 needs to search only 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 specific contribution of this chapter is in addressing these challenges through an improved search algorithm to reduce dependency analysis processing time by avoiding unnecessary searches, combined with multiple task graphs. This chapter demonstrates 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 speed-ups 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.

7.1 Target Problem Background

The global dependencies in our target 1200 MWe coal boiler problem arise from solving the radiative heat transfer equation (RTE) [53], [7]. Figure 7.1 depicts O_2 concentrations in the boiler chamber and illustrates the sheer size of the problem being modeled by Uintah and the Arches component. Both the DOE Titan and Mira systems were used to simulate coal boiler designs using different methods for computing the RTE (2.2). On Mira, the global radiation dependencies required numerous sparse, global linear solves for the discrete ordinates method [45]. For every data dependency sent out from one compute node to another, the source node could expect to receive a corresponding data dependency

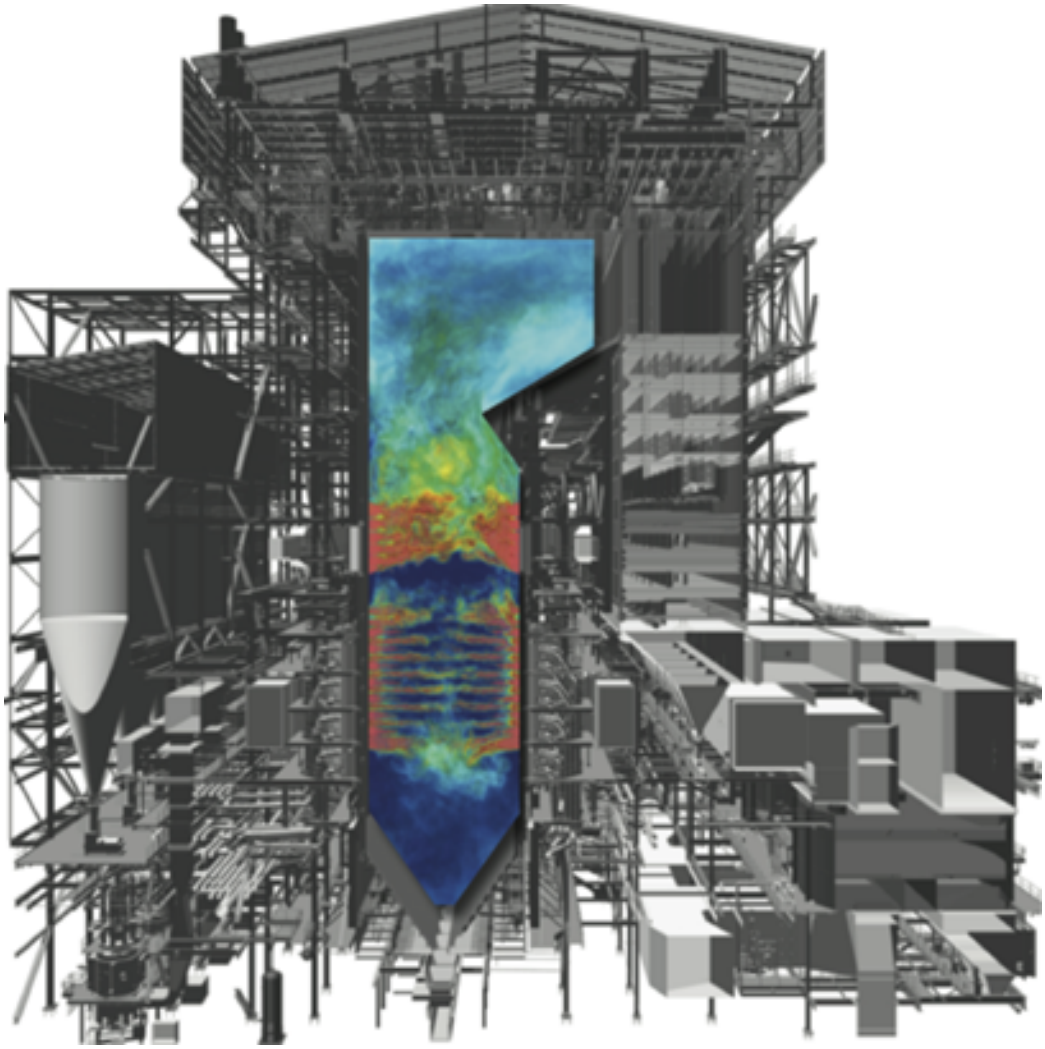

Figure 7.1. Side view of CCMSC target 1200 MWe boiler problem, showing O_2 concentrations in the boiler chamber. The boiler chamber itself is ˜90 m tall.

in return. On the Titan platform, radiative heat transfer was computed using a reverse Monte Carlo Ray Tracing (RMCRT) technique [53], 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 220,000 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** (Figure 1.5), 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.

7.1.1 RMCRT Radiation Model

The scalability of the RMCRT model has been demonstrated to 262,144 CPU cores [53] and 16,384 GPUs [54] on a benchmark problem [1]. The challenge was then in using this model in the production boiler case. The RMCRT model creates 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 [53], 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 initially used to reduce these data dependencies [53]. 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 7.2, a ray begins from the fine-mesh level partition present on the node and will eventually transition onto the global coarse-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 are updated when the ray terminates.

The goal of these simulations is to discover whether modeling such coal boilers makes it possible to further enhance their performance by using the resolutions possible only on petascale machines. On Titan specifically, the goal of our project is to show that: 1) RMCRT is a viable alternative to the discrete ordinates method for solving the RTE; and 2) hybrid architectures such as Titan can be used by employing a mixture of Arches CPU tasks and

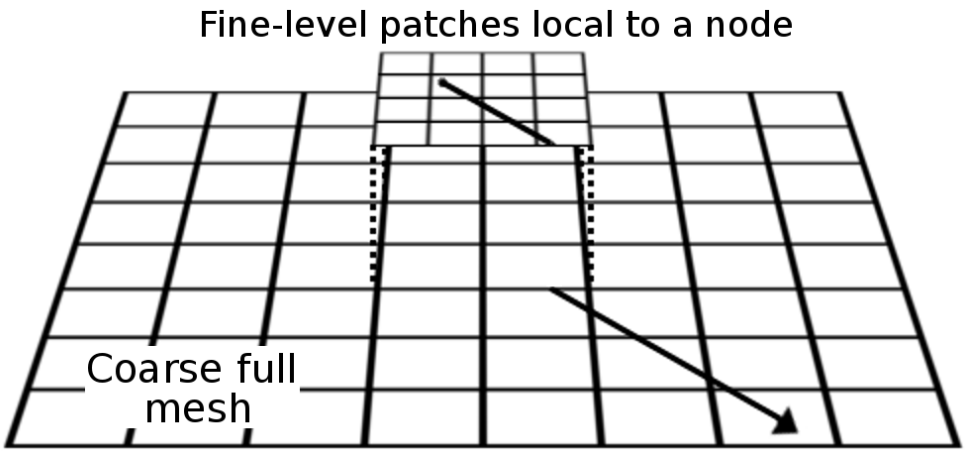

Figure 7.2. A 2D representation of an RMCRT ray as it moves across a domain with two levels of refinement. A ray begins on the fine-mesh level and transitions to the coarse-mesh level, terminating after its intensity falls below a specified threshold.

RMCRT GPU tasks in a scalable manner.

7.1.2 Arches in a Production Calculation

The Uintah Arches turbulent combustion component (Section 1.3.1) is used to compute coal multiphysics for the target boiler problem. The combination of Arches and RMCRT tasks dramatically complicated data dependency analysis. The Arches component generated close to 1000 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.

7.2 Task Graph Compilation Improvements

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. First, each node spent hours processing potential dependencies among the thousands of other nodes, with only a small fraction of these dependency checks finding actual data dependencies. Second, the simulation required reprocessing all potential dependencies

before and after each radiation solve to incorporate or exclude global dependencies time a timestep alternated the type of problem to compute. 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**.

7.2.1 Prior Global Dependency Support Within Uintah

Within Uintah, analyzing tasks for data dependencies is referred to as dependency analysis, part of the task graph compilation process. It is important to first note that for standard stencil calculations, where each compute node needs to search only surrounding nodes containing neighboring simulation data, *dependency analysis completes in milliseconds, even at scale*.

Previously, Uintah supported only a very primitive notion of a *"nonlocal halo,"* e.g., a halo that extended beyond the extents of a neighboring patch that could potentially encompass the entire computational domain, including patches from separate AMR levels. This value was defined by a single Uintah scheduler member, ***max_ghost_cells***, and was applied as a global maximum to all variables within all tasks across all patches in the computational domain. This global maximum had the effect of all variables requiring a halo depth of ***max_ghost_cells***. If RMCRT radiation tasks were in the mix, this value could be the entire domain, which not only meant severely overcommunicating unnecessary data via MPI, but also introducing severe overcompensation when performing dependency analysis between tasks that required only local communication, e.g., a standard stencil operation with stencil depth of 1-2 cells. Although this simplistic approach worked fine for RMCRT benchmark calculations [1] on a single AMR level, with relatively few tasks overall and no AMR [34], it broke down when running a full industrial boiler simulation at a production scale.

7.2.2 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. It is important to note that after patches are assigned to processors, each processor creates its own and neighbors' instances of tasks. *The neighbors' detailed tasks are created only for dependency analysis and will not be executed* [31]. 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 the identified dependencies are placed into a task graph, the corresponding MPI messages tags are created [31].

7.2.3 Example Dependency Analysis

A simplified problem demonstrating global dependencies is given in Figure 7.3, 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 with which tasks it interacts with (via halo exchange). For simulations with only

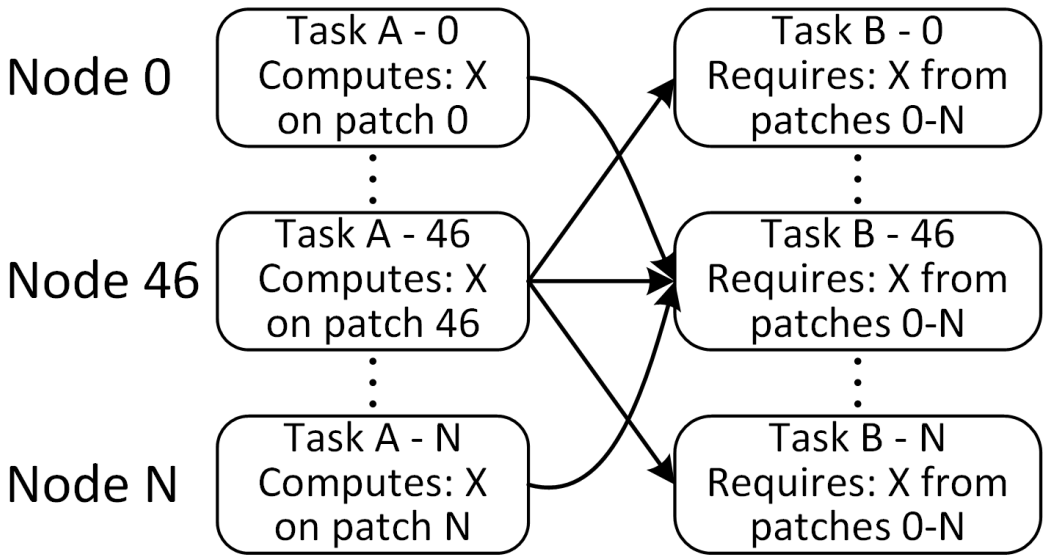

Figure 7.3. A visualization of data dependencies from the perspective of Node 46 in a simple N node problem with a global dependency on simulation variable X. After Node 46's Task A executes, the data dependencies must be sent out to $N-1$ other nodes. Similarly, before Node 46's Task B executes, it must receive data dependencies from $N-1$ other nodes.

local communication (nearest neighbor), the resulting processor neighborhood is relatively small [31], being at most 26 nodes immediately surrounding the source node (three 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.

7.2.4 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 significantly 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 that *only compute variables* and do not require halos. For example, in Figure 7.3, the set of "A" tasks does 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 7.3, the set of "A" tasks is 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 used only simulation variables requiring one cell of halo data, those tasks would be assigned a local neighborhood. If any sets of tasks shares 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 nonuniform 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.

7.3 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., 10 tasks over 100 patches would result in 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 is small with a maximum of 26 surrounding patches (three dimensions - x,y,z), and the total number of task objects is manageable. However, when considering global dependencies, the dependency analysis becomes $\mathcal{O}((ng)^2)$, effectively a search between every task/patch tuple, ng in the computational domain.

In the Titan boiler simulation, the two mesh levels used to achieve scalability [53] further increased the number of global dependencies due to the interlevel dependencies using the same global neighborhood for every task on every mesh-refinement level. The number of task objects T is then given by

$$T = \sum_{l=1}^{l_{tot}} \sum_{t=1}^{t_{tot}} \sum_{n=1}^{n_{tot}} P_{l,t,n}, \tag{7.1}$$

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 7.1 in the target boiler case prior to our improvements was

$$\mathcal{O}(n_f \cdot g_f) + \mathcal{O}(n_c \cdot g_c), \tag{7.2}$$

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 $(n_f g_f = 10,034,808) + (n_c g_c = 10,080) = \mathbf{10,044,888}$. The introduction of global and local processor neighborhoods [55] tailored for a task's variables and mesh level reduced the total number of task objects by 81x, changing the complexity in 7.2 to

$$\mathcal{O}(\frac{n_f}{p} \cdot t_{fl}) + \mathcal{O}(\frac{n_c}{p} \cdot t_{cl}) + \mathcal{O}(n_c \cdot t_{cg}) + \mathcal{O}(n_f \cdot t_{fg}), \tag{7.3}$$

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 that globally distribute coarse-level simulation variables among nodes), and t_{fg} is the number of fine-level tasks with a global neighborhood (tasks that compute on the fine level and require a global coarse mesh).

Table 7.1 illustrates the improvements from this reduction in complexity for our standard RMCRT benchmark problem [1]. 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 improvement 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 7.3,

$$\mathcal{O}(n_f \cdot t_{fg}). \tag{7.4}$$

This particular task computes on every fine-level patch and also requires a global coarse mesh. Therefore, for every fine-level patch there exist data dependencies with every coarse-level patch. From a nodal perspective, most of these are duplicate dependencies. Chapter 8 begins to address this remaining cost by analyzing dependencies on a per-node

Table 7.1. Task graph compilation improvements combined with multiple task graphs (Section 7.4) enabled scaling to 122K patches for the target boiler problem.

Initial Dependency Analysis Improvements			
# 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

basis to automatically eliminate all duplicates. When complete, this analysis will change the fourth term in 7.3 to

$$\mathcal{O}(d_f \cdot t_{fg}), \tag{7.5}$$

where d_f is the number of *nodes* containing fine-level patches, and t_{fg} is the number of fine-level tasks with a global neighborhood (tasks that compute on the fine level and require a global coarse mesh). This change will retain the use of multiple processor neighborhoods while greatly reducing the number of dependencies stored and analyzed by the Uintah runtime.

