LARGE-SCALE DISTRIBUTED RUNTIME SYSTEM FOR DAG-BASED COMPUTATIONAL FRAMEWORK

by

Qingyu Meng

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

August 2014

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Qingyu Meng**

has been approved by the following supervisory committee members:

Martin Berzins	, Chair	**04-23-2014** Date Approved
Mike Kirby	, Member	**04-23-2014** Date Approved
Mary Hall	, Member	**04-23-2014** Date Approved
Ganesh Gopalakrishnan	, Member	**04-23-2014** Date Approved
James Sutherland	, Member	**04-23-2014** Date Approved

and by **Ross Whitaker** , Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Recent trends in high performance computing present larger and more diverse computers using multicore nodes possibly with accelerators and/or coprocessors and reduced memory. These changes pose formidable challenges for applications code to attain scalability. Software frameworks that execute machine-independent applications code using a runtime system that shields users from architectural complexities offer a portable solution for easy programming. The Uintah framework, for example, solves a broad class of large-scale problems on structured adaptive grids using fluid-flow solvers coupled with particle-based solids methods. However, the original Uintah code had limited scalability as tasks were run in a predefined order based solely on static analysis of the task graph and used only message passing interface (MPI) for parallelism. By using a new hybrid multithread and MPI runtime system, this research has made it possible for Uintah to scale to 700K central processing unit (CPU) cores when solving challenging fluid-structure interaction problems. Those problems often involve moving objects with adaptive mesh refinement and thus with highly variable and unpredictable work patterns. This research has also demonstrated an ability to run capability jobs on the heterogeneous systems with Nvidia graphics processing unit (GPU) accelerators or Intel Xeon Phi coprocessors. The new runtime system for Uintah executes directed acyclic graphs of computational tasks with a scalable asynchronous and dynamic runtime system for multicore CPUs and/or accelerators/coprocessors on a node. Uintah's clear separation between application and runtime code has led to scalability increases without significant changes to application code. This research concludes that the adaptive directed acyclic graph (DAG)-based approach provides a very powerful abstraction for solving challenging multiscale multiphysics engineering problems. Excellent scalability with regard to the different processors and communications performance are achieved on some of the largest and most powerful computers available today.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

Two Dimensional (2D)

Three Dimensional (3D)

Adaptive Mesh Refinement (AMR)

Application Programming Interface (API)

Bounding Volume Hierarchy (BAH)

Computational Fluid Dynamics (CFD)

Compute Unified Device Architecture (CUDA)

Directed Acyclic Graph (DAG)

Data Warehouse (DW)

Graphics Processing Unit (GPU)

High Performance Computing (HPC)

Implicit Continuous-fluid Eulerian (ICE)

Many Integrated Core (MIC)

Message Passing Interface (MPI)

Material Point Method (MPM)

National Science Foundation (NSF)

Department of Energy (DOE)

Open Multi-Processing (OpenMP)

Partial Differential Equation (PDE)

Portable Operating System Interface Thread (Pthread)

Reverse Monte Carlo Ray-Tracing (RMCRT)

Structured Adaptive Mesh Refinement (SAMR)

Space-Filling Curves (SFC)

CHAPTER 1

INTRODUCTION

The trend in supercomputer systems moves towards larger and more diverse computer architectures with multicore CPUs and coprocessors and GPU accelerators with differing communications networks. These recent changes pose considerable challenges for achieving performance, scalability, and portability. One approach that is suggested as a suitable candidate for exascale systems is to represent computational tasks and their dependence with a directed acyclic graph (DAG) and schedule the tasks to run on computing resources at runtime [20] once the tasks they are dependent upon have completed. The tasks in the task graph may be executed in an adaptive manner and if enough tasks are available, choosing an alternate task to a waiting task may avoid communications and data movement delays.

This dissertation presents a novel and highly-scalable task scheduling and distributed runtime system to schedule and execute on both multicore CPUs and/or GPUs and/or coprocessor tasks simultaneously and dynamically for the multiscale multiphysics Uintah software. The Uintah framework provides a general purpose component-based environment for large-scale science and engineering simulations including fluid-structure interaction problems though a combination of fluid-flow solvers and/or particle-based methods for solids on structured adaptive mesh refinement (SAMR) grids. This general design [23] makes it possible for Uintah to use both data parallelism and task-based parallelism to achieve high scalability.

1.1 Background and Motivation

A widely-adopted general approach to parallel solution of large-scale scientific and engineering computing problems is to decompose a domain into subdomains that are allocated to processors. Although the tasks associated with the subdomains may be executed in a loosely synchronous manner, it is also possible to view the execution of these tasks as a workflow. This more general view of the computation as a set of coupled tasks allows greater flexibility in task execution and makes it possible to achieve greater computing efficiency.

Most scientific applications are workflow-based applications that process a huge domain. In a distributed memory architecture, the domain is usually divided into individual data sets for local computation, generally called *data parallelism*. Often applications are also structured as a sequence of computational tasks, where each sequence is executed on a different data set. Every task has its own communication and computation requirements: it reads inputs from the previous task, processes the data, and outputs results to the next task. Initial data are input to the first task and final results are obtained as the output from the last task. The parallelism is achieved when multiple tasks are executed concurrently, called *task parallelism*.

Graham et al. studied the problem of multiprocessor scheduling without communication overhead in great depth [44]. However, when communication overhead with arbitrary data size is included, the scheduling problem becomes more difficult as the trade-off between parallelism and overhead needs to be considered. To model this problem, a DAG representation is generally used, where a node reflects a task and a directed edge reflects a communication between the incident nodes.

Sarkar identified three problems [83] that need to be solved when using the task-based parallel approach: identifying the parallelism, partitioning into tasks, and scheduling tasks. Partitioning and scheduling is done either at compile time or at runtime. If both the partitioning and scheduling decision are postponed until runtime, we may be able to achieve better partitioning and partitioning results. However, the large overhead of runtime analysis necessitates very simple partitioning and scheduling algorithms. When both partitioning and scheduling are performed at compile time, overhead of runtime partitioning and scheduling are eliminated completely. The disadvantage of compile time scheduling is that it requires an estimation of task execution times and communication overhead at compile time which may simply not be accurate. By using an approach that explicitly partitions the program into tasks at compile time and schedule tasks at runtime, a macro-dataflow model can limit the runtime overhead. However, in this model, the programmer must explicitly partition the problem and the resulting tasks and their size may not make the best use of any given target machine.

In a macro-dataflow model, there is an optimal task granularity for a given machine which can minimize parallel execution time in the presence of overhead. Furthermore, an efficient near optimal task partitioning algorithm that can be used in compile time was proposed [83]. For task scheduling, the complexity of finding a smallest parallel execution time was proved to be an NP-complete problem [83].

Sinnen [86] identified the task-based parallelization process in this order: subtask decomposition, dependence analysis, and scheduling. Similar to Sarkar's model, subtask decomposition determines the program's parallelism and partitions it into subtasks. Several techniques can be used for subtask decomposition such as data decomposition, recursive decomposition, exploratory decomposition, and speculative decomposition. Furthermore, dependency analysis is considered as an important foundation for scheduling. Dependency is a precedent relationship in which one task must be completed before another task begins to run. 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, an example of task graph [65] is shown in Figure 1.1.

List scheduling and clustering are two fundamental classes of scheduling algorithms: For list scheduling algorithms, tasks are presorted into a list with topological order. After that, each task of the list will be assigned to a processor. In clustering algorithms, strongly coupled tasks are clustered before scheduling; then they can be executed at the same processor. After clustering, clusters will be assigned to processors. There is a trade-off between minimizing communication cost and maximizing the concurrency of task execution. Several advanced techniques can be used to improve the quality of scheduling. For example, we can improve the processor utilization by inserting tasks to idle slots between already scheduled nodes. Duplication of tasks can be used to reduce the communication costs.

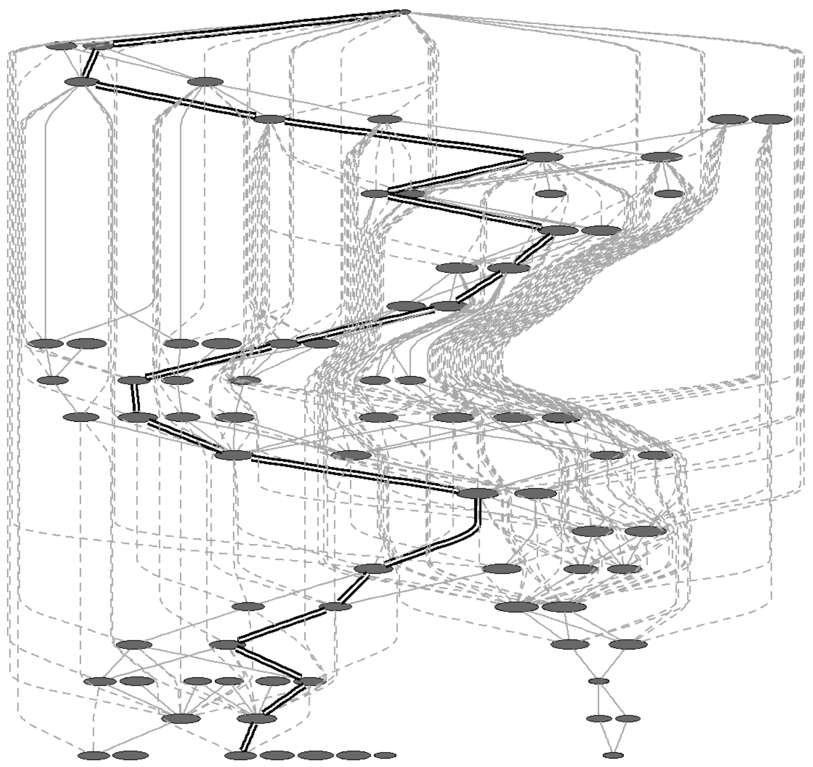


Figure 1.1. A Uintah Task Graph Example

Network topology, contention communication resources, and the involvement of processors are also included in the model to produce an accurate scheduling result.

Both Sarkar and Sinnen used graphs to represent task-based parallel programs. They built models to abstract the scheduling problem and designed several efficient algorithms based on them. The level of abstraction of those models was sufficiently close to real systems to ensure that an accurate static scheduling result can be found before runtime. Those algorithms provided important strategies to schedule tasks in a multiprocessor system.

The challenge, however, is to extend such ideas to petascale machines with complex architectures, including heterogeneous computing units, multilevel memory hierarchy, and sophisticated communication network. The working data sets and therefore the task execution times and communication latency also change dynamically and unpredictably during the simulation process. This is particularly likely to happen in applications using techniques such as adaptive mesh refinements (AMR) [31], in which the mesh (and hence the total computational work) is dynamically modified depending on the solution. There are many factors that can affect scheduling and that are hard or impossible to include in static models. The task execution time may not be easy to predict on real simulations and network latency may not be constant. Dynamic scheduling should be considered as a better approach than a purely static one as it can use all the runtime information to potentially produce a better scheduling result.

Another potential limitation of the above static scheduling algorithms is that they require a centralized task graph [86]. On large-scale machines where the number of processors could be hundreds of thousands, a task graph can easily contains millions of tasks. Processing this huge task graph is likely to become a scalability issue itself. Using a task-based approach on petascale machines requires the development of a fully distributed runtime system that can dynamically schedule computational tasks on the multicores and accelerators or coprocessors of the architecture.

1.2 DAG-based Implementations

A number of software frameworks and codes make use of task-based paradigms to solve scientific computing problems. Underlying this approach is the idea of using a directed acyclic graph (DAG) to guide the task execution. Frameworks and libraries such as Charm++ [57], TBLAS [29], and Scioto [35] all have DAG-based runtime systems. Computational entities in Charm++ can be defined using any of a variety of programming models, and the execution is enabled by a message-driven scheduler. This scheduler will automatically interleave the execution of the computational tasks. TBLAS is a task-based linear algebra library. A matrix in the TBLAS library is divided into blocks. Those blocks are mapped to different compute nodes. Tasks are then created based on output blocks. A TBLAS scheduler will automatically select a ready task and execute it. After finishing the task, dependencies are resolved causing other tasks to become ready. The Scioto framework uses a global array library to manage all distributed data. Therefore, all data are accessible using a one-sided communication operation. Tasks can only be scheduled if all its inputs are in a ″ready state″. The workload balancing is based on a voting system, which allows an idle processor to randomly steal tasks from other processors.

Uintah, the software framework considered in this study, use a DAG-based runtime system to support computational tasks from a wide range of applications on large core counts and also with AMR abilities present. The Uintah Software was originally written as part of the University of Utah Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [34]. C-SAFE, a Department of Energy ASC center, focused on providing science-based tools for the numerical simulation of accidental fires and explosions. The aim of Uintah was to be able to solve complex multiscale, multiphysics problems [22].

Uintah's component design allows these individual components to be swapped in and out, allowing them to be independently developed and tested within the entire framework. This has led to a very flexible simulation package that has been able to simulate a wide variety of problems [21]. The Uintah component approach allows the application developer to only be concerned with solving the partial differential equations on a local set of block-structured adaptive meshes, without worrying about explicit message passing calls, GPU memory operations, or notions of parallelization and load balancing. This approach also allows the developers of the underlying parallel infrastructure to focus on scalability concerns, including load balancing, task scheduling, component switching, and communications. This component-based approach to solving complex problems allows improvements in scalability to be immediately applied to applications without any additional work by the application developer. Uintah is regularly released under the MIT license.

Uintah currently contains many simulation algorithms, or components: the ICE compressible multimaterial computational fluid dynamics (CFD) formulation, the particle-based material point method (MPM) for structural mechanics, the combined fluid-structure interaction algorithm MPMICE [47], and the ARCHES combustion simulation component. Development work is also underway on a new component to provide basic molecular dynamics (MD) capabilities within Uintah. Uintah also supports a domain-specific language, Wasatch, as a simulation component which has great promise in simplifying the solution of very complex partial differential equation (PDE) problems and in automating several parts of the parallel computation pipeline in a multicore environment [23]. Uintah is regularly released as open source software [3].

1.3 Contributions

The research of this dissertation shows that combining the adaptive DAG-based approach together with a well-designed distributed runtime system can provide a very powerful abstraction for solving challenging multiscale multiphysics engineering problems on some of the largest and most powerful supercomputers available today. To address the scalability and performance challenges presented with each successive generation of supercomputers and to support new heterogeneous architectures, many generations of Uintah runtime systems have been developed.

1. Dynamic MPI Scheduler: For over a decade, Uintah used a static MPI scheduler and executed tasks from a predetermined list that was derived from the task graph. This static order was the same for all processors and did not change during runtime. Each task first posted the MPI receives, waited for all the required MPI messages to arrive, and executed. MPI sends were then posted after task execution. A limitation of this scheduler was that the overlapping of communications could only be implemented on the sending side. A single task waiting for an MPI receive causes the whole computation on a CPU core to sit idle. To address this issue, a new Uintah dynamic scheduler was developed in this research to better overlap communication and computation by using out-of-order task execution.

2. MPI Multithread Hybrid Scheduler: While dynamic MPI scheduling works well for overlapping the computations with communication on many machines by using out-of-order execution, there were still further improvements to be made. One limitation of only using MPI is that variables have to be passed through MPI messages and copied to another process' memory even if source and destination tasks are on the same multicore node. For most PDE problems solved in Uintah, ghost region copies, global meta-data, and third-party libraries consume a significant amount of memory when running at large scale. With the number of cores per node continuing to grow while memory sizes stay fixed, the memory usage limits the problem size that Uintah can solve. The solution adopted here is to design an efficient runtime system that uses MPI for internode communications and threads for each multicore node.

3. Unified Heterogeneous Scheduler: The emergence of heterogeneous systems, with additional on-node accelerators and coprocessors, presents additional design challenges in terms of effectively utilizing all computational resources on-node and managing multiple levels of parallelism. A unified heterogeneous runtime system for scheduling Uintah computational tasks on GPU and/or coprocessor has been developed. This new Uintah runtime system allows multicores or accelerators/coprocessors to post MPI Isend/Irecieve, prepare data to copy from/to a device, run CPU tasks, or run device tasks all at the same time dynamically.

4. Portability: To illustrate the new runtime system's portability, we demonstrated the Uintah software's scalability on three of the seven fastest computers as measured by the top 500 list of November 2012 [4]. These machines make use of three very different processors and networks. Some of the machines have GPU accelerators or Intel Xeon Phi coprocessors. The approach used here takes three representative and challenging Uintah applications codes and examines their scalability and performance on these very different machines.

1.4 Document Organization

Chapter 2 will discuss several related high level computational frameworks similar to Uintah. In Chapter 3 of this dissertation, the Uintah software will be outlined with the task graph generation algorithm. Chapter 4 will describe the dynamic scheduler and its improvement over the original static scheduler. The MPI multithread hybrid scheduler will be described in Chapter 5 with a master/salve model. In Chapter 6, this hybrid scheduler is improved by using a decentralized model and lock-free data structures. The scalability and performance improvement results of a full fluid-structure interaction problem will be given. Heterogeneous system support for both accelerators and coprocessors with the new Unified scheduler will be present in Chapter 7 and Chapter 8, respectively. Chapter 9 provides analysis of three Uintah components' example communications and task execution patterns on the three machines. The scalability and performance results obtained will be given with an analysis of the different cases to show how the task graph execution pattern adaptively varies to achieve scalability. Finally, the conclusion of this research and possibly future research on this topic will be given in Chapter 10. Chapter 3 and 4 correspond to paper [71], Chapter 5 and 6 correspond to papers [67, 68], respectively. Chapter 7 corresponds to papers [52, 69]. Chapter 8 and 9 correspond to papers [70, 80], respectively.

CHAPTER 2

RELATED COMPUTATIONAL FRAMEWORKS

In this chapter, several well-known frameworks similar to Uintah with block-structured adaptive mesh refinement (AMR) capability will be discussed. There are many general purpose computational frameworks that have been developed over the last few decades. The software capability of those general frameworks is not tied to a particular application. These frameworks and their application domains covered many fields from astrophysics, cosmology, and general relativity to plasma physics, and particle accelerators; from climate science to combustion, subsurface flow, turbulence, fluid-structure interactions.

BoxLib [1] is a software framework that supports the development of massively parallel block-structured AMR applications. BoxLib is written in C++ and Fortran and also includes a Fortran only implementation with limited functions. In the main branch of BoxLib, the C++ portions of the program implemented memory management, flow control, parallel communications, and I/O functions. The multigrid solvers and other numerically-intensive portions of the computation are written in Fortran90. Parallelism of BoxLib is achieved by distributing grids: internodes using MPI and on-nodes using OpenMP. Both grid and particle computations are supported in BoxLib. In grid operations, both explicit and implicit solving are supported in BoxLib. BoxLib includes single and multiple level multigrid solvers for cell-based and node-based variables. The particles in BoxLib exist on hierarchical grid meshes. BoxLib supports multiple time subcyling modes, including no subcycling and standard subcycling, for adaptive mesh simulations. BoxLib's fundamental parallel abstraction is called MutiFab. MutiFab holds the data on the union of grids at a level as multiple Fortran Array Boxes(FABs). Each FAB is an array of data on a single grid. During each MultiFab operation, the

FABs composing that MultiFab are distributed among the nodes. MultiFabs at each level of refinement are distributed independently primarily based on the use of a Morton-ordering space-filling curve. MultiFab operations are performed independently on its local data. When operations require data owned by other processors, the MultiFab operations are preceded by a data exchange between processors. BoxLib supplies a general capability for solving time-dependent PDEs on an adaptive mesh hierarchy. There are a number BoxLib-based application codes in scientific use today.

Cactus [24] was designed as a general purpose software framework for high-performance computing which employs block-structured AMR. Cactus modules consist of routines targeting logically rectangular grids; these routines are then called by the core Cactus framework. Cactus-based simulations are parallelized using both MPI and OpenMP. MPI parallelization is implemented by the driver. OpenMP parallelization is provided via a generic loop traversal infrastructure LoopControl that also provides loop tiling. The framework itself does not provide parallelism or AMR. AMR support is instead implemented via a special driver component. Carpet is the only widely-used driver in Cactus, while it would be possible to replace Carpet by an alternative driver that provides a different AMR algorithm. Application components can explicitly describe locations, shapes, and depths of refined regions. Alternatively, the application can also mark individual grid points for refinement, and Carpet will then employ a parallel tiling method to build an efficient grid structure according to marked points. Cactus modules use a domain-specific language (DSL) to describe the workflow of tasks. The framework managed a distributed data structure which contains the values of a field on every point of the grid (called grid functions), and parameter files. Cactus then provides APIs to allow infrastructure components query this information. Cactus's first application area were astrophysical simulations of compact objects involving general relativity such as black holes and neutron stars. Its most prominent user today is the Einstein Toolkit, a large set of physics modules for relativistic astrophysics simulations.

Chombo [32] is a spin-off of the BoxLib framework, having branched off from

BoxLib in 1998. Chombo shares many features with BoxLib, including the hybrid C++ and Fortran approach. Similar as Boxlib, C++ handles abstractions, memory management, I/O, and flow control, while Fortran is used for loop parallelism and stencil computations. In Chombo, an union of patches is called a BoxLayout. These maintain the mapping of patches to compute nodes. This meta-data is replicated across all processors redundantly. In cases with extreme box counts, this meta-data is compressed. Chombo keeps the Fortran Array Box (FAB) data member from BoxLib, but it is templated on data type and data centering. Instead of MultiFAB from BoxLib, Chombo has a hierarchy of templated data holders, such as LayoutData, BoxLayoutData, and LevelData. Chombo currently supports applications including magnetohydrodynamics (MHD) for tokamaks, wind turbines, solar wind and its interaction with the interstellar medium, astrophysical MHD turbulence, hydrology modelling, blood flow, subsurface reacting flow, heat transfer in nuclear reactors, etc.

Enzo [37] was originally designed to simulate the formation of large-scale cosmological structures, such as clusters of galaxies and the intergalactic medium. Enzo uses C++ for the overall code infrastructure and memory management and Fortran for most computationally intensive solvers. Enzo simulations are typically parallelized by MPI, though a hybrid OpenMP+MPI version is available and used by some applications. MPI-based parallelism takes place at the grid level, with individual grids, which are composed of a number of baryon fields, as well as particles of a variety of types, being used as the individual unit of load-balancing. Threading is typically used in situations where each MPI process has large numbers of grid patches from hundreds to thousands, and thus threading generally takes place in loops over grids. GPUs and/or the Intel Xeon Phi supports is planned. Enzo uses the block-structured adaptive mesh refinement with arbitrary grid sizes and aspect ratios. The grids in Enzo must be rectangular solids, and there are some practical limitations on grid sizes and locations. Adaptive time-stepping is used throughout the code, with each level of the grid hierarchy taking its own timestep. This adaptive timestepping is absolutely critical to the study of gravitationally-driven astrophysical phenomena, since it is the local timescale for evolution of physical systems.

FLASH [6,41] was originally designed for simulating astrophysical phenomena dominated by compressible reactive flows. FLASH's code is written in FORTRAN90 and C. FLASH's current parallelization using a hybrid MPI and OpenMP model. The vast majority of FLASH applications still operate in an MPI-only mode. Several key physics solvers have been threaded using OpenMP and are in use for production. FLASH's grid unit manages the mesh and all the associated data structures. FLASH can use three interchangeable discretization grids: 1) a uniform Grid, 2) a block-structured oct-tree-based adaptive grid using the PARAMESH library [66], and 3) a block-structured patch-based adaptive grid using Chombo [32]. FLASH's explicit physics solvers are designed to independently work with any type of underlying meshes. In this way, the application configuration can specify solvers and grids separately. The physics units that rely on their solvers interacting closely with the mesh are split into two components; the mechanics of the solvers that need to know the details of the mesh, but are agnostic to the physics, become subunits within the grid unit, while the sections that are physics-specific exist in their own separate physics units. Within the grid unit, a unified API is provided for the various underlying solvers, some of which are interfaces to libraries such as Hypre. The FLASH code is being used actively for simulations of computational astrophysics, high energy density physics, cosmology, turbulence, and biomechanical systems.

A summary of AMR framework comparison [15] is shown in Table 2.1. Although the Uintah framework will be described in detail in the next chapter, it is worth noticing that Uintah is similar to the frameworks introduced above in terms of its problem solving capability. Like many other frameworks, Uintah manages its grid and the associated data structures. A Uintah grid contains one or more grid levels. However, the refinement ratio of each level can be any arbitrary positive integer on each dimension instead of a fixed definement ratio. On each grid level, Uintah uses a bounding volume hierarchy (BVH) tree to save hexahedral mesh patches. This approach gives the Uintah framework the ability to operate on discontinuous refined grid. Uintah's component design allows these individual

Table 2.1. AMR Framework Comparison

Framework	BoxLib	Cactus	Chombo	Enzo	FLASH	Uintah
Framework Language	C++/ Fortran	C/C++	C++	C++	Fortran	C++
Application Language	Fortran	C/C++/ Fortran	Fortran	Fortran	Fortran	C++/ Fortran
Block Size	Variable	Variable	Variable	Variable	Fixed	Variable
Refine Factor	2,4	2	2,4	any int	2	any int
MPI	Block	Block	Block	Block	Block	Block
OpenMP	Block& Loop	Loop	Block	Blook& Loop	Blook& Loop	No
Pthread	No	No	No	No	No	Block& Task
GPU	No	Yes	No	Yes	Yes	Yes
MIC	No	No	No	Yes	No	Yes
Load Balancing	Yes (SFC)	Yes	Yes (SFC)	Yes (SFC)	Yes (SFC)	Yes (SFC)
Task Scheduling	No	Yes (Static)	Yes (Static)	Yes (Static)	No	Yes (Dynamic)
Scalability	196K	128K	196K	92K	512K	756K

components, such as simulation solvers, load balancers, regridders, to be swapped in and out independently. The simulation component in Uintah is written as a MPI-free tasks working on a patch. The Uintah runtime system is also distinctive in which it manages MPI communications automatically and asynchronously based on the tasks' inputs and outputs.

CHAPTER 3

UINTAH RUNTIME SYSTEM

The Uintah software framework originated in the University of Utah DOE Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [34] (9/97-3/08) which focused on providing software for the numerical modeling and simulation of accidental fires and explosions. The Uintah open-source (MIT License) software has been used to solve many different challenging fluid, solid, and fluid-structure interaction problems. The recent status of Uintah, including applications, is described by [21].

3.1 Overview

Uintah's parallel software components facilitate the solution of partial differential equations (PDEs) on structured adaptive mesh refinement (SAMR) grids. Uintah makes it possible to integrate multiple simulation components, analyze the dependencies and communication patterns between these components, and efficiently execute the resulting multiphysics simulation. Uintah contains the following major simulation components: 1) the ICE [58, 59] code for both low-speed and high-speed compressible flows; 2) the multimaterial particle-based code MPM [90] for structural mechanics; 3) the combined fluid-structure interaction (FSI) algorithm MPM-ICE [47, 48]; 4) the ARCHES turbulent reacting CFD component [56, 87] that was designed for simulation of turbulent reacting flows with participating media radiation; 5) MD component which provides Molecular Dynamics (MD) capabilities within Uintah; and 6) Wastach is a simulation component which has great promise in simplifying the solution of very complex PDE problems and in automating several parts of the parallel computation pipeline in a multicore environment [23] by using a domain-specific language, Nebo.

The component-oriented approach upon which Uintah is based [64, 73, 74] 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 in MPI, or indeed parallelization in general. This is possible as the parallel execution of the tasks is handled by a runtime system that is application-independent. This division of labor between the application code and the runtime system allows the developers of the underlying parallel infrastructure to focus on scalability concerns such as load balancing, task (component) scheduling, and communications, including accelerator or coprocessor interaction. This component-oriented parallel programming approach also makes it possible to leverage advances in the runtime system, allowing improvements in scalability to be immediately applied to applications without any additional work by the application developer. This type of programming model is ideally suited for a software environment like Uintah and has contributed significantly to its scaling success. Nevertheless, the applications developer must still write code that does not use excessive communication in relation to computation and/or a large number of global communications operations. Should this unfortunate situation occur, Uintah's detailed monitoring system is often able to identify the source of the inefficiency.

The complex engineering problems solved in Uintah require a large amount of processing power, necessitating the need for both parallelism and adaptive mesh refinement. Uintah achieves parallelism by dividing the grid into hexahedral mesh patches, which are uniquely assigned to processing processors. Figure 3.1 shows a Uintah patches which contains 64 cells. Each cell owns several types of variables: i) node centered variables, such as velocity, mass, volume, and temperature; ii) cell centered variables, such as density, internal energy, and momentum; iii) particles in cell, which also have their own variables like mass, volume, temperature, and velocity. All these variables are stored in a data warehouse, a directory-based hash map. Each variable is indexed by name, type, and the patch id of the patch to which it belongs.

Uintah components are C++ classes that follow a simple interface to establish

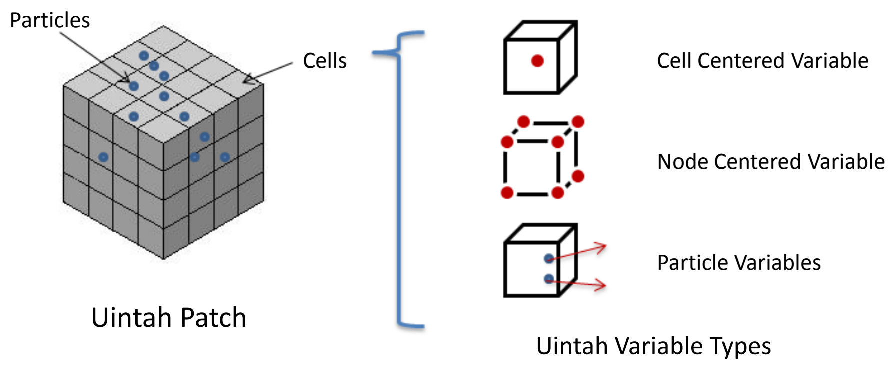


Figure 3.1. A Uintah Patch Example

connections with other components in the system. Uintah utilizes a task graph of parallel computation and communication to express data dependencies between multiple application components. The task graph is a directed acyclic graph (DAG) in which each task reads inputs from the preceding task and produces outputs for the subsequent tasks. Each task has a C++ method for the actual computation and each component specifies a list of tasks to be performed and the data dependencies between them [22].

Figure 3.2 shows an overview of the Uintah architecture. Uintah loads a selected simulation component and a grid of hexahedral cells defined in the Uintah input file. A Uintah grid can contain one or more levels with different resolutions while each level is further divided into smaller hexahedral patches. When running with adaptive mesh refinement (AMR), finer grid levels are created by the Uintah Regridder [63]. Since finer 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. After patches on each grid level are created, the Uintah load balancer is used to assign patches to MPI nodes. By profiling execution time on each patch, this load balancer can use history data to predict the further work load [62]. Once each MPI node has its patches assigned, tasks defined by the selected simulation component are then created only on local and neighbouring patches.