7.4 Temporal Scheduling

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. 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 7.3, 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, as depicted in Figure 7.4. With this approach, the type of timestep

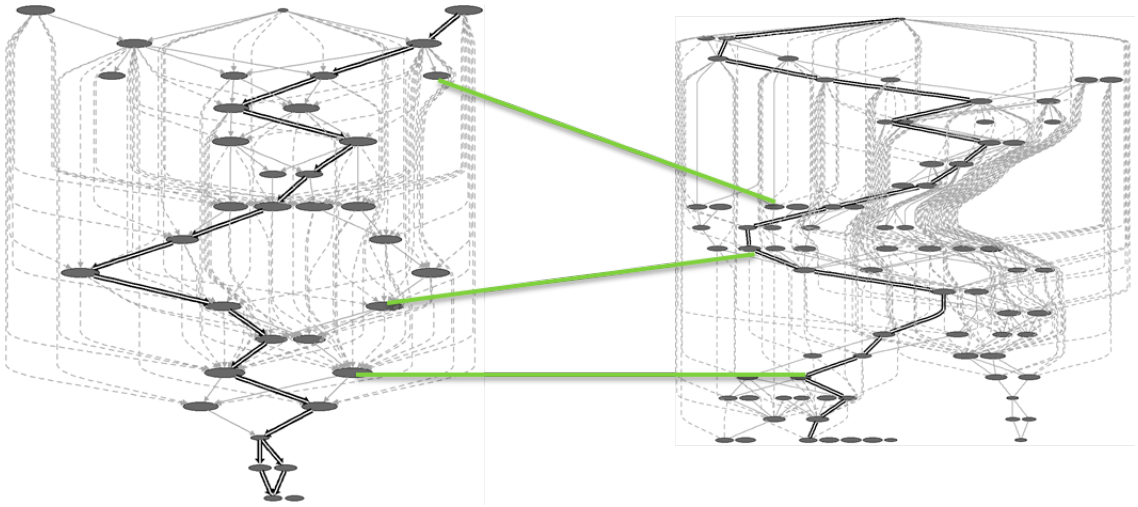

Figure 7.4. Multiple task graphs for a boiler simulation with radiation. Analysis of a task graph does not require knowledge of exactly what task graph existed in the prior timestep. Intertask graph dependencies (across timesteps) can create a graph edge with a special runtime task called `send_old_data` which associates a TG with every `DataWarehouse`.

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, and 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* (Figure 7.4) and associates it with every `DataWarehouse` 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 that 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 enumerated type is used for a set of primary task graphs, and is supplied in a header file. Tasks are then associated with a specific enumerated type:

```
// App specifies # graphs
scheduler->setNumTaskGraphs(num_task_graphs);

// Setup sensible ENUM to distinguish graphs
enum GRAPH_TYPE {
    TG_CARRY_FORWARD = 0    // carry forward TG (local comm)
  , TG_RMCRT         = 1    // rmcrt TG         (global comm)
  , NUM_GRAPHS };

// app specifies which graph to place task in
scheduler->addTask(taskRadiation,    TG_RMCRT);
scheduler->addTask(taskCarryForward, TG_CARRY_FORWARD );
```

Management of the Task pointer involved with multiple task graphs is automated through the C++11 smart pointer construct, `std::shared_ptr`:

```
// a smart pointer for shared ownership of an object, with automatic cleanup
std::shared_ptr<Task> task_sp(task); // Uintah::Task* task
```

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.

```
void SchedulerCommon::addTask(       std::shared_ptr<Task>   task
                              , const PatchSet              * patches
                              , const MaterialSet           * matls
                              , const int                     tg_num
                              ) {
  // During initialization, use only one task graph
  if (m_is_init_timestep
    m_task_graphs[m_task_graphs.size() - 1]->addTask(task, patches, matls);
  }
  else {
    // add task to all "Normal" task graphs (default value == -1)
    if (tg_num < 0) {
      for (int i = 0; i < m_num_task_graphs; ++i) {
        m_task_graphs[i]->addTask(task, patches, matls);
      }
    }
    // otherwise, add task to a specific task graph
    else {
      m_task_graphs[tg_num]->addTask(task, patches, matls);
    } } }
```

7.5 Spatial Scheduling

Uintah was originally designed for uniform task assignment on all patches across an entire domain. Adding support for *spatial scheduling*, tasks that are scheduled by the Uintah infrastructure on only a spatial subset of the computational domain, was required for spatial sweeps (Section 2.4) to perform well. As mentioned in Section 2.4, sweeps is a lightweight spatially serial algorithm in which spatial dependencies dictate its performance. These dependencies impose serialized internodal communication requirements and account for the bulk of the algorithms cost. Sweeps reduces this cost by facilitating a massive reduction in unnecessary communication for this algorithm. This support would also be used for tasks such as the RMCRT virtual radiometer. These tasks mimic an experimental heat flux sensor and are scheduled, for example, on a 2D slice within the domain. Radiometer tasks also have very little work relative to the ray tracing tasks, yet they have the same global communication costs as RMCRT.

In order to spatially schedule specific tasks, a `Uintah::PatchSubset` on which these tasks execute must be identified during the Uintah task-scheduling phase. Listing 7.1 illustrates how a radiometer task is created and scheduled within Uintah. It may be convenient to use (2.6) to accomplish this when using algorithms, such as transport sweeps.

```
1
2  //---------------------------------------------------------------------
3  // Method: Schedule the virtual radiometer.  This task has both
4  // temporal and spatial scheduling.
5  //---------------------------------------------------------------------
6
7  void
8  Radiometer::sched_radiometer( const LevelP         & level
9                              ,       SchedulerP     & sched
10                             ,       Task::WhichDW    notUsed
11                             ,       Task::WhichDW    sigma_dw
12                             ,       Task::WhichDW  celltype_dw ) {
13
14   // only schedule on the patches that contain radiometers - Spatial task scheduling
15   //   we want a PatchSet like: { {19}, {22}, {25} } (singleton subsets like level->eachPatch())
16   //     NOT -> { {19,22,25} }, as one proc isn't guaranteed to own the entire, 3-element subset.
17   PatchSet* radiometerPatchSet = scinew PatchSet();
18   radiometerPatchSet->addReference();
19   getPatchSet(sched, level, radiometerPatchSet);
20
21   int L = level->getIndex();
22   Task::WhichDW abskg_dw = get_abskg_whichDW( L, d_abskgLabel );
23
24   std::string taskname = "Radiometer::radiometer";
25   Task *tsk;
26   if (RMCRTCommon::d_FLT_DBL == TypeDescription::double_type) {
27     tsk = scinew Task(taskname, this, &Radiometer::radiometer<double>, abskg_dw, sigma_dw, celltype_dw);
28   }
29   else {
30     tsk = scinew Task(taskname, this, &Radiometer::radiometer<float>, abskg_dw, sigma_dw, celltype_dw);
31   }
32
33   tsk->setType(Task::Spatial);
34
35   Ghost::GhostType gac = Ghost::AroundCells;
36   tsk->requires(abskg_dw,     d_abskgLabel,     gac, SHRT_MAX);
37   tsk->requires(sigma_dw,     d_sigmaT4Label,   gac, SHRT_MAX);
38   tsk->requires(celltype_dw, d_cellTypeLabel, gac, SHRT_MAX);
39   tsk->modifies(d_VRFluxLabel);
40
41   sched->addTask(tsk, radiometerPatchSet, d_matlSet, RMCRTCommon::TG_RMCRT);
42
43   if (radiometerPatchSet && radiometerPatchSet->removeReference()) {
44     delete radiometerPatchSet;
45   }
46 }
47
48
49 //---------------------------------------------------------------------
50 //  Return the patchSet that contains radiometers
51 //---------------------------------------------------------------------
52
53 void
54 Radiometer::getPatchSet(       SchedulerP & sched
55                         , const LevelP     & level
56                         ,       PatchSet   * ps ) {
57
58   // find patches that contain radiometers
59   std::vector<const Patch*> radiometer_patches;
60   LoadBalancer * lb = sched->getLoadBalancer();
61   const PatchSet * procPatches = lb->getPerProcessorPatchSet(level);
62
63   for (int m = 0; m < procPatches->size(); m++) {
64     const PatchSubset* patches = procPatches->getSubset(m);
65
66     for (int p = 0; p < patches->size(); p++) {
67       const Patch* patch = patches->get(p);
68       IntVector lo = patch->getCellLowIndex();
69       IntVector hi = patch->getCellHighIndex();
70       IntVector VR_posLo = level->getCellIndex(d_VRLocationsMin);
71       IntVector VR_posHi = level->getCellIndex(d_VRLocationsMax);
72
73       if (doesIntersect(VR_posLo, VR_posHi, lo, hi)) {
74         radiometer_patches.push_back(patch);
75       }
76     }
77   }
78   size_t num_patches = radiometer_patches.size();
79   for (size_t i = 0; i < num_patches; ++i) {
80     ps->add(radiometer_patches[i]);
81   }
82 }
```

Listing 7.1. Scheduling the radiometer task using temporal and spatial scheduling.

Iterating over the phase P and two patch indices y_i and z_i allows us to collect the relevant patches to a sweeping phase P, which results in P_{max} patch subsets per octant. Algorithm 3 details this process within Uintah. These patch subsets can be reused for each intensity solve. The patch subsets are used in the Uintah task-requires call to the infrastructure that manages ghost cells, which greatly reduces communication costs. The sweep is propagated across the domain by having one independent task per stage. To propagate information from patch to patch within Uintah, a *requires-modifies* dependency chain is created, where the requires is conditioned on a patch subset relevant only to the patches on which the sweep is occurring. Algorithm 4 details this dependency chaining process. The patch subset is defined as all patches with the same phase number P, as shown in Figure 2.2.

7.6 Transport Sweeps

Table 7.2 illustrates the performance and weak scaling within Uintah of the sweeping method for radiation transport (Section 2.4 and Section 7.5) on a benchmark radiation problem, run on Mira up to 128K CPU cores. This method is experimental, and although DOM (Section 2.2) was used for the radiation solve in the full-scale Mira boiler simulations, the sweeping method introduced in this work shows great promise for radiation calculations within future boiler simulations.

For the CCMSC target boiler problems in question, the use of sweeps reduces radiation solve times by a factor of 10 relative to RMCRT and linear solve methods [10]. Adams et al. [117] also conclude that sweeps can be executed efficiently at high core counts, provided an optimal scheduling algorithm that executes simultaneous multioctant sweeps with minimal idle time is used. Within the target boiler (CFD) simulation, the previous solve can also be used as an initial guess, meaning the CFD can pollute only the initial guess so much, thus accelerating convergence and making it possible for DOM to use as few as 30-40 iterations as compared to the much larger number of backsolves shown in Table 7.2.

In contrast to a static problem, no initial guess is available, and significantly more iterations are used with DOM than is the case in a full boiler simulation (as shown in Table 7.2), in which as many as 1500 iterations are used. However, for this problem, each

Table 7.2. Weak scaling results on the model radiation problem.

Cores	16	256	4056	65536	131072	262144
Time DO	91	189	399	959	1200	1462
DO iter.	90	180	400	900	1300	1500
Sweeps Time	1.9	3.4	4.4	9.09	13.9	-

DOM iteration takes about a second. Hence, the best that DOM could achieve would be about 40 seconds, even if a good initial guess is available. In this way, sweeps outperforms both the actual observed cost and the optimistic estimated cost of DOM, with its linear solve using 40 iterations by a factor of between 4 and 10. However, the sweeping algorithm has not currently scaled beyond 128K cores due to its large memory footprint, and, additionally, it can be slower than the linear solver for systems with very high attenuation. This slowdown is because the sparse linear solvers are iterative, but they converge quickly for systems with large attenuation, as the impact of radiation can be isolated to a subset of the domain for these systems.

For systems with scattering, DOM typically lags the scattering term and then resolves until the intensities converge to within a certain tolerance. The convergence can be costly for systems where the scattering coefficients are significantly larger than the absorption coefficients. Given these very encouraging results, applying sweeps to the full problem is clearly the next step.

7.7 Simulation Results

Production run simulations were run using 7,467 nodes (119,472 cores) even though an equivalent computation was performed on 260,712 cores on Mira for other cases. The scalability of the RMCRT algorithm to 16K GPU nodes was shown even though Titan was experiencing major and sustained hardware failures in 2016 [54]. The node count was reduced to approximately half of Titan to mitigate the machine instabilities encountered for several pre-production run test cases at 16K Titan nodes. An approximately 2X speed-up was seen when using the GPUs versus the CPUs for the RMCRT radiation calculation. The production-run calculation used a patch size of 12^3, which meant that the GPUs were underutilized in comparison to the earlier benchmark problem that used significantly

larger patch sizes, thereby dramatically increasing the GPU workload [54]. The trade-off between patch sizes that utilized the processing power of the GPU versus a reasonable patch size that reduced our time per timestep for nonradiation timesteps is still an open question.

The incorporation of the RMCRT radiation model into the very complicated full coal boiler simulation with the inherent challenges of combining a code base that ran on both CPUs and GPUs proved to be challenging. The full coal boiler specification was the most complicated geometry that the Uintah Framework has ever attempted to simulate. The initial testing of the RMCRT algorithm with scalability results out to 16,384 GPU nodes was performed on a much simpler configuration. The full coal boiler geometry added a degree of complexity that required overcoming numerous technical hurdles that resulted in an overhaul of the Uintah infrastructure described above. Foremost was the reduction in the memory footprint and the time per timestep that allowed us to run the very complicated geometry specification of the full coal boiler problem.

Figure 7.5 shows a comparison of the instantaneous divergence of the heat flux computed using the CPU (left) and GPU (right) implementations of multilevel RMCRT. The divergence of the heat flux is one of the primary quantities of interest in our boiler simulations. Each view shows three slices through the interior of the computational domain. These simulations were run on 16K Titan cores using 1,024 GPUs, and the qualitative agreement is excellent. It should be emphasized that this is the first large-scale simulation on a commercial boiler design that has utilized GPUs and the RMCRT algorithm to solve the radiative heat transfer equation (RTE).

An investigation was undertaken to investigate the number of rays needed to accurately model the RTE, since the computational expense is strongly correlated with the number of rays per cell. Figure 7.6 shows the instantaneous divergence of the heat flux, at 10 physical seconds using 300 rays (left image) and 75 (right image) rays per cell. The qualitative agreement is excellent. The solution time decreases from 8.5 sec to 3.5 sec using 75 rays per cell. Figure 7.7 shows the instantaneous divergence of the heat flux computed using CPU-RMCRT on 120k Titan cores. This is the first production simulation of a coal fired boiler utilizing RMCRT to solve the RTE at these scales in parallel, and shows that RMCRT is a viable radiation approach for the future, unlike discrete ordinates.

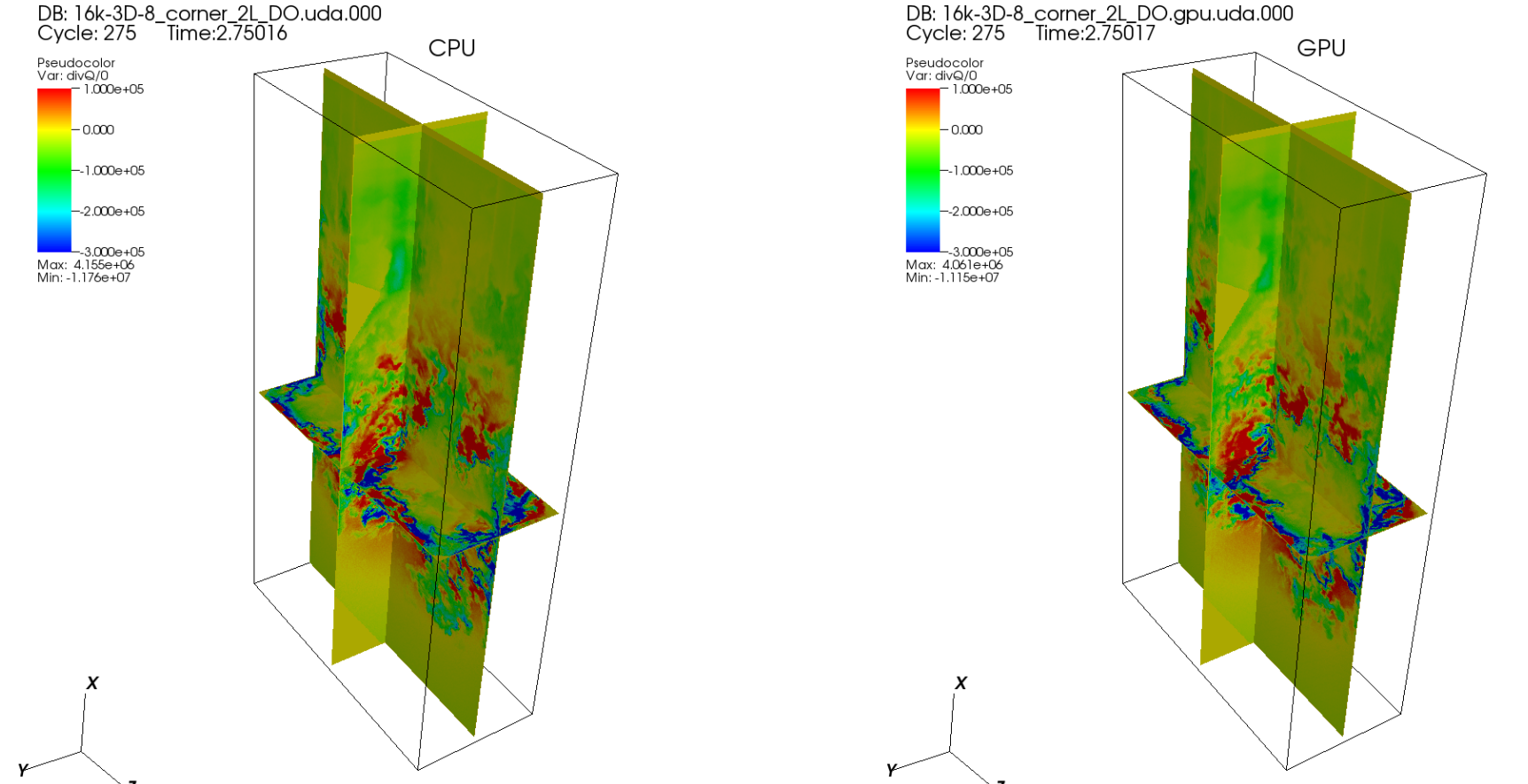

Figure 7.5. Comparison of the instantaneous divergence of the heat flux computed using the CPU and GPU for multilevel RMCRT.

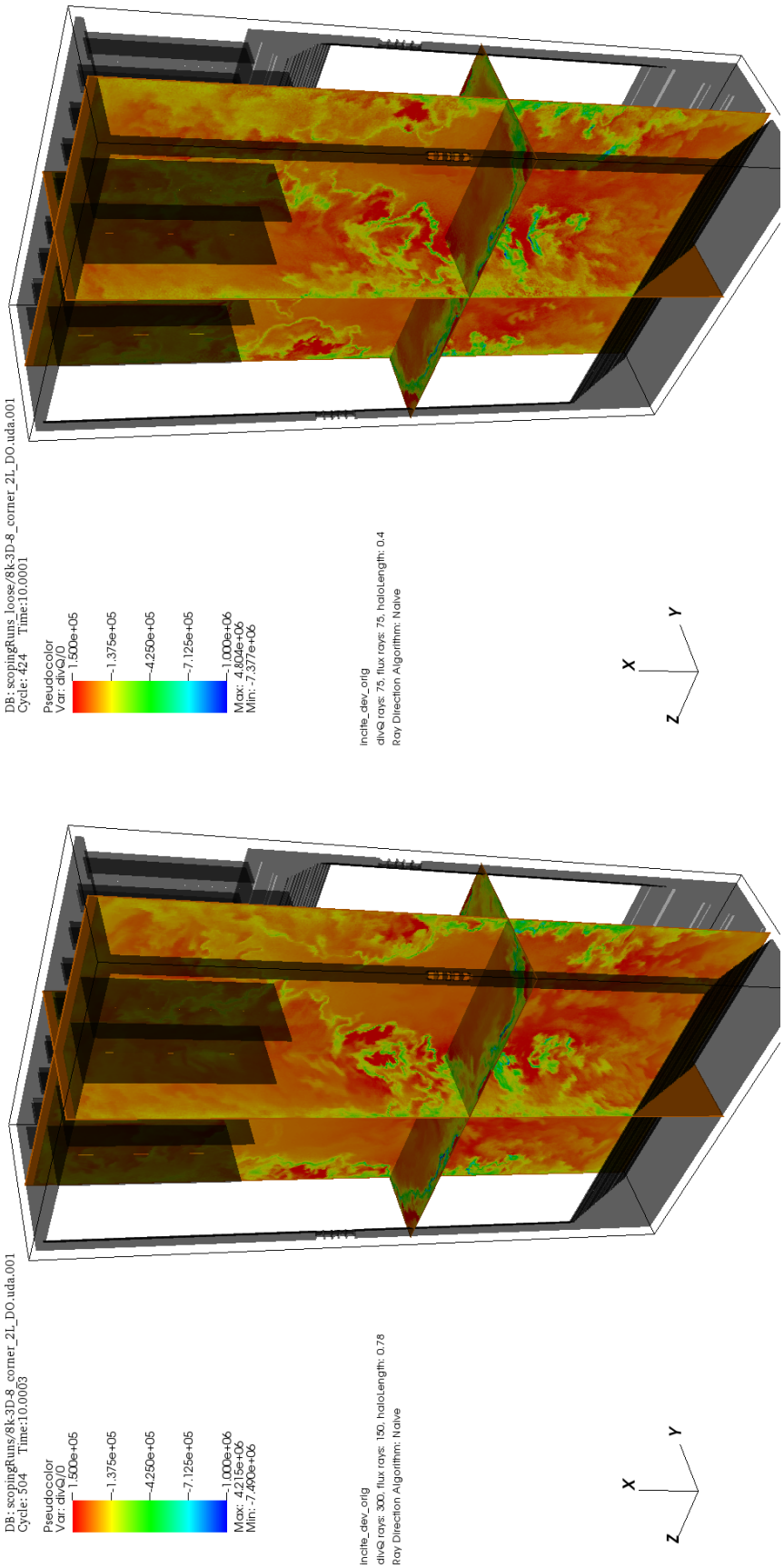

Figure 7.6. Comparison of the instantaneous divergence of the heat flux computed using 75 rays and 300 rays per cell.

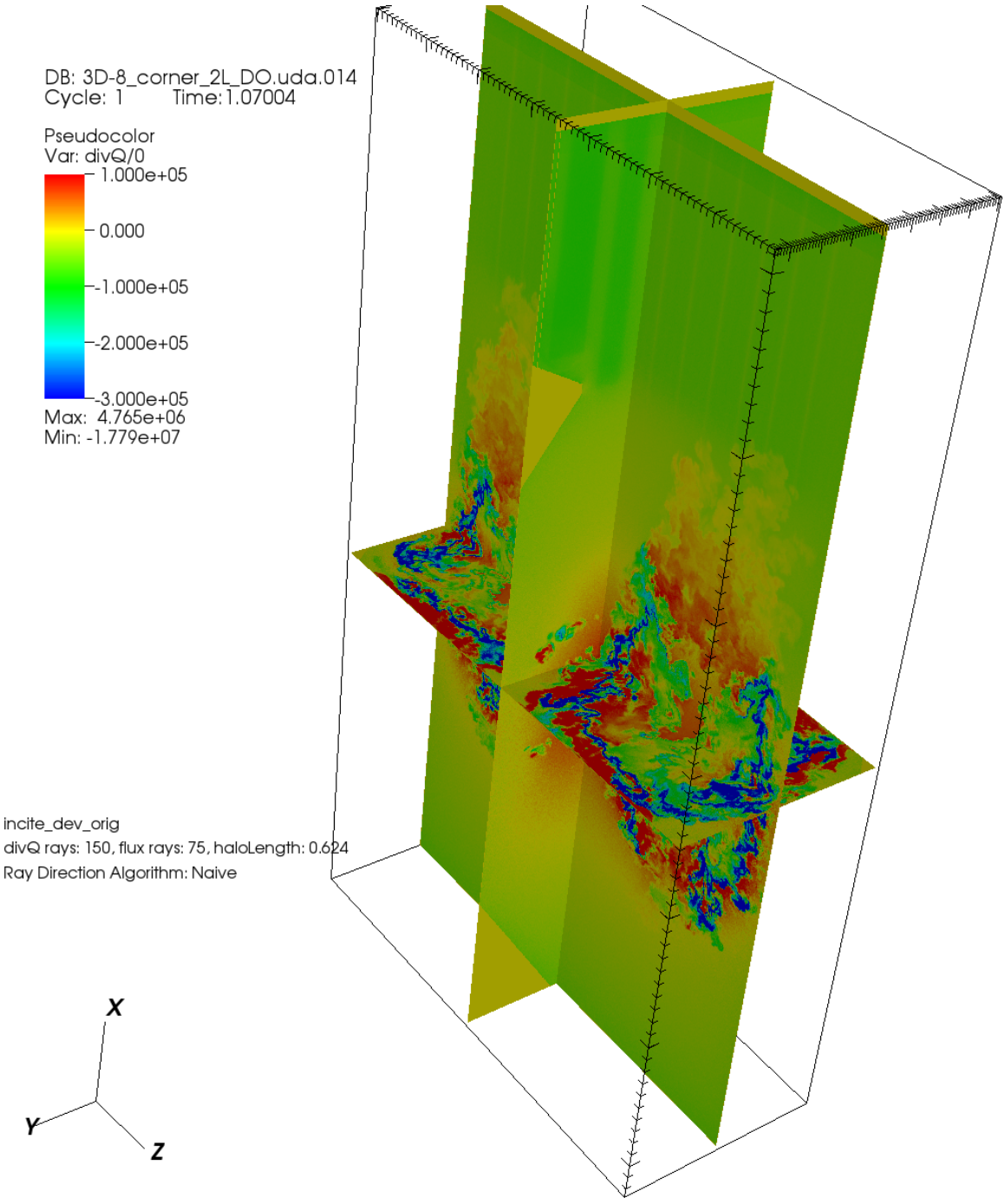

Figure 7.7. The instantaneous divergence of the heat flux computed using CPU-RMCRT on 120k Titan cores.

7.8 Scaling of Full Boiler Simulation

After addressing the scaling challenges arising from the most complicated geometry that the Uintah Framework has ever attempted to simulate, and the accompanying infrastructure changes, excellent strong scalability is achieved for the full production boiler case when using RMCRT for the radiation calculation. Table 7.3 shows the mean time per timestep (averaged over 10 timesteps involving a radiation solve) from 16,384 to 262,144 CPU cores (1k to 16K GPUs, respectively). The super linear scaling is achieved as a result of the GPU kernels being executed at maximum efficiency. These results show strong scaling out to near the full extent of Titan for a full production boiler case with Arches and RMCRT, and they are the first such results of such a complex geometry coupled to a full combustion model. Perhaps the most significant gain from a performance standpoint is that the results of the approaches outlined in this section, optimizations, and design changes have sped up the radiation solve by two orders of magnitude when using sweeps at 262K CPU cores.

Table 7.3. Strong scaling results: full boiler with GPU-based RMCRT implementation.

Cores/GPUs	16k/1k	32k/2k	64k/4k	128k/8k	256k/16k
Time (sec)	821.13	407.31	202.69	99.39	55.06

CHAPTER 8

ADDRESSING FINAL TASK GRAPH COMPLEXITY

This chapter focuses on the remaining complexity involved with Uintah's task graph compilation process, which was introduced in Section 7.3. This process is required for efficient, out-of-order (with respect to a topological sorting of tasks) execution of computational tasks [50]. Large-scale parallel applications with complex global data dependencies beyond those of reductions pose significant scalability challenges to automated dependency analysis, specifically the internodal challenges identifying the all-to-all communication of data dependencies among compute nodes. For Uintah, the predominant scalability concern is the sheer size and complexity of the resulting task graphs at production scale when running problems with global dependencies such as radiative heat transfer, an important physical process and a key mechanism in a class of challenging engineering and research problems. The principal aim of this chapter is to demonstrate how the final complexity term of the task graph dependency analysis phase can be significantly reduced when considering global data dependencies at large scale, e.g., 128,000 CPU cores and beyond. Prior to this work, this fourth, and dominant, complexity term is

$$\mathcal{O}(n_f \cdot t_{fg}), \tag{8.1}$$

where n_f is the number of fine-level patches and t_{fg} is the number of fine-level tasks with a global neighborhood. These tasks compute on the fine level and require a global coarse mesh.

8.1 Existing Task Graph Complexity

A strategy used by other AMT runtimes that have an explicit representation of the task graph is to execute the task graph as it is being constructed.

Within Uintah, however, this is done in two distinct phases, compilation and execution.

Uintah's use of a static task graph is largely related to its automated MPI message generation (a hallmark of Uintah), for which dependency analysis must be completed prior to the execution phase under the current design. For standard stencil calculations, where each compute node needs to search only surrounding nodes containing neighboring simulation data, dependency analysis typically completes in milliseconds, and only a few seconds at scale. In the past, Uintah has relied on amortizing the small cost of the compilation phase (typically acceptable in applications that do only local communication) over a significant number of iterations. The complexity of generating Uintah's distributed task graph for applications doing only local communication is shown to be $\mathcal{O}(\frac{n}{p} \cdot log(\frac{n^2}{p}))$, where n is the number of patches and p is the number of processes (MPI ranks) [31]. Each rank will then have $\frac{n}{p}$ local patches. In the context of the CCMSC target boiler runs when employing multilevel RMCRT with global dependencies at large scale, this small cost is no longer the case, as mentioned in Chapter 7.

8.1.1 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 is 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. MPI messages are created after all the identified dependencies are placed into a task graph [31]. The task graph execution phase is then ready to commence.