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 a directed acyclic

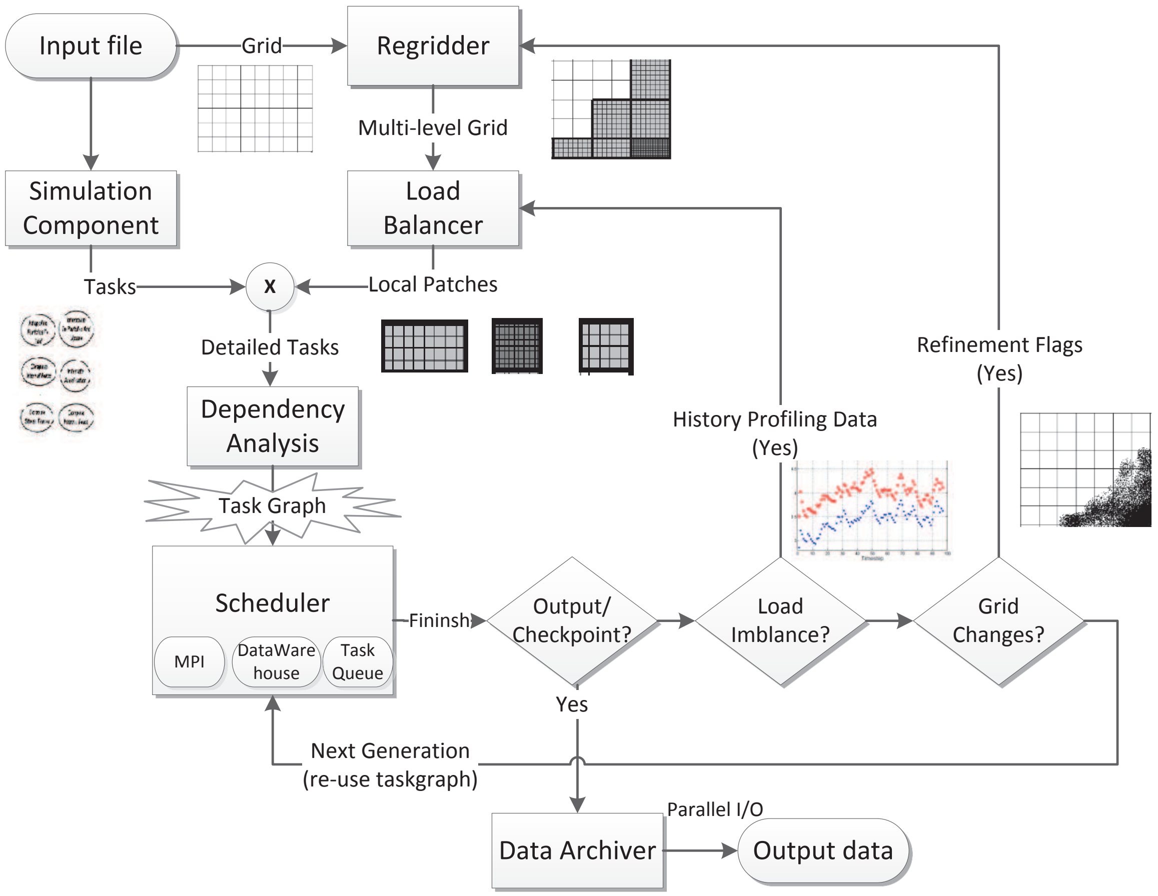


Figure 3.2. Overview of Uintah Components

task graph in which a node represents a task and a directed edge represents data flows. During the task graph compilation process, the correctness of the task graph is also checked by finding cycles or disconnect edges/nodes in the graph. This checking allows missing or ambitious dependencies (requires without computes, double computes) or cyclic dependencies (i.e., two or more tasks depend on each other) to be reported to the simulation component developer. This reporting helps developers to write correct and consistent task inputs and outputs. In the case when a dependency that connects two tasks associated with patches on the same MPI node is found, it is tagged as an internal dependency. Similarly, in the case when a dependency that connects two tasks associated with patches on different MPI nodes is found, an external dependency is created. At the end of task graph compilation, an MPI message tag is assigned to each external dependency. Task graph compilation can be done in parallel without any communications between MPI nodes. Once the task graph is compiled, it can be used for multiple timesteps without being recompiled unless the grid changes, such as when new patches are created or deleted by the SAMR regridder or when the load balancer moves patches from one node to another. Each MPI node runs a private runtime system that makes scheduling decision locally and communicates with other nodes regarding data dependencies when necessary.

3.2 Distributed Task Graph

Uintah simulation components are written as a set of Uintah tasks. In Uintah, the scheduler is responsible for computing the dependencies of tasks, determining the order of execution, and ensuring that the correct interprocess communication via MPI is made when necessary. Uintah uses a call back task design [33]. A task may be related to a single equation or stage of a simulation algorithm. A simulation component contains a list of these user-written tasks by defining input variables, output variables, and call back functions. Those tasks will be given to a scheduler, and the scheduler determines when to call each task during the execution.

3.2.1 Tasks

In order to create a Uintah task, the programmer specifies variables which are required for the task's computation, variables the task computes, and a call back function which performances computing on a generic patch. The following example equation shows the algorithm of the fourth stage of ICE, which computes face-centered velocities, according to the function:

$$\vec{U}^f = f(\Delta t, P_{eq}, \vec{g}, \rho, \vec{U}).$$

By specifying a task name (`ICE::computeVel_FC`) and a call back function pointer (`&ICE::computeVel_FC`), the Uintah task can be created:

```
Task task=new Task("ICE::computeVel_FC",
  &ICE::computeVel_FC);
```

In this algorithm, the requirements of this task include the following input variables:

1. Δt : *delT* global timing variable from previous timestep;
2. P_{eq} : *press_equil_CC* cell centered pressure variable from the current timestep;
3. $\vec{g}$: *sp_vol_CC* cell centered volume variable from the current timestep;
4. ρ : *rho_CC* cell centered density variable from the current timestep;
5. $\vec{U}$: *vel_CC* cell centered velocity variable from the previous timestep.

where Δt is a per level global variable. The variables P_{eq}, $\vec{g}$, ρ, $\vec{U}$ need one neighbouring cell data value from neighboring patches. These neighbouring cells of data, referred to as ghost cells, are copied locally to fulfill the data requirement of the ICE discretization stencil that calculating a derivative in one mesh patch may need information from an adjunct patch [58]. Variables exist either on a patch or a mesh level and have various types, such as FaceCenter, CellCenter, or Global. During the simulation, variables are stored in a dictionary data structure, the data warehouse. Variables that existed on a previous timestep are stored in the old data warehouse (OldDW) and variables that are computed in the current timestep are stored in the new data warehouse (NewDW). At the end of each timestep, the variables in the NewDW are mapped to the OldDW for the next timestep in the simulation and a new NewDW is initialized. That is to say, variables from the last timestep should be queried from the OldDW; variables from current timestep should be queried from the NewDW. In this example, the requirements for task `ICE::computeVel_FC` can be set up as:

```
Ghost::GhostType  gac = Ghost::AroundCells;
task->requires(OldDW, delT, getLevel(p));
task->requires(NewDW, press_equil_CC, gac,1);
task->requires(NewDW, sp_vol_CC, gac, 1);
task->requires(NewDW, rho_CC, gac, 1);
task->requires(OldDW, vel_CC, gac, 1);
```

From the algorithm, this task computes $\vec{U}^f$ on all three faces of the cell: *uvel_FC*, *vvel_FC*, *wvel_FC*. They are all face centered variables. All output variables are stored in NewDW, e.g.,

```
task->computes(uvel_FC);
task->computes(vvel_FC);
task->computes(wvel_FC);
```

Finally, the task is added to the scheduler component with specifications regarding which patches and materials are associated with the actual computation.

```
scheduler->addTask(task, patches);
```

For more complex problems involving multiple materials and multiphysics calculations, a subset of the materials may only be used in the calculation of particular tasks. The Uintah framework allows for the independent scheduling and computation of variables associated with an individual material within a multiphysics calculation.

3.2.2 Load Balancing

A Uintah simulation grid can contain one or more levels with different resolutions while each grid level is further divided into smaller hexahedral patches. When running with SAMR, finer grid levels are created by the Uintah Regridder [63]. As the simulation progresses, individual grid cells can also be tagged for refinement. The regridder will take flags, and, wherever there are refinement flags, patches are constructed around them on a finer level. Since finer 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.

After patches on each grid level are created, the Uintah load balancer is used to partition and assigned patches to MPI nodes. By profiling execution time on each patch, this load balancer can use history data to predict the further work load [62]. After regridding, these patches are partitioned and assigned to different processing resources by the load balance algorithm. Uintah's load balancer determines a reasonable allocation of patches to nodes using measurement and geometric information [62]. The load balancer attempts to guarantee that an equal amount of work is distributed to each processor allowing for optimal scaling of the simulation to multiple processors. The weight for each patch is predicted through certain criteria, such as history weights, number of particles, number of cells, etc. In additional to reducing the communication cost, the load balancing algorithm clusters neighboring patches together because communication is predominantly local in that only a small area of ghost cells around each patch needs to be communicated.

3.2.3 Detailed Tasks

In the Uintah framework, each patch will create its own instance of a task which is referred to as a *detailed* task. Suppose a Uintah component designed M tasks, and there are a total of N patches in the grid, a total of $M \times N$ detailed tasks will be created globally. It is nontrivial to generate a centralized directed acyclic graph(DAG) by creating one edge per dependency between detailed tasks.

1. Centralized Version In order to be more precise regarding the form of a task graph, we use the flowing definition:

Definition 1. A centralized task graph is a two-tuple
$G_{Global} =< T_g, D_g >$, where T_g is a set of nodes and D_g is a set of direct edges. Each node $t_i \in T_g$ is a detailed task associated to a patch in the global mesh and a task.

There is an edge $d < t_i, t_j > \in D_g$ if there is a dependency that t_i need to be executed before t_j.

The complexity of creating a centralized task graph will be nearly $O(|T_g| \log |T_g|)$. Since the number of tasks on a patch M is a constant, the complexity can be written as $O(N \log N)$. There will be thousands to millions of patches created in total depending on what problem size we are running. A centralized version of task graph will thus clearly not scale on large simulations with high resolution meshes. Therefore, Uintah uses a distributed algorithm to generate task graphs.

2. Distributed Version 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 actually executed. Suppose the number of processors is P; each processor approximately has N/P local patches.

Definition 2. A distributed task graph is a two-tuple
$G_{Global} = < T_l \bigcap T_n, D_d >$, where T_l is a set of locally detailed tasks and T_n is a set of neighbor detailed tasks. Each node $t_i \in T_l$ is a detailed task associated with a local patch. Each node $t_i \in T_n$ is a detailed task associated with a patch in its neighborhood. These is an edge $d < t_i, t_j > \in D_d$ if t_i need to be executed before t_j, $t_i \in T_l$ or $t_j \in T_l$.
The complexity of creating a distributed task graph in Definition 2 will be approximately [64]:

$$O(|T_l| \log |T_l + T_d|) = O(\frac{N}{P} \log \frac{N^2}{P}).$$

Consequently, a distributed version of the task graph will scale if the ratio of N/P is sufficiently bounded.

These Uintah detailed tasks contain all the necessary information for the scheduler to analyze data dependencies and execute the tasks in a completely distributed manner. Figure 3.3 shows the data structure of a detailed task in the Uintah scheduling system. A detailed task contains the following information: 1) Patch: the patch that the current detailed task will process, as assigned by the load balancer. 2) Task-related information such as task name, task type, call back function. 3) Input: Variables required for the computation in this task. These

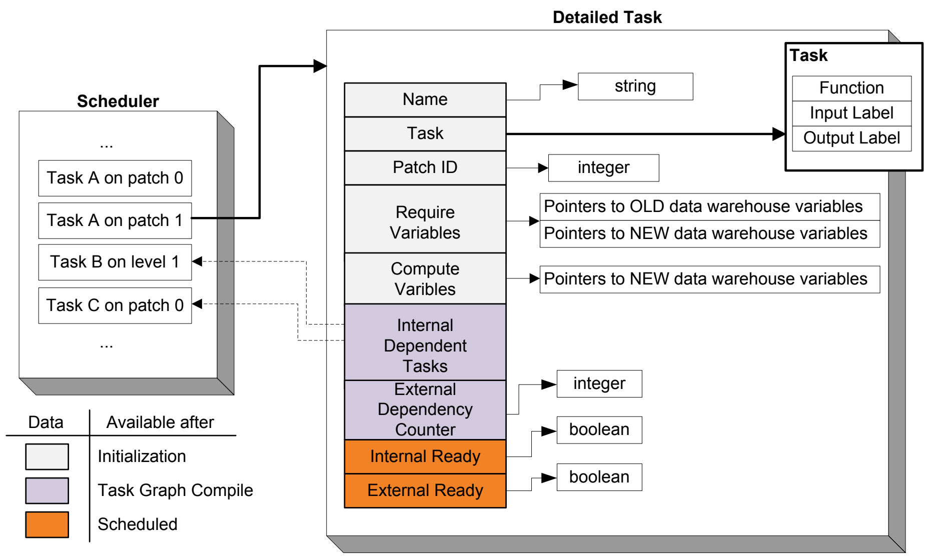


Figure 3.3. Data Structure of Detailed Task

variables may come from the task's patch or from neighbor patches. 4) Output: Variables computed by this task. These variables will be written to local memory.

After the task graph is compiled, each detailed task also contains: an internal dependency pointer that links to tasks which require variables from this task, an external dependencies counter that specifies the number of MPI messages that need to be received from the other processor. During run time, there are also some task status flags. These flags indicate whether or not a task has all its internal data, external data, is running, or has finished, respectively.

By using this design, computing patches and variables are not owned by individual tasks. They are stored in an on-demand data warehouse, a directory-based data structure. This enables the data warehouse to do the allocation and deallocation work automatically. Also there are no MPI calls inside tasks. All the MPI communication buffers are also created and destroyed automatically by the data warehouse. A detailed task is essentially a runtime instance for a task on a specific patch, and the smallest schedulable unit in Unitah.

3.2.4 Task Dependency

As the simulation component programmer writes tasks sequentially and does not explicitly define dependencies between tasks, in order to ensure the task will run in a correct order, Uintah's scheduler will automatically detect these dependencies. If there exists a data dependence between tasks, the scheduler can determine which task precedes another. There are two types of dependencies in the Uintah framework: internal dependencies and external dependencies. Internal dependencies are between patches on the same processor and external dependencies are between patches on different processors. Thus internal dependencies imply a necessary order where external dependencies also specify necessary communication.

1. Internal Dependency The Uintah scheduler detects read after write (RAW), write after read (WAR), and write after write (WAW) dependencies based on the task inputs and outputs. Each Uintah task always has the input and output variables defined through requires and computes function. Therefore, the scheduler can go through all the detailed tasks to match the patch and variables information.

Whenever two tasks access the same variable in the same patch, the scheduler detects a data dependency and updates the detailed tasks to put a dependency link between them. In the RAW case, a variable is computed by the previous task and required by the second. In the WAR case, a variable is required by the previous task, but the second task updates its value. In the WAW case, both tasks compute the same variable. Since WAR and WAW dependencies can be removed by renaming, we only consider the true dependencies.

2. External Dependency In Uintah, almost every external dependency comes from input variables with a ghost cell requirement, in that a task may require the variable from multiple additional layers of cells around its patch, as demonstrated by the variables *press*, *rho*, and *velocity* in the previous example task. Figure 3.4 shows two patches are assigned to two different processors; the task on patch 1 requires one additional layer of ghost cells which are on patch 0. Since all detailed tasks of neighbors are also created, whenever a task requires ghost cells, we can always find the corresponding originating task which computes that variable. If the originating task has been assigned to the same processor, an internal dependency will be added; otherwise, an external dependency batch object will be created. The external dependency batch objects will later be used for MPI messages combination and tag assignment.

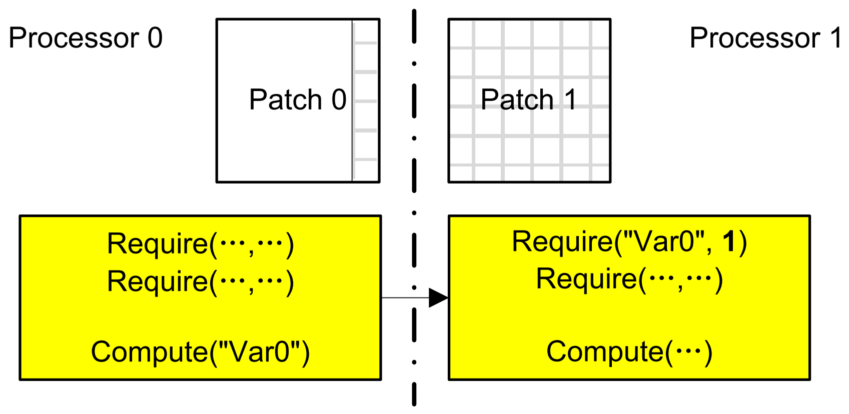


Figure 3.4. Detecting External Dependencies

Since the external dependencies for detailed tasks are computed in this distributed environment, each node only computes its own side of sends and receives. Uintah's distributed task graph can then guarantee that those sends and receives will match each other without additional communication.

3.2.5 Task Graph Compilation

Once all tasks and data dependencies are detected, each processor creates a distributed directed acyclic graph (DAG) by creating one-edge-per-variable dependency between tasks. An initial graph is generated once we have processed all data dependencies and made edges. For example, in Figure 3.5 (middle), the graph has

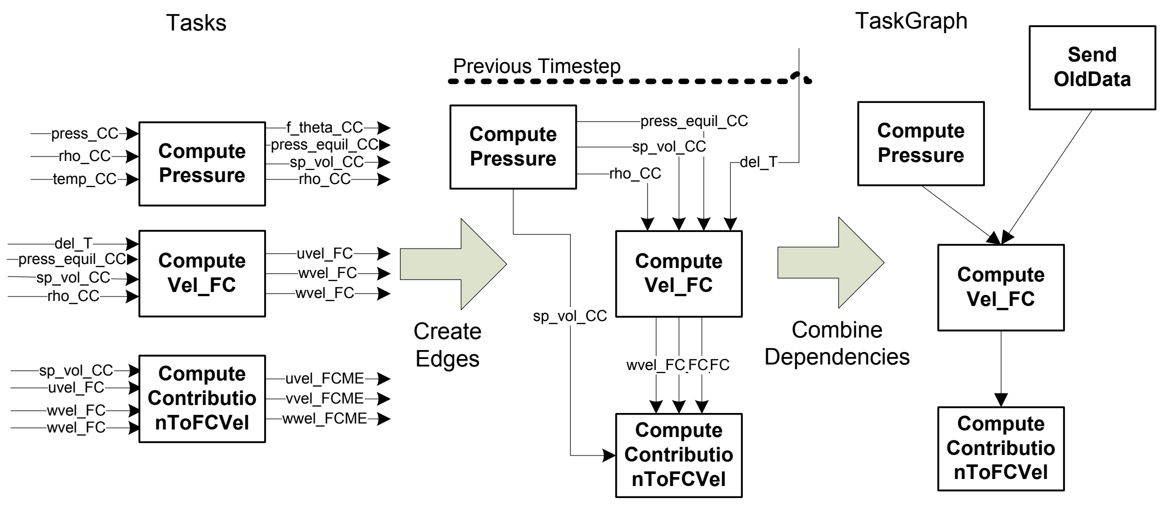


Figure 3.5. Task Graph Compilation

a lot of redundant dependencies. If the number of variables is large, the overhead for tracing the availability of all input variables will be dramatically increased. In addition, a task graph will be executed many times and may need to be simplified to record dependencies between tasks. Also, for those tasks requiring old data warehouse variables, a special system task called *SendOlddata* will be generated by the Unitah infrastructure to prepare old data warehouse variables by copying necessary variables from the previous timestep. As a result, all dependencies from the previous timestep will be replaced by dependencies from *SendOlddata* task. A dependency is also removed if it can be recursively represented by other dependencies. Figure 3.5 (right) shows part of the compiled task graph.

External dependencies are also combined if they will send data to a same destination detailed task. MPI message tags are assigned after message combination has taken place. Each detailed task will then initialize an external dependency counter to trace the outstanding MPI messages. After a task graph is compiled, the scheduler will continue to execute the same task graph on each timestep until the grid is refined or the simulation component decides the task list is no longer valid.

3.3 Data Warehouse

Uintah variables are stored in a distributed dictionary data structure called the on-demand data warehouse. The data warehouse is an abstraction of a global single-assignment memory, with automatic data lifetime management and storage reclamation. The dictionary uses three elements to index a variable: variable name, variable type, and patch id. A variable in the data warehouse is a reference-counted pointer to the local memory where the data are stored. The variable type is used to identify the data structure and for managing memory, e.g., automatic cleanup. Besides the common data types such as integer, double, and vector, Uintah also defined its own set of variable types. For example, the particle variable type associates with a particle with its location. Grid variable types including FaceCenter type or CellCenter type associates with a face of cell or a center of cell, respectively. Grid variables are typically 2D or 3D array-structured values with geometric information. Patch id is used to identify in which patch the variables are located physically.

The on-demand data warehouse not only contains local patch variables but also contains foreign variables from other processors. A task can read the variables from all local and foreign patches by calling get function to get the data pointer but can only write to its own patch by calling put function. All the temporary memory that task allocated in its own code should be discarded when it finished. In this way, a task is limited to work on its own memory and exchange data only through the data warehouse. If a task sets up its input and output variables correctly, the variables of related patches will be ready in the data warehouse to read and write when the task is being scheduled. In addition, the data warehouse will also track the life span of all variables. The data warehouse will also clean up variable memory if no future tasks are going to use that variable.

CHAPTER 4

DYNAMIC SCHEDULER

In moving Uintah to petascale machines, such as Ranger[1] and Kraken[2], it was initially observed that there was a substantial increase in MPI communication time when using a larger numbers of cores. The time spent waiting for communication comes from the dependencies between computing tasks distributed to different processors. This wait time is a combination of time spent waiting for data to be computed by another task and time spent waiting for the data to be transmitted through the network. Although Uintah's task scheduler is designed to reduce this wait time by automatically overlapping communication and computation, this wait time is linked to the fixed order of execution of the tasks.

In order to address this wait time, we considered the design of a dynamic task scheduling mechanism in Uintah [64, 74, 75] by allowing the tasks to run not in a sequential order, but dynamically, asynchronously and out-of-order according to the runtime information like a dataflow programming model. As Uintah is a general computational framework, it supports various tasks which may have asynchronous communication with different neighbors or calls to third party libraries such as PETSc. This dynamic scheduler must therefore be robust enough to guarantee that all these tasks compute the correct results.

We accomplished this by putting fine-grained computational tasks in a directed acyclic graph (DAG) and by isolating task memory. To achieve high scalability, we use a decentralized scheduling scheme for a distributed memory system.

[1]Ranger is a NSF parallel computer at the Texas Advanced Computing Center with about 64K Intel Xeon CPU cores.

[2]Kraken is a NSF parallel computer at the National Institute for Computational Sciences with about 110K AMD Opteron CPU cores.

That is, each node schedules its tasks privately and communicates with other nodes regarding data dependencies only when necessary. Furthermore, Uintah's scheduler respects task priorities and supports scheduling tasks which require a global synchronization operation. In order to create as many independent tasks as possible (to prevent processors from becoming idle), we allow multiple versions of memory by adding a variable version table. This can help the system to remove certain task dependencies and generate more independent tasks.

4.1 Original Static Design

Uintah's task graph approach provides a high degree of automated parallelism. The task graph in Uintah was originally used with static analysis of the data dependencies of user-defined tasks. As shown in Figure 4.1, the scheduler generated a correct order of tasks for later execution through a task graph compilation. The execution order was originally identical for all processors and the simulation process in Uintah was synchronized. In the approach adopted here, new data structures for the task graph are added to support dynamic execution of tasks without changing the task interfaces for users.

The original task scheduler in Uintah uses asynchronous MPI communication and combines messages which have the same source and destination. These techniques can overlap some communication and computation and reduce the data transmit time through the network. For example, after a task is finished, the

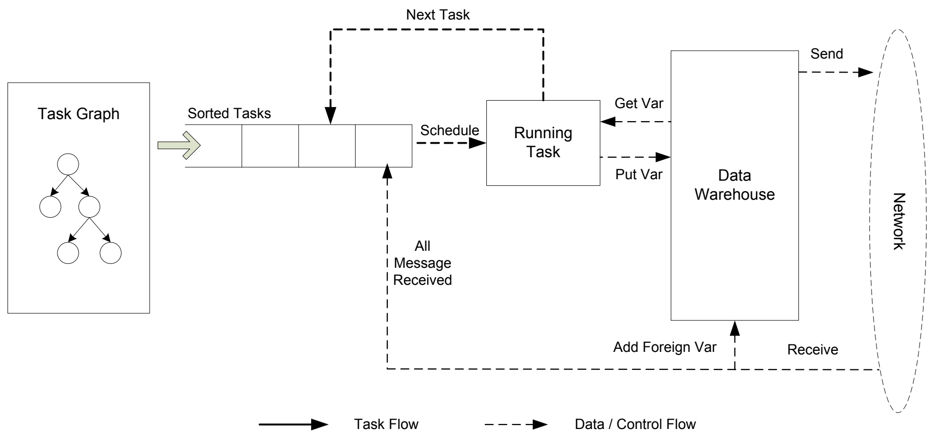


Figure 4.1. Uintah Static MPI Task Scheduler

processor can execute a new task while sending the messages produced by the last task, but a task must wait for the required messages arrival before it can be executed and the computation cannot start without the data contained in these messages. The original Uintah task scheduler generated a task graph to statically analyze task dependencies and combine MPI messages. The task graph is a directed acyclic graph (DAG) in which each node in the graph represents a task. Directed edges are used to represent a data dependency or MPI communication. After the static analysis is complete, the task execution order is determined and the scheduler runs tasks based on this order.

If all tasks in the same period take the same amount of time to execute, there will be little time spent waiting for data to arrive, as all the data are computed and ready to be sent out at the same time when the whole simulation is synchronized. Uintah supports AMR, in which the workloads for different patches may not be equal, and particles move from a cell to another cell during the simulation, and so the task workload per patch with particles is not constant. The result of communications delays and variations in execution time is the time spent waiting for data dependencies to be the majority of Uintah's MPI Wait. Measurements show that this type of wait is as much as 80 percent of the total MPI wait time in Uintah. In order to reduce the task wait time and further improve the performance of Uintah simulations, we will now investigate an alternate scheduling algorithm which can dynamically execute tasks.

4.2 Dynamic Runtime System Design

In order to address the executive wait time described above, a new task scheduler was developed that solves this problem by dynamically determining the order during execution to overlap communication and computation. In particular, the architecture of the runtime system has been extended to support out-of-order execution of tasks with respect to the task graph.

4.2.1 Tasks Ready Queues

Instead of using a fixed precomputed execution order list in Uintah's static scheduler, the new dynamic scheduler has two task queues (Figure 4.2): the

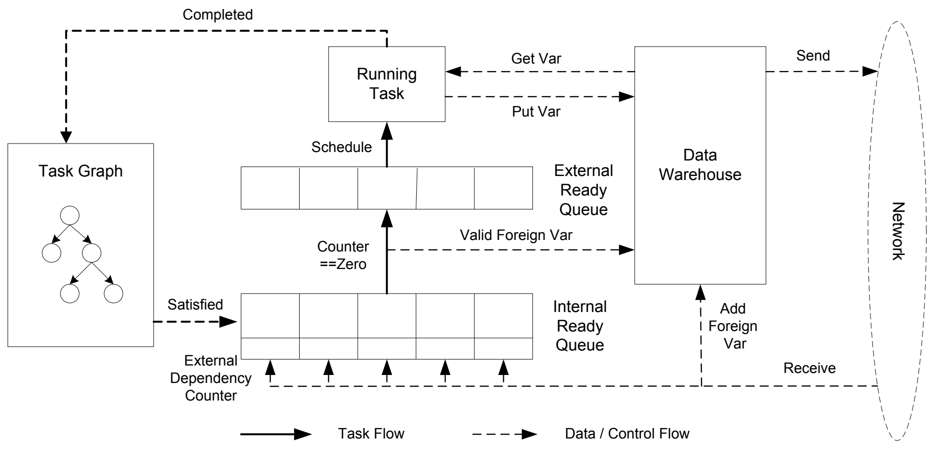


Figure 4.2. Uintah Dynamic MPI Task Scheduler

internal ready queue and external ready queue. After the task graph is compiled, all the presatisfied tasks will be placed in the internal ready queue. The value of the counter for tracking outstanding MPI messages is set according to information provided by the task graph. When this counter reaches zero, the communication phase is complete and the task is ready to be executed. At that point, it is placed in the external ready queue. When scheduling a task, the scheduler chooses a task in the external ready queue based on a prioritization algorithm. The scheduler prefers to execute tasks on the external boundary patches first when possible, or execute the tasks on internal patches while waiting for the communication for boundary patches. This is described in Section 4.3.2.

When a task is completed, the task graph will check if a task's local dependencies are satisfied. Newly satisfied tasks will also be placed in the internal ready queue and have their external dependencies initialized. The new scheduler allows multiple tasks to wait for communication at the same time; a task can also be executed when other tasks are waiting for foreign variables which are owned by other processors to arrival. To prevent conflicting access on an uncompleted foreign variables, the variable needs to be set to valid after communication is finished, and then it can be accessed by tasks.

When the scheduler begins to run, the tasks will at first wait for all internal dependencies to be satisfied and then wait for MPI messages to arrive. If a task finally reaches the external ready queue, that means that it can be executed immediately; all the variables it requests are available. As long as the external queue is not empty, the processor always has tasks to run. This can help to overlap task execution time with wait time for communication and has been found to be critical in obtaining salabilities at large core counts.

4.2.2 Variable Versioning

In Uintah, different tasks may require the same variable on the same neighboring patch multiple times: 1) They may need different ghost cells in the same patch; 2) They may need the input variables that are about to be modified. The original data warehouse was designed for static scheduling and so has one variable under each key. As tasks are executed in a fixed order, a new variable will replace an existing one. But when tasks are running in an out-of-order way, multiple copies of the same variable may exist at the same time. In order to let the correct values be available for each requesting task, we have created multiple versions of variables under the same key. The data warehouse is thus modified to automatically select a proper version of a variable according to the task's requests.

For example, in Figure 4.3, patch 0 is assigned to processor 0, patch 1,2,3 are assigned to processor 1. If three tasks on patch 1,2,3 all require ghost cells of variable $v1$, three regions A,B,C of the variable on patch 0 need to be sent to and stored at processor 1. Combining all the three regions and sending a single message to save a variable in the original data warehouse will create new data dependencies. This removes the possibility that a task on patch 1 may run when region A is received and region B and C are still waiting for data. To allow these tasks to be scheduled

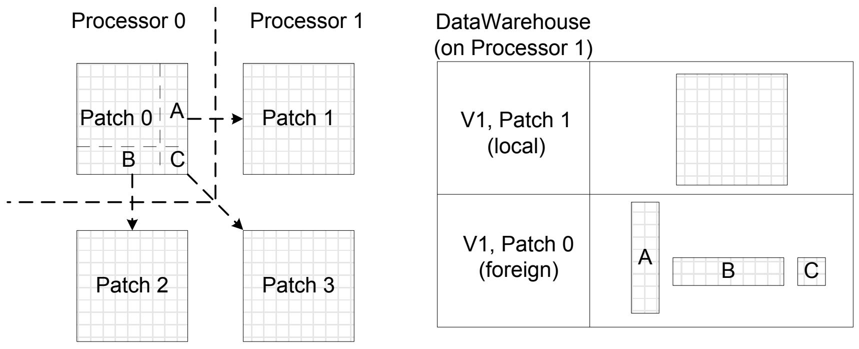


Figure 4.3. Region Versions of Foreign Variable

independently, the data warehouse uses variable versions to store all the regions on the same patch.