8.1.2 Initial Complexity Reductions

Task graph performance issues initially began manifesting with the multilevel RMCRT benchmark [1] at scales of over 32,000 cores when doing initial computational experiments in [53]. These issues necessitated preliminary algorithmic improvements to the task graph compilation algorithm. The original complexity of this operation was $\mathcal{O}(n_1 \cdot log(n_1) +$

$n_2 \cdot log(n_2))$, and after optimization became $\mathcal{O}(n_1 \cdot log(n_1)) + \mathcal{O}\left(\frac{n_2}{p} \cdot log(n_2)\right)$, where n_1 is the number of patches on coarse level, n_2 is the number of patches on fine level, and p is the number of processes (MPI ranks). In the context of the full-scale CCMSC production boiler runs, a reemergence of task graph compilation issues was observed because of the complexity of the problem, mixture of local and global task dependencies, and the sheer volume of tasks per node. This observation required a more detailed look at the complexity of these operations.

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 1,000 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 (three dimensions - x,y,z), and the total number of task objects is manageable. However, when considering global dependencies, the dependency analysis becomes $\mathcal{O}((ng)^2)$, a search between every task/patch tuple, ng in the computational domain.

In the CCMSC production boiler simulation, the two mesh refinement levels used by RMCRT (Section 2.3) to achieve scalability [53] further increased the number of global dependencies due to the interlevel dependencies using the same global neighborhood for every task on every mesh refinement level. The number of task objects T is then given by

$$T = \sum_{l=1}^{l_{tot}} \sum_{t=1}^{t_{tot}} \sum_{n=1}^{n_{tot}} P_{l,t,n}, \tag{8.2}$$

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.** The generalized complexity of the task object count, T, shown by (8.2) in the CCMSC production boiler case prior to our improvements was

$$\mathcal{O}(n_f \cdot g_f) + \mathcal{O}(n_c \cdot g_c), \tag{8.3}$$

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. The work covered in Section 7.2.4, to split the previous single-processor neighborhood into global and local processor neighborhoods [55], reduced the total number of task objects by 81x and reduced the complexity in (8.3) to

$$\mathcal{O}(\frac{n_f}{p} \cdot t_{fl}) + \mathcal{O}(\frac{n_c}{p} \cdot t_{cl}) + \mathcal{O}(n_c \cdot t_{cg}) + \mathcal{O}(n_f \cdot t_{fg}), \tag{8.4}$$

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 that globally distribute coarse-level simulation variables among nodes); and t_{fg} is the number of fine-level tasks with a global neighborhood (tasks that compute on the fine-level and require a global coarse mesh).

In the full CCMSC production boiler simulation at 119K cores, the task graph processing was reduced 93% from 4.5 hours to roughly 20 minutes [55]. This remaining time was deemed acceptable at the time because 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 were, at the time, thought to be largely due to one global radiation production task associated with the dominant fourth term of (8.4), $\mathcal{O}(n_f \cdot t_{fg})$. This particular task computes on every fine-level patch and also requires a global coarse mesh. Therefore, for every fine-level patch there exist data dependencies with every coarse-level patch. This chapter focuses on reducing the complexity of the fourth term of (8.4), $\mathcal{O}(n_f \cdot t_{fg})$.

8.2 Algorithm Analysis, Computational Experiments, and Results

A significant challenge in analyzing, identifying, and ultimately addressing issues with the dominant fourth term of (8.4), $\mathcal{O}(n_f \cdot t_{fg})$, is the sheer scale at which the problem begins to manifest. In other words, at scales tractable for most performance analysis tools and debuggers, the time complexity of task graph compilation in the context of global dependencies is not apparent or producible in any way. For the RMCRT radiation benchmark [1], we begin to see the growth in time only at around 32k-64k cores,

assuming a domain decomposition resulting in two Uintah mesh patches being assigned to each physical core on a distributed system. For many of the available debuggers and performance tools on the larger DOE systems, these core counts are challenging if not impossible to manage. To further exacerbate the problem, Uintah has moved to a nodal shared memory model [52], where the traditional approach of using one MPI rank per physical core has become one MPI rank per compute node (or NUMA node within a compute node) [33]. This approach is also crucial in eliminating redundant halo and global metadata information, significantly decreasing the nodal memory footprint. Each MPI process then spawns multiple Pthreads (`std::thread`) that are pinned to physical cores and execute tasks using shared memory data structures.

Supporting the level of asynchrony and latency hiding necessary for Uintah's AMT runtime to scale to large core counts requires the use of MPI_THREAD_MULTIPLE, where any arbitrary thread can make both point-to-point and collective MPI calls. Most available performance profiling tools, e.g., Allinea (now part of ARM) MAP, CrayPat, HPCToolkit, etc., simply do not support MPI_THREAD_MULTIPLE or provide only limited support. Other tools, such as TAU, have a build system that becomes a complete usage barrier for a system like Uintah, which is 1-million+ lines of heavily templated C++11 code, that uses many modern language constructs. Open|Speedshop (a tool developed with DOE–NNSA funding) was the only tool that enabled collection of useful information at large scale.

Ultimately, after weeks attempting to get more technologically advanced tools working for the CCMSC boiler problem without success, this study was forced to to use more primitive *printf-based* methods to collect information to be parsed and reasoned about. What follows is a detailed account of this process on three machines currently ranked 7th, 17th, and 33rd on the *TOP500* list as of June 2018.

8.2.1 Scaling Experiments

The experiments, profiling, and testing for this work were conducted across a range of DOE Office of Science and DOE National Nuclear Security Administration (NNSA) systems, each offering large core counts necessary for testing at scale (e.g., $>$ 100,000 cores).

- **OLCF Titan** – a Cray KX7 system located at Oak Ridge National Laboratory, where each node hosts a 16-core AMD Opteron 6274 processor running at 2.2 GHz, 32 GB

DDR3 memory and 1 NVIDIA Tesla K20x GPU with 6 GB GDDR5 ECC memory. The entire machine offers 299,008 CPU cores and 18,688 GPUs (1 per node) and over 710 TB of RAM. Titan uses a Cray Gemini 3D Torus network, 1.4 μs latency, 20 GB/s peak injection bandwidth, and 52 GB/s peak memory bandwidth per node.

- **ALCF Mira** – a IBM Blue Gene/Q system located at Argonne National Laboratory that enables high-performance computing with low power consumption. The Mira system has 49,152 nodes, each having 16 1600 MHz PowerPC A2 cores per node, providing a total of 786,432 CPU cores. Each node has 16 GB of RAM, and the network topology is an integrated 5D torus with hardware assistance for collective and barrier functions and 2GB/sec bandwidth on all 10 links per node. The latency of the network varies between 80 ns and 3 ms at the farthest edges of the system. The interprocessor bandwidth per flop is close to 0.2, which is higher than many existing machines. There are two I/O nodes for every 128 compute nodes, with one 2 GB/s bandwidth link per I/O node. Mira uses the GPFS file system. Ranks are assigned with locality guarantees on the machine.
- **LLNL Vulcan** – an unclassified IBM Blue Gene/Q system located at Lawrence Livermore National Laboratory. Vulcan has 24,576 nodes, each having 16 1600 MHz PowerPC A2 cores per node, providing a total of 393,216 CPU cores. Each node has 16 GB of RAM, and the network topology is an integrated 5D torus with hardware assistance for collective and barrier functions and 2GB/sec bandwidth on all 10 links per node. This system is nearly architecturally identical to the Mira system.

8.2.2 Inefficiencies in Processor Neighborhood Creation

As mentioned in Section 7.2.2, the first step in the Uintah task graph dependency analysis process is for a set of nodes to be 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.

Based on data collected at small scale (1024 nodes – 16,384 CPU cores) on the OLCF Titan system, the focus in this section is on the member method of the `Uintah::LoadBalancer` class, `LoadBalancerCommon::create_neighborhood()`. The

input parameters are an old and new `Uintah::Grid` object (for AMR regridding), and two outputs, `m_neighbor_ranks` and `m_neighbor_patches`, which are of type `std::set`, used for eliminating duplicates and return elements in order. These sets contain all the neighboring processor ranks and all the neighboring patch pointers, `m_neighbor_ranks` and `m_neighbor_patches`, respectively, and are vital to automated MPI message and tag generation within Uintah. The `std::set` is implemented as a red-black tree, which has an insertion and find complexity of $\mathcal{O}(log(n))$, where n is the number of elements in the `std::set`.

`LoadBalancerCommon::create_neighborhood()` runs for each MPI rank and iterates through all patches in the computational grid to identify all patches that a particular rank owns. For each local patch, a helper function in the Level class, `level::select_patches()` is called to query within a given index range to find all neighboring patches. This query is done through Uintah's bounding volume hierarchy (BVH) patch tree (Section 8.2.3). These "neighbors" are inserted into the `std::set`, `m_neighbor_patches`, and neighboring process IDs (MPI ranks) are inserted into `m_neighbor_ranks`. `m_neighbor_patches` is used to query whether a patch is in the processor neighborhood of the current MPI rank, and `m_neighbor_ranks` is used to query whether a particular MPI rank is within a neighborhood. Looking at 8.4, $\mathcal{O}(n_f \cdot t_{fg})$, it is clear that when considering simulations with millions of patches in the computational domain across multiple AMR levels, these operations become expensive. From the information collected at small scale (1024 nodes – 16,384 cores) on OLCF Titan, profiling tools showed that more than 50% of the overall time *at small scale* during task graph compilation is spent on insertions into and finds within `m_neighbor_ranks` and `m_neighbor_patches`.

After significant investigation of the surrounding and related code, it became clear sorting, in this case, is not actually required for the either neighboring ranks or patches to be sorted. The design decision to use `std::set` was likely made long before the C++11 standard, when set properties were needed and the complexity arising from running problems 1,000 times larger was not a consideration. Much of the original Uintah code dates back to the late 1990s and early 2000s. As of C++11, the `std::unordered_set` has been available, part of the standard library, which offers average $\mathcal{O}(1)$ insertion and find time complexity. With significant refactoring and testing at scale on all three machines used in

this study (Titan, Mira, and Vulcan), `m_neighbor_ranks` and `m_neighbor_patches` are now of type `std::unordered_set`, and the complexity involved with insertion and finds for these vital data structures has been reduced to less than 1%, even at 128,000 CPU cores. As a result of this work, these optimizations are now a part of production Uintah code.

8.2.3 Bounding Volume Hierarchy (BVH) Tree

The current data structure used to store all patches in Uintah's computational grid is implemented as a bounding volume hierarchy (BVH) tree structure. This BVH tree is much like a k-dimensional (KD) tree, a data structure used for organizing some number of points in a space with k dimensions, effectively a binary search tree where data in each node are a k-dimensional point in space. In other words, it is a space partitioning data structure useful for range and nearest neighbor searches. For quick patch queries within a given index range (used in processor neighborhood construction, among other Uintah infrastructure tasks), patches are stored in a tree structure where nodes in each level are divided in the maximum range dimension. To facilitate efficient searches, patches are sorted on each grid level with complexity $\mathcal{O}(n \cdot log(n))$ and are equally divided into two sets. With $log(n)$ levels in total, the complexity of the BVH tree constructor becomes $\mathcal{O}(n \cdot log(n)^2)$. Each patch set query has a time complexity of $\mathcal{O}(k \cdot log(n)))$, where k represents the number of patches returned. Although this operation was investigated in depth as a seemingly obvious consumer of CPU cycles, the time consumed in general for patch queries was surprisingly insignificant. This time was made negligible by caching of patch sets when repeated searches for large patch sets were made.

8.3 Reducing the Complexity of $\mathcal{O}(n_f \cdot t_{fg})$

The remaining 20 minutes of task graph compilation time (Section 8.1.2) in the CCMSC production boiler run was deemed acceptable at the time. However, this cost is untenable for current, shorter running simulations and would certainly grow to be intractable with an order of magnitude increase in problem size. These larger target problems are currently planned by the Utah CCMSC for pre-exascale and exascale systems. Addressing the fourth term in 8.4, $\mathcal{O}(n_f \cdot t_{fg})$, then becomes mandatory to ensure the scalability of the Uintah task graph compilation process when considering global data dependencies. From here

forward, this fourth term, $\mathcal{O}(n_f \cdot t_{fg})$ will be referred to as the *"global complexity term,"* the reduction of which is the principal aim of this work.

The first objective is then to test the hypothesis that the true complexity of the *global complexity term* is in fact $\mathcal{O}(n_f \cdot t_{fg})$, where n_f is the total number of fine-level patches and t_{fg} is the number of fine level tasks with a global neighborhood (tasks that compute on the fine level and require a global coarse mesh). The typical technique would normally be to run the problem through a performance profiler, identifying hot spots, addressing these hot spot, and re-profiling. This process would be repeated until such time as the application performance met certain criteria. As mentioned in Section 8.2, however, the growth in task graph compilation time due to the *global complexity term* was reproducible only at large scale, e.g., 32k-64k cores (with one mesh patch per core). Few tools can run at these scales, and of those few, even fewer support the multithreaded MPI techniques employed by Uintah (level-3 thread support, MPI_THREAD_MULTIPLE). It is important to note that a single 64k CPU core job, for example, (relatively small) may take days to run and return results after it has been queued. Queue times on the larger DOE systems can be days, and in the event of an error or a profiling tool malfunction, the job queueing and waiting process is repeated. These wait times can be even longer for jobs requiring $>$ 100k cores. After weeks of working to get more technologically advanced tools to work for this problem, we decided to move to more primitive *printf-based* methods, and internal diagnostic instrumentation within Uintah to collect information to be parsed and reasoned about.

At times, these primitive, *printf-based* methods can generate gigabytes of text files for the core counts used, which are then parsed, sorted, etc. This information can, for example, lead to understanding deadlock (e.g., identifying missing ranks in MPI collective operations) and growth in data structure size at scale, among many other data points. Through experience with these approaches, an intuition is developed for how to localize problems within potentially massive volumes of data. For this problem, focusing on the growth in the global processor neighborhood [55] became the key insight. As the patch counts and problem size grow, the size of the global processor neighborhood (used for dependency analysis) grows correspondingly. In other words, the size of the global processor neighborhood was always equal to the total number of patches in the entire computational

domain, across all AMR levels. This discovery verifies the accuracy of the hypothesis on the complexity of the *"global complexity term,"* $\mathcal{O}(n_f \cdot t_{fg})$.

The complexity arises as hypothesized [55], from the fine-level tasks with a global neighborhood (tasks that compute on the fine-level and require a global coarse mesh). The complexity in global dependency analysis is encountered not only for the fine-level task alone, but also for each global property involved with that fine-level task. Thus, the term is most accurately described as

$$\mathcal{O}(t_{gp} \cdot n_f \cdot t_{fg}), \tag{8.5}$$

where t_{gp} is the total number of global properties, n_f is again the total number of fine-level patches, and t_{fg} is again the number of fine-level tasks with a global neighborhood.

8.3.1 A Proof of Concept Solution

Prior to moving to a nodal shared memory model, Uintah used an MPI-only approach with a single MPI rank per physical core. Typically, a problem was specified such that domain decomposition resulted in a single mesh patch being assigned per MPI rank (per core in this case). When considering a nodal shared memory model, all patches are owned by a single MPI rank and visible to all threads spawned by the main thread of execution within this rank. Thus, for a 16-core node, the number of patches per rank increases by a factor of 16 in this case (a factor of cores-per-node in general), which means, in terms of task graph dependency analysis, that most (15/16 in this particular case) dependencies are actually redundant. The solution is then rooted in analyzing dependencies on a per-node basis to automatically eliminate duplicate global dependencies, significantly simplifying dependency analysis.

The challenge becomes not only culling these duplicate dependencies, but also preserving the way Uintah's automated MPI engine packages, receives, and stores dependencies within its *DataWarehouse* [35], a distributed data store (shared among all threads in a compute node) in which all Uintah simulation data are stored based on a specific <`Variable, PatchID, MaterialID`> 3-tuple key.

This viable solution in its entirety is the topic of future work discussed in Section 8.4. This preliminary work, however, demonstrates at scale how the elimination of duplicate dependencies and analyzing dependencies on a per-node basis facilitates a dramatic re-

duction in task graph compilation time for a radiative heat transfer benchmark problem [1] using RMCRT as the radiation model and leveraging Uintah's AMR infrastructure. When complete, the fourth term in 8.4 will change to $\mathcal{O}(d_f \cdot t_{fg})$, where d_f is the number of *nodes* containing fine-level patches. This work will also provide a path forward in order to implement these changes within Uintah for production boiler simulations.

8.3.2 Testing the Proof of Concept Solution

For these computational experiments, a proof of concept Uintah implementation of the per-node dependency analysis is tested. In this implementation, the duplicate dependencies are culled prior to dependency analysis and subsequent task graph compilation. These tests are solely concerned with task graph compilations times, specifically isolating the dependency analysis phase. To this end, these tests are designed to first compile and execute the initialization task graph (Uintah's zeroeth timestep, from which tasks initialize the simulation for subsequent timesteps). After the initial timestep is run, the regular task graph is compiled, *considering global dependencies*. Postcompilation, the execution of regular tasks is short circuited and the simulation ends. This approach is perhaps the only way to test these ideas, as execution of regular tasks needing global properties would fail, as the <`Variable, PatchID, MaterialID`> 3-tuple key for 15/16 fine level patches per node are missing from the `DataWarehouse`. The implementation for a potential full solution is outlined in Section 8.4.

The target problem focuses on a two-level AMR problem based on benchmark described in [1], which exercises all the main features of the AMR support within Uintah in addition to the radiation physics required by the CCMSC production boiler problem. A fine-level halo region of four cells in each direction, x, y, z, was used. The AMR grid consists of two levels with a refinement ratio of four, the fine mesh being four times more resolved than the coarse radiation mesh. The total number of cells on the fine level was 512^3 (135,796,744 total grid cells). The total number of cells on the coarse level was 128^3 (287,496 total grid cells). Each compute core was assigned a single patch from the fine-mesh level. Details on how the marching algorithm proceeds across these AMR levels are described in [53]. Five separate configurations were run for this problem on the LLNL Vulcan system (homogeneous compute nodes with 16 physical cores each). Each problem

listed below was on the same fixed problem size described above, but run with differing fine-mesh level patch counts.

1) 8,192 cores (512 nodes) – 8,192 fine-mesh patches;
2) 16,384 cores (1k nodes) – 16,384 fine-mesh patches;
3) 32,768 cores (2k nodes) – 32,768 fine-mesh patches;
4) 65,536 cores (4k nodes) – 65,536 fine-mesh patches;
5) 131,072 cores (8k nodes) – 131,072 fine-mesh patches.

Table 8.1 shows task graph compilation times before and after elimination of duplicate global dependencies and their subsequent analysis. These results are quite promising and look to offer a scalable approach for task graphs within Uintah for problems involving global dependencies, and specifically for radiative heat transfer calculations within the Utah CCMSC target boiler problem on current and emerging architectures. Demonstrating these significant reductions in compile times at scale warrants pursuing the full implementation, which is outlined in Section 8.4. The speed-ups shown in Table 8.1 are nearly uniform by a factor of *cores-per-node* as mentioned in Section 8.3.1, ˜15 in this case, confirming the change in the global complexity term from $\mathcal{O}(n_f \cdot t_{fg})$ to $\mathcal{O}(d_f \cdot t_{fg})$ as predicted in Section 8.3.1.

8.4 Conclusions and Future Directions

Perhaps the most significant result to come from this work has been to demonstrate, at scale, that the complexity of Uintah's task graph compilation phase can be significantly reduced in the context of globally coupled problems such as radiative heat transfer. This result is critical to the mission of the Utah CCMSC as it seeks to use radiation models like RMCRT in its target boiler simulations, not only on current systems, but eventually

Table 8.1. Task graph compile times before and after optimizations for the Burns and Christon Benchmark [1] problem. Results obtained on the LLNL Vulcan system.

#Cores	8,192	16,384	32,768	65,536	131,072
Time (sec) before	101.76	194.56	394.09	782.58	1559.11
Time (sec) after	6.86	13.17	26.23	52.71	104.64
Speedup (X)	14.83	14.77	15.02	14.85	14.90

at exascale. This work shows this complexity reduction to be possible, and the following sections outline the key elements needed for an implementation of these ideas to work for production code. Any changes would build upon the initial work done in [55], retaining the use of multiple processor neighborhoods, which is the foundation for greatly reducing the number of dependencies stored and analyzed by the runtime. Auxiliary insights from the in-depth investigations into Uintah's core infrastructure to understand the problems related to global task graph dependencies are also covered here.

8.4.1 Proposed Production Solution

A full implementation to cull duplicate global dependencies, although conceptually not differing from what has been implemented for this work, would require extensive interaction with the Uintah MPI engine, the mechanism that automatically packs, sends, and unpacks *"foreign variables"* (halo information from another node for a specific variable) and places this halo information into the `DataWarehouse` under a specific <`Variable, PatchID, MaterialID`> 3-tuple key. This key is then used by a task to retrieve a variable's data with requested halo for computation. Uintah's current model expects a specific `DataWarehouse` entry (3-tuple key) for each variable residing on each patch, regardless of the halo size requested. This model is a function of the original Uintah design being specific to algorithms requiring only nearest neighbor, or local communication. Because of this design, data from the coarser levels are retrieved from the Uintah `DataWarehouse` for each fine-level patch on a node. More specific information on this implementation follows in Section 8.4.2.

8.4.2 DataWarehouse and MPI Engine Modifications

The Uintah `DataWarehouse` design automatically generates MPI messages and keeps multiple versions of variables for out-of-order scheduling and execution [31], as different tasks may require the same variable on the same neighboring patch multiple times for differing ghost cell requirements. Tasks may also need input variables prior to modification. In order to support these and other scenarios, the "on-demand" DataWarehouse provides the application the illusion it has access to memory it does not actually own (via the task input specification, where the ghost cell requirement is requested).

Within Uintah's `DataWarehouse` sits the `DWDataBase` with its `KeyDatabase`, the combi-

nation of which implements thread-safe, ordered, associative arrays. The `DWDatabase` class contains methods that are templated on `DomainType`, which can be either `Patch` or `Level`. The ‘‘`Level Database`’’ has been typically used for variables not specific to a patch, such as infrastructure variables used for reductions. This ‘‘`Level Database`’’ stores a single copy of variable for a particular MPI process (node). This feature will likely need to be leveraged to store a shared copy of each global properties (per-mesh level based on Uintahs level-upon-level approach to AMR).

The Uintah MPI engine will then have to be modified to know when it it is shipping a global property from the sending side, and from the receiving side, it will need to be modified to place the single, shared copy of global data into this ‘‘`Level Database`’’. This portion of the refactoring is nontrivial, dealing with 20K+ lines of code across dozens of source files that effect nearly everything within the Uintah infrastructure. Estimated time to perform this could potentially be several months of dedicated time, including testing, etc., at large scale. The application tasks requesting these global variables will need to be modified to search the level database for these shared copies of global variables. The application portion of the required refactoring will be far more straightforward than the internal modifications to the MPI engine and DataWarehouse.

8.4.3 Annotations for Dependency Optimizations

For a full implementation of the ideas presented here to work for a large framework like Uintah, certain application-specific hints or annotations will need to be provided for efficiency, presumably as part of the problem specification. In Uintah, this problem specification is via the XML input file, written to conform to the Uintah input file specification. These annotations will allow the Uintah infrastructure to efficiently short circuit redundant or unnecessary dependency analysis. The primary annotations that would help this effort include the following:

- Listing of global properties (fields) that need to be communicated. For example, in the case of RMCRT, currently four global radiative properties are communicated via the all-to-all communication phase. With this listing, the interlevel dependency checks on these properties could be clustered as opposed to one-by-one as the communication phase currently operates.

- Annotating pairwise interlevel dependencies. Using integer IDs for each level, dependency checks on variables between levels that have no data dependencies would be eliminated. Considering three AMR levels, interlevel dependencies could be specified as <GlobalLevelDeps> [0,1],[1,2] <GlobalLevelDeps>. This annotation would eliminate unnecessary interlevel dependency analysis up front.
- Directionality of interlevel data dependence. For example, consider a two-level case for RMCRT, which for global radiative properties imposes global dependencies from coarse to fine only. Annotations on directionality would eliminate the coarse-to-coarse, specifically intralevel dependency checks, which currently need to happen for other AMR simulations.
- A statement as to the existence of global properties for a particular simulation. This existence is ultimately determined by the Uintah infrastructure due to individual tasks that request *"global halos"* for a particular variable for computation. The Uintah infrastructure would benefit from knowing about the certainty of global dependencies a priori.

8.4.4 Uintah Global Metadata

Historically, Uintah has stored a global mapping of *"patch ⇒ owning-rank"* locally (per MPI process) to facilitate MPI message generation, enabled by the dependency analysis detailed in this work. The data structures `m_neighbor_ranks` and `m_neighbor_patches` covered in Section 8.2.2 play a majority part in this global Uintah metadata approach. It is clear that at some point, the nodal memory footprint may become an issue with this approach, specifically when orders of magnitude more patches are used than in simulations being run today. The insights provided from this work suggest a new mechanism for storing patches and their processor mappings is likely needed in the future. This mechanism could be based on a system where all patches are compressed and distributed across all processors. Each thread within Uintah's current nodal shared memory model would have only a partial view of the whole domain, and communication requests would be sent when information on other patches is needed. This communication pattern is currently what occurs during the regridding process, and so would not be a significant jump conceptually.

CHAPTER 9

SUMMARY AND CONCLUSIONS

This dissertation has introduced the challenges posed by the trend toward larger and more diverse computer architectures, with or without co-processors and GPU accelerators, and differing communications networks, when running general purpose parallel software for solving multiscale and multiphysics applications. The principal exascale-candidate application motivating this research has been based on a large-eddy simulation (LES) aimed at predicting the performance of a commercial, 1200 MWe ultra-super critical (USC) coal boiler, with radiation as the dominant mode of heat transfer. This research clearly demonstrates how scalable modeling of radiation is currently one of the most challenging problems in large-scale simulations, due to the global, all-to-all physical and resulting computational connectivity.

The principal aims of this research have been to demonstrate how the Uintah AMT runtime can be adapted, making it possible for complex multiphysics applications with radiation to scale on current petascale and emerging exascale architectures. In this dissertation, these aims have been achieved through: 1) the use of the Uintah AMT runtime system; 2) adapting and leveraging Uintah's adaptive mesh refinement (AMR) support to dramatically reduce computational analysis, communication volume, and nodal memory footprint for radiation calculations; and 3) efficiently orchestrating the all-to-all communication required of radiation through a task graph dependency analysis phase, designed to efficiently handle global dependencies.

This research has ultimately shown that it is the combination of these approaches that makes it possible to scale large industrial simulations such as the USC boiler problem with radiation. Without these approaches, combined with massive parallelism and distributed memory, the target boiler simulations would simply not be possible.

Chapter 1 introduced the importance and relevance of asynchronous many-task (AMT)

programming models and runtime systems, which are becoming more widely considered as a way to address the scalability and performance challenges the exascale machine model poses to current-generation HPC codes [16]. These challenges primarily involve increased concurrency, nodal heterogeneity, deep memory hierarchies, and variable performance due to thermal throttling, all requiring adaptivity with respect to the order of execution of computational tasks. Traditional bulk synchronous parallel (BSP) approaches tend to overspecify a computation by imposing particular parallel execution order, limiting the possibilities for exploiting additional levels of parallelism. Unlike the BSP approach, the AMT model views a program as a flow of data control and data relationships (inputs and outputs) between tasks. The runtime system then extracts the appropriate level of parallelism based on these relationships.

As discussed in Chapter 1, the principal components in the task-based parallelization process are: 1) subtask decomposition; 2) dependence analysis; and 3) task scheduling. Using explicit task dependencies, the scheduling operation can be deferred to the AMT runtime system. When every task is explicitly annotated with the set of tasks on which it depends, the order in which tasks are launched by the application does not limit the possible schedules, a key point to note. A natural way to consider an application's collection of tasks and their dependencies is a graph, with nodes in the graph representing tasks and edges between nodes representing explicit dependencies between tasks. This graph is often called a *task graph*. In this chapter, it was shown that dependency analysis is considered an important foundation for scheduling, and is a precedent relationship in which one task must be completed before another task begins to run [52]. The principal issue for this graph, outside of the scheduling and execution of the tasks it contains, is how the graph is constructed, e.g., whether the graph is constructed and analyzed at compile time, at runtime, etc., and the complexity of this operation. This research has illuminated the importance of how a graph-based approach must also scale well to very large task graphs.

In Chapter 2, a detailed summary of radiation modeling (approaches to solve the radiative transport equation (RTE) shown by (2.2)) was provided, with a survey of approaches used, both in general, and within Uintah itself. Thermal radiation in the target boiler simulations is loosely coupled to the computational fluid dynamics (CFD) due to time-scale

separation and is the rightmost source term in the conservation of energy equation, shown by (2.1). Through this process, the divergence of the heat flux for every subvolume in the domain (and radiative heat flux for surfaces, e.g., boiler walls) is computed by (2.5), as rays accumulate and attenuate intensity according to the RTE for an absorbing, emitting, and scattering medium. This dissertation has focused on the reverse Monte Carlo ray tracing (RMCRT), a photon Monte Carlo method, and the transport sweeps method, a formulation of the discrete ordinates method. In the case of RMCRT, a statistically significant number of rays (photon bundles) are traced from a computational cell to the point of extinction. This method is then able to calculate energy gains and losses for every element in the computational domain. As mentioned in Chapter 2, the process is considered "reverse" through the Helmholtz Reciprocity Principle, e.g., incoming and outgoing intensity can be considered as reversals of each other [69]. Within the Uintah RMCRT module, rays are traced backwards from the detector, thus eliminating the need to track ray bundles that never reach the detector [63]. Rather than integrating the energy lost as a ray traverses the domain, RMCRT integrates the incoming intensity absorbed at the origin, where the ray was emitted. RMCRT is more amenable to domain decomposition, and thus, so is Uintah's parallelization scheme due to the backward nature of the process [8] and the mutual exclusivity of the rays themselves.