As mentioned in Chapter 2, variable renaming can be used to avoid false dependencies (WAR and WAW). A variable can be renamed and therefore be written into another memory location other than the conflicting variable. For example, if variable v is both an input variable and an output variable of a task, we can rename the output variable v to v_new. However, in some situations, such as calling a library whose output must be at the same memory location of its input, variable renaming cannot be used. In Uintah, the programmer can define a modifiable variable requirement for a Uintah task to allow the task to read and write at the same variable. When scheduling, dependencies will be added to a local task graph to enforce that any task that requires a newer version of this variable will not be executed before the modifiable task. As multiple time versions of a variable under the same name will be sent through the network, the newer time version of a variable will be appended to the end of version list under the same key. The Uintah data warehouse can then select a correct version of variable for the task which requires it.

Each variable may have several versions under the same label during execution. This increased the memory usage in the data warehouse. Our experiments show that the new structure uses around 10 percent more memory. This appears to be an acceptable overhead for the increase of performance and scalability that results.

4.2.3 Synchronization Phases

Tasks that require the result of a global communication also require a specialized scheduling mechanism when these tasks are running out of order. Those global tasks are created when: a) A task computes a global variable which needs to be updated through the whole grid. i.e., computation of the total mass of the system. b) A task calls a third party library which needs the MPI communicator as an argument, i.e., calling PETSc. These global tasks will create one instance on each processor instead of one on each patch and need to be scheduled everywhere in the system at the same time. In a static scheduler, as all tasks are executed in a fixed order, the global tasks do not need special treatment, but when a task runs out of order, two issues are noticed: deadlock and load imbalance.

Due to the limitation of MPI, there are no nonblocking reduction operations provided to us in MPI versions 1&2. Such ability is provided in a recent adopted MPI standard version 3. If global tasks run in an out-of-order way, processors may not make progress if they are both blocked in different MPI reduction calls. The load imbalance problem shows itself when processors choose a different path before executing a global synchronization task. As they need to synchronize at that task, when one processor has finished more tasks than another processor, a load imbalance is observed.

To solve these two problems, tasks are divided into different phases in which each phase contains only one global task. The scheduler only executes the global task if all of other tasks in its phase have completed then moves to the next phase. In this way, global task will be execute in a fixed order. In addition, the scheduler allows nonglobal tasks to be executed in an earlier phase but not a later phase.

4.3 Improvement and Experimental Results

The new dynamic scheduler has produced a significant performance benefit in lowering both the MPI wait time and the overall runtime. In this section, we present performance results of a dynamic scheduler with various benchmarks to demonstrate and analyzes its advantage. These tests were preformed on Kraken at National Institute for Computational Sciences, the University of Tennessee and on Ranger at Texas Advanced Computing Center, the University of Texas at Austin.

4.3.1 Dynamic Scheduling Speedup

Component timing results show that our new dynamic scheduler significantly reduced the task communication wait time. Figure 4.4 and Figure 4.5 show the percent reduction of both wait time (which is as high as 90% in some cases) and total execution time on Ranger and Kraken. The example problem used is a two material compressible Navier Stokes type problem that models the movement of one material through another at high speed as in [62]. This problem was chosen as it is a typical example of the problems solved by Uintah and is also challenging due to the variable and unpredictable workload resulting from the use of AMR.

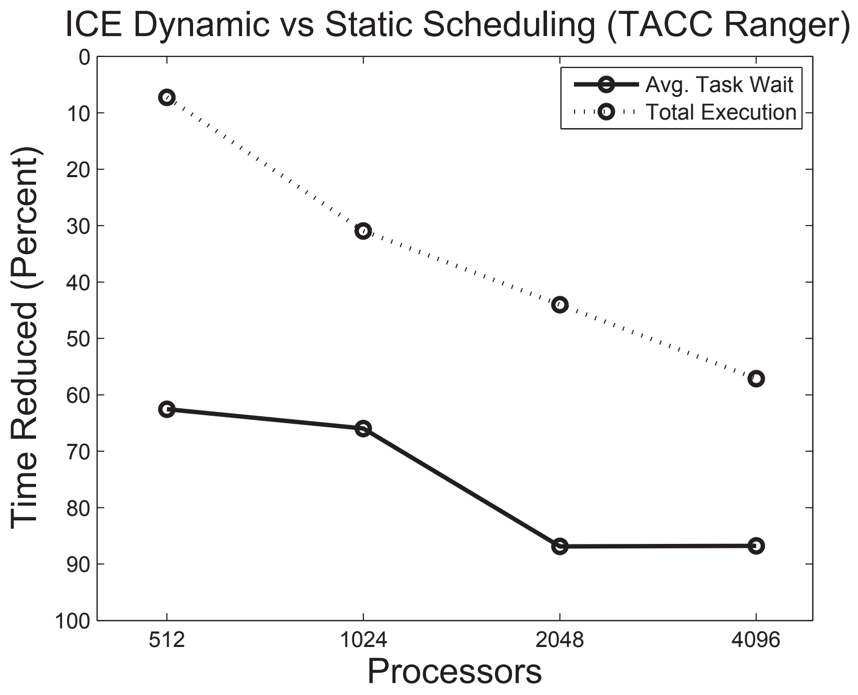


Figure 4.4. Dynamic Scheduling Speedup, Strong Scaling

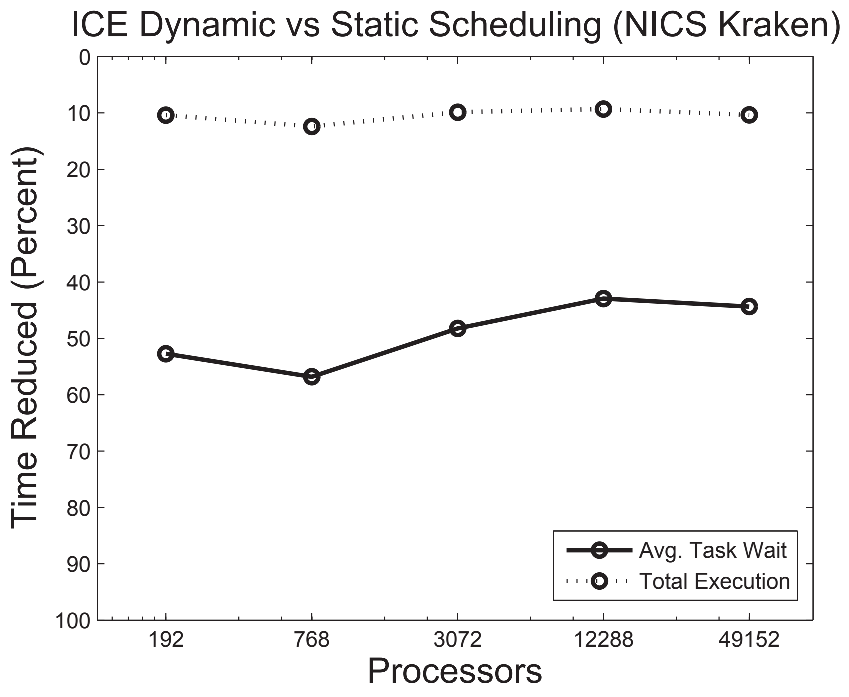


Figure 4.5. Dynamic Scheduling Speedup, Weak Scaling

In the context of this dissertation, we define strong scaling as a decrease in execution time when a fixed size problem is solved on more cores, while weak scaling should result in constant execution time when more cores are used to solve a correspondingly larger problem. The results on Ranger (Figure 4.4) were computed on a fixed problem size (strong scaling) with 24578 patches of 16^3 cells. Task wait time from 512 to 4096 processors is reduced by about 65% to 90%. The overall execution time is reduced up to 50% on runs with 49K cores, as when we use more processors, the part of MPI wait is increasing. The results on Kraken (Figure 4.5) were produced on a fixed problem size per processor (weak scaling) with 8 patches of 16^3 cells on each processor. Task wait time from 192 to $48K$ processors is reduced by around 50 to 60%. The MPI wait time is small part of total run time on Kraken, due to the benefit of a faster communication network. The overall execution time is reduced by nearly constant 10% on larger processor counts. These results show that this approach is especially beneficial on systems with slow and less consistent communication, a situation that may arise on a possibly very large future system with very large core counts.

4.3.2 Task Priority

As Uintah does not have a global view of its task graph, traditional scheduling algorithms based on the knowledge of a complete task graph cannot be used here. We designed and tested different algorithms which use only local tasks' status and the local part of a task graph. The performance of dynamic scheduling depends on how well the task executions overlap the communication between processors. As long as a processer's external ready queue is not empty, the processor will always have a task to run while waiting for incoming messages. That is to say, the scheduler will have more opportunity to reduce wait times if the external ready queue is longer. One way to lengthen the external ready queue is to give the priority to the task which can generate more ready tasks. Several prioritization algorithms are designed to maintain a priority external task queue. Once a task's external inputs are available, it is inserted into an appropriate position in this priority queue. The processer will always pick the top task of the priority queue to run, which in turn is the task with the highest priority.

The prioritization algorithms we tested here are: i) Random: Randomly give out priority. ii) First Come First Serve (FCFS): Give priority to the task which is earliest satisfied. iii) Patch Order: Give priority to the task according to its patch's geometric position (e.g., from left to right). iv) MostMessages: Give priority to the task which can satisfy most external dependencies (a.k.a., the task will send out most MPI messages).

The ready queue length, wait time, and overall runtime on an ICE problem with the above task algorithms are shown in Table 4.1. Results show that dynamic scheduler needs an effective prioritization algorithm to perform well. We discovered that a prioritization algorithm which can maintain a larger queue length led to a lower wait time on the basis of this and other experiments. The Random and FCFS algorithms do not take the communication into account and their scheduling results are worse than others. The Patch Order algorithm uses the patch's topologically sorted order to guide the execution. This causes the scheduled task order trend to a fixed order and more delays during communication synchronization. We chose MostMessages as our default prioritization algorithm, as it favors the MPI sending tasks, which can reduce the MPI waiting time of neighbors nodes.

4.3.3 Granularity Effects

We can also increase the size of the external ready queue by reducing the patch size. Uintah's patch design allows the user to easily change the size and data layout which can affect performance. When the patch size is smaller, there are more patches per processor. Therefore, more tasks are created per processor and the size of the external ready queue increases. The following granularity results are generated from a fixed ICE problem running with 24K cores on Kraken. If patches are smaller, there are more patches per processor, the average length of the task

Table 4.1. Prioritization Algorithms Effect

Algorithm	Random	FCFS	PatchOrder	MostMsg.
Queue Length	3.11	3.16	4.05	4.29
Wait Time	18.9	18.0	7.0	2.6
Overall Time	315.35	308.73	187.19	139.39

ready queue increases, and the task wait time is lower.

Figure 4.6 shows that if we use smaller patches, the task wait time is small, but the overhead of regridding, patch migration, and task scheduling is relatively large. As a result, the program's overall execution time will decrease first and then increase, depending on which part of the effects dominate. This experiment also shows that the 12x12x12 is an optimal patch size for this ICE problem running on Kraken with 24K cores. From other experiments, this optimal patch size may change when solving different problems or running on different machines.

4.4 Summary

We discovered that the time spent waiting for communication in Uintah is due to dependencies between computing tasks distributed across different processors. A new dynamic task scheduler that allows better overlapping of the communication and computation is designed and evaluated in this study to improve the performance of Uintah for petascale architecture. Uintah framework's component design allows us to replace its original static task scheduler without changing the user's

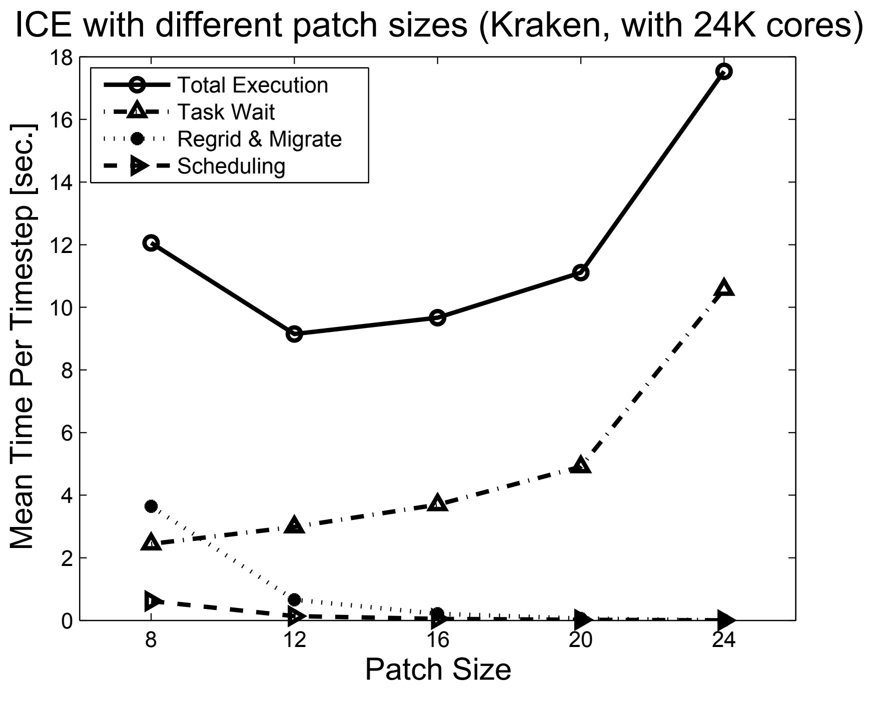


Figure 4.6. Granularity Effects

interfaces or codes. In order to support asynchronous, out-of-order scheduling of computational tasks, the new scheduler can determine the execution order of tasks according to both task graph and runtime information by putting tasks in a distributed directed acyclic graph (DAG) and further isolating task memory. This new approach is shown to significantly reduce the communication wait time on large-scale fluid-structure examples.

The next step will be to develop a new task scheduler to include a multithreaded option to take advantage of the most recent and emerging multicore architectures as well as future GPU-like architectures. The new scheduler will use MPI for internode communication and multithreaded task graph execution within nodes.

CHAPTER 5

HYBRID SCHEDULER - MASTER/SLAVE MODEL

Using the approaches described in the previous chapters, we showed that Uintah scales well to about 98K cores with a dynamic scheduler for some applications [22], including a sympathetic explosion modeling problem, funded by the NSF PetaApps Program, that is one of our main applications' driving examples. As we approach problem sizes requiring greater than 100K cores on machines such as Jaguar[1] and Kraken the memory requirements of these problems require a close examination of the overall memory usage within the Uintah framework. The typical message passing paradigm that Uintah operated under was that any data that needed to be shared to a neighboring processor must be passed via MPI. For multicore architectures, the process of passing data that is local to a node is both wasteful in terms of latency from MPI sends and receives and in the duplication of identical data that is shared between cores.

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

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

In this chapter, we look at how to extend the novel approach of Uintah to the use of this hybrid model. In contrast to many other approaches, the Uintah task-based model lends itself better to the use of Pthreads, see [12], rather than OpenMP.

5.1 Uintah Global Data Structures

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

5.1.1 Ghost Cell Data in Uintah

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

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

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

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

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

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

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

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

In Uintah, n_{el} is often in the range 2-4, $n_p = 12$ [71], and $n_h = 2$, thus giving a halo overhead of 100% if $n_{el} = 2$ and a halo overhead of 50% if $n_{el} = 4$. Of course this is a considerable simplification and for a mesh partition that is not cubic when stored on a core, the halo may be even larger. These numbers correspond to a similar overhead identified in Cactus [94].

5.1.2 Global Meta-data in Uintah

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

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

$$N_{md} = 7.5NT_c\, n_{el}^3.$$

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

Although only having one copy of the mesh data per node will help to reduce the memory requirements, this may not completely solve the problem when the number of cores and nodes approaches those predicted for exascale machines [20].

5.1.3 A Model for Memory Saving in Uintah

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

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

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

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

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

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

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

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

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

5.2 Hybrid Runtime System Design

As was shown in the previous section, a limitation of pure MPI scheduling is that tasks which are created and executed on the same node cannot share data and this causes excessive memory usage on a multicore architecture. The new multithreaded MPI scheduler described below solves this problem by dynamically assigning tasks to worker threads during execution and sharing the same infrastructure components between threads. The architecture of the runtime system has been extended to support multithreaded execution. Compared to Uintah's dynamic MPI scheduler, the new multithreaded MPI scheduler has one control thread and several worker threads per MPI process. The control thread holds all infrastructure components such as the regridder, the load balancer, the task graph, and the data warehouse and has read and write access to them.

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

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

5.2.1 Control Thread

The control thread has two task queues (Figure 5.1): the internal ready queue and external ready queue. After the task graph is compiled, all the presatisfied tasks will be placed in the internal ready queue. The value of the counter for tracking outstanding MPI messages is set according to information provided by the task graph. When this counter reaches zero, the communication phase is complete and the task is ready to be executed. At that point, it is placed in the external ready queue. When scheduling a task, the scheduler chooses a task in the external ready

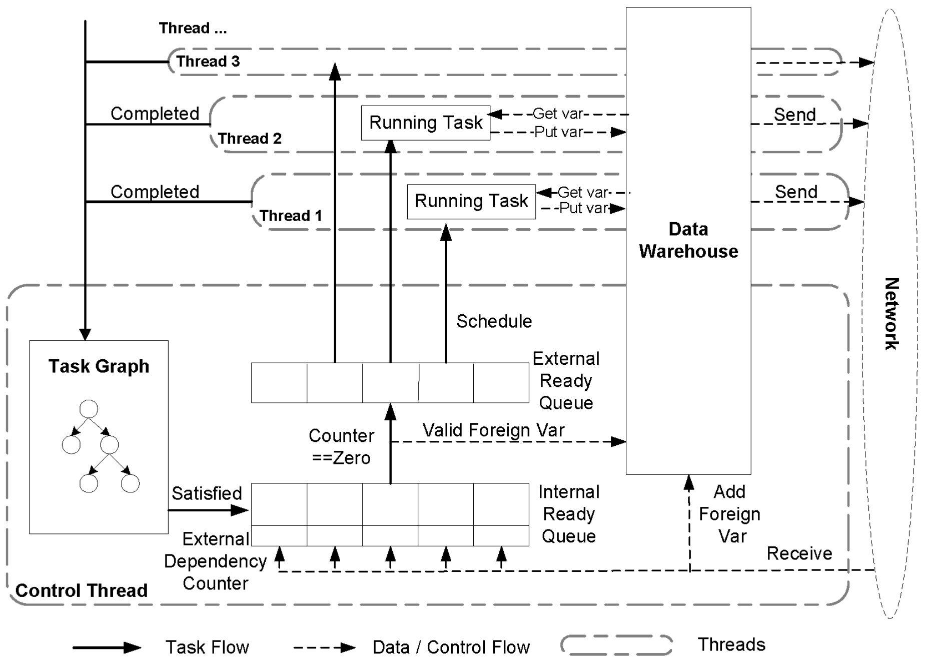


Figure 5.1. Uintah Hybrid Multithreaded MPI Scheduler

queue based on a prioritization algorithm.

At the time when a task is being scheduled, the control thread will select an idle thread as a target thread and assign a task to it. After the assignment, the control thread will then wake up this target worker thread through the worker thread's condition variable signal. If all threads are busy, the control thread will block itself by waiting based on its own condition variable until a worker thread signals it. Since MPI receives and task dependencies are also processed by control thread, when the external ready queue is empty, the control thread will call MPI_Testsome or MPI_Waitsome to process or wait for incoming MPI messages. The control thread has three states: processing MPI receives and task dependencies, blocked while requesting idle thread, and blocked in MPI wait. The algorithm for the control thread must be carefully designed to handle all these cases. A simplified version of control thread code is shown in Algorithm 5.1. The task external ready

Algorithm 5.1 Control Thread

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

queue is referred to as ReadyQ here, and the internal ready queue is not shown.

5.2.2 Worker Thread

In Uintah's multithreaded MPI scheduler, each worker thread has been made easily manageable in that its data structure only contains few variables for thread controlling and status recording. The scheduler will create a number of worker threads according to user's specification. When those threads are initialized, they will immediately block on their run conditions. Algorithm 5.2 shows the main loop of every worker thread. When a run signal is called by the control thread,

Algorithm 5.2 Worker Thread

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

the worker thread will be awoken and then start running tasks. After each task is executed, its MPI sends will also be posted by the worker thread. As the MPI send operation only requires read-only access to the data warehouse, multiple messages can be sent out concurrently. The worker thread will then ask the control thread for its next task and block on a run condition again until the next task is assigned. If the control thread is blocked, a signal from the worker thread will wake the control thread up. When a task is completed, the worker thread will also check if any task's local dependencies are satisfied based on the task graph. Any newly satisfied task will be placed in the internal ready queue waiting for the control thread's process.

5.2.3 Thread-safe Data Warehouse

As mentioned above, the core scheduler component that stores simulation variables is the data warehouse. The data warehouse is a hashed-map-based dictionary which maps variable name and patch id's to the memory address of a variable. Each task can get its read and write variable memory by querying the data warehouse with a variable name and a patch id. The task dependencies of the task graph guarantee that there are no memory conflicts on local variables' access, while variable versioning guarantees that there are no memory conflicts on foreign variables access. These mechanisms have been implemented for supporting out-of-order task execution in our previous work using a dynamic MPI scheduler [71]. This means that a task's variable memory has already been isolated. Hence, no locks are needed for reads and writes on a task's variables memory.

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

5.2.4 Task Requirements

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

5.2.5 Global Synchronization

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

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

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

5.3 Improvement and Experimental Results

The aim of this section is to examine whether the hybrid memory version of Uintah reduces the memory requirement sufficiently for us to consider running larger problems. In what follows, measurements of the memory usage were obtained by the MallocTrace memory profiling library described in [62]. Two prototypical simulation studies were used to compare the hybrid multithreaded/MPI approach versus the MPI approach. The two metrics that we looked at were memory usage and run time. The ICE algorithm was tested in both the single level and AMR case using a simulation of the transport of two fluids with a prescribed initial velocity of Mach two. For this problem, the conservation of mass, momentum, and energy equations were solved for two inviscid fluids. The fluids exchange momentum and heat through the exchange terms in the compressible Navier Stokes governing equations. This simulation is an explicit formulation and utilized the w-cycle execution model [48,50] for timestepping in which proportionally smaller timesteps were used on adaptively refined mesh patches. This problem exercises all of the main features of ICE and amounts to solving eight partial differential equations, along with two point-wise solves, and one iterative solve [22,62].

This AMR ICE benchmark [62] involved three runs of varying sizes denoted by A, B, and C. The refinement algorithm used tracked the interface between the two materials, causing the simulation to regrid often while maintaining a fairly constant sized grid, which allows the scalability to be more accurately measured. This criteria led to each problem being about four times as large as the previous one. All three runs are based on a same 3 level adaptive mesh problem but with different resolutions. Run A uses a 64x64x64 coarse level resolution and contains in total 26.8 million cells on all 3 levels of the refined mesh. Run B uses a 128x128x128 coarse level resolution and contains 108 million cells. Run C uses a 256x256x256 coarse level resolution and contains 435 million cells. Thus, the problem size of run C is about four times that of run B and the problem size of run B is also about four times that of run A. The refinement ratio of all three runs is 4 to 1. Three sets of experiments were run on the 12 cores per node of Kraken: a pure MPI case, a case where the 12 cores were split into two times 6 threads, and a third case of 12 threads per node. The results with 12 threads were slightly better, but not significantly different and so are reported here. The scalability results for cases A, B, and C are shown in Figure 5.2. Weak scalability as defined in Section 4.3.1 is represented by the almost horizontal lines and strong scalability by the almost straight diagonal lines in the three cases. The scalability results of these runs are similar to those reported in [22].

Table 5.1 and Table 5.2 show the reductions in memory and the relative CPU times when using the hybrid approach. In the strong scaling cases, the memory saving increases when running with more cores. This follows from the analysis in the previous section, because for the same grid, the ghost cell data increase as a proportion of the total data when running with more cores, but a fixed size problem size, hence the saving is larger when the number of cores increases. However, the saving of CPU time decreases and sometimes slightly exceeds the pure MPI case when running with more cores. The reason is that most of CPU savings come from eliminating in-socket MPI communications. When the number of cores increases,

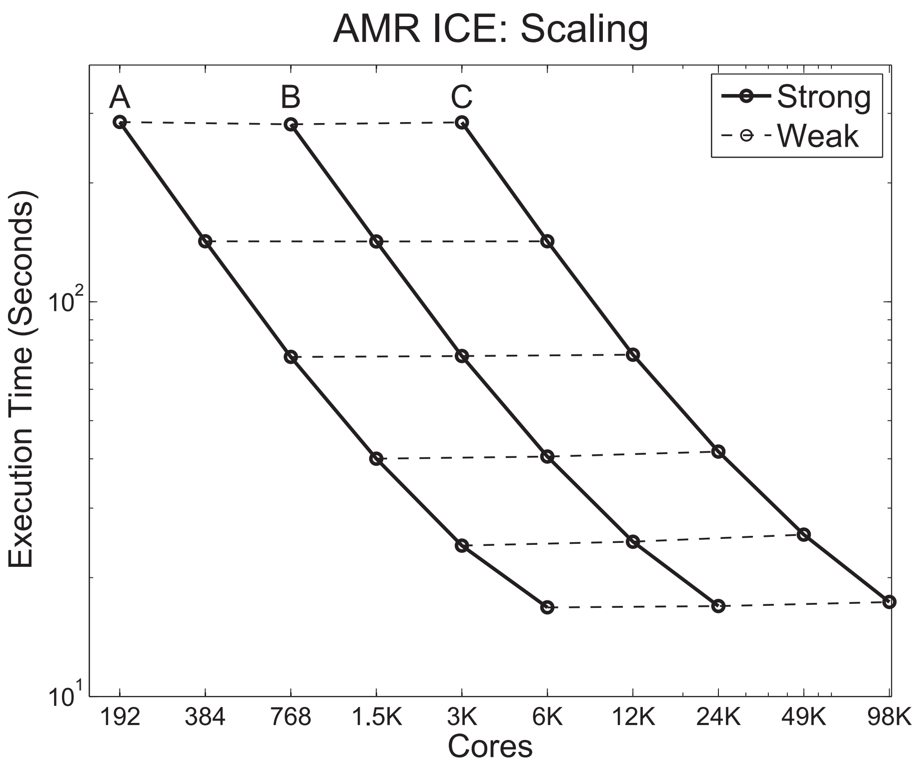


Figure 5.2. Hybrid Scheduling Scalability Results

Table 5.1. AMR ICE Relative Memory with Hybrid Approach Compared to MPI

ICE Strong/Weak Runs Memory % Used Relative to MPI						
Weak	Strong	Run A	Strong	Run B	Strong	Run C
Run	Cores	%	Cores	%	Cores	%
1	192	50	768	60	3072	61
2	384	46	1536	53	6144	47
3	768	43	3072	44	12288	36
4	1536	34	6144	33	24576	27
5	3072	25	12288	24	49152	18
6	6144	19	24576	17	98304	11

Table 5.2. AMR ICE Relative CPU Time with Hybrid Approach Compared to MPI

ICE Strong/Weak Runs CPU % Used Relative to MPI						
Weak	Strong	Run A	Strong	Run B	Strong	Run C
Run	Cores	CPU%	Cores	CPU%	Cores	CPU%
1	192	85	768	85	3072	88
2	384	84	1536	85	6144	91
3	768	90	3072	90	12288	95
4	1536	86	6144	90	24576	97
5	3072	98	12288	99	49152	100
6	6144	107	24576	104	98304	101

the amount of in-socket MPI communication decreases. Hence the saving of CPU time decreases when running with more cores. The overhead due to the use of the threaded approach, principally that of locking on the data warehouse and other nonthreaded components such as the load balancer, will offset the savings from eliminating in-socket MPI communications.

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

Using the hybrid multithreaded MPI scheduler, we have also been able to successfully run both the AMR problem and the non-AMR fluid structure interaction problem described in Sections 6 and 7 of [22] on as many as 196K cores on Jaguar, with good scaling results, due to the reduced memory requirement. These reductions in memory are illustrated by the two material CFD test problems from [62] used on Jaguar using 110K cores that could not have been previously run due to memory constraints. This problem had a resolution of 2048^3 cells and 128^3 patches distributed amongst 110,592 cores on Jaguar. The overall memory use per node was reduced from 13.5 GB per node to 1GB per node (12 cores) when running the same size problem using the nonthreaded MPI scheduler with 98K cores. Attempts were made to run this same problem on 110K cores with the MPI scheduler, but the problem size was too large and we ran out of memory on each node. The hybrid MPI/threaded approach thus allows us to consider problems that were previously out of our scope due to memory constraints.

5.4 Summary

These results show great memory savings, and show great promise so far on 200K cores on Jaguar. However, alongside these memory improvements, it is the case that further algorithmic improvements will be needed, particularly for fluid-structure interaction problems, for Uintah to be routinely used with 200-300K cores. This further work will be focused on two aspects: 1) further improve the performance and scalability of Uintah with hybrid scheduler, 2) make Uintah's particle system thread-safe and allow MPMICE component to be able to take advantage of this multithread approach. This work ultimately leads to the development of a new runtime system in the next chapter.

CHAPTER 6

HYBRID SCHEDULER - DECENTRALIZED MODEL

A significant part of the challenge in moving from petascale to exascale problems will perhaps be to ensure that multiphysics multiscale codes that reflect real applications can be run in a scalable way on parallel computers with large core counts. The multiscale challenge involved arises from the need to use approaches such as adaptive mesh refinement while the multiphysics challenge is typified by different physics being used in different parts of the domain with different computational loads in these different spatial domains. In addition, it is necessary to couple these domains with the different physics and this coupling is an interesting challenge in itself as the coupling algorithm may be more complex than the algorithms used away from the interface. These challenges are further compounded by the anticipated relative memory per core potentially shrinking [14]. Furthermore, the node architectures are becoming more complex with ever-increasing core counts and with the potential for the use of accelerators as part of the node [13], on machines such as Titan[1] and Stampede[2].

This chapter will start to address some of these challenges by developing scheduling algorithms and associated software that will be used to tackle fluid-structure interaction algorithms in which mesh refinement is used to resolve the structure within the Uintah Computational Framework [33,74,75] described above. The Uintah code was designed to solve fluid-structure interaction problems arising

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

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

from a number of physical scenarios. Uintah uses a combination of computational fluid dynamics algorithms in its ICE solver [48, 50] and couples ICE to a particle-based solver for solids known as the Material Point Method (MPM) [90, 91]. The Adaptive Mesh Refinement (AMR) approach used in Uintah is that of multiple levels of regularly refined mesh patches. This mesh is also used as a scratch pad for the MPM calculation of the movement and deformation of the solid, [62, 63]. Uintah has been shown to scale successfully with many fluid and solid problems with adaptive meshes, [22,68]; however, the scalability of fluid-structure interaction problems has proven to be somewhat more challenging. This is at least partly because the particles that represent the solid in Uintah can freely move across mesh patch boundaries in a way that was not known beforehand and partly because not all the domain is composed of the solid or fluid. There are two key components in the approach that will be described here to improve the scalability of fluid-structure interaction. The first component concerns access to data stored at the level of a node in Uintah. In Uintah, a multicore node is treated similarly to a miniature shared memory system, and only one copy of Uintah's global data needs to be stored per multicore node in the data warehouse that Uintah uses, thus saving a considerable amount of memory overall [68] as shown above. This approach also makes it possible to migrate particles across the cores in a node without using MPI. However, for this approach to be successful, it is necessary to design a data warehouse that large numbers of cores can simultaneously access without contention.