In Chapter 3, a survey of current leading AMT runtimes and programming models, as well as a cross section of those under initial or active development, was provided. The current AMT community clearly represents a broad range of different design points and philosophies within the design space of AMT models. The AMT runtimes and models covered in Chapter 3 comprise the majority of those available today as well as those under initial development; however, not one offers support for *automated global halos*. Legion and Charm++ remain the only two models that offer the closest conceivable approach to match what has been accomplished within Uintah for global calculations. Legion and Charm++ both require the application developer to explicitly supply far more characteristics of data dependencies. A global dependency problem such as radiative heat transfer would require explicitly writing tasks for all pairs of communicating global properties across the computational domain. For the CCMSC target boiler problem, global dependencies would result in millions of hand coded tasks in all AMT runtimes surveyed, *an untenable and intractable*

solution. The Uintah AMT runtime has been the clear choice for the CCMSC target boiler simulations with radiation as the dominant heat transfer mode.

The introduction of radiation as the design driver for the Utah CCMSC was provided in Chapter 4, which also introduced the central role radiation modeling plays in the challenges inherent in the CCMSC target boiler simulation, specifically showing that significant scalability challenges inherent in the RMCRT problem exist. Although the investigation of scalability with respect to the different processors and communications performance concluded that the adaptive DAG-based approach provides a very powerful abstraction for solving challenging multiscale, multiphysics engineering problems, the key discovery related to the global dependency problem within Uintah, related to this dissertation, was Uintah's inability to complete RMCRT simulations beyond 16,384 CPU cores due to intractable task graph compilation times.

Chapter 5 presented strong and weak scaling results for the RMCRT radiation model, to 262,144 CPU cores, and demonstrated that through leveraging the multilevel AMR infrastructure provided by the Uintah framework, a scalable approach to radiative heat transfer using reverse Monte Carlo ray tracing was possible for CPU-based architectures. It is important to note that *although the AMR methods used in this work are not necessarily new, the application of these methods to radiative heat transfer algorithms and their scalability is novel.* These strong and weak scaling results provide a promising alternative to approaches to radiation modeling such as discrete ordinates. Using the cost model for communication and computation developed in [53], we can now predict how our approach to radiation modeling may scale and perform on current, emerging, and future architectures. The calculations demonstrated in this chapter were determined as ideal candidates for large-scale accelerator use, employing large numbers of rays for every cell in the computational domain, specifically, using the whole of machines such as Titan with accelerators. Adapting this work to leverage the on-node GPUs of Titan [54] was detailed in Chapter 6.

Strong scaling results for the multilevel RMCRT radiation model to 16,384 GPUs were presented in Chapter 6, which demonstrated that radiative heat transfer problems can be made to scale within Uintah on current petascale heterogeneous systems through a combination of reverse Monte Carlo ray tracing (RMCRT) techniques and AMR to reduce the amount of global communication. The results presented here offer a promising approach

to modeling radiative heat transfer within Uintah on massive heterogeneous architectures. This approach ultimately enables the Utah CCMSC to run the target 1200 MWe boiler problem on current and emerging GPU-based architectures at large scale, as shown in Chapter 7.

The work in Chapter 6 has also demonstrated *the necessity of choosing optimal data structures and algorithms to efficiently expose concurrency.* An important software design lesson is illustrated in this chapter, specifically, how maintaining critical sections around serial data structures in legacy code increases code complexity and the likelihood of the introduction of difficult race conditions and deadlock scenarios, especially when using mixed concurrency models, namely *MPI + (Pthreads + Nvidia CUDA)*. Furthermore, this chapter showed the necessity for frameworks like Uintah to better manage limited memory through the use of custom allocators that allow us to choose better allocation policies for different objects and to better utilize available resources, improving nodal throughput. The specific contribution of this portion of my dissertation is the development of a scalable radiation model for current and emerging heterogeneous architectures, made widely available through the Uintah open-source framework.

The full-scale industrial CCMSC target boiler was introduced in Chapter 7, where all research and developments from previous chapters culminate. This chapter also showed that a broad class of large-scale multiphysics applications requiring long-range interactions, such as molecular dynamics [4], cosmology [5], neutron transport [6], and radiative heat transfer [7] calculations, each 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 in terms of both computational analysis and memory storage if the amount of data to be sent is large in contrast, for example, to an MPI reduction.

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. For this industrial boiler problem, the DOE Titan and Mira systems were used to simulate coal boiler designs using different methods for computing the RTE (Equa-

tion (2.2)). On Mira, the global radiation dependencies required numerous sparse, global linear solves for the discrete ordinates method [46]. On the Titan platform, radiative heat transfer was computed using a reverse Monte Carlo Ray Tracing (RMCRT) technique [53], which requires replication of radiative properties to facilitate local ray tracing. The Titan boiler case utilized the Uintah asynchronous many-task (AMT) runtime system [35], [50], *which managed the scheduling and execution of over 8 million computational tasks on 119,000 CPU cores and 7,500 GPUs simultaneously.*

In these boiler simulations, the dominant mode of heat transfer is radiation, which presents significant challenges for AMT runtime systems [16] due to the all-to-all nature of radiation. The principal challenge related to this dissertation 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 [31]. 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 needs to search only 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 the standard computational fluid dynamics (CFD) calculation, or 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 specific contribution of this chapter is in addressing these challenges through an improved search algorithm to reduce dependency analysis processing time by avoiding unnecessary searches, combined with temporal scheduling, e.g., multiple primary task graphs. This chapter demonstrated 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 speed-ups 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.

Final task graph complexity for fully scalable task graphs with radiation within Uintah was covered in Chapter 8, with future directions provided. This chapter reminds the reader of the strategy used by other AMT runtimes that have an explicit task graph representation, which is to execute the task graph as it is being constructed. Within Uintah, however, this is done in two distinct phases, compilation and execution. Uintah's use of a static task graph is largely related to its automated MPI message generation (a hallmark of Uintah), for which dependency analysis must be completed prior to the execution phase under the current design. For standard stencil calculations, where each compute node needs to search only surrounding nodes containing neighboring simulation data, dependency analysis typically completes in milliseconds, and only a few seconds at scale. In the past Uintah has relied on amortizing the small cost of the compilation phase (typically acceptable in applications which do only local communication) over a significant number of iterations. The complexity of generating Uintah's distributed task graph for applications doing only local communication is shown to be $\mathcal{O}(\frac{n}{p} \cdot log(\frac{n^2}{p}))$, where n is the number of patches and p is the number of processes (MPI ranks) [31]. Each rank will then have $\frac{n}{p}$ local patches. This chapter provides a stark reminder that, in the context of the CCMSC target boiler runs when employing multilevel RMCRT with global dependencies at large-scale, *this is no longer the case.*

The principal contributions in this chapter were: 1) providing a research solution that demonstrated it was possible to significantly reduce the complexity of the fourth term of 8.4, $\mathcal{O}(n_f \cdot t_{fg})$ for CCMSC boiler calculations; and 2) developing the high-level design for what a more generalized solution with a subquadratic complexity in dependency analysis would be for global calculations in general.

The research solution tested in this chapter was based on analyzing dependencies on a per-node basis to automatically eliminate duplicate global dependencies, significantly simplifying dependency analysis. This preliminary work demonstrates, at scale, how the elimination of duplicate dependencies and analyzing dependencies on a per-node basis

facilitates a dramatic reduction in task graph compilation time for a radiative heat transfer benchmark problem [1] when using RMCRT as the radiation model. The challenge for a full, generalized production solution extends beyond simply culling these duplicate dependencies, and involves preserving the way Uintah's automated MPI engine packages, receives, and stores dependencies within its *DataWarehouse*, in which all Uintah simulation data are stored based on a specific <`Variable, PatchID, MaterialID`> 3-tuple key. When complete, the fourth term in (8.4) will change to $\mathcal{O}(d_f \cdot t_{fg})$, where d_f is the number of *nodes* containing fine-level patches. This chapter additionally provides a path forward in order to implement these changes within Uintah.

The broader impact of this work may ultimately include algorithmic developments for related problems with pervasive all-to-all type communications in general, such as long-range electrostatics in molecular dynamics, and will be of importance to a broad class of users, developers, scientists, and students for whom such problems are presently a bottleneck.

9.1 Conclusion and Lessons Learned

As mentioned in Chapter 1, the exponential growth in HPC over the past 20 years has fueled a wave of scientific insights and discoveries, many of which would not be possible without the integration of HPC capabilities.

Many of these discoveries not only answer scientific questions, but also inform public policy. The primary challenge in moving codes to new architectures at exascale is that although present codes may have good scaling characteristics on some present architectures, these codes may likely have components that are not suited to the extreme scale of new computer architectures, or to the complexity of real-world applications at exascale. These challenges, for example, may involve potentially billion-way concurrency, multiple levels of heterogeneity (at both hardware and software levels) with multilevel memories, and a proposed target power ceiling of 20-40 megawatts (MW) for 1 exaflop, likely leading to power capping, nonuniform node-level performance and diminishing memory bandwidth and capacity relative to FLOP count. The same bandwidth limitations also apply to the I/O system at nearly all levels. The challenge of resilience is also not well understood on architectures that are not yet defined. Nevertheless, the possibility of more frequent

faults leads to consideration of practical resilience strategies. Although existing science and engineering problems will of course be addressed at exascale, we have the opportunity to solve a new generation of ever more challenging problems. The complexity of these next-generation problems imposes challenges in that the algorithms and computational approaches used will need to be considered to achieve scalability.

AMT runtimes like Uintah are attractive at petascale and exascale as the runtime approach shelters the application developer from the underlying parallelism and complexities introduced by future architectures, *aiming to mitigate these complexities and challenges at the runtime level*. Ultimately, the AMT approach seeks to accelerate scientific insight by reducing both developer time and time to solution in simulations.

9.1.1 Reproducibility and Out-of-Order Execution

With HPC growing into such a powerful tool for scientific inquiry, computational reproducibility has more recently become a focus in scientific computing, specifically uncovering nondeterministic errors in large-scale simulations. This loss of reproducibility became a viable concern when systems combined parallelism with nondeterminism, e.g., distributed computing combined with out-of-order execution. With the levels of parallelism available in current and emerging HPC systems (GPUs, manycore, and other accelerator-based platforms), combined with the semantics of out-of-order execution inherent in AMT runtimes, ensuring computational reproducibility becomes challenging if not impossible.

For the Uintah AMT runtime system, the end goal, apart from shielding the application from the details of the underlying parallelism and architectural complexities, is to execute code efficiently and portably on a broad range of architectures. Within an AMT runtime, task execution often proceeds in an opportunistic manner (e.g., execution based on message arrival times) with the execution order changing across differing systems and networks. Due to the noncommutative nature of floating point computations, simulations can have different execution orders and could possibly run on different processing elements altogether across runs. Code may be executed efficiently by an AMT runtime, but the nature of how, when, and where code is executed can be impossible to predict on modern HPC systems. Subsequently, computational results will be by definition, nonreproducible,

due to the nondeterministic computational execution of tasks. For Uintah, it then becomes imperative to maintain a way to allow for fully deterministic execution.

With this in mind, a result of this work has been to explore reinstrumenting the Uintah runtime with the capability to employ a fully deterministic execution strategy for any simulation. Specifically, such a strategy involves a topological sort of tasks, combined with a guaranteed way to execute tasks in this order by the runtime, on specified processing elements, eliminating any prioritization of tasks. At a minimum, this approach will allow application developers to determine how variant the nondeterministic execution of their codes are with respect to a deterministic ordering and execution.

9.1.2 Challenges of Running at Large Scale

The scaling challenges faced in this dissertation have become apparent only by running challenging problems at very high core counts, stressing areas of Uintah infrastructure code in ways never before seen, specifically Uintah's task-graph compilation phase. A parting message in this regard is the emphasis on the need for tools that help us understand how the behavior of a code at *small scale* can provide insight as to its scaling characteristics at *large scale*. This need has prompted the development of tools that understand, in detail, the behavior of every task in a simulation executed by an AMT runtime.

The past approach has effectively been to run problems at larger and larger scales, anticipating some portion of the code to slow down considerably, and then to use lightweight, internal Uintah diagnostics, combined with large-scale debuggers, to localize the problem well enough that the offending code can be reasoned about. The reason for this approach is that Uintah has had "correct" code that has become a road block to scalability. This scalability barrier may be illustrated by considering the case of the RMCRT radiation model, for which there is potentially global connectivity. When considering RMCRT, compiling the task graph involves every compute node interacting with every other, an N_{nodes}^2 process. Similarly, the global communications required in radiation grow to flood the communications network unless mesh refinement is used to limit the amount of traffic [53]. In both cases, profiling would have predicted these difficulties.

The proposed tools would aim to accomplish two main tasks. The first is to determine the performance of each task executed by the AMT runtime. As each user task deals

with mesh patches of size n^3 mesh cells, repeat runs with different tasks will allow a least squares approach to project forward to larger values of n. At the same time, system tasks will also potentially have a reliance on both the number of mesh points in a patch, n, and p, the number of communicating MPI processes, resulting in the relationship np. The envisioned approach is then to do two sets of runs to determine both the weak and strong scaling characteristics of user, as well as system tasks, and to use this information to predict future performance not only at larger core counts but also at larger patch sizes.

APPENDIX

PUBLICATIONS

This section provides a comprehensive listing of all publications to date, with DOI. Appendix A.1 includes publications central to this dissertation, while Appendix A.2 provides additional, related publications.

A.1 Related Publications

1. ***Co-First Author*** S. Kumar, **A. Humphrey**, W. Usher, S. Petruzza, B. Peterson, J. Schmidt, D. Harris, B. Isaac, J. Thornoc, T. Harman, V. Pascussi, and M. Berzins, "Scalable Data Management of the Uintah Simulation Framework for Next-Generation Engineering Problems with Radiation," in *Supercomputing Frontiers, Springer International Publishing, pp. 219–240, 2018.* DOI: 10.1007/978-3-319-69953-0_13

2. ***Awarded Best Paper*** B. Peterson, **A. Humphrey**, J. Schmidt, and M. Berzins, "Addressing Global Data Dependencies in Heterogeneous Asynchronous Runtime Systems on GPUs," in *Third International IEEE Workshop on Extreme Scale Programming Models and Middleware, held in conjunction with SC17: The International Conference on High Performance Computing, Networking, Storage and Analysis, 2017.*
   ISBN: 978-3-319-69953-0 DOI: 10.1145/3152041.315208

3. **A. Humphrey**, D. Sunderland, T. Harman, and M. Berzins, "Radiative Heat Transfer Calculation on 16384 GPUs Using a Reverse Monte Carlo Ray Tracing Approach with Adaptive Mesh Refinement," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW) 2016, IEEE International (17th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2016)).*
   DOI: 10.1109/IPDPSW.2016.93

4. **A. Humphrey**, T. Harman, M. Berzins, and P. Smith, "A Scalable Algorithm for Radiative Heat Transfer Using Reverse Monte Carlo Ray Tracing," in *Proceedings of the International Supercomputing Conference (ISC15)*, Springer LNCS, Volume 9137, pp. 212-230, Frankfurt, Germany, 2015. DOI: 10.1007/978-3-319-20119-1_16

5. **A. Humphrey**, Q. Meng, M. Berzins, D. Caminha B De Oliveira, Z. Rakamaric, and G. Gopalakrishnan, "Systematic Debugging Methods for Large Scale HPC Computational Frameworks," in *Computing in Science & Engineering*, vol. 16, no. 3, pp. 48-56, May-June 2014. DOI: 10.1109/MCSE.2014.11

6. **A. Humphrey**, Q. Meng, M. Berzins, and T. Harman, "Radiation Modeling using the Uintah Heterogeneous CPU/GPU Runtime System," in *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond (XSEDE '12)*. ACM, New York, NY, USA, Article 4, 8 pages. DOI: 10.1145/2335755.2335791

7. Q. Meng, **A. Humphrey**, and M. Berzins, "The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System," in *Proceedings International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12) 2012 SC Companion*: pp. 2441,2448, 10-16 Nov. 2012.
DOI: 10.1109/SCC.2012.6674233

8. Q. Meng, **A. Humphrey**, J. Schmidt, and M. Berzins, "Investigating applications portability with the Uintah DAG-based Runtime System on Petascale supercomputers," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 96, 12 pages. DOI: 10.1145/2503210.2503250

A.2 Additional Related Publications

1. B. Peterson, **A. Humphrey**, D. Sunderland, J.C. Sutherland, T. Saad, H. K. Dasari, and M. Berzins, "Automatic Halo Management for the Uintah GPU-Heterogeneous Asynchronous Many-Task Runtime," in *International Journal of Parallel Programming, 2018.* DOI: 10.1007/s10766-018-0619-1

2. B. Peterson, **A. Humphrey**, D. Sunderland, T. Harman, H. C. Edwards, and M. Berzins, "Demonstrating GPU Code Portability Through Kokkos on an Asynchronous Many-Task Runtime on 16384 GPUs," in *Journal of Computational Science and Engineering, Volume 27, pp 303 – 319, 2018.* DOI: 10.1016/j.jocs.2018.06.005

3. Z. Yang, D. Sahasrabudhe, **A. Humphrey**, and M. Berzins, "A Preliminary Port and Evaluation of the Uintah AMT Runtime on Sunway TaihuLight," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW) 2018, IEEE International (19th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2018)), 2018.* DOI: 10.1109/IPDPSW.2018.00155

4. A. Sanderson, **A. Humphrey**, J. Schmidt, and R. Sisneros, "Coupling the Uintah Framework and the Visit Toolkit for Computational Steering and Parallel In Situ Data Analysis and Visualization," in *3rd International Workshop on In Situ Visualization: Introduction and Applications, ISC 2018.* DOI: 10.1109/IPDPSW.2018.00155

5. J. Holmen, **A. Humphrey**, D. Sunderland, and M. Berzins, "Improving Uintah's Scalability Through the Use of Portable Kokkos-Based Data Parallel Tasks," in *Proceedings of the Practice and Experience in Advanced Research Computing (PEARC17) 2017.* DOI: 10.1145/3093338.3093388

6. D. Sunderland, B. Peterson, J. Schmidt, **A. Humphrey**, J. Thornock, and M. Berzins, "An Overview of Performance Portability in the Uintah Runtime System Through the Use of Kokkos," in *Proceedings of the Second International Workshop on Extreme Scale Programming Models and Middleware (ESPM2).* IEEE Press, Piscataway, NJ, USA, 44-47. DOI: 10.1109/ESPM2.2016.012

7. M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, **A. Humphrey**, Q. Meng, J. Schmidt, and C. Wight, "Extending the Uintah Framework through the Petascale Modeling of Detonation in Arrays of High Explosive Devices," in *SIAM Journal on Scientific Computing* 2016 38:5, S101-S122. DOI: 10.1137/15M1023270

8. B. Peterson, H. Dasari, **A. Humphrey**, J. Sutherland, T. Saad, and M. Berzins, "Reducing Overhead in the Uintah Framework to Support Short-Lived Tasks on GPU-Heterogeneous Architectures," in *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC '15)*. ACM, New York, NY, USA, Article 4 , 8 pages.
DOI: 10.1145/2830018.2830023

9. B. Peterson, N. Xiao, J. Holmen, S. Chaganti, A. Pakki, J. Schmidt, D. Sunderland, **A. Humphrey**, and M. Berzins, "Developing Uintahs Runtime System For Forthcoming Architectures," Refereed paper presented at *the RESPA 15 Workshop at Supercomputing 2015 Austin Texas, SCI Institute, 2015.*

10. J. Bennett, R. Clay, G. Baker, M. Gamell, D. Hollman, S. Knight, H. Kolla, G. Sjaardema, N. Slattengren, K. Teranishi, J. Wilke, M. Bettencourt, S. Bova, K. Franko, P. Lin, R. Grant, S. Hammond, S. Olivier, L. Kale, N. Jain, E. Mikida, A. Aiken, M. Bauer, W. Lee, E. Slaughter, S. Treichler, M. Berzins, T. Harman, **A. Humphrey**, J. Schmidt, D. Sunderland, P. McCormick, S. Gutierrez, M. Schulz, A. Bhatele, D. Boehme, P. Bremer, and T. Gamblin, "ASC ATDM Level 2 Milestone #5325 Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms," *Sandia National Laboratories, 2015*.

11. J. K. Holmen, **A. Humphrey**, and M. Berzins, "Exploring Use of the Reserved Core," in *High Performance Parallelism Pearls*, Edited by J. Reinders and J. Jeffers, Elsevier, pp. 229-242. 2015. DOI: 10.1016/b978-0-12-803819-2.00010-0

12. Q. Meng, **A. Humphrey**, J. Schmidt, and M. Berzins, "Preliminary Experiences with the Uintah Framework on Intel Xeon Phi and Stampede," in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery (XSEDE '13)*. ACM, New York, NY, USA, Article 48, 8 pages.
DOI: 10.1145/2484762.2484779

13. D. Oliveira, Z. Rakamaric, G. Gopalakrishnan, **A. Humphrey**, Q. Meng, and M. Berzins, "Practical Formal Correctness Checking of Million-Core Problem Solving Environments for HPC," in *2013 5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE),* IEEE Press, Piscataway, NJ, USA, 75-83. DOI: 10.1109/SECSE.2013.6615102

14. M. Berzins, J. Schmidt, Q. Meng and **A. Humphrey**, "Past, Present and Future Scalability of the Uintah Software," in *Proceedings of the Extreme Scaling Workshop (BW-XSEDE '12)*, University of Illinois at Urbana-Champaign, Champaign, IL, USA, Article 6, 6 pages. ACMID: 2462083

15. J. Lv, G. Li, **A. Humphrey**, and G. Gopalakrishnan, "Performance Degradation Analysis of GPU Kernels," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011 EC2 Workshop), 2011*.

16. **A. Humphrey**, C. Derrick, G. Gopalakrishnan, and B. Tibbitts, "GEM: Graphical Explorer of MPI Programs," in *2010 39th International Conference on Parallel Processing Workshops*. DOI: 10.1109/ICPPW.2010.33 10.1145/1879211.1879248

17. A. Vo, G. Gopalakrishnan, S. Vakkalanka, **A. Humphrey**, and C. Derrick, "Seamless Integration of Two Approaches to Dynamic Formal Verification of MPI Program," in *Proceedings of Programming Languages Design and Implementation, First Workshop on Advances in Message Passing (AMP Workshop, co-located with PLDI), 2010*.

REFERENCES

[1] S. P. Burns and M. A. Christon, "Spatial domain-based parallelism in large-scale, participating-media, radiative transport applications," *Numer. Heat Trans., Part B: Fundam.*, vol. 31, no. 4, pp. 401–421, 1997.

[2] U. D. of Energy, "Exascale computing project," 2017, https://exascaleproject.org/.

[3] J. Russell, "Doug Kothe on the race to build exascale applications," 2017, https://www.hpcwire.com/2017/05/29/doug-kothe-race-build-exascale-applications/.

[4] H. Fangohr, A. R. Price, S. J. Cox, P. A. de Groot, G. J. Daniell, and K. S. Thomas, "Efficient methods for handling long-range forces in particle-particle simulations," *J. Comput. Phys.*, vol. 162, no. 2, pp. 372–384, Aug. 2000.

[5] V. Springel, "The cosmological simulation code gadget-2," *Monthly Notices Roy. Astron. Soc.*, vol. 364, no. 4, pp. 1105–1134, 2005.

[6] J. Briesmeister, "Mcnp – a general monte carlo n-particle transport code, version 4c," Los Alamos National Lab, Tech. Rep. LA-13709-M, 2000.

[7] J. P. Jessee, W. A. Fiveland, L. H. Howell, P. Colella, and R. B. Pember, "An adaptive mesh refinement algorithm for the radiative transport equation," *J. Comp. Phys.*, vol. 139, no. 2, pp. 380–398, 1998.

[8] X. Sun, "Reverse Monte Carlo ray-tracing for radiative heat transfer in combustion systems," Ph.D. dissertation, Dept. of Chem. Eng., University of Utah, Salt Lake City, UT, 2009.

[9] I. Veljkovic and P. E. Plassmann, "Scalable photon Monte Carlo algorithms and software for the solution of radiative heat transfer problems," in *Proc. 1st Int. Conf. High Performance Comput. Commun.*, ser. HPCC'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 928–937.

[10] S. Kumar, A. Humphrey, W. Usher, S. Petruzza, B. Peterson, J. A. Schmidt, D. Harris, B. Isaac, J. Thornock, T. Harman, V. Pascucci, and M. Berzins, "Scalable data management of the Uintah simulation framework for next-generation engineering problems with radiation," in *Supercomputing Frontiers*, R. Yokota and W. Wu, Eds. Springer International Publishing, 2018, pp. 219–240.

[11] J. Luitjens, "The scalability of parallel adaptive mesh refinement within Uintah," Ph.D. dissertation, School of Comput., University of Utah, Salt Lake City, UT, USA, 2011.

[12] Scientific Computing and Imaging Institute, "Uintah Web Page," 2015, http://www.uintah.utah.edu/.

[13] I. ANSYS, "Fluent Web Page," 2014, http://www.ansys.com/Products/.

[14] L. Los Alamos National Security, "Los Alamos National Laboratory Transport Packages," 2014, http://www.ccs.lanl.gov/CCS/CCS-4/codes.shtml.

[15] D. Kerbyson, "A look at application performance sensitivity to the bandwidth and latency of infiniband networks," in *Parallel and Distributed Processing Symp., 2006. IPDPS 2006. 20th Int.*, Apr. 2006, p. 7.

[16] J. Bennett, R. Clay, G. Baker, M. Gamell, D. Hollman, S. Knight, H. Kolla, G. Sjaardema, N. Slattengren, K. Teranishi, J. Wilke, M. Bettencourt, S. Bova, K. Franko, P. Lin, R. Grant, S. Hammond, S. Olivier, L. Kale, N. Jain, E. Mikida, A. Aiken, M. Bauer, W. Lee, E. Slaughter, S. Treichler, M. Berzins, T. Harman, A. Humphrey, J. Schmidt, D. Sunderland, P. McCormick, S. Gutierrez, M. Schulz, A. Bhatele, D. Boehme, P. Bremer, and T. Gamblin, "ASC ATDM level 2 milestone #5325: asynchronous many-task runtime system analysis and assessment for next generation platforms," Sandia National Laboratories, Tech. Rep., 2015.

[17] O. Sinnen, *Task Scheduling for Parallel Systems*. Hoboken, NJ, USA: John Wiley & Sons, 2007, vol. 60.

[18] I. Martin and F. Tirado, "Relationships between efficiency and execution time of full multigrid methods on parallel computers," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 6, pp. 562–573, Jun. 1997.

[19] W. Martin, A. Majumdar, J. Rathkopf, and M. Litvin, "Experiences with different parallel programming paradigms for Monte Carlo particle transport leads to a portable toolkit for parallel Monte Carlo," Apr. 1993.

[20] B. Lewis and D. J. Berg, *Multithreaded Programming with Pthreads*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[21] G. Bronevetsky and B. R. de Supinski, "Complete formal specification of the openmp memory model," *Int. J. Parallel Programming*, vol. 35, no. 4, pp. 335–392, Aug. 2007.

[22] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.

[23] M. Berzins, "Status of release of the Uintah computational framework," Scientific Computing and Imaging Institute, Tech. Rep. UUSCI-2012-001, 2012.

[24] B. Kashiwa and E. Gaffney, "Design basis for CFDLIB, tech. rep. la-ur-03-1295," 2003.

[25] J. E. Guilkey, T. B. Harman, A. Xia, B. A. Kashiwa, and P. A. McMurtry, "An Eulerian-Lagrangian approach for large deformation fluid-structure interaction problems, part 1: algorithm development," in *Fluid Structure Interaction II*. Cadiz, Spain: WIT Press, 2003.

[26] J. Spinti, J. Thornock, E. Eddings, P. Smith, and A. Sarofim, "Heat transfer to objects in pool fires," *Transport Phenomena in Fires*, vol. 20, p. 69, 2008.

[27] J. Luitjens and M. Berzins, "Improving the performance of Uintah: a large-scale adaptive meshing computational framework," in *2010 IEEE Int. Symp. Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–10.

[28] M. Berzins, Q. Meng, J. Schmidt, and J. C. Sutherland, "Dag-based software frameworks for pdes," in *European Conf. Parallel Processing*. Berlin, Heidelberg: Springer, 2011, pp. 324–333.

[29] J. Luitjens and M. Berzins, "Scalable parallel regridding algorithms for block-structured adaptive mesh refinement," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 13, pp. 1522–1537, 2011.

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

[31] Q. Meng, J. Luitjens, and M. Berzins, "Dynamic task scheduling for the Uintah framework," in *2010 3rd Workshop on Many-Task Computing on Grids and Supercomputers*. IEEE, 2010, pp. 1–10.

[32] Q. Meng, M. Berzins, and J. Schmidt, "Using hybrid parallelism to improve memory use in the Uintah framework," in *Proc. 2011 TeraGrid Conference: Extreme Digital Discovery*. ACM, 2011, p. 24.

[33] Q. Meng and M. Berzins, "Scalable large-scale fluid–structure interaction solvers in the Uintah framework via hybrid task-based parallelism algorithms," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 7, pp. 1388–1407, 2014.

[34] A. Humphrey, Q. Meng, M. Berzins, and T. Harman, "Radiation modeling using the Uintah heterogeneous cpu/gpu runtime system," in *Proc. 1st Conf. Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the Campus and Beyond*, ser. XSEDE '12. New York, NY, USA: ACM, 2012, pp. 4:1–4:8.

[35] Q. Meng, A. Humphrey, and M. Berzins, "The Uintah framework: a unified heterogeneous task scheduling and runtime system," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov. 2012, pp. 2441–2448.

[36] J. K. Holmen, A. Humphrey, D. Sutherland, and M. Berzins, "Improving Uintah's scalability through the use of portable kokkos-based data parallel tasks," in *Proc. Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, ser. PEARC17, no. 27, 2017, pp. 27:1–27:8.

[37] B. Peterson, A. Humphrey, J. H. T. Harman, M. Berzins, D. Sunderland, and H. Edwards, "Demonstrating GPU code portability and scalability for radiative heat transfer computations," *J. Comput. Sci.*, Jun. 2018.

[38] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel and Distributed Comput.*, vol. 74, no. 12, pp. 3202 – 3216, 2014, Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[39] P. J. Smith, R.Rawat, J. Spinti, S. Kumar, S. Borodai, and A. Violi, "Large eddy simulation of accidental fires using massively parallel computers," in *18th AIAA Computational Fluid Dynamics Conf.*, Jun. 2003.