The second component concerns how the cores themselves request work. In the model proposed in Chapter 4 and in [68], a single centralized controller allocates work to the cores. This approach has worked well on the Kraken and Jaguar XT5 architectures. This approach breaks down when there are more cores per node, and when communications are faster, as on the new Jaguar XK6 machine that presently consists of the CPU part of the proposed Titan machine. The solution, as will be shown, is to move to a distributed approach in which the cores themselves are able to request work.

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

The challenge addressed here is similar to that addressed in the PRONTO work [16, 17] for the solution of contact problems in which the contact algorithm requires more work than takes place in the rest of the domain. There are many examples of other parallel fluid-structure interaction work [27, 36, 43, 45], but the approach adopted here is somewhat different to many as it relies heavily upon the asynchronous nature of Uintah.

6.1 Uintah's Fluid-Structure Interaction Methodology

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

Uintah uses a full *multimaterial* approach in which each material is given a continuum description and is defined over the complete computational domain. Although at any point in space the material composition is uniquely defined, the multimaterial approach adopts a statistical viewpoint whereby the material (either fluid or solid) resides with some finite probability. To determine the probability of finding a particular material at a specified point in space, together with its current state (i.e., mass, momentum, energy), multimaterial equations are used. The algorithm that uses a common framework to treat the coupled response of a collection of arbitrary materials is described below. This methodology follows the ideas of Kashiwa et al. [58, 60]. Individual equations of state are needed for each material to determine relationships between pressure, density, temperature, and internal energy. Constitutive models are also required to describe the stress for each material based on appropriate input parameters (deformation, strain rate, history variables, etc.). In addition to those parameters, the multimaterial nature of the equations also requires closure for the volume fraction of each material.

6.1.1 The ICE Multimaterial CFD Approach

In order to represent fluids in its multimaterial CFD formulation, Uintah uses the ICE (Implicit Continuous-fluid Eulerian) method [49], further developed by Kashiwa and others at Los Alamos National Laboratory [60]. The use of a cell centered, finite volume approach is convenient for multimaterial simulations in that a single control volume is used for all materials. This is particularly important in regions where a material volume goes to zero, as by using the same control volume for mass and momentum, if the material volume tends to zero, then the associated mass and momentum also similarly tend to zero. The technique allows wide generality in the types of problems that can be simulated.

The Uintah implementation of the ICE technique uses operator splitting in which the solution consists of a separate Lagrangian phase where the physics of the conservation laws are computed and an Eulerian fluids phase in which the material state is transported via advection to the surrounding cells. The general solution approach is well-developed and described in [47, 48, 50, 97].

6.1.2 The Material Point Method

Solids in Uintah are represented by the particle method known as the Material Point Method (MPM) [90, 91]. MPM is a powerful technique for computational solid mechanics and has found favor in applications such as those involving complex geometries, large deformations, and fractures; see [21] for these and many other examples. MPM is an extension to solid mechanics of FLIP [25], which is a particle-in-cell (PIC) method for fluid flow simulation [26]. Uintah also uses an implicit formulation of MPM [46]. In MPM, Lagrangian particles or material points are used to discretize the volume of a solid material. Each particle carries state information (e.g., mass, volume, velocity, and stress) about the portion of the volume that it represents. The MPM method typically uses a cartesian grid as a computational scratchpad for computing spatial gradients. This grid may be arbitrary, and in Uintah it is the same grid used by the accompanying multimaterial ICE CFD component. Particles are usually created on the finest level of the mesh and are always mapped back to the background grid according to their coordinates. The initial physical state of the solid is projected from the computational nodes to the cell centers collocating the solid material state with that of the fluid. This common reference frame is used for all physics that involve mass, momentum, or energy exchange among the materials. This results in a tight coupling between the fluid and solid phases. This coupling occurs through terms in the conservation equations, rather than explicitly through specified boundary conditions at interfaces between materials. Since a common multifield reference frame is used for interactions among materials, typical problems with convergence and stability of solutions for separate domains communicating only through boundary conditions are alleviated. Considerable improvements in MPM and its analysis [82,89,96,99] have resulted from work connected to the Uintah code.

6.1.3 Uintah Fluid-Structure Algorithm

The combination of MPM and ICE, MPMICE, is Uintah's fluid-structure interaction component. One of the challenges in multiphysics simulations is that the coupling algorithms involve calculations with each of the methods used in the domains that are coupled. Such calculations impose extra work in the coupling domain and also involve calls to the functions that implement the individual methods being coupled. In the Uintah coupling algorithm, there are twelve steps. Some of these steps apply only to the fluid (as labeled by ICE), others apply only to the solid (as labeled by MPM), while other steps apply to both the fluid and the solid (as labeled by MPMICE). If each phase of this algorithm is used as a synchronization point, then the parts of the domain not containing a multiphysics interface will be locked out while the interface calculation proceeds. The twelve steps used in the Uintah coupling algorithm are [50]:

1. Interpolate particle state to grid, MPM;
2. Compute the equilibrium pressure, ICE;

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

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

6.1.4 Scaling Challenges

The motivating fluid-structure interaction problem used here arises from the simulation of explosion of a small steel container filled with a solid explosive (PBX-9501). The explosive ignites and begins to burn, converting the solid into a high temperature gas which in turn causes the container to pressurize. As the pressure increases, the container expands and eventually ruptures violently. The benchmark problem used for this scalability study is the transport of a small cube of steel container piece inside of the PBX product gas at an initial velocity of Mach two. The simulation used an explicit formulation with the lock-step timestepping algorithm which advances all levels simultaneously. This problem exercises all of the main features of ICE, MPM, and AMR and amounts to solving eight partial differential equations, along with two point-wise solves, and one iterative solve as described in [22, 62]. The ICE method [58, 60, 96] is used to model the gas and the MPM [90] method is used to model the solid. The interactions between the gas and the solid that cause the block of material to move are modeled using the algorithm described in [47, 48, 50]. This benchmark also included a model for the deflagration of the explosive and the material damage model ViscoSCRAM [19] in order to be representative of the type of target calculations at which Uintah is aimed [22]. The simulation utilized three mesh refinement levels with each level being a factor of four more refined than the previous level. The starting solution with fluid velocity arrows and the refined spatial mesh around the moving block of material to this problem is shown in Figure 6.1 as is the solution at the end of the run again with with fluid velocity arrows and the refined spatial mesh around the moving object. The initial solution with velocity arrows is shown on top left and its mesh is shown on top right; the final Solution with velocity arrows is shown on bottom left and its mesh is shown on bottom right. The refinement algorithm used tracked the interface between the solid and the fluid, causing the simulation to regrid often while maintaining a fairly constant sized grid, which allows the scalability to be more accurately measured. This criteria led to each problem being about four times as large as the previous one.

This test problem was originally run on NSF's Kraken system [65] in 2011 with the scaling results shown in Figure 6.2. In this figure, the average time per timestep is compared against the number of cores used. The benchmark involved four strong scaling runs of varying size. Each of these runs uses an initial coarse mesh with doubled resolution with respect to the previous run. The refinement criteria used in MPMICE refines the mesh anywhere that particles exist. This refinement criteria led to each problem being about eight times as large as the previous one. In the four cases shown, the number of mesh cells were 619K, 3.8M, 21.3M, and 152M, respectively, while the number of MPM points used to represent the solid were 2.1M, 16.8M, 134M, and 1.1B in each of the strong scaling cases associated with the four solid lines. In the case of the run with the largest number of points,

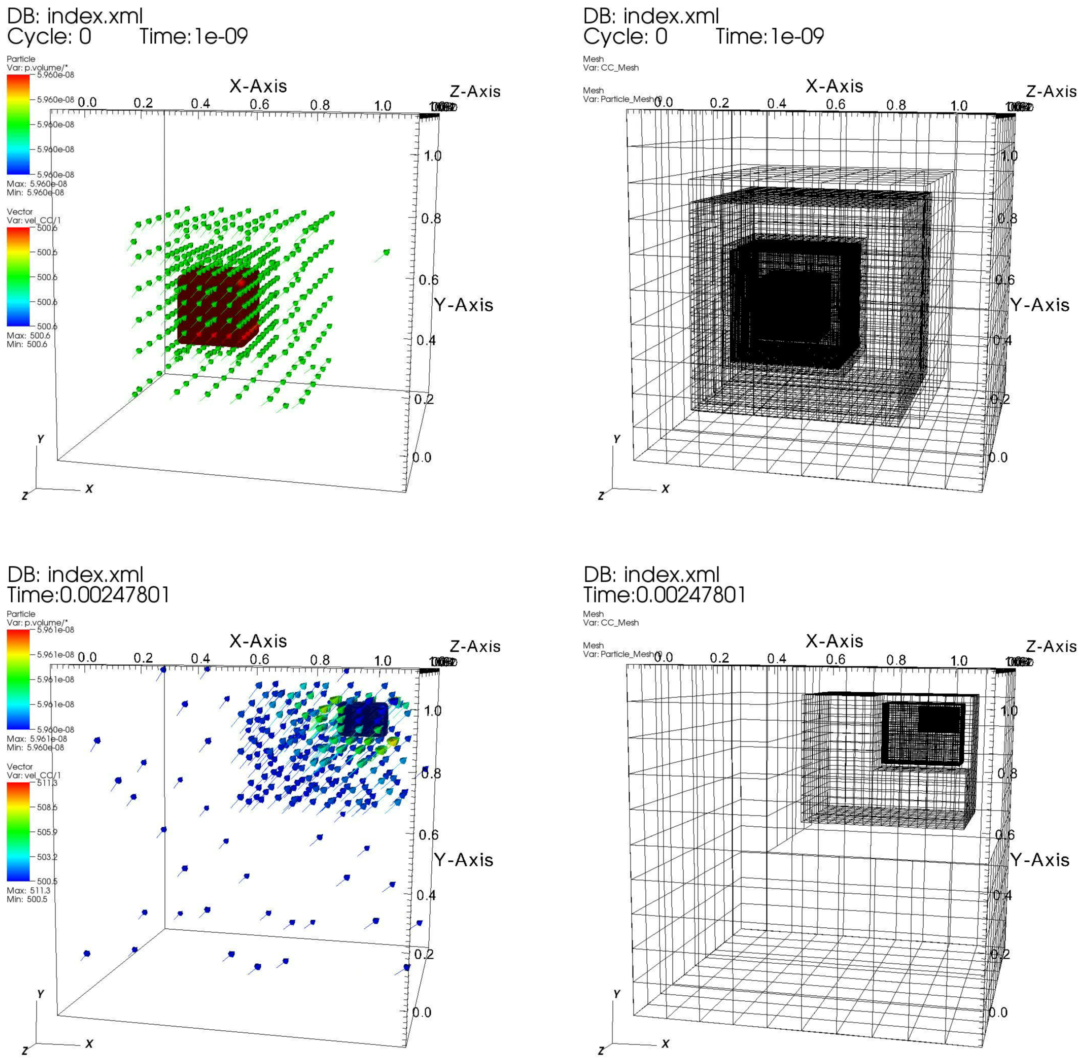


Figure 6.1. Uintah AMR MPMICE Simulation Solution

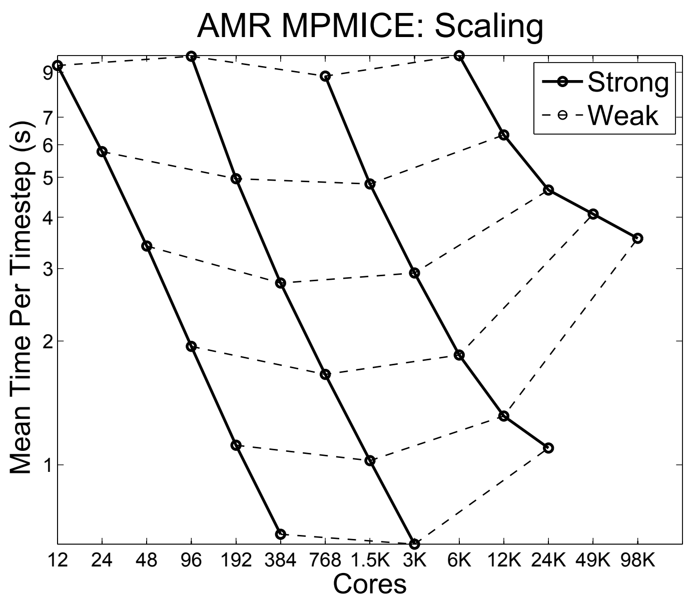


Figure 6.2. AMR MPMICE Initial Scalability Results

the rightmost solid line, the strong scaling clearly breaks down as the line turns up. In this case, the code has only only 16% relative efficiency [65] at the final point with respect to the run in the top leftmost corner. The dashed lines show weak scaling information in which the lines should be horizontal if weak scaling occurs when moving to a larger core count with the same amount of work per core. In the case of this problem at larger core counts, both weak as defined above and strong scaling as defined in Section 4.3 break down.

In this example, as the solid moves through the domain, the number of particles in a certain area changes significantly. Figure 6.3 shows how the number of particles and execution time changes during the simulation at two different locations. Location A is at the front of the object while location B is at one of the edges of the object. Figure 6.3 shows that the computing cost of a patch with particles is about six times as great as the cost of a patch without particles. This causes a serious load imbalance issue which leads to poor scaling results as shown in Figure 6.2. Even with the measurement-based load balancer [62], the computational work in a region with particles is hard to precisely predict. When running with pure MPI scheduler, if one core has finished its assigned work faster than the other cores, it has to stay idle and wait for them to finish even if they are in the same node.

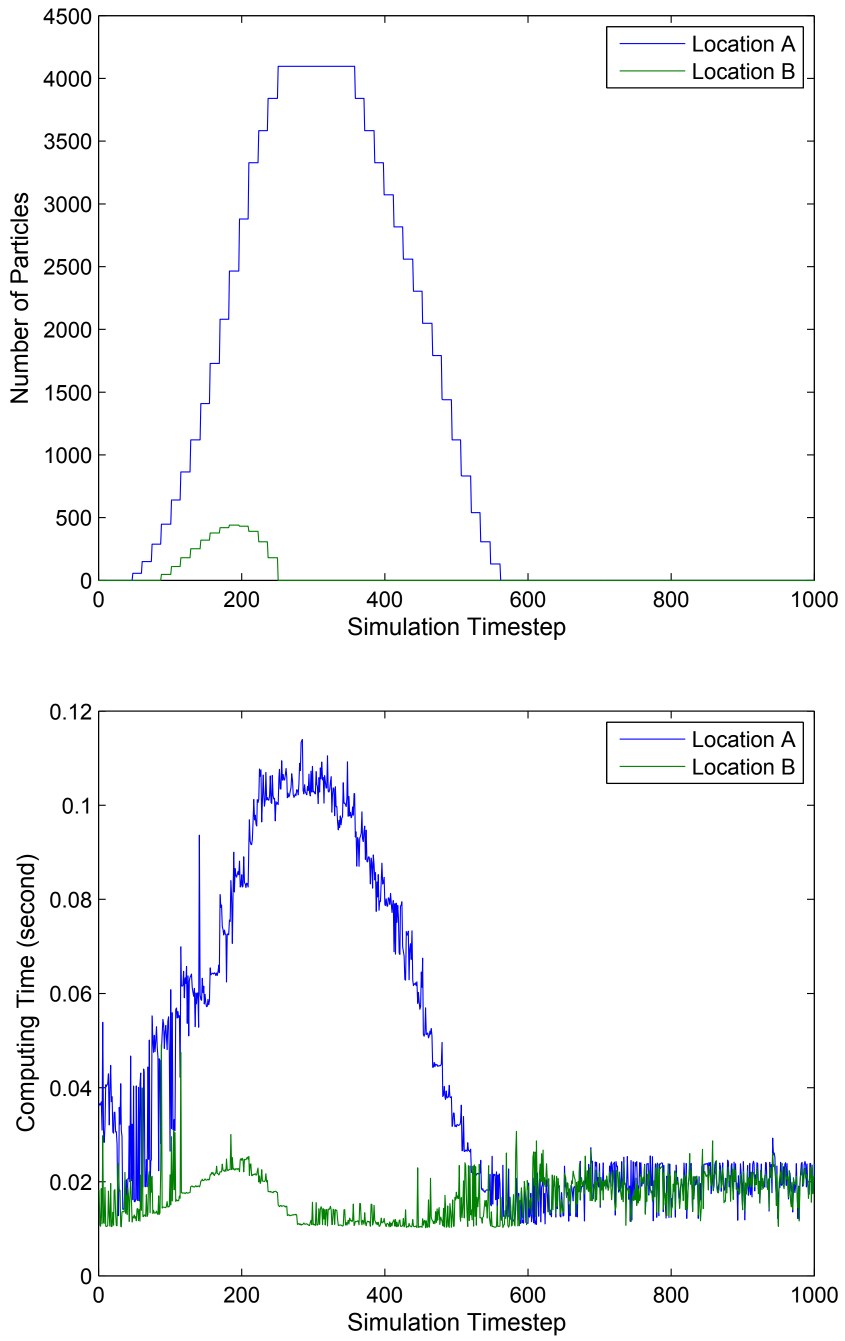


Figure 6.3. MPM Particles and Task Execution Time

With the same code, adaptive mesh refinement scaled well to 98K cores, [68]. The challenges that arose in the scaling of this fluid-structure interaction problem directly motivated and influenced the work presented here.

6.2 Decentralized Hybrid Runtime System Design

A potential bottleneck in the master slave model hybrid scheduler is that a worker thread may become idle if the control thread cannot respond to its next ready task request quickly enough. In order to guarantee a short response time, the control thread was assigned to a dedicated core. This approach led to this core being under-utilized when running with small number of cores, as there was not enough work to keep the control thread busy. The solution to this was to design a new decentralized multithreaded scheduler to allow all threads to process MPI sends and receives or to execute tasks concurrently without a control thread.

Instead of asking the control thread for a ready task, the threads in the decentralized multithreaded scheduler directly pull tasks from the two ready queues. When a thread pulls a task from the internal ready queue, then MPI nonblocking receives will be posted. Also, when a thread pulls a task from the external ready queue, then a task's call-back function will be executed and MPI sends will be posted after task execution. Figure 6.4 shows the threads and shared data structures in this new design. Furthermore, each thread must keep checking the task queues and MPI receives when it is idle as there is no longer a central controller. As this could lead to busy-locking on those shared resources when multiple threads become idle and keep acquiring read locks, a two-stage execution approach is used.

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

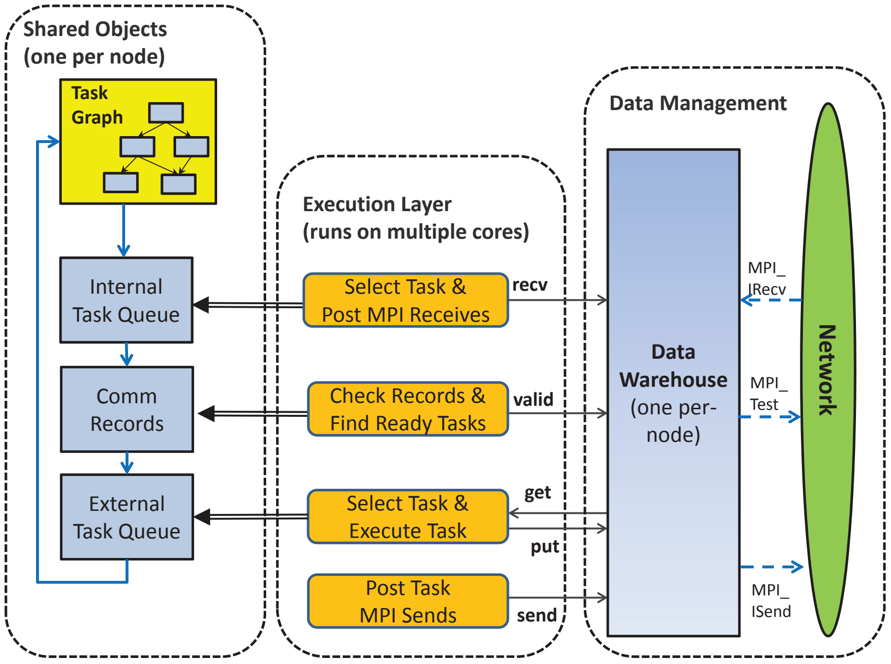

Shared Objects (one per node)
Task Graph
Internal Task Queue
Comm Records
External Task Queue
Execution Layer (runs on multiple cores)
Select Task & Post MPI Receives
Check Records & Find Ready Tasks
Select Task & Execute Task
Post Task MPI Sends
recv
valid
get
put
send
Data Management
Data Warehouse (one per-node)
MPI_ IRecv
MPI_ Test
MPI_ ISend
Network

Figure 6.4. Thread and Shared Data Structures of Decentralized Hybrid Scheduler

resources will be slowed down by this mutex and priority is given to threads that are able to update the task queue or to post MPI receives.

Table 6.1 shows a performance comparison between the decentralized and master-slave models in a single node. In this case, the decentralized model outperforms the master-slave model on all runs up to 32 cores per node. By monitoring the CPU utilization of each core, it was confirmed that the decentralized model solved the issue of under utilization of the core that runs the control thread. Furthermore, when one master control thread with 1, 3, 7, 15, and 31 worker threads is used, the CPU loads on the control thread increase linearly and are about 0.3%, 0.7%, 1.7%, 3.0%, and 6.9%, respectively. There is an increase in master control thread CPU usage with increasing numbers of worker threads. This makes it difficult to balance the control thread workload with the worker threads. While

Table 6.1. Execution Time: Master-Slave vs Decentralized

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

the master-slave model will likely hit a control thread bottle neck when number of cores per node increases, in contrast the decentralized scheduler is able to fully utilize all available cores on-node, regardless of the number of cores.

6.3 Uintah Hybrid Parallelism Improvements

This section will discuss how the MPI multithreaded hybrid approach was used to overcome the challenges with regard to scaling fluid-structure problems. Our previous published work used the multithreaded MPI approach for fluids problems without particles by using ICE algorithm. Unlike all the tasks in ICE components which were thread-safe, several race conditions were found to exist on MPM simulation components. Most of these race conditions were due to the use of global temporary data structures instead of local ones and were easily fixed. However, much of the existing framework related to particles had to be rewritten to guarantee thread-safety.

6.3.1 Reducing Particle Relocation Costs

As noted above, in Uintah, solid objects are represented by MPM particle variables. During a timestep, each particle's new location co-ordinates and other physical attributes will be computed by MPM tasks and saved in the data warehouse. If a particle's position moves from one patch to another, it is the infrastructure's responsibility to map those particle variables back into the background grid according to their new locations. This is done by inserting a relocation task to the task graph. During this process, each particle's new "owner" patch will be located in the patch BVH tree according to its new co-ordinates. If the new "owner" patch is located on the same node, only a simple re-indexing of the patch's particle variable array is required. However, if the new "owner" patch is located on another node, the particle information is transmitted using MPI. In particle relocation, only the sending node has knowledge of the new particle location and how many particles and their variables need to be transferred. The destination node must know the source node id first, as MPI nonblocking receives cannot be posted without a source rank id. To transfer the source node id to all destination nodes, Uintah uses an MPI_scatter message which can distribute a vector to all nodes concurrently. In this way, for any particle that is moved using MPI, two messages are required. Thus the cost of moving a particle off a node is much more expensive than that of moving a particle inside a node or core. As illustrated in Figure 6.5, when using a hybrid MPI multithreaded approach, the number of particles that need to be sent is significantly reduced when using one MPI process per node as opposed to one MPI process per core, as all transfers internal to a node no longer use MPI.

6.3.2 Load Balancing Improvements

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

6.3.3 Using Lock-free Data Structures

A major overhead of the multithreaded scheduler approach [68] is due to the use of locking to protect shared data structures. This overhead keeps increasing with the number of cores per node as contention for acquiring locks also increases. A significant reduction in tasks waiting was obtained by eliminating large amounts of locking overhead through a redesign of some of the shared data structures so as to make them lock-free, in particular by using hardware-based atomic operations [40],

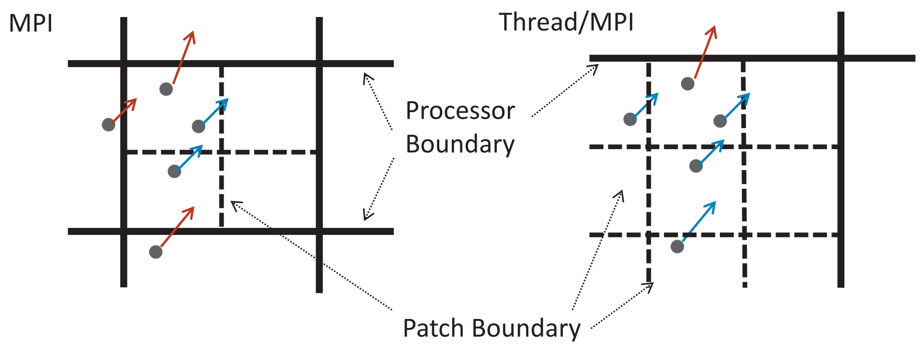


Figure 6.5. Particle Relocation Example

which are supported by modern CPUs. These new data structures can be made to be more efficient than using the traditional Pthread read write lock and mutex [40].

Uintah has three types of variables: 1) Grid variables exist everywhere in the simulation grid for flow simulations. Grid variables on the same node can share a combined 3D array with different memory windows. As both the memory window and the 3D array are reference counted, the associated memory will be deallocated when no longer referenced. 2) Particle variables exist only at a certain point for solid MPM simulations in Uintah. Particles in a Uintah patch are saved in a simple vector and indexed by a subset of that vector. By saving recent query ranges, particle sets are cached so as to speed up later queries. 3) Reduction variables are designed to combine multiple values provided by different tasks. The reduction variable operator must be associative, so that the order of computation will not alter the final value, apart from rounding errors. When the same reduction variable is computed multiple times on the same node, the locally reduced value will be updated immediately and stored in the data warehouse. After all local tasks have computed reduction variables, the global value of the reduction variable will be calculated through a call to MPI_AllReduce. As mentioned above, Uintah variables can share the same memory window and 3D array. In fact, most of Uintah objects such as grid level, variable label, and MPI buffers are also reference counted so that they can be easily deallocated when no longer needed. When running in multithreaded mode originally, reference counters were protected by a fixed size array of mutexes to ensure correctness. Once a new reference counted object was created, a mutex from this array was assigned to it in a round-robin fashion. This design allowed many objects to share a mutex instead of each object creating its own as there may be thousands of reference counted objects and dynamically creating thousands of mutexes is too expensive. However, potential false-conflicts may exist when accessing two unrelated objects which happen to share the same mutex. This reference counting lends itself to the use of atomic operations, [40]. A new reference counting class was implemented in Uintah by using *add_and_fetch* and *sub_and_fetch* atomic operations to replace the Pthread mutex vector.

As described above, all tasks depend on the hashed multimap data warehouse to look up and save variables by using a patch id and variable name key. Each data warehouse has a Pthread read/write lock to protect the hashed multimap. A read-only lock needs to be acquired when a task looks up a variable's memory address from the data warehouse. A write lock needs to be acquired when a new variable needs to put a result into the data warehouse either from a computational task or from MPI message. Based on our timing results on Uintah read-write locks in Table 6.2, the data warehouse lock was seen to be the largest single source of overhead, far exceeding the four other main data structures that used locks on a node. For this reason, the hashed multimap data warehouse was redesigned to be lock-free by using a two-step look up strategy. During task graph compilation, a hash map containing keys of all locally computed and required variables is created on-the-fly. This hash map will not change during task execution and can also be shared by multiple data warehouses until the task graph needs to be recompiled. The actual container of variables in each data warehouse will be a preallocated vector which has exactly the same size as this hash map. The value of this precomputed hash map will be the index to the container vector. When accessing a variable, the data warehouse will first use the hash-map key to locate the variable from its private container vector. As the precomputed hash map is read-only during the task execution, no lock is needed to protect it. When updating the container vector, atomic operations are used to achieve consistency among threads.

A simplified version of the Variable Inserting Algorithm for putting a new variable into the data warehouse using the *compare_and_swap* atomic operation is shown in Algorithm 6.1. This algorithm inserts a variable to a linked list atomically. The head of this linked list is saved in the container vector. A Variable Combining Algorithm for reducing reduction variables using the *test_and_set* atomic operation

Table 6.2. Shared Data Structure Locking Cost

Data Structure	DW	Level	Particle	TaskQ(int)	TaskQ(ext)
Read Lock(s)	7.86	0.066	0.056	0.035	0.009
Write Lock(s)	12.14	0.007	0.001	0.064	0.011

Algorithm 6.1 Variable Inserting: Put a variable into the data warehouse

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

is shown in Algorithm 6.2. This algorithm combines the new value of a variable

Algorithm 6.2 Variable Combining: Reduce a variable into the data warehouse

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

with the existing value in the data warehouse. When multiple threads try to update this value, any two threads can compute the combined reduction value without being serialized. As these atomic operations are used on our redesigned data structures, Pthread locks are no longer needed to protect a long critical session when accessing the hashed multimap data structure.

6.4 Experimental Results

This section will consider whether or not hybrid parallelism can sufficiently improve the performance and scalability of fluid-structure interaction problems in Uintah. We will also examine the performance difference between using a lock-free data structure and a traditional lock-protected data structure. The prototypical simulation study used in Section 6.1.4 was used to compare the hybrid multi-threaded/MPI approach, the multithreaded/MPI with lock-free data warehouse approach and the MPI approach.

6.4.1 Single Node Performance Improvement

The first comparison is between the hybrid multithreaded MPI approach and the lock-free data warehouse on a single shared memory node. This test was run on a Jaguar Cray XK-6 external node with 32 AMD interlagos cores. Figure 6.6 shows the speedups when running with 2, 4, 8, 16, and 32 cores. Three sets of strong scaling benchmark results were gathered by using the dynamic MPI scheduler, the decentralized multithreaded MPI hybrid scheduler with the old Pthread locking data warehouse and the decentralized multithreaded MPI hybrid scheduler with the lock-free data warehouse. The input files for all three runs are identical and generate 887K particles on an AMR grid. The multithreaded scheduler with the lock-free data warehouse is 1.4X faster than with Pthread locking data warehouse

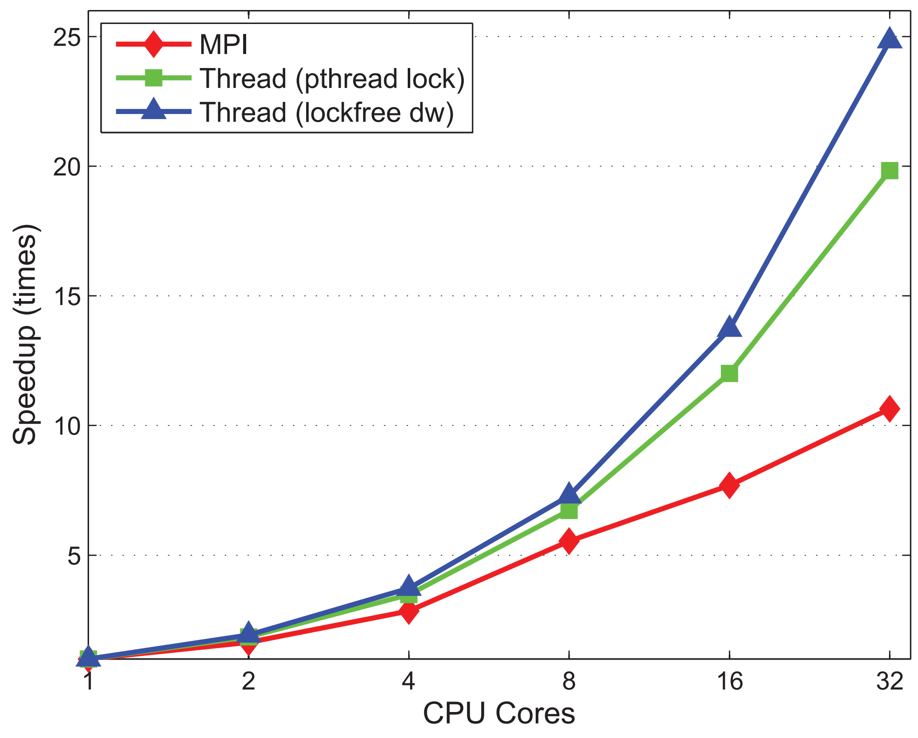


Figure 6.6. Performance Comparison on Single Node

and 2.4X faster than using MPI only.

Table 6.3 shows the execution time comparison results when running with different numbers of MPI processes and with different numbers of threads per MPI process. CPU affinity was used to guarantee that each thread is assigned to a dedicated core. Threads in the same MPI process were assigned to nearby cores so as to limit any possible cache-coherence overheads. These benchmark results show that the optimized performance for this problem is achieved when running with the maximum number of available threads per MPI process, also when increasing the number of threads to be larger than the number of cores per node. The results of using 64 and 128 threads on a 32 core node are about 9% slower than when using 32 threads. As Uintah threads are lightweight, we are able to run 1024 threads per node, which is much larger than the 32 available cores and the additional scheduling overhead is about 11%. With MPI-only, the execution times for 64 and 128 MPI processes on a 32 core node are 6.60 and 8.63 seconds, 27% and 67% slower than 32 MPI process. We are unable to obtain a result for 1024 MPI processes per node as this exceeds available memory due to the heavy memory footage by using MPI process.

6.4.2 Overall Scalability

The scalability benchmark involved four runs with varying problem sizes using a similar approach to that in Section 6.1.4 and the same test problem. Each run uses a mesh that was refined by a factor of two in each dimension with respect to the previous run. As the mesh is refined where particles exist, eight times as many particles will be created than on the original coarse mesh. The number of particles created in the four runs are 7.1 million, 56.6 millon, 452.9 millon, and 3.62 billion, respectively. This leads to each run being approximately eight times as large as the previous run. As we will also generate weak scaling results at the same time,

Table 6.3. Mixed MPI and Threads Performance

nMPI	32	16	8	4	2	1	1	1	1
nThread	1	2	4	8	16	32	64	128	1024
ExecTime(s)	5.18	3.77	3.05	2.79	2.62	2.22	2.40	2.41	2.47

the problem size per node needs to be constant. Therefore, eight times as many cores were used from one run to the next. These tests were run on the Jaguar Cray XK-6 machine with up to 256K cores and with 16 cores per node. Figure 6.7 shows the scaling results for four benchmark tests. Weak scalability is represented by the almost-horizontal dashed lines and strong scalability by the almost-straight diagonal solid lines in the four runs. The strong scaling efficiency of the largest problem (the rightmost solid line) is 68% at 256K cores relative to the base case of 16K cores.

By using optimized hybrid multithreaded MPI algorithms, the number of MPI ranks involved in communication is reduced without much performance overhead. For the MPMICE problem, overheads as percentages of local computations are: 1.2% for allocation variables and put in the data warehouse, 6.9% for querying and assembling variables from the data warehouse, 0.6% for querying particle sets from the data warehouse, 1.5% for reading tasks from the work queue, 1.2% for inserting tasks into the work queue with priority. The major single overhead (6.9%) of the

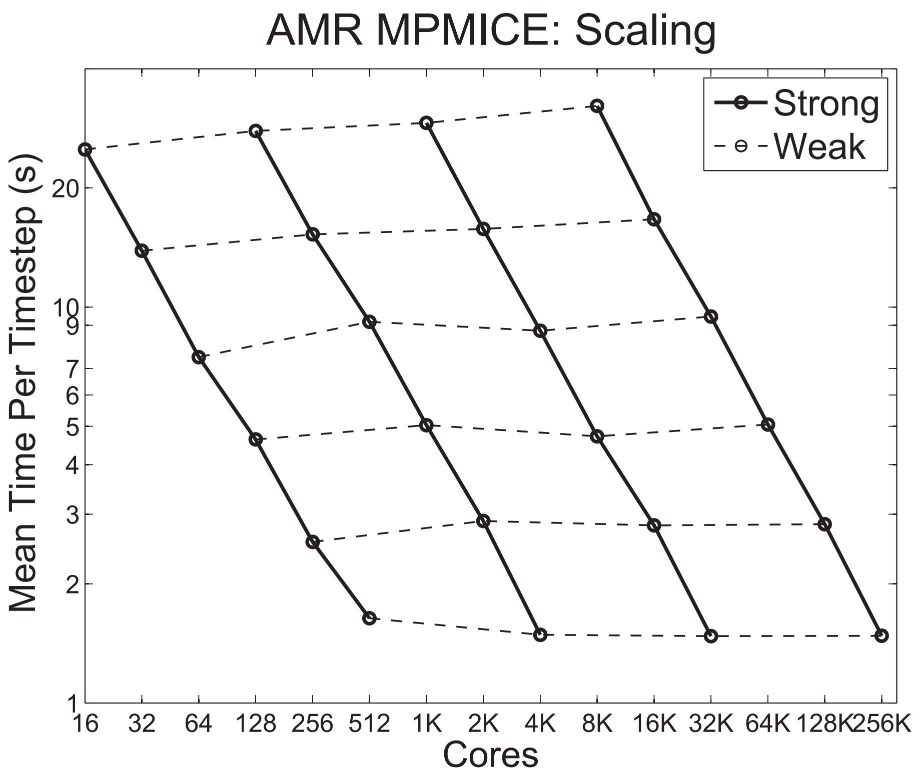


Figure 6.7. Improved AMR MPMICE Scalability Results

Uintah framework is querying and assembling variables, in particular when: 1) Mapping patches between different AMR levels, 2) Allocating new memory and copying data to this new memory to hold both patch center and halo regions when preallocation is not possible.

This approach leads to better weak scaling. Figure 6.8 shows the weak scaling efficiency compared to the previous MPI only benchmark result. The base cases to calculate efficiencies are 24 cores for MPI runs on Kraken and 64 cores for the hybrid multithreaded MPI runs on Jaguar XK6. These two series are the same as the second from bottom dashed weak scaling line in Figure 6.2 and the bottom dashed weak scaling line in Figure 6.7. The results show significant improvement of weak scaling efficiency when using hybrid multithreaded MPI approach, especially on runs with large core counts.

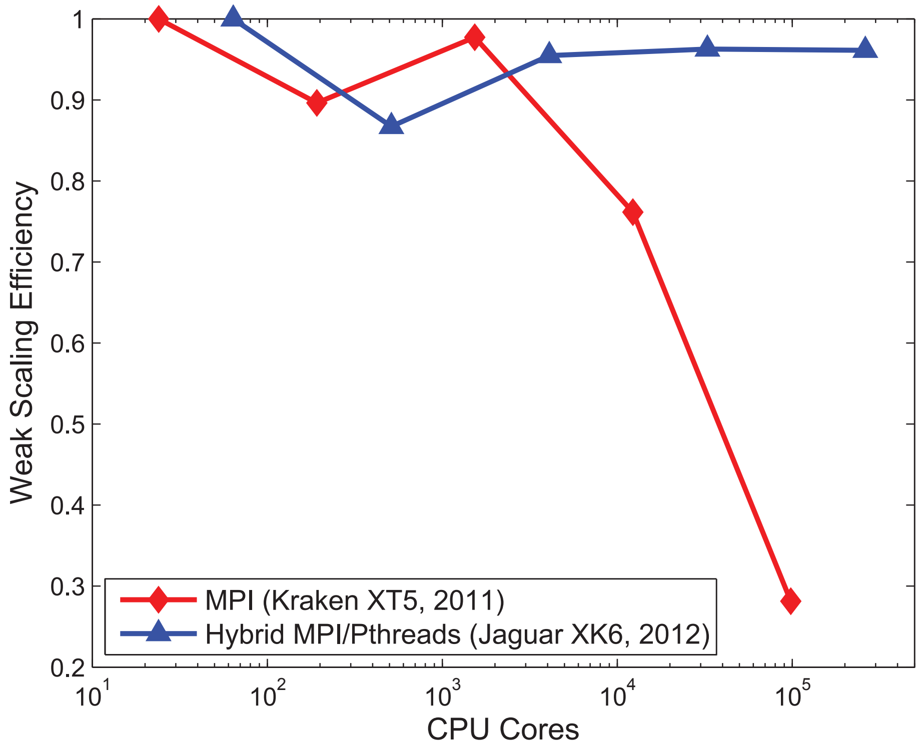


Figure 6.8. Weak Scaling Efficiency Comparison

6.5 Summary

These results presented here show great performance improvements, and also show good scalability so far on 256K cores on Jaguar Cray XK-6 by using the hybrid multithreaded MPI scheduler and the new lock-free data structures. However, alongside the lock-free data warehouse, many other shared data structures such as task queues, particle subset caches, and patch BVH trees still need to be redesigned to be lock-free. Even though these data structures are accessed less frequently than the data warehouse, the waiting time for acquiring locks related to these structures is going to grow as the number of core per node increases. The future removal of all these locks and making Uintah fully lock-free will improve the scaling here still further on present and future many-core and multicore machines. The present decentralized scheduler will also be used instead of the current master-slave model used in the Uintah GPU scheduler [52]. This will require the current GPU controller thread routines to be rewritten to guarantee thread-safety, but will allow these scalability results to be extended to heterogeneous CPU-GPU architectures.

CHAPTER 7

UNIFIED SCHEDULER - GPU SUPPORT

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 [68]. Such frameworks must address the formidable scalability and performance challenges associated with running on these systems, and must also insulate application developers from the inherent complexity of the parallelism involved. Traditional systems are now commonly augmented with graphics processing units (GPUs). Software framework designs must consider these heterogeneous architectures and additionally plan for future many-core designs.

In this chapter, we describe the evolution of Uintah's hybrid multithreaded MPI runtime system [68] to support, schedule, and execute both CPU and GPU tasks simultaneously, without a central control thread. In this study, this new design is examined in the context of a hierarchical GPU-based ray tracing radiation transport model that provides Uintah with additional capabilities for heat transfer and electromagnetic wave propagation. Preliminary results from computational experiments on TitanDev[1] are presented.

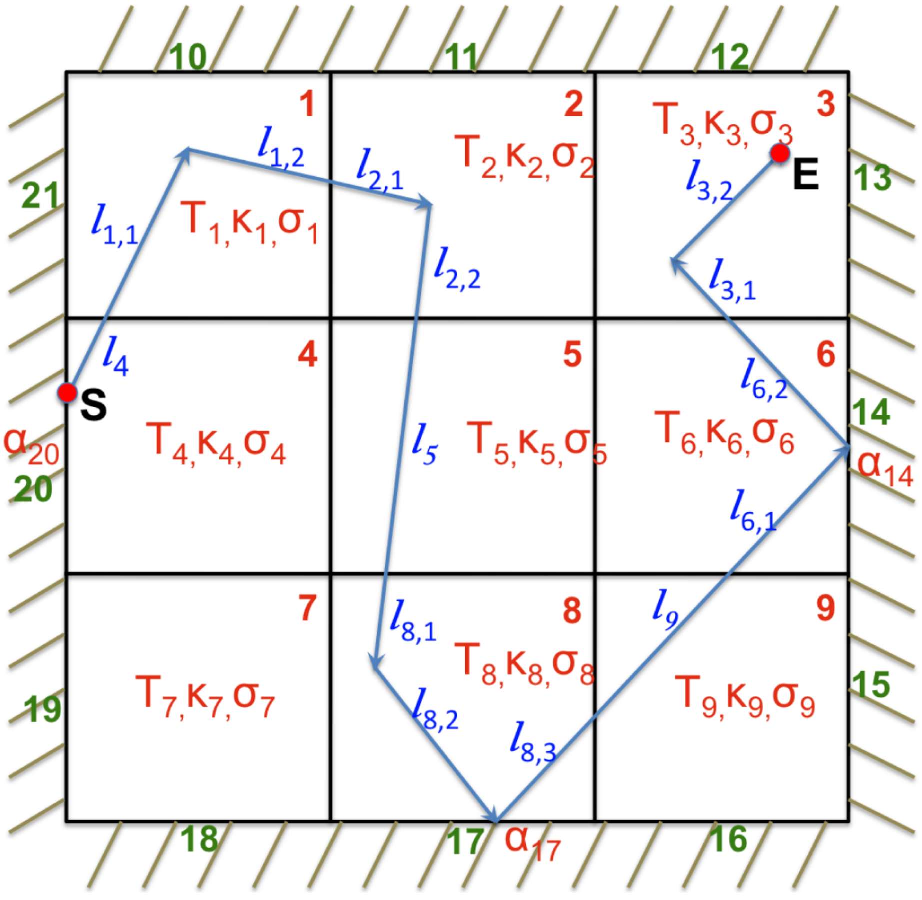
7.1 Radiation Modeling

We extended 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 [55, 78]; however, other

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

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 on using the RMCRT methods, as described by [72]. 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 [72]. 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. As Sun [92] points out, RMCRT is more amenable to domain decomposition and thus parallel implementation due to the backward nature of the process. Figure 7.1 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,



10
11
12
1
2
3
T_3, κ_3, σ_3
E
13
21
$l_{1,2}$
$l_{2,1}$
T_2, κ_2, σ_2
$l_{3,2}$
$l_{1,1}$
T_1, κ_1, σ_1
$l_{2,2}$
$l_{3,1}$
l_4
4
5
6
$l_{6,2}$
S
14
α_{20}
T_4, κ_4, σ_4
l_5
T_5, κ_5, σ_5
T_6, κ_6, σ_6
α_{14}
20
$l_{6,1}$
7
$l_{8,1}$
8
l_9
9
T_8, κ_8, σ_8
19
T_7, κ_7, σ_7
$l_{8,2}$
T_9, κ_9, σ_9
15
$l_{8,3}$
18
17
α_{17}
16

Figure 7.1. Outline of Reverse Monte Carlo Ray Tracing absorption coefficient κ_i, scattering coefficient σ_i, and appropriate path lengths $l_{i,j}$. 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 RMCRT, there are considerable challenges in the efficient implementation of RMCRT as it is an all-to-all method, where all of the geometry information and property model information for the entire computational domain must be present on each processor [92]. This nature severely limits the size of the problem that can be computed due to memory constraints, especially with large highly resolved physical domains. 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 [54]. The hybrid memory approach of Uintah 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 *data warehouse* so it encompasses the patch in a stair step fashion.

CPU-only scalability studies of the RMCRT for the benchmark problem as described by Burns and Christon [28] were run on a single level [54] with 256^3 cells, using 25 & 100 rays per cell. Each scaling run was run for 10 timesteps, 1 patch per processor, and the mean time per timestep was computed. These preliminary results show reasonable scaling up to 16K cores on Titan; above this, 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.

7.2 GPU Runtime System Design

This section describes a intermediate version of CPU-GPU scheduler which then leads to the development of a Unified scheduler in Section 7.3. In the same fashion that Uintah insulates the application developer from the parallelism its infrastructure provides via the multithreaded CPU scheduler, the hybrid CPU-GPU version also hides and carefully manages details related to GPU memory allocation and transfer. Associated with each Uintah task is a C++ method which is used to perform the actual computation. In the context of the hybrid CPU-GPU scheduler, a GPU task is represented by an additional C++ method that is used for GPU kernel setup and invocation. This design uses Nvidia CUDA C/C++ exclusively for both the Uintah infrastructure and user GPU tasks.

Central to the master-slave design of the hybrid multithreaded/MPI CPU-GPU scheduler [52] is the multistage queuing architecture for efficient scheduling of CPU and GPU tasks. The CPU-GPU scheduler utilized four task queues: an internal ready and external ready queue for CPU tasks and an additional pair of queues for the GPU: one for initially ready GPU tasks, those that have their requisite simulation variable data copies from host-to-device pending, and a second for the corresponding device-to-host data copies pending completion. It should be noted that both GPU task queues are priority queues and thus preserve a given task priority algorithm established by the scheduler itself.

The CPU-GPU scheduler also maintains a set of queues for CUDA *stream* and *event* handles (one per device representing separate CUDA 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 [52]. These *stream* and *event* handles provide a mechanism to detect completion of asynchronous memory copies without a busy wait, using `cudaEventQuery(event)`. This allows querying the status of all device work preceding the most recent CUDA 4.0 API call to `cudaEventRecord()` [2]. On systems with multiple on-node GPUs, the hybrid CPU-GPU scheduler additionally manages a CUDA calling context for each device.

First, if a task's internal dependencies were satisfied, then that task is placed in the CPU internal ready queue where it waits 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. 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. 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 [52].

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

7.3 Unified Runtime System Design

The natural design progression given the success of the original CPU-GPU scheduler as presented in [52], and the generally superior performance and potential of the decentralized model as in the previous chapter, was to extend the decentralized CPU design to heterogeneous systems, allowing all threads to process MPI sends and receives and to execute both CPU and GPU tasks concurrently without a control thread. Through this design extension, a unified multithreaded runtime system and approach to scheduling Uintah computational tasks has been developed. The Unified Scheduler and runtime system, shown in Figure 7.2, is the principal contribution in this work and allows Uintah to not only exploit current heterogeneous architectures, but also plans for emerging and future many-core designs. Much of this design path has been motivated by machines such as NSF Keeneland and the DOE Titan and NSF Stampede systems. In order to adapt the Uintah Computational Framework for hybrid CPU-GPU architectures, we elected [52] to use Nvidia CUDA C/C++ for numerous reasons; namely, looking at the upgrade path of the Jaguar XK6 system to Titan and also the Keeneland Initial

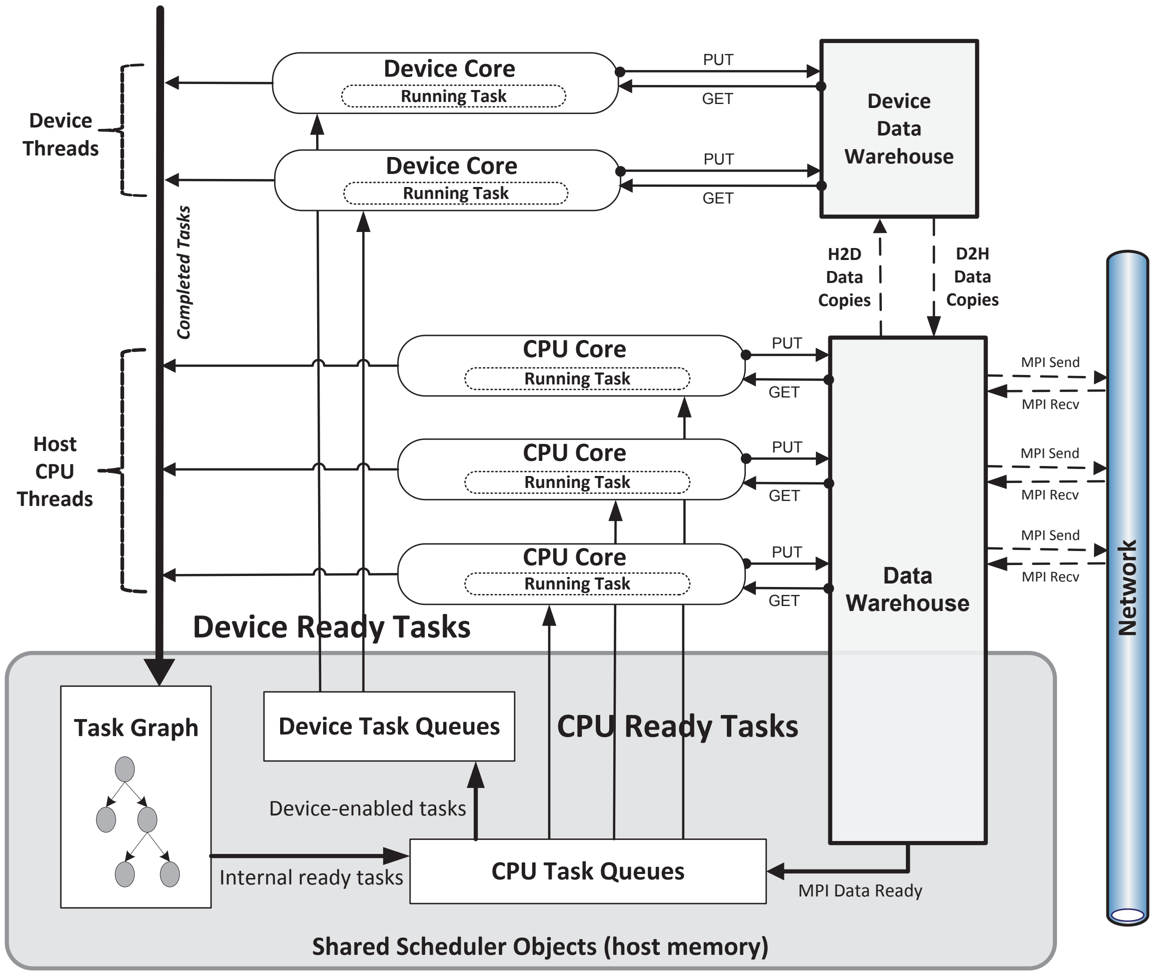


Figure 7.2. Uintah Unified Scheduler

Delivery System (KIDS), we see a trend in the use or planned use of Nvidia GPUs.

7.3.1 Decentralized Model

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

In the same way that access to CPU-only task data in the *data warehouse* must be guaranteed to be thread-safe, access to the current data structures that track corresponding GPU data must be similarly protected. As described in [52], before a GPU task is placed into the GPU host-to-device copy queue, the Unified scheduler initiates the device memory allocations and asynchronous host-to-device data copies for the task's simulation variables. To carry out these operations, the data warehouse must be queried by the Unified scheduler for the location and size of the data required for computation on the GPU. It is here that space in the data warehouse for the result of the GPU computation is also allocated on the host. These operations produce sets of pointers to device and host memory for both a task's requires(input) and computes(output) variables that must be managed. Additionally, host memory pointers are registered by the Unified scheduler to be copied to the GPU via DMA using a call to `cudaHostRegister()` combined with the `cudaHostRegisterPortable` flag from the CUDA 4.1 API. This creates page-locked memory from preallocated host memory that is considered page-locked by all CUDA contexts and ultimately accelerates PCIe transfers and eliminates resetting of CUDA contexts when referencing the registered host memory [52]. This information must also be tracked in order to cleanly unregister the page-locked host memory when a task has completed. All of this pointer information is kept in a set of maps maintained by the Unified scheduler. Access to each of these maps must also be guaranteed thread-safe. Here, access to these data structures is currently infrequent as the overall number of GPU-enabled Uintah tasks is relatively low. Hence access to the maps can be regulated by the same read-write locks used in the task queues without significant overhead. However, as more Uintah tasks are ported to the GPU, this could become a potential bottleneck. This issue is then addressed through the creation of a GPU data warehouse in the following section that encapsulates these maps and uses the similar data structures and algorithms used in the current Uintah data warehouse to support concurrent lock-free queries in the GPU task kernel.

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

The run method for each thread also exposes a potential performance bottleneck in that the Unified scheduler contains a critical section that is protected by mutex, a

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

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

7.3.2 GPU Data Warehouse

We developed a set of new Uintah GPU task APIs to make the Uintah simulation component developer easier to write GPU task kernel. As shown in Figure 7.3, this new Uintah GPU interfaces included new and old GPU data warehouses

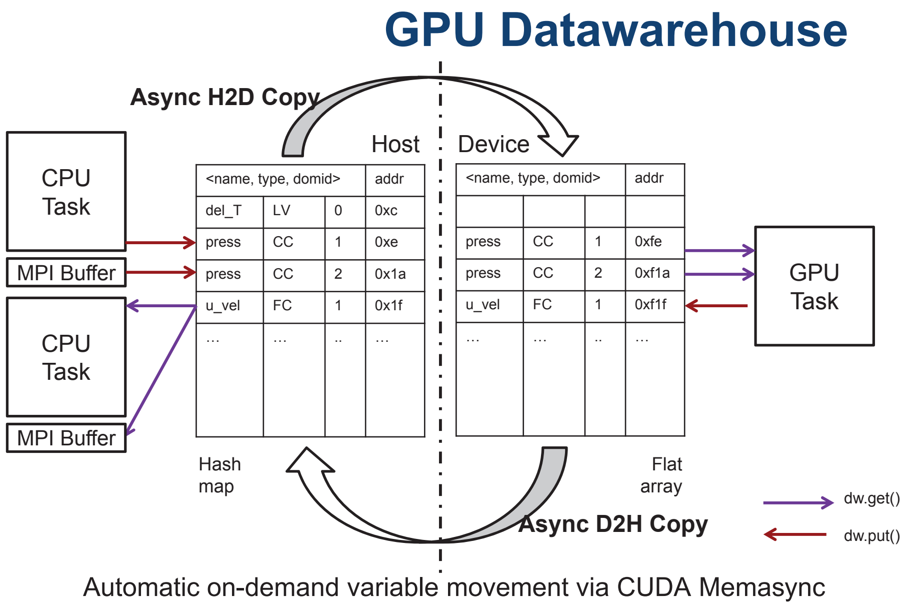


Figure 7.3. Uintah GPU Data Warehouse Implementation

which are similar to the CPU data warehouses and working as a variable memory manger. The GPU data warehouse is a variable directory that maps a variable name and patch or level id key pair to the variable's GPU memory locations. Once correct requires and computes are specified, the Uintah framework will do the CudaMalloc/H2D/D2H memory copy asynchronously and automatically. In this way, memory movement can overlap with GPU kernel execution, CPU task execution, and MPI communications to hide latency. As each Uintah GPU task uses one CUDA stream, a kernel will be launched when all its required variables are ready in the device memory and multiple kernels and two-way memory copies can be executed concurrently.

In a Uintah GPU tasks device code, the component developer can call get function to query the preloaded variable location from the GPU data warehouse by using the variable name, patch ID, and marital index key in device function. As the device memory is now managed by the data warehouse, variables can now exist beyond a task kernel execution and a timestep, when CUDA memcopy can be skipped when variables already exist on GPU to reduce PCI-E memory bandwidth usage. In addition, when there are multiple GPUs that exist in a single node, the GPU data warehouse also knows the variables' location across multiple devices so GPUDirect could be used to avoid memory copies through the host CPU.

7.4 Experimental Results

In evaluating the relative performance improvements of the Unified scheduler, several initial tests were performed. The first test looks at CPU only data from a single 32-core Cray XE6 node and compares execution times of the CPU-only master-slave model [68] to the new Unified scheduler. The second test looks at data from a single 12-core, 3-GPU heterogeneous node, comparing execution times of the hybrid CPU-GPU master-slave model [52] to the new Unified scheduler. Lastly, we plot scaling data from runs on the DOE Jaguar system (CPU-only) and compare Uintah's MPI-only scheduler to its multithreaded schedulers (master-slave model). These plots also include data from TitanDev, comparing GPU and CPU implementations of the RMCRT problem from [52].

Table 7.1 shows a CPU-only performance comparison between the master-slave and Unified models on a single Cray XE6 node (two 16-core AMD Opteron 6200 Series processors each with Interlagos cores @2.6GHz) for a combined MPMICE problem using AMR. In this case, the Unified model outperforms the master-slave model on all runs up to 32 cores. By monitoring the CPU utilization of each core, it was confirmed that the Unified model solved the issue of load imbalance of the core that runs the control thread and the cores running the worker threads. Furthermore, when one master control thread with 1, 3, 7, 15, and 31 worker threads is used, the CPU loads on the control thread increase linearly and are about 0.3%, 0.7%, 1.7%, 3.0%, and 6.9%, respectively. There is an increase in master control thread CPU usage with increasing numbers of worker threads.

Table 7.2 shows a hybrid CPU-GPU performance comparison between the master-slave and Unified models on a 12-core heterogeneous node (two Intel Xeon X5650 processors each with Westmere 6-core @2.67GHz, 2 Nvidia Tesla C2070 GPUs and 1 Nvidia GeForce 570 GTX GPU) for the GPU-enabled Reverse Monte Carlo Ray Tracer (RMCRT) presented in [52] with 25 rays per cell and a problem size of 41^3. This is the benchmark problem from [28]. In this case, the Unified model outperforms the master-slave model on all runs up to 12 cores. These results also confirm that the performance bottleneck found in the decentralized model is even more pronounced in the presence of additional GPU tasks, with performance when using 2 and 12 threads respectively being 16% to 37% faster for the Unified model for this problem.

Table 7.1. Execution Time: CPU-only Master-Slave vs Unified

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

Table 7.2. Execution Time: CPU-GPU Master-Slave vs Unified

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

Figure 7.4 shows the Uintah strong scaling results when using MPI-only, multithreaded MPI and multithreaded MPI with GPU schedulers on two different problems: AMR MPMICE and RMCRT Raytracing. In these results, the processing unit baseline *N* for AMR MPMICE, Thread/MPI AMR MPMICE, Thread/MPI/GPU RayTracing, and Thread/MPI/GPU RayTracing are 6144 CPU cores, 8192 CPU cores, 16 CPU cores, and 16 CPU cores with 1 GPU, respectively. For a large-scale MPMICE AMR problem, Uintah originally scaled up to 96K CPU cores with MPI only on the DOE Jaguar XK6 system. By using the multithreaded MPI scheduler (decentralized), Uintah can achieve significantly better scalability, up to 256K CPU cores on Jaguar. This simulation used 3.62 billion particles with three refinement grid levels. For the GPU-enabled Reverse Monte Carlo Ray Tracer (RMCRT) problem, 100 rays per cell were used with a problem size of 128^3.