[40] J. Pedel, J. N. Thornock, and P. J. Smith, "Large eddy simulation of pulverized coal jet flame ignition using the direct quadrature method of moments," *Energy & Fuels*, vol. 26, no. 11, pp. 6686–6694, 2012.

[41] S. Gottlieb, C.-W. Shu, and E. Tadmor, "Strong stability-preserving high-order time discretization methods," *SIAM Review*, vol. 43, no. 1, pp. 89–112, 2001.

[42] R. D. Falgout, J. E. Jones, and U. M. Yang, "The design and implementation of hypre, a library of parallel high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, A. M. Bruaset and A. Tveito, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 267–294.

[43] S. B. Pope, *Turbulent Flows*. Bristol, United Kingdom: Cambridge University Press, 2000.

[44] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland, "Large scale parallel solution of incompressible flow problems using Uintah and Hypre," in *2013 13th IEEE/ACM Int. Symp. on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 458–465.

[45] G. Krishnamoorthy, R. Rawat, and P. J. Smith, "Parallel computations of radiative heat transfer using the discrete ordinates method," *Numerical Heat Transfer*, vol. 47, no. 1, pp. 19–38, 2004.

[46] G. Krishnamoorthy, R. Rawat, and P. Smith, "Parallelization of the p-1 radiation model," *Numerical Heat Transfer, Part B: Fundamentals*, vol. 49, no. 1, pp. 1–17, 2006.

[47] X. Sun and P. J. Smith, "A parametric case study in radiative heat transfer using the reverse Monte Carlo ray-tracing with full-spectrum k-distribution method," *J. Heat Transfer*, vol. 132, no. 2, p. 024501, 2010.

[48] I. Hunsaker, T. Harman, J. Thornock, and P. Smith, "Efficient parallelization of RMCRT for large scale LES combustion simulations," in *20th AIAA Computational Fluid Dynamics Conf.*, 2011, p. 3770.

[49] S. Amarasinghe, D. Campbell, W. Carlson, A. Chien, W. Dally, E. Elnohazy, M. Hall, R. Harrison, W. Harrod, and K. Hill, "Exascale software study: software challenges in extreme scale systems," *DARPA IPTO, Air Force Research Labs, Tech. Rep*, pp. 1–153, 2009.

[50] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Investigating applications portability with the Uintah dag-based runtime system on petascale supercomputers," in *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 96:1–96:12.

[51] S. J. Treichler, "Realm: Performance portability through composable asynchrony," Ph.D. dissertation, Dept. Comp. Sci., Stanford University, Stanford, CA, USA, 2016.

[52] Q. Meng, "Large-scale distributed runtime system for DAG-based computational framework," Ph.D. dissertation, School of Comput., University of Utah, Salt Lake City, UT, USA, 2014.

[53] A. Humphrey, T. Harman, M. Berzins, and P. Smith, "A scalable algorithm for radiative heat transfer using reverse Monte Carlo ray tracing," in *High Performance Computing*, ser. Lecture Notes in Computer Science, J. M. Kunkel and T. Ludwig, Eds. Springer International Publishing, 2015, vol. 9137, pp. 212–230.

[54] A. Humphrey, D. Sunderland, T. Harman, and M. Berzins, "Radiative heat transfer calculation on 16384 GPUs using a reverse Monte Carlo ray tracing approach with adaptive mesh refinement," in *2016 IEEE Int. Parallel and Distributed Processing Symp. Workshops (IPDPSW)*, May 2016, pp. 1222–1231.

[55] B. Peterson, A. Humphrey, J. Schmidt, and M. Berzins, "Addressing global data dependencies in heterogeneous asynchronous runtime systems on GPUs," in *Proc. 3rd Int. Workshop Extreme Scale Programming Models and Middleware*, ser. ESPM2'17. New York, NY, USA: ACM, 2017, pp. 1:1–1:8.

[56] J. Tencer and J. R. Howell, "Coupling radiative heat transfer in participating media with other heat transfer modes," *J. Brazilian Soc. Mech. Sci. Eng.*, vol. 38, no. 5, pp. 1473–1487, 2016.

[57] D. Balsara, "Fast and accurate discrete ordinates methods for multidimensional radiative transfer. Part i, basic methods," *J. Quantitative Spectroscopy and Radiative Transfer*, vol. 69, no. 6, pp. 671 – 707, 2001.

[58] D. Joseph, P. Perez, M. El Hafi, and B. Cuenot, "Discrete ordinates and Monte Carlo methods for radiative transfer simulation applied to computational fluid dynamics combustion modeling," *J. Heat Transfer*, vol. 131, no. 5, p. 052701, 2009.

[59] I. Hunsaker, "Parallel-distributed, reverse Monte-Carlo radiation in coupled, large eddy combustion simulations," Ph.D. dissertation, Dept. of Chem. Eng., University of Utah, Salt Lake City, UT, USA, 2013.

[60] J. R. Howell, "The Monte Carlo in radiative heat transfer," *J. Heat Transfer*, vol. 120, no. 3, pp. 547–560, 1998.

[61] I. Veljkovic, "Parallel algorithms and software for multi-scale modeling of chemically reacting flows and radiative heat transfer," Ph.D. dissertation, Dept. of Comp. Sci. and Eng., Pennsylvania State University, University Park, PA, USA, 2006.

[62] K. Viswanath, I. Veljkovic, and P. E. Plassmann, "Parallel load balancing heuristics for radiative heat transfer calculations," in *Proc. 2006 Int. Conf. Scientific Computing (CSC), 2006*, 2006, pp. 151–157.

[63] M. F. Modest, "Backward Monte Carlo simulations in radiative heat transfer," *J. Heat Transfer*, vol. 125, no. 1, pp. 57–62, 2003.

[64] S. Silvestri and R. Pecnik, "A fast GPU Monte Carlo radiative heat transfer implementation for coupling with direct numerical simulation," *arXiv preprint arXiv:1810.00188*, 2018.

[65] K. Deshmukh, M. Modest, and D. Haworth, "Direct numerical simulation of turbulence–radiation interactions in a statistically one-dimensional nonpremixed system," *J. Quantitative Spectroscopy and Radiative Transfer*, vol. 109, no. 14, pp. 2391–2400, 2008.

[66] S. Ghosh and R. Friedrich, "Effects of radiative heat transfer on the turbulence structure in inert and reacting mixing layers," *Physics of Fluids*, vol. 27, no. 5, p. 055107, 2015.

[67] S. Silvestri, A. Patel, D. Roekaerts, and R. Pecnik, "Turbulence radiation interaction in channel flow with various optical depths," *J. Fluid Mech.*, vol. 834, pp. 359–384, 2018.

[68] R. D. Falgout, J. E. Jones, and U. M. Yang, "The design and implementation of hypre, a library of parallel high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, A. M. Bruaset and A. Tveito, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 267–294.

[69] B. Hapke, *Theory of Reflectance and Emittance Spectroscopy*, 2nd ed. Cambridge University Press, Cambridge, United Kingdom, 2012.

[70] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[71] J. Amanatides and A. Woo, "A fast voxel traversal algorithm for ray tracing," in *Eurographics*, 1987, pp. 3–10.

[72] S. Moustafa, M. Faverge, L. Plagne, and P. Ramet, "3d Cartesian transport sweep for massively parallel architectures with PaRSEC," in *2015 IEEE Int. Parallel and Distributed Processing Symp.*, May 2015, pp. 581–590.

[73] M. M. Mathis, N. M. Amato, and M. L. Adams, "A general performance model for parallel sweeps on orthogonal grids for particle transport calculations," in *Proc. 14th Int. Conf. Supercomputing*, ser. ICS '00. New York, NY, USA: ACM, 2000, pp. 255–263.

[74] S. D. Pautz and T. S. Bailey, "Parallel deterministic transport sweeps of structured and unstructured meshes with overloaded mesh decompositions," *Nuc. Sci. and Eng.*, vol. 185, no. 1, pp. 70–77, 2017.

[75] S. J. Plimpton, B. Hendrickson, S. P. Burns, W. M. III, and L. Rauchwerger, "Parallel sn sweeps on unstructured grids: Algorithms for prioritization, grid partitioning, and cycle detection," *Nuc. Sci. and Eng.*, vol. 150, no. 3, pp. 267–283, 2005.

[76] S. Plimpton, B. Hendrickson, S. Burns, and W. McLendon, "Parallel algorithms for radiation transport on unstructured grids," in *Supercomputing, ACM/IEEE 2000 Conf.* IEEE, 2000, pp. 25–25.

[77] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *Proc. Int. Conf. on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 66.

[78] L. V. Kale and S. Krishnan, "Charm++: a portable concurrent object oriented system based on c++," in *ACM Sigplan Notices*, vol. 28, no. 10. ACM, 1993, pp. 91–108.

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

[80] L. V. Kale and G. Zheng, *Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects*. Hoboken, NJ, USA: John Wiley & Sons, Ltd, 2009, ch. 13, pp. 265–282.

[81] P. P. L. U. of Illinois at Urbana-Champaign, "Charm++ parallel programming system manual," 2019, http://charm.cs.illinois.edu/manuals/html/charm++/manual.html.

[82] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: a task based programming model in a global address space," in *Proc. 8th Int. Conf. Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[83] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: exploiting heterogeneity to enhance scalability," *Comp. in Sci. & Eng.*, vol. 15, no. 6, pp. 36–45, 2013.

[84] T. Bailey, W. D. Hawkins, M. L. Adams, P. N. Brown, A. J. Kunen, M. P. Adams, T. Smith, N. Amato, and L. Rauchwerger, "Validation of full-domain massively parallel transport sweep algorithms," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2014.

[85] A. Buss, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: standard template adaptive parallel library," in *Proc. 3rd Annu. Haifa Experimental Systems Conf.* ACM, 2010, p. 14.

[86] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[87] J. Wilke, D. Hollman, N. Slattengren, J. Lifflander, H. Kolla, F. Rizzi, K. Teranishi, and J. Bennett, "Darma 0.3. 0-alpha specification," Sandia National Laboratory (SNL-CA), Livermore, CA, USA, Tech. Rep., 2016.

[88] T. Sterling, M. Anderson, and M. Brodowicz, "A survey: runtime software systems for high performance computing," *Supercomputing Frontiers and Innovations*, vol. 4, no. 1, pp. 48–68, 2017.

[89] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, and M. Lee, "The open community runtime: a runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conf. (HPEC)*. IEEE, Sept. 2016, pp. 1–7.

[90] T. Mattson, R. Cledat, Z. Budimlic, V. Cavé, S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar, "Ocr: the open community runtime interface," *Technical report*, 2015.

[91] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, "Habaneroupc++: a compiler-free PGAS library," in *Proc. 8th Int. Conf. Partitioned Global Address Space Programming Models*. ACM, 2014, p. 5.

[92] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, and F. Schlimbach, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[93] K. Spafford, J. S. Meredith, and J. S. Vetter, "Quantifying numa and contention effects in multi-gpu systems," in *Proc. 4th Workshop General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 11:1–11:7.

[94] 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/.

[95] U. D. of Energy Oak Ridge Natioanl Laboratory and O. R. L. C. Facility, "Titan Web Page," 2011, http://www.olcf.ornl.gov/titan/.

[96] C. 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. 15th ACM Mardi Gras Conf: From lightweight mash-ups to lambda grids.*, ser. MG '08. New York, NY, USA: ACM, 2008, pp. 18:1–18:9.

[97] P. Balaji, A. Chan, and R. T. E. L. W. Gropp, "Non-data-communication overheads in MPI: analysis on Blue Gene/P," in *Proc. 15th Euro. PVM/MPI Users' Group Meeting on Recent Advances in PVM and MPI*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 13–22.

[98] M. Pernice and B. Philip, "Solution of equilibrium radiation diffusion problems using implicit adaptive mesh refinement," *SIAM J. Sci. Comput.*, vol. 27, no. 5, pp. 1709–1726, 2005.

[99] N. Corporation, "Nvidia Developer Zone Web Page," 2018, https://docs.nvidia.com/cuda/index.html.

[100] T. K. Group, "OpenCL Web Page," 2018, https://www.khronos.org/opencl/.

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

[102] TOP500.Org, "Top500 Web Page," 2012, http://www.top500.org/list/2012/11/.

[103] M. Faghri and S. Senden, Eds., *Heat Transfer to Objects in Pool Fires*. Southampton, UK: Wit Press, 2008, vol. 20.

[104] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, and C. Wight, "Extending the Uintah framework through the petascale modeling of detonation in arrays of high explosive devices," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. S101–S122, 2016.

[105] R. Thakur, R. Rabenseifner, and W. D. Gropp, "Optimization of collective communication operations in MPICH," *Int. J. High Performance Comput. Applications*, vol. 19, no. 1, pp. 49–66, 2005.

[106] C. E. Goodyer and M. Berzins, "Parallelization and scalability issues of a multilevel elastohydrodynamic lubrication solver," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 4, pp. 369–396, 2007.

[107] A. Brandt and A. Lubrecht, "Multilevel matrix multiplication and fast solution of integral equations," *J. Comput. Physics*, vol. 90, no. 2, pp. 348–370, 1990.

[108] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[109] S. Lee, T. Johnson, and E. Raman, "Feedback directed optimization of tcmalloc," in *Proc. Workshop Memory Systems Performance and Correctness*, ser. MSPC '14. New York, NY, USA: ACM, 2014, pp. 3:1–3:8.

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

[111] 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*, ser. Lecture Notes in Computational Science and Engineering, vol. 41. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 341–350.

[112] B. Van der Holst, G. Toth, I. Sokolov, K. Powell, J. Holloway, E. Myra, Q. Stout, M. Adams, J. Morel, and S. Karni, "Crash: a block-adaptive-mesh code for radiative shock hydrodynamics–implementation and verification," *Astrophysical J. Suppl. Series*, vol. 194, no. 2, p. 23, 2011.

[113] C. Burstedde, L. C. Wilcox, and O. Ghattas, "`p4est`: scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM J. Sci. Comput.*, vol. 33, no. 3, pp. 1103–1133, 2011.

[114] R. W. Townson, X. Jia, Z. Tian, Y. J. Graves, S. Zavgorodni, and S. B. Jiang, "GPU-based Monte Carlo radiotherapy dose calculation using phase-space sources," *Physics in Medicine and Biology*, vol. 58, no. 12, p. 4341, 2013.

[115] J. Bédorf, E. Gaburov, M. S. Fujii, K. Nitadori, T. Ishiyama, and S. P. Zwart, "24.77 pflops on a gravitational tree-code to simulate the Milky Way galaxy with 18600 GPUs," in *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 54–65.

[116] A. Gray and K. Stratford, *Ludwig: Multiple GPUs for a Complex Fluid Lattice Boltzmann Application*. Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series, 2013.

[117] M. P. Adams, M. L. Adams, W. D. Hawkins, T. Smith, L. Rauchwerger, N. M. Amato, T. S. Bailey, and R. D. Falgout, "Provably optimal parallel transport sweeps on regular grids," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, USA, Tech. Rep., 2013.