Figure 7.5 isolates the CPU vs GPU scaling results in an effort to better clarify the scaling breakdown in the GPU implementation of the RMCRT problem. In these results, the processing unit baseline *N* for Thread/MPI RayTracing and Thread/MPI/GPU RayTracing are 16 CPU cores and 16 CPU cores with 1 GPU,

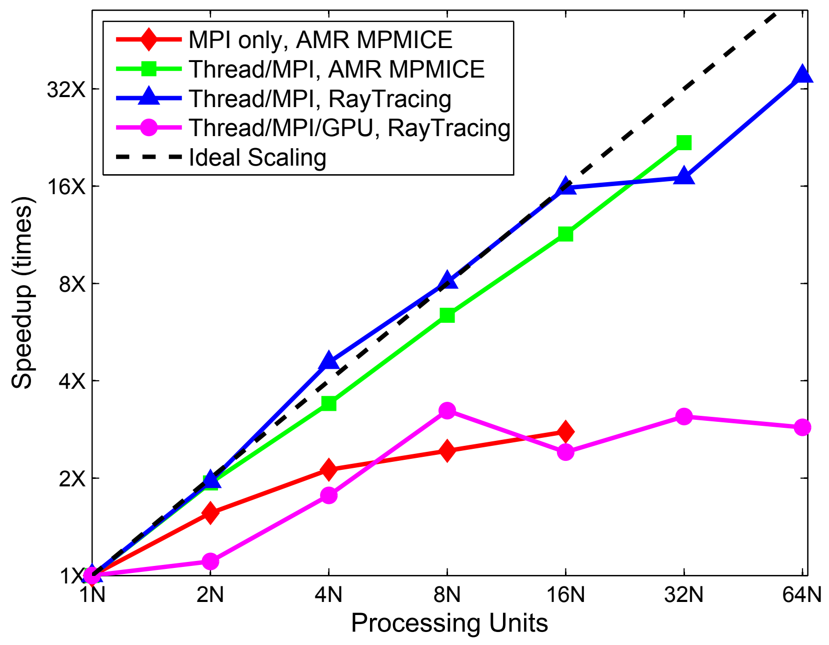


Figure 7.4. Uintah Scalability Comparisons

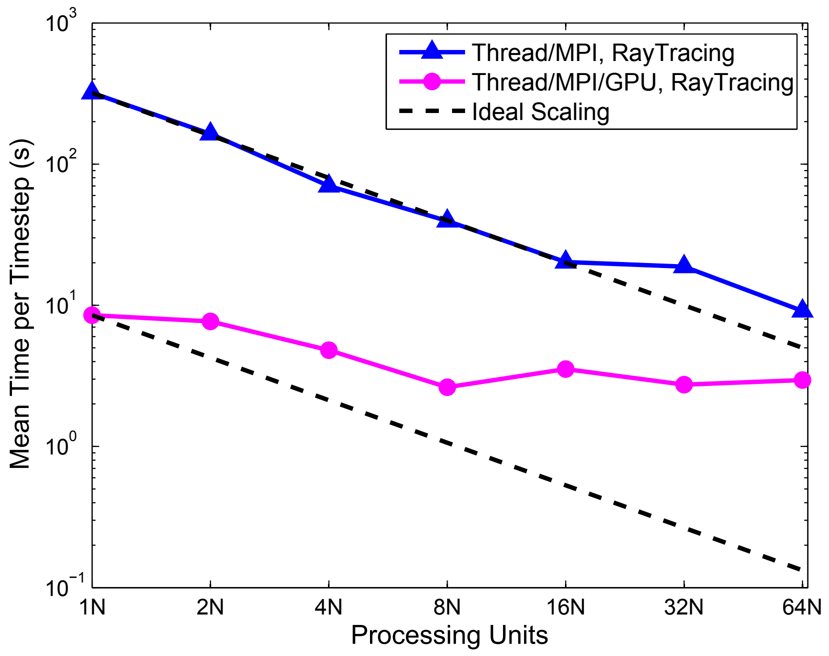


Figure 7.5. RMCRT Scalability with CPU/GPU

respectively. Although the mean time per timestep for the GPU implementation is still considerably lower than the CPU implementation at this point (up to 64 GPUs), ultimately there is insufficient work, and the GPU implementation is subject to the same communication costs as the CPU implementation [52] due to the all-to-all nature involved with radiation modeling.

7.5 Summary

We have shown that our Unified multithreaded scheduler design is capable of utilizing all on-node computational resources on current and emerging multicore and heterogeneous systems efficiently and automatically. This work has also illustrated how our Unified design keeps the application developer insulated from the multiple levels of parallelism inherent in heterogeneous systems by a separation of the user implemented tasks from the Uintah runtime system. We have also shown preliminary results that confirm the decentralized multithreaded design used in the Unified scheduler not only outperforms previous designs, but is also well positioned to efficiently exploit emerging and future many-core architectures.

Through the development of Unitah's Unified scheduler, the data warehouse lock was seen to be the largest single source of overhead based on timing results on Uintah read-write locks, and in the way the data warehouse has been made efficient with a lock-free implementation, we are also considering an efficient, lock-free GPU data warehouse. Additionally, we would like to pursue designing a mechanism for the Unified scheduler to decide at runtime whether to run a particular task on a CPU core or on a GPU, with plans to extend Uintah's scheduler to support such coprocessor designs as well. And, with the eminent arrival of the massive-scale heterogenous DOE Titan in late 2012, larger scaling runs to further test our Unified scheduler and runtime design will also be performed. Given that Titan will potentially have 18K or more Nvidia Kepler K20 GPUs, we will also be leveraging the advanced features available through CUDA 5.0 and Kepler, specifically Dynamic Parallelism to further improve GPU utilization by the Uintah framework.

CHAPTER 8

UNIFIED SCHEDULER - MIC SUPPORT

In the previous chapter, we described how Uintah has been extended to run on heterogenous CPU/GPU architectures while this is one solution to the fact that individual processing units consisting solely of CPU's are no longer increasing in speed from generation to generation, while the demands on system architects for increased density and power efficiency steadily increase. With these demands in mind, traditional systems are now also augmented with an increasing number of graphics processing units or coprocessors such as the Intel Xeon Phi. This architectural trend is most notable in machines such as the XSEDE resources Stampede[1].

In this chapter, we detail our experiences moving Uintah onto the TACC Stampede system with its Intel Xeon Phi coprocessors using the Uintah Unified Scheduler and Runtime System to support, schedule, and execute both host CPU and coprocessor tasks simultaneously. Throughout this chapter, we refer to the **Intel Xeon Phi Coprocessor (MIC Architecture)** as **Xeon Phi** when referring to the coprocessor in general, and **MIC** when we talk specifically about the architecture of the Xeon Phi. We explore the various usage models provided by the Xeon Phi with a key aim of understanding the portability of a general purpose framework such as Uintah on such an architecture. Although the Xeon Phi symmetric model is given focus in this work, as it best fits the current Uintah model, our work here clearly illustrates that the Directed Acyclic Graph or DAG [23] approach used by Uintah provides the ability to leverage all usage models provided by the

[1]Stampede is a Dell PowerEdge C8220 cluster, administered by TACC with 6,400+ Dell PowerEdge server nodes, each with 32GB memory, 2 Intel Xeon E5 (8-core Sandy Bridge) processors, and an Intel Xeon Phi Coprocessor (MIC Architecture) [8].

Xeon Phi. Ultimately, we provide results from computational experiments using the host-only, native, and symmetric models using two challenging computational simulations, one being an incompressible flow calculation (host only) and the other a fluid-structure interaction problem (native and symmetric models) with adaptive mesh refinement (AMR).

8.1 Xeon Phi Programming Models

Xeon Phi provides five programming models: Host-only, MIC native, offload, reverse offload, and symmetric. Our focus will be on the four models in Figure 8.1, and will not cover the reverse offload model, as it is not yet supported by the Intel MPI implementation. In the host-only model, programs run only on host CPUs in the system without any utilization of the Xeon Phi coprocessors. Host processors between multiple nodes can communicate though MPI. This model is similar to running on most other CPU-only clusters. The Xeon Phi native model uses only the Xeon Phi coprocessors in the system, disregarding the host CPUs. On a Xeon phi card, a very basic version of Linux is installed. After being compiled to MIC binary, a program can then run on the Xeon Phi directly and can use using MPI and OpenMP/Pthreads. The offload model is similar to using accelerators such as

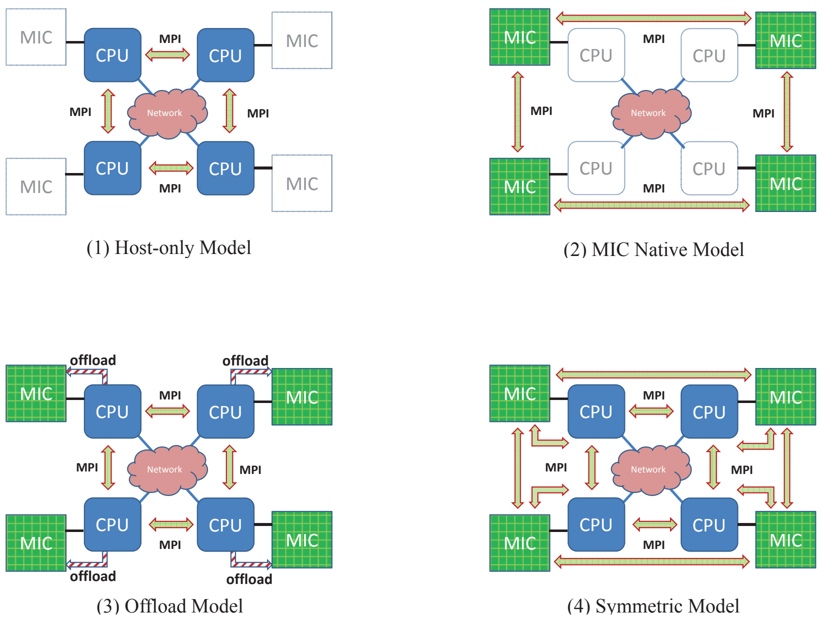


Figure 8.1. Xeon Phi Execution Models

a GPU (in conjunction with OpenACC [7]), where the program runs on host CPU and uses offload directives to run certain parts of the computation on Xeon Phi. In this model, all MPI messages are sent and received by host processor. Reverse offload is similar though to offload mode in that the offload region simply runs on host CPU while MPI ranks are run on the Xeon Phi. For the symmetric model, programs can run on both the host CPU and the Xeon Phi coprocessor card natively. MPI messages can be processed by host CPU and Xeon Phi directly.

There are two MPI libraries available on Stampede, Intel MPI and MVAPICH. MVAPICH does not yet have a build for Xeon Phi, but host-only and offload models are supported at this time. Intel MPI has both host and MIC builds and supports four MPI communication modes besides host only:

1. within a single Xeon Phi coprocessor,
2. between the Xeon Phi coprocessor and the host CPU inside one node,
3. between multiple Xeon Phi coprocessors inside one node,
4. between the Xeon Phi coprocessors and the host CPUs between several nodes.

8.2 Native Model

As the Intel Xeon Phi is based on X86 technology, porting existing code to the Xeon Phi is relatively easy. Most codes, including Uintah, can be compiled to run on the Xeon Phi by simply adding the *-mmic* compiler flag. The Uintah framework infrastructure code and most of its simulation components are written in C++, with some legacy components written in Fortran. Both C++ and Fortran are supported by the Intel compiler for the MIC architecture. The parallel programming libraries used by Uintah, MPI, and Pthread are also supported natively. However, Uintah depends on many third party libraries such as *libxml2* and *zlib*. Those libraries are not currently installed on the Xeon Phi and needed to be built. To get both Uintah and the other libraries built, cross compiling is required, as the binaries compiled with the *-mmic* compiler flag cannot run on the head nodes of Stampede. As Uintah uses autotools for its build system, only minor changes were made to support cross compiling. We were able to get a native Uintah build up and running on a single Xeon Phi card within 24 hours of having access to the machine.

When running on a single Xeon Phi card, Uintah uses both MPI and Pthreads for parallelization. When running with Pthreads on a shared memory node, Uintah also uses lock-free data structures to allow concurrent access to shared objects such as the data warehouse (a simulation variable repository) without using high-level and typically high-overhead Pthread read-write locks. This lock-free data warehouse uses built-in atomic operations that are supported in the gcc compiler such as *fetch_and_add* and *compare_and_swap*. Those gcc built-ins are not supported in earlier versions of the Intel compiler. However, this issue has been solved by using equivalent atomic operations in older Intel compilers or by using the newer Intel compiler. Figure 8.2 shows strong scaling results of the Uintah AMR MPMICE simulation on a single Xeon Phi card comparing pure MPI, Pthreads with read-write locks and Pthreads with lock-free data structures. Two MPI ranks or Pthreads per Xeon Phi core are used for this benchmark. These results show that Uintah performs and scales better when using a combination of MPI and Pthreads as opposed to an MPI-only approach.

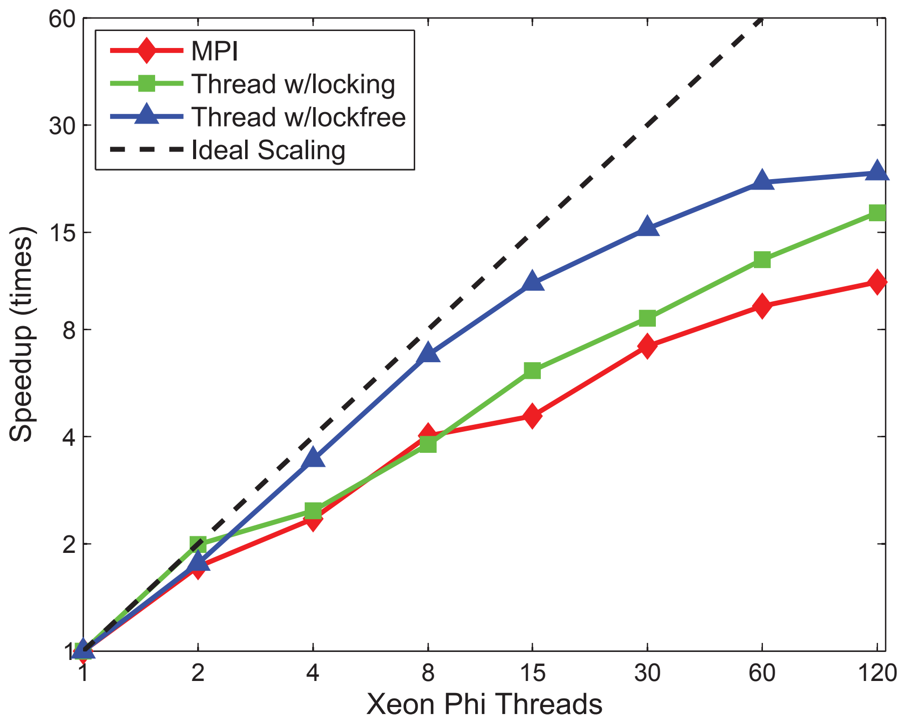


Figure 8.2. Uintah Scalability on MIC Native Model

8.3 Offload Model

Although the directive-based approach, using the Xeon Phi synchronous offload model, seems the most attractive to use initially, we discovered this model is more difficult to implement than we originally anticipated for a general purpose framework like Uintah. In order to use this pragma-based offload model, all functions called from the Xeon Phi must be defined with the offload attribute:

```
__target(mic)
```

Due to the complexity of the heavily templated Uintah code, we essentially need to define almost everything with this attribute or rewrite a particular task with a simple C/C++ structure, avoiding the complexities of the infrastructure code. For Uintah to make effective use of this model, the Xeon Phi asynchronous offload features must be used. These features include:

1. asynchronous data transfer,
2. asynchronous compute,
3. memory management without data transfer.

Using these asynchronous API offerings, PCIe latency can be hidden by overlapping MPI communication with computation on both the host CPU and the Xeon Phi coprocessor. The key component in making this work is to implement a mechanism to detect completion of the asynchronous data copies to-and-from the coprocessor. This approach is nearly a perfect analog to the mechanism created in [52] to orchestrate and manage asynchronous data copies to-and-from on-node GPUs. In the context of the Xeon Phi asynchronous offload model, an offload region can be executed asynchronously when a signal clause is included with the directive. All asynchronously offloaded data and computation can be associated with this signal clause. Detecting completion of this operation is achieved with explicit API calls. For example, the API call:

```
_Offload_signaled(mic_no, &c)
```

tests whether the computation signaled with *c* has finished. This is a nonblocking mechanism to check if offload has been completed.

Using the Xeon Phi asynchronous offload features, we simply generalize the existing *GPU* task queues to become *device* task queues and add the associated logic to the Unified Scheduler and Runtime System from [69] to become what was shown in Figure 7.2. This implementation is currently underway and testing it is part of our future work.

Uintah's DAG-based runtime system allows full utilization of all available cores on the host CPU and Xeon Phi coprocessors easily through the symmetric programming model. The simulation grid in Uintah is partitioned into hexahedral patches by a highly scalable regridder and assigned to nodes by a measurement-based load-balancer [22]. In each MPI process, the Uintah runtime system will schedule tasks on local patches by using a local task graph and data warehouse. The task graph is a DAG [23] which is compiled by making connections on task's required and computed variables. The Uintah scheduler uses the task graph to determine the order of execution, assign tasks to local computing resources, and ensure that the correct interprocess communication is performed. Uintah uses Pthreads for intranode task scheduling. Each core directly pulls tasks from multistage ready task queues without any intranode communications taking place. This runtime system is shown to fully use all available cores on-node, regardless of the number of cores.

When running with the Xeon Phi symmetric model, two binaries are required, one for host CPU(s) and one for Xeon Phi coprocessor. Since the Xeon Phi has significantly more cores than the host CPU, more threads are created in MPI ranks running on the Xeon Phi than MPI ranks running on the host CPU. In a typical Uintah run, we create 120 threads per Xeon Phi and 16 threads (one per core) for the host CPU(s). For example, to run symmetric mode, we used the following command line:

```
mpirun.hydra -n 4 ./sus -nthreads 16 input.ups;
-n 4 ./sus-mic -nthreads 120 input.ups
```

This will run Uintah on 4 CPU hosts with 16 threads per host and 4 Xeon Phi cards with 120 threads per card at the same time.

8.4 Symmetric Model

With some MPI ranks running on one architecture while other MPI ranks run on a different architecture, it is important to make sure that all ranks execute in a consistent way. Errors may happen when control logic-based results differ between the Xeon Phi and host CPU, such as MPI messages based on floating point calculations. In Uintah, a common operation when running with AMR is to find cells in a finer level based on a point that is computed from coarser level, which are then sent from the finer level cells to coarser level. Figure 8.3 shows a real AMR example in Uintah, in which a point is computed by the division of two double precision numbers that are known globally to all MPI ranks. The algorithm guaranteed that all ranks should compute this point as the same value such that the sending side will pick the same interval of cells as the receiving side (left side: host-only model). However, while the algorithm is consistent, when one rank runs on the Xeon Phi, the computed value may be inconsistent. In this example, the CPU side receiver picks intervals beginning with 162; however, the Xeon Phi sender picks interval beginning with 161. Hence, an MPI buffer mismatch error occurs due to a floating point operation that is not consistent between the Xeon Phi

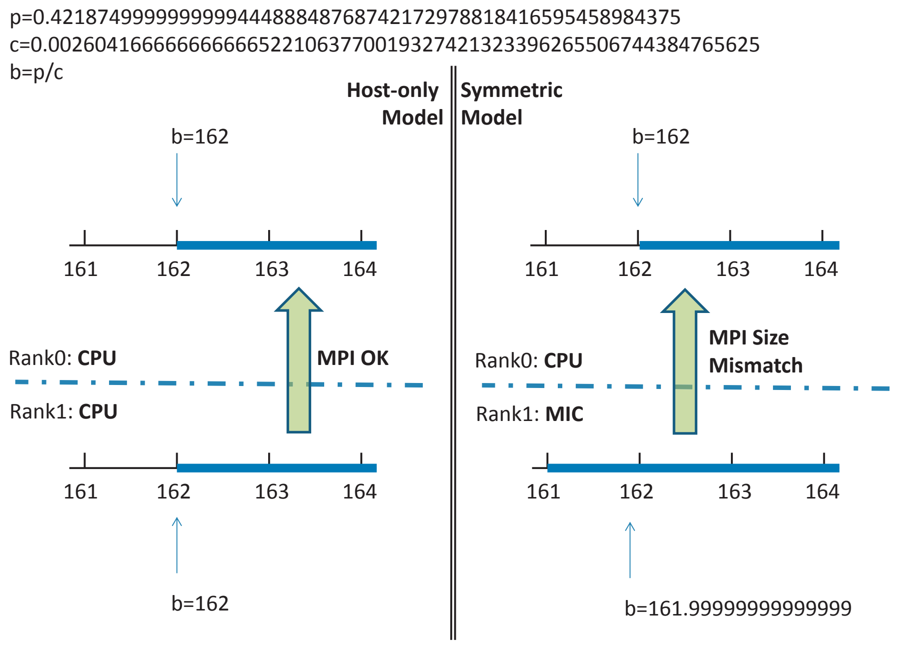


Figure 8.3. MPI Error from a Floating Point Inconsistency

coprocessor and host CPU (right side: symmetric model). To fix this error, a higher precision compiler flag was used at the cost of lower performance for this method.

Figure 8.4 shows preliminary scaling results on Stampede with multiple Xeon Phi cards and host nodes using the symmetric model. Using this model, Uintah can strong scale up to 16 Xeon Phi cards (the current Stampede MIC development queue limit); however, the scaling efficiency is limited due to load imbalance between the host CPU and Xeon Phi. The reason being that the Uintah load balancer currently assigns host MPI ranks and Xeon Phi ranks the same workload. We detected a load imbalance up to 60% for this benchmark. The workload ratio of CPU to Xeon Phi should be computed based on profiling. We will develop a new load balancer to profile and predict the work load on the host CPU and Xeon Phi card separately to solve this problem.

8.5 Summary

We have described our preliminary experiences with Stampede using the Uintah Computational Framework with an emphasis on understanding the performance

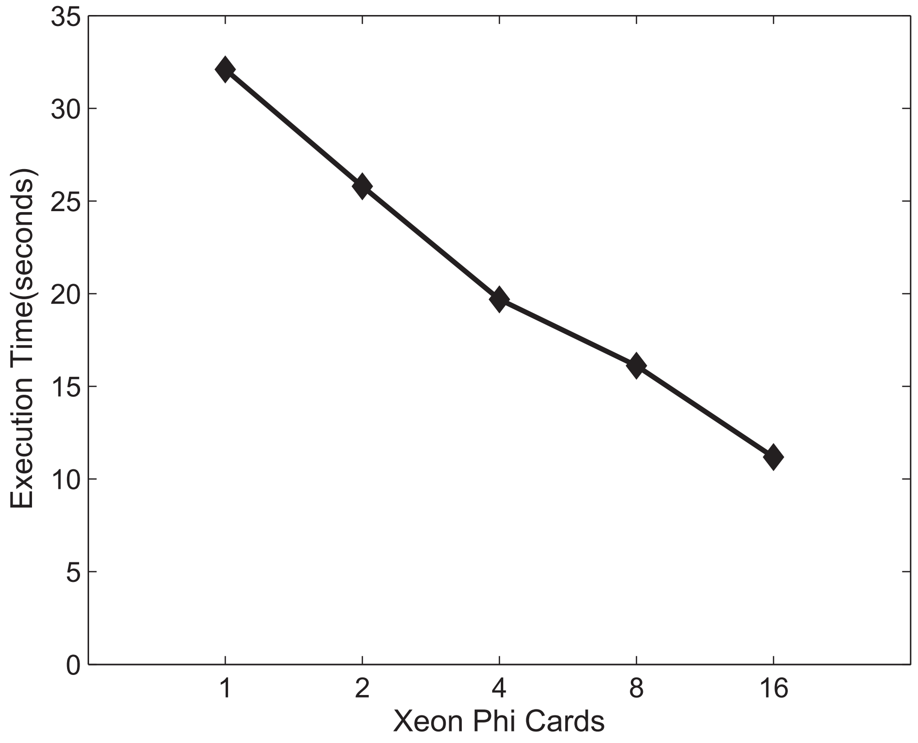


Figure 8.4. Uintah Scalability on MIC Symmetric Model

implications of the new Intel Xeon Phi Coprocessor (MIC Architecture). Using only the host CPUs for computations, Stampede is nearly 3X faster than Kraken for a complex reacting flow CFD calculation. The Uintah architecture has a runtime environment which has been shown to be highly adaptable to the heterogeneous architectures that are emerging in the high performance computing world [52, 69]. This adaptability has allowed Uintah to utilize the range of usage models provided by the Xeon Phi. Of these usage models, we found the symmetric model to best fit Uintah, and required only very small modifications to the Uintah runtime system to use both the host CPUs and Xeon Phi together. Using the Xeon Phi symmetric model yielded good strong scaling characteristics up to 16 Xeon Phi cards (the Stampede MIC development queue limit at the time).

Due to different performance characteristics between the host CPU and the Xeon Phi, our scaling efficiency was limited. This will require us to develop an improved load balancer as part of our future work on Stampede to make efficient use of the Xeon Phi symmetric model. Specifically, the load balancer needs to be updated to distribute a given workload according to which processing unit an MPI process is running on. This will expand the current forecast method to profile the host CPU and Xeon Phi separately, as the Xeon Phi and host CPU have different levels of concurrency. For the Xeon Phi, finer patch sizes should be used to keep the many available threads busy and for the host CPU, larger patches are needed to better utilize the larger cache. This change will require the Uintah regridder to be able to generate different patch sizes based on the target processing unit.

To efficiently use the Xeon Phi asynchronous offload model, work is now underway within the Uintah runtime system to generalize its existing *GPU* task queues to become *device* task queues with associated logic. Using this design, we hope to provide the Uintah framework with an additional way to achieve high performance from the Xeon Phi coprocessor. We have also discovered the necessity in making use of the long vector units available on the Xeon Phi, and will so investigate explicitly using its 512-bit vector instructions as the C++ iterator loops currently used throughout Uintah cannot be easily be optimized automatically by the compiler.

CHAPTER 9

PORTABILITY AND SCALABILITY

The aim in this chapter is to illustrate that this combination of portability and scalability can be demonstrated with the Uintah software on three of the seven fastest computers as measured by the top 500 list of November 2012 [4], DOE's Titan and Mira and NSF's Stampede, [5, 8, 10]. These machines make uses of three very different processors and networks and will be discussed in this chapter. Two of the machines, Titan and Stampede, have GPU accelerators and Intel Xeon Phi coprocessors, respectively. The approach used here will be to take three representative and challenging Uintah applications codes and to examine their scalability and performance on these very different machines. The three applications are:

1. Fluid-structure interaction with adaptive mesh refinement in an example used by [67] and an explosive array simulation in [18];
2. Radiation modeling through raytracing with significant amounts of global communication as used by [52] on a variety of GPU accelerators and CPUs; and
3. Turbulent combustion on a fixed-mesh requiring large-scale linear solves with the Hypre iterative solver [84].

These three applications have very different communications requirements and work patterns. The approach used here will be to run these three applications at large scale on each of the target machines and to demonstrate how the Uintah runtime manages to achieve scalability through adaptive execution of its task graph and in the context of three different communications patterns for each of the three applications. The salient features of three target architectures and the porting of the Uintah code from the Titan architecture to the other two architectures are described in Section 9.1. Section 9.2 will describe the simulation components of the Uintah used to solve problems and analysis of the components' example communications and task execution patterns on the three machines. In Section 9.3, the scalability and performance results obtained will be given with an analysis of the different cases to show how the task graph execution pattern adaptively varies to achieve scalability. Overall we will show that with these applications, we are able to achieve good scalability on machines like Mira and Stampede when coming from our starting point of scalability on Titan [67]. Moreover, such scalability comes without significant porting effort.

9.1 Target Architectures

The three machines considered here illustrate some of the architectural differences in processor and network performance that may give rise to portability challenges. Two of these machines, Titan and Stampede, have been introduced briefly and used for benchmark in previous chapters. In this section, we compare those three machines together in details. The main architectural features of these machines are summarized in Table 9.1.

9.1.1 Titan

Titan is currently the fastest machine of the Top 500 list from November 2012 [4] with a theoretical peak of 27 petaflops and ranked third on the Green 500 list for energy efficiency. Each heterogeneous Cray XK7 node is equipped with a 16-core AMD Opteron 6274 processor running at 2.2 GHz, 32 gigabytes of DDR3

Table 9.1. System Specifications: Titan, Stampede, and Mira

SYSTEM	Titan	Stampede	Mira
Vendor / Type	Cray XK7	Dell Zeus C8220z	IBM Blue Gene/Q
CPU	AMD Opteron 6200	Intel Xeon E5-2680 Intel Xeon Phi SE10P	PowerPC A2
Cores	299,008	102,400 (host) 390,400 (phi)	786,432
Accel / Co-proc	Nvidia Tesla K20	Intel Xeon Phi	none
Mem per node	32GB	32GB	16GB
Interconnect	Gemini	InfiniBand	5D Torus

memory, and a single Nvidia Tesla K20 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. Even though the GPUs have a slower clock speed (732 MHz versus 2.2 GHz) than the CPUs, each GPU contains 2,688 CUDA cores. The overall system design was to use the CPU cores to allocate tasks to the GPUs rather than directly processing the data as in conventional supercomputers. Titan's Cray Gemini network is a 3D Torus with a latency of about 1.4 μseconds, with a peak of over 20 GB/second of injection bandwidth per node and a memory bandwidth of up to 52 GB/second per node.

From a software development perspective, Titan offers perhaps the greatest portability challenge for computational scientists to efficiently run on both host CPU and GPU. The Uintah Framework has been extended to incorporate a new task scheduler which offers three different ways of scheduling tasks to include MPI, threads, and GPUs. To harness the computational power of the GPU, a new GPU kernel must be developed for each task. However, the Uintah framework infrastructure provides a seamless way of using the CPUs to allocate tasks and move data back and forth to the GPU without explicit calls from the computational stack.

9.1.2 Stampede

The NSF Stampede machine is ranked seventh in the top 500 [4] and provides an interesting alternative to Titan as it contains Intel's new coprocessor technology, the Xeon Phi. Each of the 6400 Stampede nodes has two eight core Xeon E5-2680 operating at 2.7GHz with a 61 core Intel Xeon Phi coprocessor with cores operating at 1.0GHz. The system interconnect is via a fat-tree FDR InfiniBand (IB) interconnect, [9], with a bandwidth of 56GB/second.

Stampede provides five programming models: Host-only, MIC native, offload, reverse offload, and symmetric. Uintah currently only uses the host-only, native, and symmetric models, as these were the fastest and most portable options, as described in [70]. A Uintah MIC offload model scheduler is under development.

In the host-only model, programs run only on host CPUs in the system without any utilization of the Xeon Phi coprocessors. Host processors between multiple nodes can communicate though MPI. This model is similar to running on most other CPU-only clusters. For the symmetric model, programs can run on both the host CPU and the Xeon Phi coprocessor card natively. MPI messages can be processed by host CPU and Xeon Phi directly. Minor modifications were made to the Uintah Framework to incorporate the hybrid scheduler which can schedule tasks on Xeon Phi in the same manner as for the host. The advantage of the Xeon Phi coprocessor over the Titan GPU is that computational tasks running on the Xeon Phi do not have to be developed from scratch, which is what is required for tasks running on Titan's GPUs.

9.1.3 Mira

The Mira system is a new DOE open science 10-petaflop IBM Blue Gene/Q system installed at Argonne National Laboratory. Mira is designed to solve large-scale problems and has low power requirements and was ranked the fourth fastest supercomputer in the world as recorded by top500.org in November 2012 [4]. Each Mira compute node is equipped with 16 PowerPC A2 core processors running at 1.6GHz and 16 GB of SDRAM-DDR3 memory. The simple, low-power Blue Gene /Q cores support 4 hardware threads per core. With in-order execution, there is no instruction-level parallelism, so the key goal is to use multiple threads per core to maximize instruction throughput. The challenge in using Mira is that the lower floating point performance of the cores requires the use of a larger core count which in turn has the potential to stress a communications network that is very different from those of the other two machines considered here. With the Blue Gene/Q software stack including standard runtime libraries for C, C++ and Fortran, Uintah was able to run on Mira without modification. Only minor changes to the Uintah build system were necessary to recognize Blue Gene/Q-specific installation locations for MPI and compiler wrappers.

As Mira's power PC cores run at only 1.6GHz with a much simplified instruction set as compared to the Intel processors or AMD processors as used in Titan or Stampede, there are substantial differences in performance. The communications network on Mira 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. This performance is offset by having only 1GB of memory per core, which can be problematic for certain applications, as we will see in the scaling results. Thus Mira has perhaps the best communications bandwidth per node relative to its computational power, and Stampede perhaps the (relatively) weakest of the three, while Titan is perhaps in between unless substantial use is made of its GPU cards.

9.2 Simulation Components

The three representative problems cover a broad range of typical Uintah applications ranging from fluid-structure interaction to turbulent combustion. In order to add a deliberate contrast to these cases, we have also included the modeling of radiation using raytracing. A massively parallel implementation of this last problem involves severe challenges in that there is a substantial amount of global communication.

9.2.1 MPMICE

As noted in Chapter 6, fluid-structure-interaction problems represent an important and computationally demanding class of problems that have been part of the landscape for which Uintah was originally conceived. Broadly speaking, fluid-structure-interaction problems require the solution of the general multimaterial computational fluid dynamics (CFD) formulation coupled to a solid mechanics computation. The MPMICE component uses the algorithm [47] to solve the governing multimaterial formulation for the Navier-Stokes equations coupled to the MPM Lagrangian particle method for discretizing the solid mechanics. Additional subcomponents are implemented such as various equations of state, constitutive models, and solids$\rightarrow$gas reaction models.

The challenges with the scalability of this problem arise with the complex combination of solids, fluids, and mesh refinement; see [67] for a full description of the changes to the runtime system needed in order to achieve scalability on this problem and others like it. Given the difficulty of achieving scalability, it is by no means clear that it will be possible to achieve scalability on all three of the target machines.

Figure 9.1 shows the send and receive volume time distribution of communications for the case of three sample timesteps as three separate lines. The top three subfigures show the data volumes sent over time by a Titan, Stampede, and Mira node while the bottom three subfigures show the data received by a Titan, Stampede, and Mira node. In each timestep, the wall clock time is normalized and divided into 30 equal intervals. The MPI message volume for each interval is plotted as the y axis. The initial peak of data sent at the left of the top three subfigures shows the transmission of the data that was computed at the end of the last timestep. For example, as soon as current timestep starts, all the ghost cell data from old data warehouse can be sent out immediately. Most of this initial data sent involves local communication and can be posted immediately.

However, due to the network latency and bandwidth limitations, those messages continue to be received over several of the 30 time intervals shown. The relative delay in receiving these initial sends on Stampede is longer than on the other two machines. For Stampede, the send and receives use the Sandy Bridge

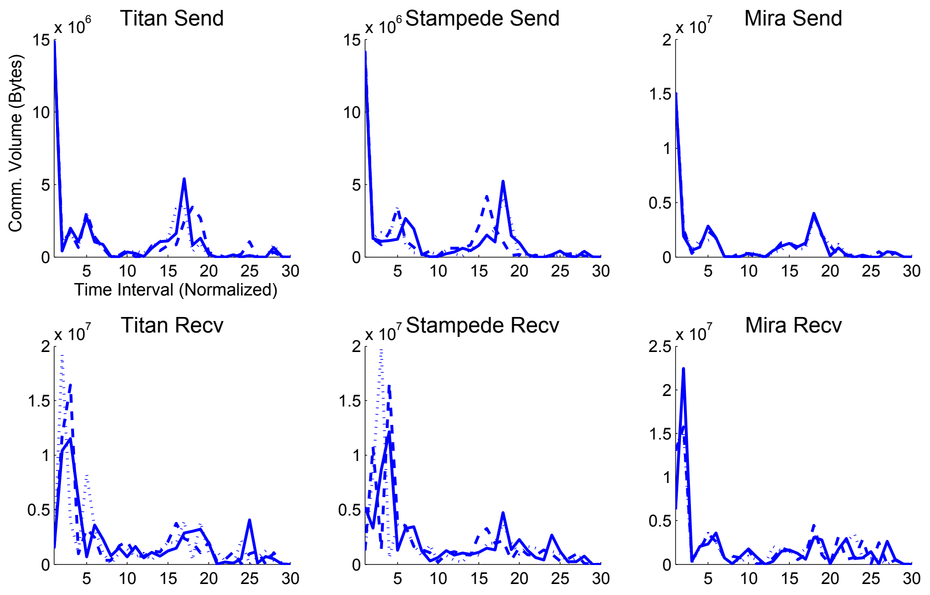


Figure 9.1. Communication Measurement of AMR MPMICE

nodes via IB. The top three subfigures show several later peaks of data sent after the initial sends. All those later peaks are due to the requirement to send ghost-cell data on newly computed variables by later tasks in the same timestep. By observing the receive side, we can see the receiving delays for the later peaks vary both from machine to machine and from timestep to timestep. Those changes may be related to many possible situations, e.g., network bandwidth usage of other users in the same machine which is hard to predict by using traditional static analysis of DAG's critical path. Uintah uses dynamic task scheduling that can automatically pick ready tasks to overlap those unpredictable communication delays.

This overlapping is done by the Uintah task scheduler moving later tasks to execute earlier than would otherwise be the case. Figure 9.2 shows the real scheduling task order in the y direction and its ordinal designed task order in the x direction. The solid line represents how tasks will be executed when static scheduling is used. The scatter points (x) show the task execution order when the multiqueue scheduler is used. All tasks below the solid diagonal line are executed earlier than they would be if static execution is used. All tasks above the solid diagonal line are executed later than they would have been if static execution had been used.

The MPMICE scheduling results show that many tasks are moved to a region close to x-axis. For example, some task originally designed to be execute as in between the $1300th$ and $1400th$ tasks are moved to approximately the first 100 tasks to be run. In this way, the time that would be spent waiting for initial MPI sends to arrive is hidden. We also see that for Stampede, more tasks are moved to be executed earlier than on Titan and Mira; for example, see tasks numbered from $500 - 700$ for static execution. This observation matches the comment that some messages are delayed in Stampede, so the scheduler can move tasks to overlap this delay accordingly. Overall, this shows how the task execution order depends on a combination of the core clock speeds and the communications performance.

9.2.2 ARCHES

The ARCHES component is a multiphysics Large Eddy Simulation (LES) numerical algorithm used to solve for the reacting flow field and heat transfer arising in

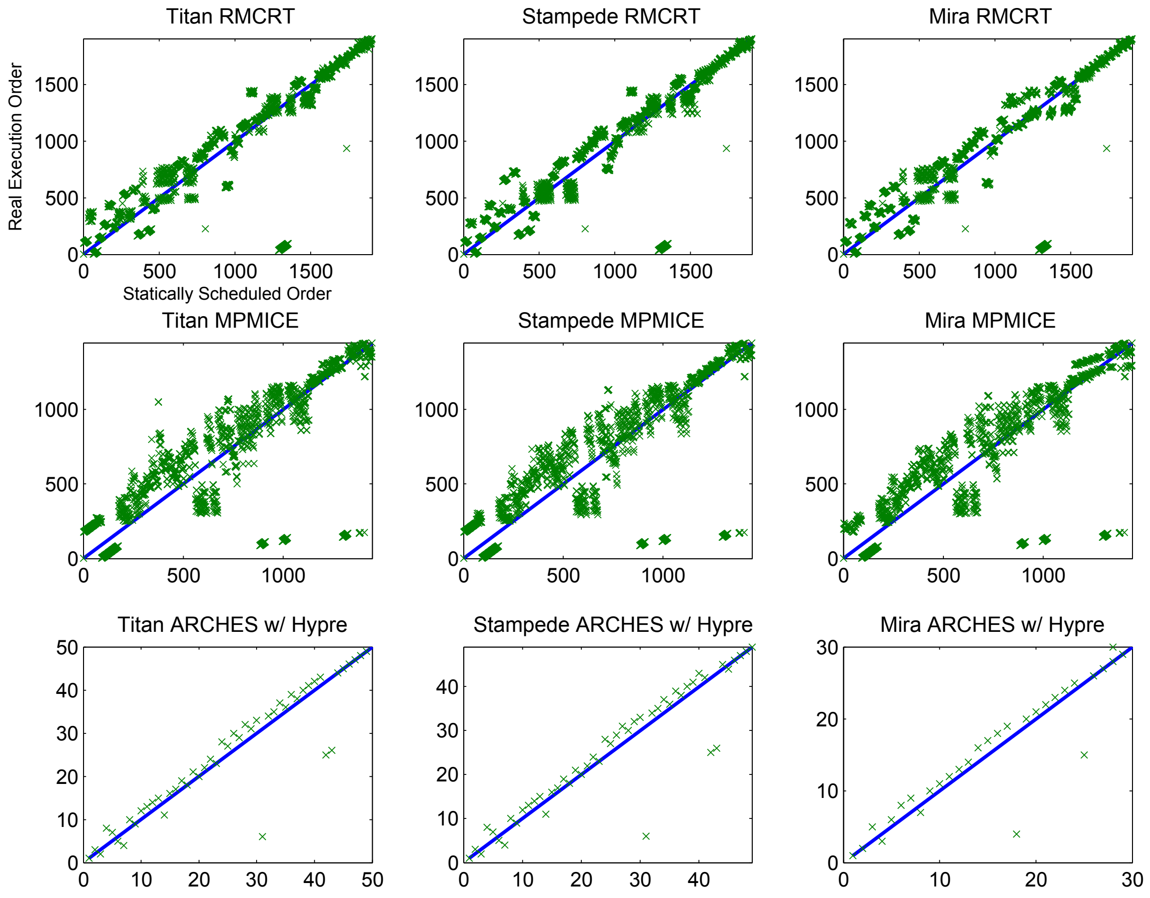


Figure 9.2. Task Scheduling Result

a number of computationally demanding simulations such as oxy-coal fired boiler simulations, oil recovery, and complicated pipe mixing. In addition to solving the multimaterial Navier-Stokes equations, subgrid models are used to describe the complicated turbulence of momentum and species transport.

The ARCHES component solves the coupled Navier-Stokes equations for mass, momentum, and energy conservation equations. The algorithm uses a staggered finite-volume mesh for gas and solid phase combustion applications [38, 76, 77]. The discretized equations are integrated in time using an explicit strong-stability preserving third-order Runge-Kutta method [42]. Spatial discretization is handled with central differencing where appropriate for energy conservation or flux limiters (e.g., scalar mixture fractions) to maintain realizability. In contrast to the explicit formulation of ICE, ARCHES uses the low-mach, pressure formulation which requires a solution of an implicit pressure projection at every timestep using the Hypre linear solver package [39].

For momentum and species transport equations, a dynamic, large eddy turbulence closure model is used to account for subgrid velocity and species fluctuations [79]. The gas phase chemistry for coal combustion is represented using a chemical equilibrium model parameterized by the coal gas and secondary oxidizer stream mixture fractions [88]. The energy balance includes the effect of radiative heat-loss/gains in the IR spectra by solving the radiative intensity equation using a discrete-ordinance solver [61]. The solution procedure solves the intensity equation over a discrete set of ordinances which is formulated as a linear system similar to the pressure projection equation and is solved using Hypre [39]. The gas phase chemistry is parameterized by the two mixture fractions and heat-loss terms and preprocessed in a tabular form for dynamic table look-up during the course of the LES simulation.

The challenging nature of this problem lies in the complex physics and variety of numerical techniques used. Good scalability has been achieved with the Taylor Green Vortex Problem described in [84]. This involved a careful use of both data structures and options in the Hypre code so it is far from clear that the same scalability will transfer from Titan to the other two machines.

Figure 9.3 also shows the send and receive volume time distribution as the similar way to that of MPMICE. Again the initial peak of the data sent also distributes the data that are computed from the last timestep and these messages can be sent out immediately. Unlike MPMICE, ARCHES is not continually receiving data. The reason is that the ARCHES component calls Hypre to do a number of iterative linear solves. During those linear solution phases, the MPI communicator is passed to the Hypre library. In particular, when Hypre is running, we can see there are several intervals when there are no receives seen by ARCHES, as Uintah is not able trace any communications conducted within the Hypre library. This effectively means that data cannot be passed to a Uintah task unless all processors have finished that specific Hypre task. MPI_Irecv for all tasks after this Hypre solve (essentially a global synchronization) can only be posted after Hypre has finished, thus resulting in several peaks in the data received.

Although Hypre now supports the use of a hybrid OpenMP/MPI parallel approach, we have so far been unable to make use of this in a way that is consistent with the Uintah multithreaded approach on a multicore node and so used our MPI only scheduler for this problem. Therefore, there are far less tasks per MPI node for ARCHES as we can see from its task execution order plots in Figure 9.2.

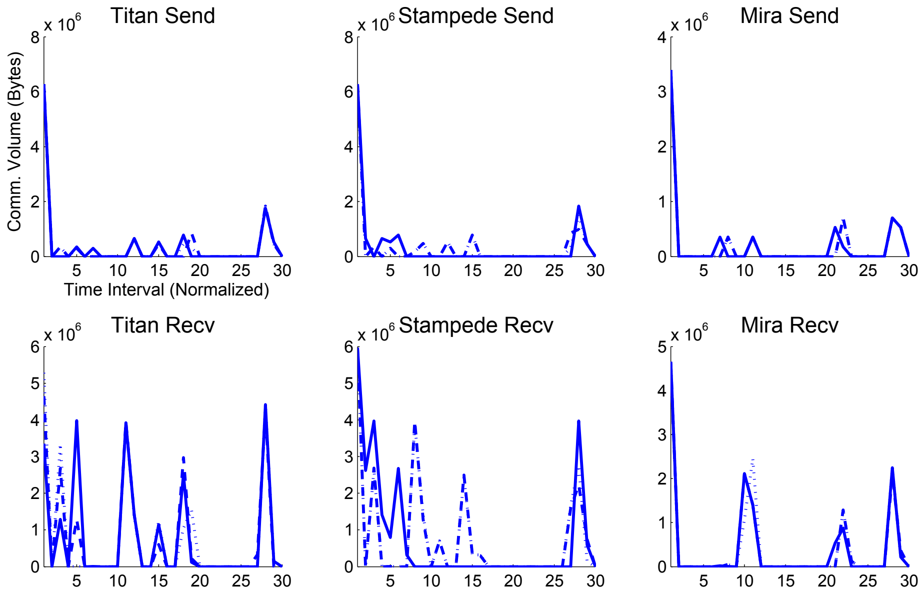


Figure 9.3. Communication Measurement of ARCHES

The multiple global synchronization points further limited the task scheduler's ability to migrate Uintah tasks, as the scheduler only allows tasks to be moved before a global synchronization point, and not after to avoid scheduling deadlock. However, we can still see that some tasks are moved to be executed early to overlap the initial sends.

9.2.3 RMCRT

Scalable modeling of radiation is important in a number of multiple applications areas such as heat transfer in combustion [87], neutron transport modeling [30] in nuclear reactors, and astrophysics modeling [100] and is currently one of the most challenging problems in present-day large-scale computational science and engineering, due to the global nature of radiation. An approach using Reverse Monte Carlo Ray Tracing (RMCRT) is used here for radiation modeling in which rays are followed back from the source to the various origins. While this is efficient in that it does not follow rays that do not reach the source, the computational and communications complexity is still potentially prohibitive on a modern heterogenous machine even though rays may be traced independently on a single GPU accelerator at high speed.

The solution often adopted and used here is that multiple length scales are used to ensure that the amount of communication and computation is reduced in a way that is consistent with achieving accuracy. This may be illustrated by Figure 9.4. This figure shows a dense computational fluids mesh and for two of the computational fluids cells shows the radiation mesh. The radiation mesh may be coarsened rapidly away from the particular cell in which the (reverse) rays originate. The number of rays is kept constant per cell, as are the heat fluxes that are calculated.

The approach used by [52, 53, 92, 93] is to store the entire geometry on each single multicore node and to calculate the partial heat fluxes on the boundaries due to the radiation originating locally. Suppose that there are N_{total} fine mesh cells. The algorithm then involves a broadcast of all the data associated with every cell to every other cell on all the other processors. This involves a multiple of N_{total}^2 in terms of total communication. The volume of communication in this case may overwhelm

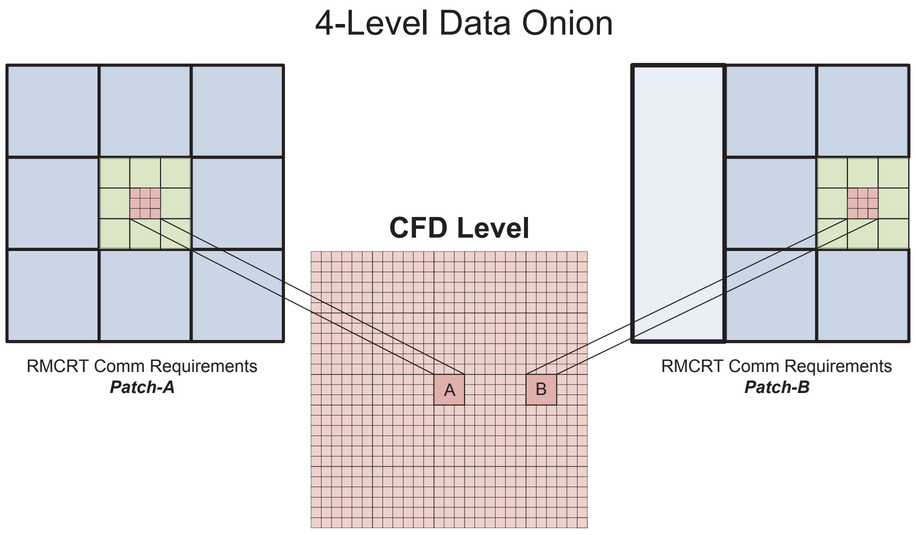


Figure 9.4. RMCRT Mesh Coarsening Scheme

the system for large problems in our experience. The alternative is to use coarser resolutions as shown in Figure 9.4 in which a fine mesh is only used close to each point and a coarse mesh used further away. The use of adaptive meshes in the context of radiation is well understood with more traditional approaches [30, 81, 100], such as the Discrete Ordinates (DO) method used in the ARCHES combustion component of the Uintah code, [56]. However, the DO approach as used with ARCHES is costly and may consume as much as 60-70% of the calculation time. In applications where such high accuracy is important, RMCRT can become more efficient than DO approaches. In particular, RMCRT can potentially reduce the cost on shared memory machines [51,92,93] and on distributed memory [52,53], with GPU accelerators [69]. A simple analysis of the two level scheme of [53] breaks the method down into the following steps:

1. Replicate the geometry (once) and coarsen mesh solution of temperature and absorption coefficients (every timestep) on all the nodes using allgather; This has a complexity of $\alpha log(p) + \beta \frac{p-1}{p}(N/r)^3$ for p cores with N^3 elements per mesh patch on a core are coarsened by a factor of r, where α is the latency and β the transmission cost per element [95].
2. Carry out the computationally very intensive ray-tracing operation locally. Suppose that we have r_a rays per cell, then each ray has to be followed through

as many as λN_G coarse mesh cells, where $N_G \approx Np/r$, or a multiple of this if there is reflection and where $0 \leq \lambda \leq \sqrt{3}N$. The total work is thus the sum of the fine mesh on each node contribution and the contribution from all the coarse mesh cells: $(\lambda N^4 + \lambda N_G^4)W_{ray}$, where W_{ray} is the work per ray per cell.

3. Distribute the resulting divergences of heat fluxes back to all the other nodes, again this cost is $\alpha log(p) + \beta\frac{p-1}{p}(N/r)^3$.

The relative costs of computation vs. communication are then given:

$$\lambda N^4(1+(p/r)^4)W_{ray}r_a \;\; vs \;\; 2(\alpha log(p) + \beta\frac{p-1}{p}(N/r)^3).$$

Thus for enough rays r_a with enough refinement by a factor of r on the coarse radiation mesh, it looks likely that computation will dominate. A key challenge is that storage of $O(N_G^3)$ will be required on a multicore node and so only coarse and AMR mesh representations will be possible in a final production code at very large core counts. Although preliminary, this analysis can be extended to fully adaptive meshes, rather than the two level case considered here.

The send and receive volume time distribution for RMCRT, shown in Figure 9.5, indicates that the volume of initial data sent dominates. As we discussed above, RMCRT is a relatively simple algorithm in that most of requisite data is only required from the last timestep and can be sent out once the current timestep starts. However, unlike the AMR MPMICE component for which most messages are local, RMCRT requires an all-to-all data transfer and the data transfer time is limited by the global bandwidth. From the bottom of Figure 9.5, it is seen that some MPI messages on Titan are delayed. We can also see large periods of time in which no data are being read, as RMCRT is computationally expensive and has a relatively simple variable exchange data pattern when compared to the complexity and frequency of communications in the multiphysics AMR MPMICE component.

Figure 9.2 shows that in order to overlap the huge initial send of data, the scheduler moved tasks as much as possible to be executed early (close to the x-axis). In other words, some tasks that should be statically scheduled for execution near the end of a timestep are moved to the beginning of the timestep. Furthermore, the MPI message delay on Titan is addressed by the scheduler moving tasks more aggressively than for Mira and Stampede.

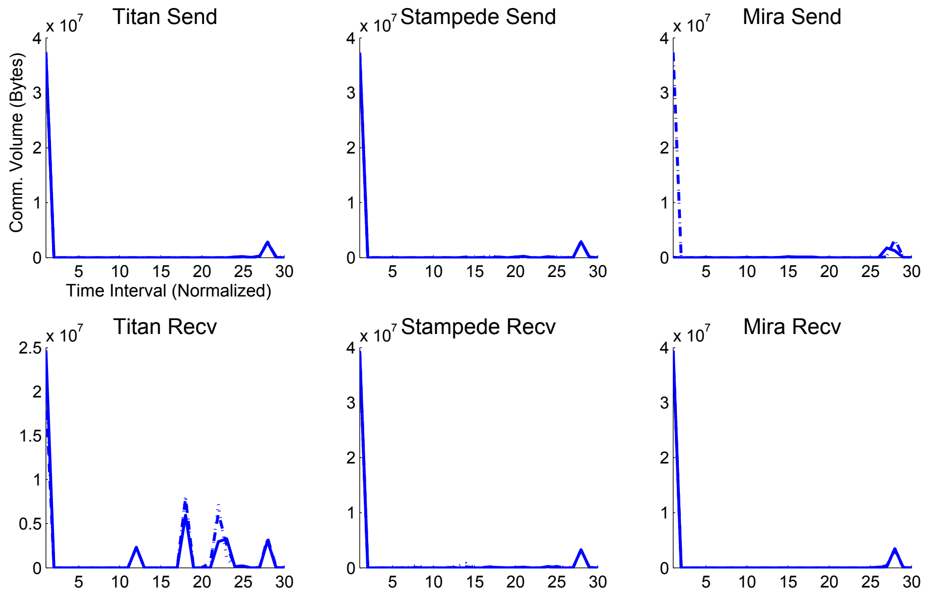


Figure 9.5. Communication Measurement of RMCRT

9.3 Scaling Results

In this section, we will show the scalability results for AMR MPMICE, ARCHES, and RMCRT on the three target machines described above: Titan, Stampede, and Mira. We also include preliminary GPU scaling results for RMCRT here. Coprocessor scaling results are covered in [70]. We define strong scaling as a decrease in execution time when a fixed size problem is solved on more cores, while weak scaling should result in constant execution time when more cores are used to solve a correspondingly larger problem. AMR MPMICE scaling results from the newly commissioned Blue Water[1] machine are also presented.

9.3.1 Strong Scaling of MPMICE

Perhaps the most satisfactory result from this study is shown in Figure 9.6 which displays the strong scaling results from three AMR MPMICE problems. The AMR MPMICE problems exercise all of the main features of AMR, ICE, and MPM and also include a model for the deflagration of the explosive and the material damage model ViscoSCRAM [67]. The simulation grid utilized three refinement

[1]Blue Water is a Cray cluster, administered by NCSA with 22,640 XE6 and 4,224 XK7 nodes, the XE6 nodes have 64 GB memory per node and the XK7s have 32 GB memory per node. [11].

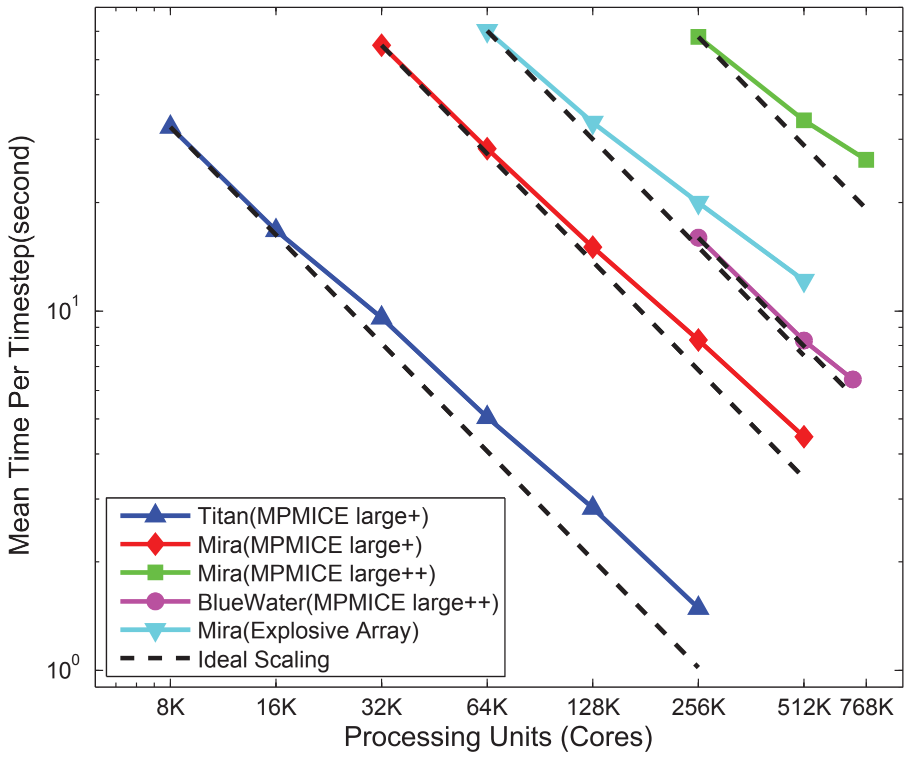


Figure 9.6. AMR MPMICE Strong Scaling

levels with each level being a factor of four more refined than the previous level. For large+ problems, a total of 3.62 billion particles and 160 million cells are created on three AMR levels. These tests were run on Titan and Mira with up to 512K cores and with 16 threads per MPI node. For large++ problems, a total of 29.0 billion particles and 3.8 billion cells are created on three AMR levels. These tests were run on Mira with up to 756K cores and with 16 threads per MPI node and on Blue Water with up to 704K cores and with 32 threads per MPI node. The explosive array problem [18] is designed to resolve the underlying physics of a highway accident in which detonation occurred during the transportation of about 35000 lbs of seismic boosters on Highway 6 in Utah. In this simulation, 7.7 billion particles and 1.3 billion cells are created in three AMR levels. The benchmark results come from runs on Mira with up to 512K cores with 16 threads per core.

In order to get excellent scaling results, we tested and found out that the optimal patch configuration for AMR MPMICE problems should fit the following two requirements. 1) The number of patches on each level should be tuned as close as possible but not more than the number of cores on the largest run. 2) The patch size should be at least 8x8x8. The second requirement overrides the first one, that is, even though there is not enough patch in a particular level for all CPU cores, we cannot divided the patch further into a smaller size that is less than size 8x8x8. The reason is that when patch size is smaller than 8x8x8, the cost of a patch's MPI messages will be too high comparing to the cost of its computation for runtime system to overlap this cost. This lower bound of patch size should be considered as machine dependent, and could change on further machines. In addition to choosing a good patch size for different levels, it is also important to line up the patch boundaries in finer level to patch boundaries in coarse level. An easy way to achieve this is to choose finer level patch size that can divide coarser level patch size in each dimension without a reminder. For example, when coarse level patch size is 8x8x8, it is better to have finer level patch size set up as 16x16x8 rather than 12x12x12. We observed that the later choice of patch size will lead to higher MPI communication imbalance.

In this benchmark problem, the simulation grid changes once per every 40 to 50 timesteps, the same as reported on [65]. The grid changes once its finest level patches cannot hold all the particles in them. The overhead of this regridding process, including creating new grid, compiling new task graph, and moving old grid data to new grid, is less than 3 percent of overall execution time. This is a result of many improvements that have been made to reduce the cost of the regridding process for MPMICE AMR runs, including removal of topological sort in task graph compiling algorithm, looping through only neighbours instead of all processers when assigning cores to once-per-proc task, and computing the refine patch sets in parallel.

9.3.2 Weak Scaling of MPMICE and ARCHES

Figure 9.7 shows the weak scaling results from the AMR MPMICE problem. This problem is the same type of problem with three refinement levels shown previously for strong scaling, but with different resolutions on each run. The average numbers of particles and cells per node are set as close as possible. In the same number of

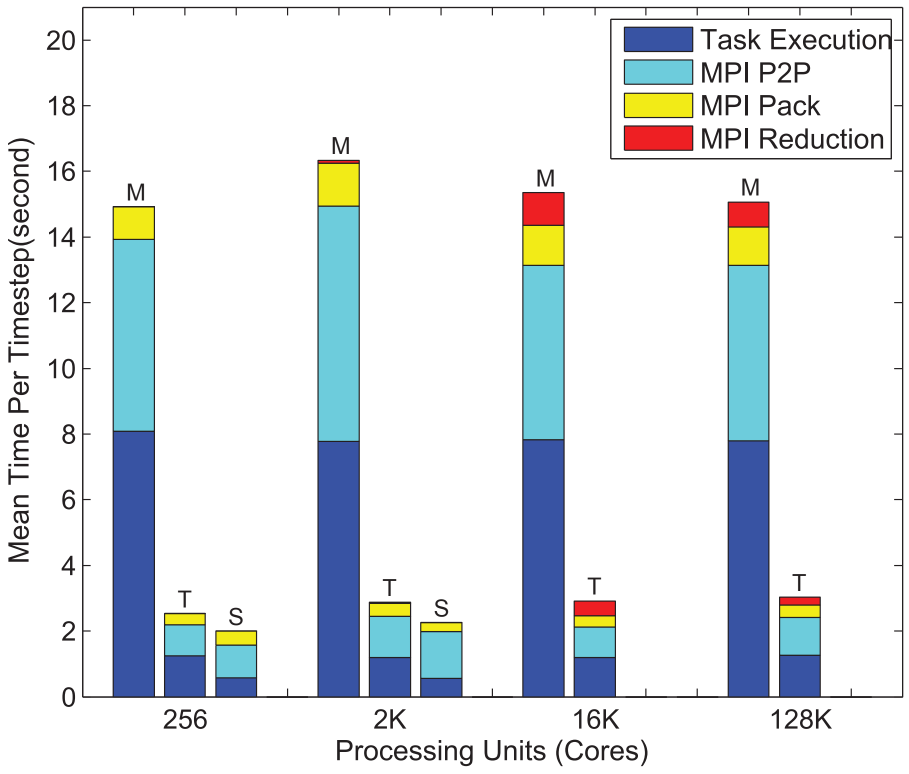


Figure 9.7. AMR MPMICE Weak Scaling

cores group, from left to right are results from Mira, Titan, and Stampede, denoted by M, T, and S, respectively. The times for task execution, message passing, and packing are shown in different colors. The weak scaling for all three machines for this example is almost perfect with higher than 95% efficiency.

The weak scalability of the ARCHES component is shown in Figure 9.8 with each core group from left to right representing the results for Mira, Titan, and Stampede, denoted by M, T, and S, respectively. For Mira and Titan, core counts ranged from 2K to 128K, and for Stampede, the core counts ranged from 2K to 64K. The problem uses a fixed resolution (42^3 cells per patch) with a single patch per core. Each timestep was broken down into linear solver time (Hypre Time) and Uintah time. For each timestep, the solution to a large (74,000 unknowns per core), sparse system of equations (Pressure-Poisson equation) is solved. The Hypre solver parameters used included the following: conjugate gradient method with the PFMG multigrid preconditioner and a red-black Gauss-Seidel relaxer. The weak scaling efficiency is 88% on Mira at 128K cores, 79% on Titan at 128K cores, and 56% on Stampede at

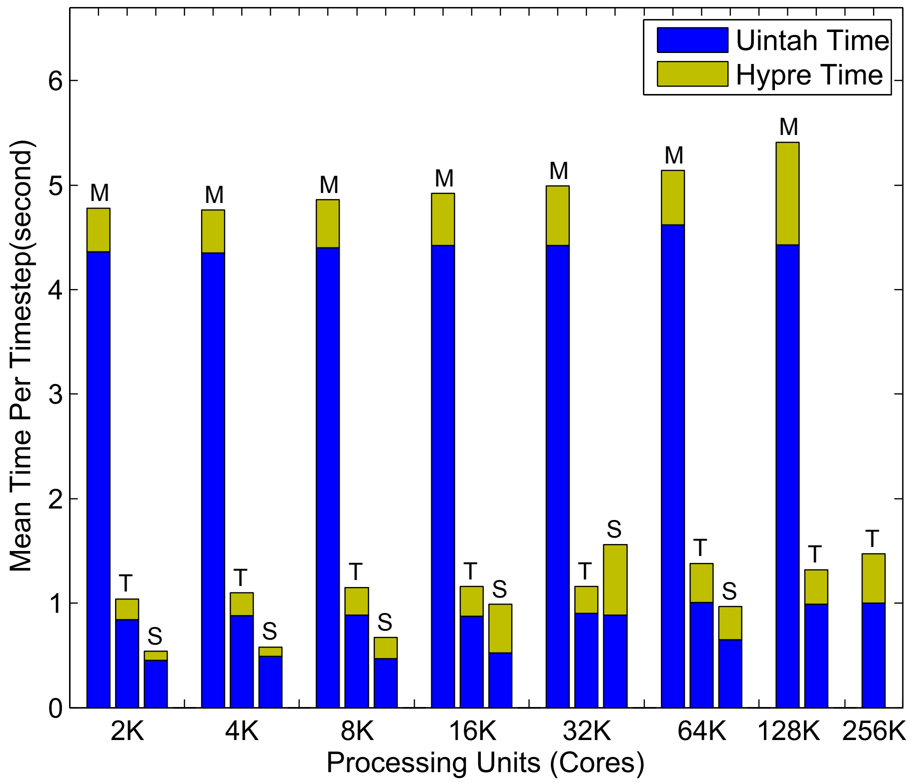


Figure 9.8. ARCHES with Hypre Weak Scaling

64K cores. However, we can see that in most cases, the Uintah scaling component is better than the Hypre component. The efficiency losses come mostly from the Hypre solving phase which has a log(P) term, where P is the number of cores [84]. The relatively slower Mira cores (compared to Titan and Stampede) contributes to the significantly slower mean time/timestep. However, the network topology of Mira is well balanced and shows excellent weak scalability out to 128K cores. In contrast to Mira, the faster cores of Stampede and Titan magnify any slight timing variabilities as core counts were increased. The variabilities for Stampede were especially pronounced for 16K and 32K cores. The variability in timing may be due in part to system loads and network traffic impacting the Uintah simulation. We encountered issues with Mira when attempting to scale beyond 128K cores with an out of memory condition, which may be due to the limited memory per core. Future work will include using a newer version of the Hypre library with additional threading support along with Uintah's threaded task scheduler. It is anticipated that this combination will reduce the memory requirements for the larger core count simulations.

9.3.3 Strong Scaling of RMCRT

Although solving the radiative transport equation using methods such as the parallel Discrete Ordinates Method (DOM) has been shown to scale [85] through the use of Hypre [39], the cost is significant due to the large number of systems of linear equations required by this method. RMCRT has been shown to significantly reduce this cost. However, RMCRT is an all-to-all method, where all geometry information and property model information for the entire computational domain must be present on each processor. This presently limits the size of the problem that can be computed due to memory constraints.

For the RMCRT problem, a two-level coarsening scheme with a refinement ratio of 2 was used. The finer, CFD level used a resolution of 256^3 with one patch per core in each case and the coarse, RMCRT level used a resolution of 64^3 with a single patch. 10 rays per cell were used in the RMCRT portion of the calculation and the mean time per timestep was averaged over 10 timesteps. Figure 9.9 shows strong scaling results from the RMCRT benchmark case detailed in Section 9.2 to 16k cores. These results are significant in that other adaptive mesh codes using similar approaches in radiative shock hydrodynamics, astrophysics, and cosmology; e.g., Crash and Enzo [98, 100] report scaling to a maximum of near 1000 cores. The hybrid memory approach used by Uintah has also contributed to our results, as only one copy of the geometry is needed per multicore node.

The results in Figure 9.9 also show that in this case, Titan outperforms Stampede (host native mode) and Mira. In the case of the AMR MPMICE problem, Stampede and Titan perform similarly. The difference in these two cases would appear to be better network performance of Titan for the very large amounts of all-to-all communication required by the RMCRT problem.

Strong scaling results for a single-level, GPU-enabled RMCRT problem are shown by [52] for a prototype Uintah testbed component with a resolution of 128^3 on the TitanDev system. This Uintah component executes an extension of the RMCRT benchmark problem described by Burns and Christon in [28]. In moving

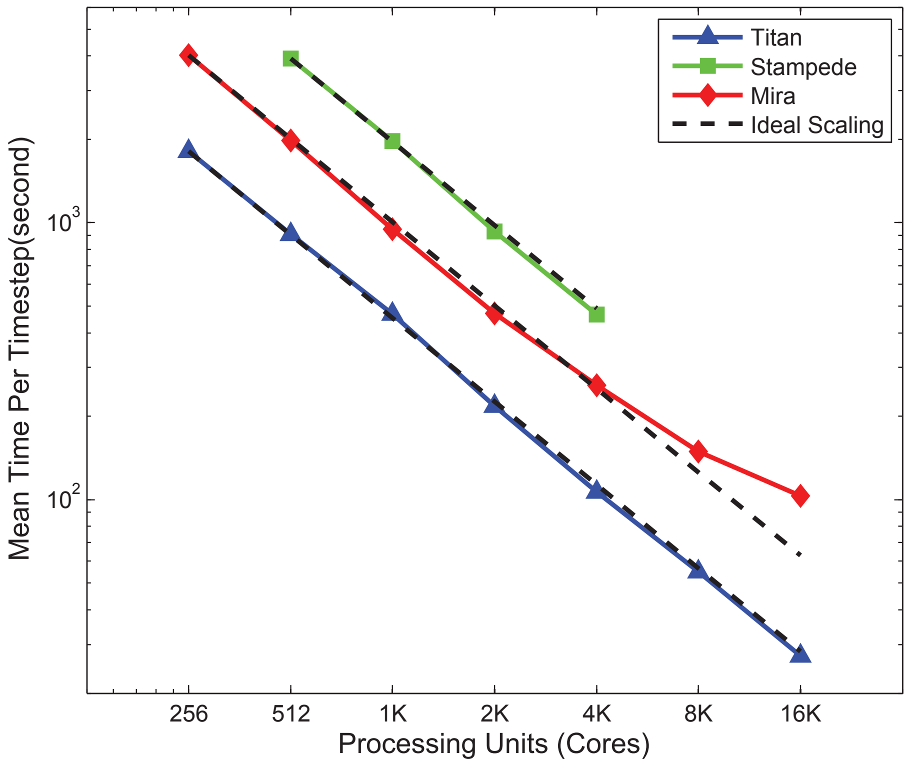


Figure 9.9. RMCRT Strong Scaling

to the full Titan system with its new Nvidia K20 GPUs, we found scaling results consistent to those in [52]. The only significant difference was that the Kepler GPUs were nearly three times faster than their Fermi predecessor.

Although we were able to run successfully on Titan beyond 256 nodes, utilizing the on-node GPUs, the GPU implementation quickly runs out of work and strong scaling begins breaking down. The all-to-all nature of this problem severely limits the size of the problem that can be computed, and hence does not scale well due to memory constraints involved with large highly resolved physical domains [52]. To address this scalability issue and as part of future work, we will modify our RMCRT GPU implementation to leverage the multilevel coarsening scheme discussed in Section 9.2.

9.4 Summary

The porting requirements of Uintah can vary depending on the nature of the heterogeneous platform, and the dominant changes that must occur are focused primarily in the runtime system. For machines which require vendor-specific extensions such as Nvidia CUDA, the component code does require changes to make use of these extensions. However, the runtime system does provide mechanisms to minimize changes in the component code. Porting Uintah to new heterogenous systems only requires changes to the runtime system such as schedulers and data warehouse, with little to no changes to infrastructure code. Porting Uintah task code can vary depending on the nature of the heterogeneous platform; however, the Uintah runtime system does provide convenient mechanisms to port any subset of task code. We have also shown that weak and strong scalability is achieved when the runtime environment is allowed to flexibly schedule the execution order of computational tasks to overlap computation with communication. It is the combination of the DAG representation of tasks with a runtime environment that can schedule tasks based on a precompiled dependency analysis in a preferred order that yields scalability on a variety of problems with widely different communication requirements, and on systems with very different architectures. A major remaining challenge is to extend Uintah to move beyond being able to use accelerators and coprocessors to achieve scalability across the whole of machines like Titan, Mira, and Stampede for very broad problem classes including components such as radiation.

CHAPTER 10

CONCLUSION AND FUTURE WORK

In this dissertation, I have covered the history and evolution of Uintah in the context of its task schedulers and runtime systems, all leading up to the development of the Unified heterogeneous task scheduler and runtime system described in this work. Our conclusion is 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.

There are several potential research topics to extend this research further. The runtime system could be improved by implementing an auto-tuning mechanism to select patch size from profiling results and runtime information. This approach could lead to a better cache reuse if an optimized patch size is chosen. Moreover, when Uintah is run on different computing units, such as multicore CPUs, GPUs, or MICs, different patch size may be required to get better performance. Communication, on the other hand, may also have different granularity requirements for better overlapping the computations. It could be hard to find a universal patch size to satisfy all these requirements. If the runtime system can group patches together dynamically for CPU processing, GPU/MIC processing, or communications, we could then have a good cache reuse, GPU occupancy, and communication performance at the same time.

During this research, many challenges come from debugging and managing complexity on the large scale. Most of petascale machines no longer provide back end access or core dump files to user. Large-scale commercial debugger's debugging session caused racks of the machine to crash and can be resolved only by the creation of a special build of debugger itself. These restrictions make Uintah's development and performance analysis on petascale much more difficult. By manually looking up the address pointers print-out and then mapping them to the code offline, we extracted limited but useful information for debugging. Another technique is to reproduce the target issue in a small scale so that common debugging tools can be used. To identify key performance and scalability issues of Uintah, we have employed Uintah's built-in monitoring functions to locate components needing improvement. Third-party profiling tools were then used to localize the exact code, consuming the most CPU time. We also utilized manually inserted timers to confirm profiling results and to verify the improvement once changes were made. In order to provide an easier way to overcome those challenges, systematic debugging and performance monitoring approaches may be required for future Uintah development.

Using directed acyclic graphs to represent computational tasks combined with an asynchronous and dynamic runtime system provides an effective way to achieve scalable systems on disparate heterogenous and homogenous high performance computer systems. High efficiency is achieved when the runtime environment is allowed to flexibly schedule the execution order of the various computational tasks. This approach leads to scalability on a variety of problems with widely different communication requirements, and on systems with very different architectures. To extend Uintah to being able to use accelerators and coprocessors more efficiency with hardware's new abilities to solve new challenging scientific and engineering problems requires additional and continue work on the runtime system.

APPENDIX

PUBLICATIONS

- Q. Meng, J. Luitjens, and M. Berzins. "Dynamic Task Scheduling for the Uintah Framework". In Proceedings of *the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10)*, IEEE 2010
- Q. Meng, M. Berzins, and J. Schmidt. "Using Hybrid Parallelism to Improve Memory Use in the Uintah Framework." In Proceedings of *the 2011 TeraGrid Conference (TG11)*, Salt Lake City, Utah, ACM, 2011.
- 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 2nd Conference of the Extreme Science and Engineering Discovery Environment (XSEDE 2013)*, ACM, 2013.
- Q. Meng, A. Humphrey, and M. Berzins. "The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System". In Digital Proceedings of *the Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC12)*, IEEE, 2012.
- 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*, John Wiley & Sons, Ltd., 2013
- 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 *the International Conference for High Performance Computing, Networking, Storage and Analysis(SC'13)*, ACM/IEEE 2013

- M. Berzins, J. Luitjens, Q. Meng, T. Harman, C.A. Wight, and J.R. Peterson. "Uintah - A Scalable Framework for Hazard Analysis." In Proceedings of *the 2010 TeraGrid Conference (TG'10)*, 2010, ACM.

- 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 (XSEDE 2012)*, 2012, ACM.

- M. Berzins, Q. Meng, J. Schmidt, and J. Sutherland. "DAG-based Software Frameworks for PDEs." In the Proceedings of *the Workshop on Algorithms and Programming Tools for Next-Generation High-Performance Scientific Software, Lecture Notes in Computer Science (LNCS)*, Springer, 2013

- M. Berzins, J. Schmidt, Q. Meng, A. Humphrey. "Past, Present, and Future Scalability of the Uintah Software", In Proceedings of *the Extreme Scaling Workshop*, ACM, 2012

- J. Beckvermit, J.R. Peterson, T. Harman, S. Bardenhagen, C.A. Wight, Q. Meng, M. Berzins. "Multiscale Modeling of Accidental Explosions and Detonations" *Computing in Science and Engineering* Volume 15, Issue 4, pp. 76-86, IEEE/AIP, 2013

REFERENCES

[1] BoxLib user's guide, 2011. https://ccse.lbl.gov/BoxLib.

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

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

[4] Top500 Web Page, 2012. http://www.top500.org/list/2012/11/.

[5] Argonne Leadership Computing Facility Mira Web Page, 2013. https://www.alcf.anl.gov/mira.

[6] FLASH user's guide, 2013. http://flash.uchicago.edu/.

[7] OpenACC member companies and CAPS Enterprise and CRAY Inc and The Portland Group Inc (PGI) and NVIDIA, OpenACC Web Page, 2013. http://www.openacc-standard.org/.

[8] Texas Advanced Computing Center Stampede User Guide, 2013. http://www.tacc.utexas.edu/user-services/user-guides/stampede-user-guide.

[9] Texas Advanced Computing Center Stampede Web Page, 2013. http://www.tacc.utexas.edu/resources/hpc/stampede.

[10] U.S. Department of Energy Oak Ridge National Laboratory and Oak Ridge Leadership Computing Facility Titan Web Page, 2013. http://www.olcf.ornl.gov/titan/.

[11] National Center for Supercomputing Applications BlueWater User Guide, 2014. https://bluewaters.ncsa.illinois.edu/blue-waters.

[12] Akhter, S., and Roberts, J. *Multi-core Programming*. Intel Press, 2006.

[13] Amarasinghe, S., Campbell, D., Carlson, W., Chien, A., Dally, W., Elnohazy, E., Hall, M., Harrison, R., Harrod, W., Hill, K., et al. Exascale software study: Software challenges in extreme scale systems. Tech. rep., 2009.

[14] Amarasinghe, S., Campbell, D., Carlson, W., Chien, A., Dally, W., Elnohazy, E., Hall, M., Harrison, R., Harrod, W., K.Hill, Hiller, J., Karp, S., Koelbel, C., D.Koester, Kogge, P., J.Levesque, Reed, D., Sarkar, V., R.Schreiber, Richards, M., Scarpelli, A., J.Shalf, A.Snavely, and Sterling, T. Exascale computing study: Technology challenges in achieving exascale

systems. Tech. Rep. ECSS Report 101909, Georgia Institute of Technology, 2009.

[15] Anshu Dubey, Ann Almgrena, J. B. M. B. S. B. A survey of high level frameworks in block-structured adaptive mesh renement packages.

[16] Attaway, S., Heinstein, M., and Swegle, J. Coupling of smooth particle hydrodynamics with the finite element method. *Nuclear engineering and design 150*, 2 (1994), 199–205.

[17] Attaway, S. A., Barragy, E. J., Brown, K. H., Gardner, D. R., Hendrickson, B. A., Plimpton, S. J., and Vaughan, C. T. Transient solid dynamics simulations on the sandia/intel teraflop computer. In *Supercomputing, ACM/IEEE 1997 Conference* (1997), IEEE, pp. 58–58.

[18] Beckvermit, J., Peterson, J., Harman, T., Bardenhagen, S., Wight, C., Meng, Q., and Berzins, M. Multiscale modeling of accidental explosions and detonations. *Computing in Science and Engineering 15*, 4 (2013), 76–86.

[19] Bennett, J. G., Haberman, K. S., Johnson, J. N., and Asay, B. W. A constitutive model for the non-shock ignition and mechanical response of high explosives. *Journal of the Mechanics and Physics of Solids 46*, 12 (1998), 2303–2322.

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

[21] Berzins, M. Status of release of the uintah computational framework. SCI Technical Report UUSCI-2012-001, SCI Institute, University of Utah, 2012.

[22] Berzins, M., Luitjens, J., Meng, Q., Harman, T., Wight, C., and Peterson, J. Uintah - a scalable framework for hazard analysis. In *TG '10: Proc. of 2010 TeraGrid Conference* (2010), ACM.

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

[24] Blazewicz, M., Hinder, I., Koppelman, D. M., Brandt, S. R., Ciznicki, M., Kierzynka, M., Löffler, F., Schnetter, E., and Tao, J. From physics model to results: An optimizing framework for cross-architecture code generation. *Scientific Programming*.

[25] Brackbill, J., and Ruppel, H. Flip: A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *Journal of Computational Physics 65*, 2 (1986), 314–343.

[26] Brackbill, J. U. Particle methods. *International Journal for Numerical Methods in Fluids 47*, 8-9 (2005), 693–705.

[27] Bungartz, H.-J., Benk, J., Gatzhammer, B., Mehl, M., and Neckel, T. Partitioned simulation of fluid-structure interaction on cartesian grids. In *Fluid Structure Interaction II*. Springer, 2010, pp. 255–284.

[28] Burns, S. P., and Christen, M. A. Spatial domain-based parallelism in large-scale, participating-media, radiative transport applications. *Numerical Heat Transfer, Part B: Fundamentals 31*, 4 (1997), 401–421.

[29] Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing 35*, 1 (2009), 38 – 53.

[30] Clouse, C. Parallel deterministic neutron transport with amr. In *Computational Methods in Transport*, F. Graziani, Ed., vol. 48 of *Lecture Notes in Computational Science and Engineering*. Springer Berlin Heidelberg, 2006, pp. 499–512.

[31] Colella, P., Bell, J., Keen, N., Ligocki, T., Lijewski, M., and van Straalen, B. Performance and scaling of locally-structured grid methods for partial differential equations. *Journal of Physics: Conference Series 78* (2007), 012013.

[32] Colella, P., Graves, D., Ligocki, T., Martin, D., Modiano, D., Serafini, D., and Straalen, B. V. Chombo software package for AMR applications: design document.

[33] Davison, J., Germain, S., Mccorquodale, J., Parker, S. G., and Johnson, C. R. Uintah: A massively parallel problem solving environment. In *Proc. of the 9th IEEE Intl. Symposium on High Performance and Distributed Computing* (2000).

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

[35] Dinan, J., Krishnamoorthy, S., Brian, L., Nieplocha, J., and Sadayappan, P. Scioto: A framework for global-view task parallelism. In *Parallel Processing, 2008. ICPP '08. 37th International Conference on* (9-12 2008), pp. 586 –593.

[36] Dostál, Z., Vondrák, V., Horák, D., Farhat, C., and Avery, P. Scalable feti algorithms for frictionless contact problems. In *Domain Decomposition Methods in Science and Engineering XVII*. Springer, 2008, pp. 263–270.

[37] Enzo astrophysical AMR code, 2013. http://enzo-project.org/.

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

[39] Falgout, R., Jones, J., and Yang, U. *Numerical Solution of Partial Differential Equations on Parallel Computers*, vol. UCRL-JRNL-205459. Springer-Verlag, 51, 2006, ch. The Design and Implementation of Hypre, a Library of Parallel High Performance Preconditioners, pp. 267–294.

[40] Fraser, K. *Practical lock-freedom*. PhD thesis, PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[41] Fryxell, B., Olson, K., Ricker, P., Timmes, F. X., Zingale, M., Lamb, D. Q., MacNeice, P., Rosner, R., Truran, J. W., and Tufo, H. Flash: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical Journal, Supplement 131* (2000), 273–334.

[42] Gottlieb, S., Shu, C., and Tadmor, W. Strong stability-preserving high-order time discretization methods. *Siam Review 43*, 1 (2001), 89–112.

[43] Götz, J., Iglberger, K., Stürmer, M., and Rüde, U. Direct numerical simulation of particulate flows on 294912 processor cores. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis* (2010), IEEE Computer Society, pp. 1–11.

[44] Graham, R., Lawler, E., Lenstra, J., and Kan, A. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics 5* (1979), 287–326.

[45] Grinberg, I., and Wiseman, Y. Scalable parallel simulator for vehicular collision detection. *International Journal of Vehicle Systems Modelling and Testing 8*, 2 (2013), 119–144.

[46] Guilkey, J., and Weiss, J. Implicit time integration for the material point method: Quantitative and algorithmic comparisons with the finite element method. *International Journal for Numerical Methods in Engineering 57*, 9 (2003), 1323–1338.

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

[48] Guilkey, J. E., Harman, T. B., Xia, A., Kashiwa, B. A., and McMurtry, P. A. An Eulerian-Lagrangian approach for large deformation fluid-structure interaction problems, part 1: Algorithm development. In *Fluid Structure Interaction II* (2003), WIT Press.

[49] Harlow, F. H., and Amsden, A. A. Numerical calculation of almost incompressible flow. *Journal of Computational Physics 3*, 1 (1968), 80–93.

[50] Harman, T. B., Guilkey, J. E., Kashiwa, B. A., Schmidt, J., and McMurtry, P. A. An eulerian-lagrangian approach for large deformationfluid-structure interaction problems, part 1:multi-physics simulations within a modern computationalframework. In *Fluid Structure Interaction II* (2003), WIT Press.

[51] Howell, J. R. The monte carlo in radiative heat transfer. *Journal of Heat Transfer 120*, 3 (1998), 547–560.

[52] Humphrey, A., Meng, Q., Berzins, M., and Harman, T. Radiation Modeling Using the Uintah Heterogeneous CPU/GPU Runtime System. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment (XSEDE 2012)* (2012), ACM.

[53] Hunsaker, I., Harman, T., Thornock, J., and Smith, P. Efficient Parallelization of RMCRT for Large Scale LES Combustion Simulations. Paper AIAA-2011-3770. 41st AIAA Fluid Dynamics Conference and Exhibit, 2011.

[54] Hunsaker, I., Harman, T., Thornock, J., and Smith, P. J. Efficient Parallelization of RMCRT for Large Scale LES Combustion Simulations. No. 2011-3770 in Volume 1, pp. 2714–2724.

[55] Jessee, J. P., Fiveland, W. A., Howell, L. H., Colella, P., and Pember, R. B. An adaptive mesh refinement algorithm for the radiative transport equation. *Journal of Computational Physics 139*, 2 (1998), 380–398.

[56] J.Spinti, Thornock, J., Eddings, E., Smith, P., and Sarofim, A. Heat transfer to objects in pool fires, in transport phenomena in fires. In *Transport Phenomena in Fires* (2008), WIT Press.

[57] Kale, L. V., and Krishnan, S. Charm++: Parallel Programming with Message-Driven Objects. In *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 175–213.

[58] Kashiwa, B., and Gaffney., E. Design basis for cfdlib. Tech. Rep. LA-UR-03-1295, Los Alamos National Laboratory, 2003.

[59] Kashiwa, B., Lewis, M., and Wilson, T. Fluid-structure interaction modeling. Tech. Rep. LA-13111-PR, Los Alamos National Laboratory, 1996.

[60] Kashiwa, B. A., and Rauenzahn, R. M. A cell-centered ICE method for multiphase flow simulations. Tech. Rep. LA-UR-93-3922, Los Alamos National Laboratory, 1994.

[61] Krishnamoorthy, G. *Predicting Radiative Heat Transfer in Parallel Computations of Combustion*. PhD thesis, University of Utah, December 2005.

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

[63] Luitjens, J., and Berzins, M. Scalable parallel regridding algorithms for block-structured adaptive mesh refinement. *Concurrency and Computation: Practice and Experience 23*, 13 (2011), 1522–1537.

[64] Luitjens, J., Worthen, B., Berzins, M., and Henderson, T. Scalable parallel amr for the uintah multiphysics code. In *Petascale Computing Algorithms and Applications*, D. Bader, Ed. Chapman and Hall/CRC, 2007.

[65] Luitjens, J. P. *The Scalability of Parallel Adaptive Mesh Refinement Within Uintah*. PhD thesis, The University of Utah, 2011.

[66] MacNeice, P., Olson, K., Mobarry, C., de Fainchtein, R., and Packer, C. PARAMESH: A parallel adaptive mesh refinement community toolkit. *Computer Physics Communications 126*, 3 (2000), 330–354.

[67] Meng, Q., and Berzins, M. Scalable large-scale fluid-structure interaction solvers in the Uintah framework via hybrid task-based parallelism algorithms. *Concurrency and Computation: Practice and Experience* (2013).

[68] Meng, Q., Berzins, M., and Schmidt, J. Using Hybrid Parallelism to Improve Memory Use in the Uintah Framework. In *Proc. of the 2011 TeraGrid Conference (TG11)* (2011).

[69] Meng, Q., Humphrey, A., and Berzins, M. The Uintah Framework: A Unified Heterogeneous Task Scheduling and Runtime System. In *Digital Proceedings of Supercomputing 12 - WOLFHPC Workshop* (2012), IEEE.

[70] Meng, Q., Humphrey, A., Schmidt, J., and Berzins, M. Preliminary Experiences with the Uintah Framework on Intel Xeon Phi and Stampede. In *The 2nd Conference of the Extreme Science and Engineering Discovery Environment (XSEDE 2013)* (2013), ACM.

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

[72] Modest, M. F. Backward Monte Carlo Simulations in Radiative Heat Transfer. *Journal of Heat Transfer 125*, 1 (2003), 57–62.

[73] Parker, S. G. C-safe uses linux hpcc in fire research. *Technology for Higher Education, Syllabus 16* (2003).

[74] Parker, S. G. A component-based architecture for parallel multi-physics pde simulation. *Future Generation Computing System 22*, 1 (2006), 204–216.

[75] Parker, S. G., Guilkey, J., and Harman, T. A component-based parallel infrastructure for the simulation of fluid structure interaction. *Engineering with Computers 22*, 3 (2006).

[76] Pedel, J., Thornock, J., and Smith, P. Ignition of co-axial oxy-coal jet flames: data collaboration of experiments and simulations. *Combustion and Flame Accepted for publication* (2012).

[77] Pedel, J., Thornock, J., and Smith, P. Large simulation of pulverized coal jet flame ignition using the direct quadrature method of moments. *Energy and Fuels Accepted for publication* (2012).

[78] Pernice, M., and Philip, B. Solution of equilibrium radiation diffusion problems using implicit adaptive mesh refinement. *SIAM J. Sci. Comput. 27*, 5 (2005), 1709–1726.

[79] Pope, S. B. *Turbulent Flows*. Cambridge Press, 2000.

[80] Q. Meng, A. Humphrey, J. S., and Berzins, M. Investigating Applications Portability with the Uintah DAG-based Runtime System on PetaScale Supercomputers. In *the International Conference for High Performance Computing, Networking, Storage and Analysis(SC'13)* (2013), IEEE/ACM.

[81] Rijkhorst, E.-J., Plewa, T., Dubey, A., and Mellema, G. Hybrid characteristics: 3d radiative transfer for parallel adaptive mesh refinement hydrodynamics. *Astronomy and Astrophysics 452*, 3 (2006), 907–920.

[82] Sadeghirad, A., Brannon, R., and Burghardt, J. A convected particle domain interpolation technique to extend applicability of the material point method for problems involving massive deformations. *International Journal for Numerical Methods in Engineering 86*, 12 (2011), 1435–1456.

[83] Sarkar, V. *Partitioning and scheduling parallel programs for execution on multiprocessors*. PhD thesis, 1987. UMI Order No. GAX87-23080.

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

[85] Schmidt, J., Thornock, J., Sutherland, J., and Berzins, M. Large Scale Parallel Solution of Incompressible Flow Problems using Uintah and Hypre. Tech. Rep. UUSCI-2012-002, Scientific Computing and Imaging Institute, 2012.

[86] Sinnen, O. *Task scheduling for parallel systems*, vol. 60. Wiley. com, 2007.

[87] Smith, P. J., R.Rawat, Spinti, J., Kumar, S., Borodai, S., and Violi, A. Large eddy simulation of accidental fires using massively parallel computers. In *18th AIAA Computational Fluid Dynamics Conference* (June 2003).

[88] Smoot, L., and Smith, P. *Coal Combustion and Gasification*. Plenum Press, 1985.

[89] Steffen, M., Kirby, R. M., and Berzins, M. Decoupling and balancing of space and time errors in the material point method (mpm). *International journal for numerical methods in engineering 82*, 10 (2010), 1207–1243.

[90] Sulsky, D., Chen, Z., and Schreyer, H. A particle method for history-dependent materials. *Computer Methods in Applied Mechanics and Engineering 118*, 1-2 (1994), 179–196.

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

[92] Sun, X. *Reverse Monte Carlo ray-tracing for radiative heat transfer in combustion systems*. PhD thesis, Dept. of Chemical Engineering, University of Utah, 2009.

[93] Sun, X., and Smith, P. J. A parametric case study in radiative heat transfer using the reverse monte-carlo ray-tracing with full-spectrum k-distribution method. *Journal of Heat Transfer 132*, 2 (2010).

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

[95] Thakur, R., Rabenseifner, R., and Gropp, W. D. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications 19*, 1 (2005), 49–66.

[96] Tran, L., and Berzins, M. Impice method for compressible flow problems in uintah. *International Journal for Numerical Methods in Fluids, Note: Published online 20* (2011).

[97] Tran, L., Kim, J., and Berzins, M. Solving time-dependent pdes using the material point method, a case study from gas dynamics. *International journal for numerical methods in fluids 62*, 7 (2010), 709–732.

[98] van der Holst, B., Toth, G., Sokolov, I., Powell, K., Holloway, J., et al. Crash: A Block-Adaptive-Mesh Code for Radiative Shock Hydrodynamics - Implementation and Verification. *Astrophys.J.Suppl. 194* (2011), 23.

[99] Wallstedt, P., and Guilkey, J. An evaluation of explicit time integration schemes for use with the generalized interpolation material point method. *Journal of Computational Physics 227*, 22 (2008), 9628–9642.

[100] Wise, J. H., and Abel, T. enzo+moray: radiation hydrodynamics adaptive mesh refinement simulations with adaptive ray tracing. *Monthly Notices of the Royal Astronomical Society 414*, 4 (2011), 3458–3491.