PORTABLE, SCALABLE APPROACHES FOR IMPROVING ASYNCHRONOUS MANY-TASK RUNTIME NODE USE

by

John Keith Holmen

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

Computing

School of Computing

The University of Utah

May 2022

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **John Keith Holmen**
has been approved by the following supervisory committee members:

Martin Berzins ,	Chair(s)	**03/29/2022** Date Approved
Robert Michael Kirby II ,	Member	**03/29/2022** Date Approved
Hari Sundar ,	Member	**03/31/2022** Date Approved
Mary W. Hall ,	Member	**04/04/2022** Date Approved
Victor Eijkhout ,	Member	**03/25/2022** Date Approved

by **Mary W. Hall** , Chair/Dean of
the Department/College/School of **Computing**
and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

This research addresses node-level scalability, portability, and heterogeneous computing challenges facing asynchronous many-task (AMT) runtime systems. These challenges have arisen due to increasing socket/core/thread counts and diversity among supported architectures on current and emerging high-performance computing (HPC) systems. This places greater emphasis on thread scalability and simultaneous use of diverse architectures to maximize node use and is complicated by architecture-specific programming models.

To reduce the exposure of application developers to such challenges, AMT programming models have emerged to offer a runtime-based solution. These models overdecompose a problem into many fine-grained tasks to be scheduled and executed by an underlying runtime to improve node-level concurrency. However, task execution granularity challenges remain, and it is unclear where and how shared memory programming models should be used within an AMT model to improve node use. This research aims to ease these design decisions with consideration for performance portability layers (PPLs), which provide a single interface to multiple shared memory programming models.

The contribution of this research is the design of a task scheduling approach for portably improving node use when extending AMT runtime systems to many-core and heterogeneous HPC systems with shared memory programming models. The success of this approach is shown through the portable adoption of a performance portability layer, Kokkos, within Uintah, a representative AMT runtime system. The resulting task scheduler enables the scheduling and execution of portable, fine-grained tasks across processors and accelerators simultaneously with flexible control over task execution granularity. A collection of experiments on current many-core and heterogeneous HPC systems are used to validate this approach and inform design recommendations. Among resulting recommendations are approaches for easing the adoption of a heterogeneous MPI+PPL task scheduling approach in an asynchronous many-task runtime system and furthermore to ease indirect adoption of a performance portability layer in large legacy codebases.

For my parents and my brother, Jacob.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

Adaptive Mesh Refinement	(AMR)
Asynchronous Many-Task	(AMT)
Application Programming Interface	(API)
Advanced Scientific Computing Advisory Committee	(ASCAC)
Center for the Simulation of Accidental Fires and Explosions	(C-SAFE)
Carbon Capture Multidisciplinary Simulation Center	(CCMSC)
Central Processing Unit	(CPU)
Compute Unified Device Architecture	(CUDA)
Department of Energy	(DOE)
Direct Quadrature Method of Moments	(DQMOM)
Exascale Computing Project	(ECP)
General Electric	(GE)
Graphics Processing Unit	(GPU)
High Performance Computing	(HPC)
Haswell	(HSW)
Implicit Continuous-fluid Eulerian	(ICE)
Knights Landing	(KNL)
Large-Eddy Simulation	(LES)
Lawrence Livermore National Laboratory	(LLNL)

Maxwell	(MAX)
Many Integrated Core	(MIC)
Message Passing Interface	(MPI)
Material Point Method	(MPM)
National Nuclear Security Administration	(NNSA)
National Research Center of Parallel Computer Engineering & Technology	(NRCPC)
National Science Foundation	(NSF)
Oak Ridge Leadership Computing Facility	(OLCF)
Open Multi-Processing	(OpenMP)
Portable Operating System Interface	(POSIX)
Pressure Poisson Equation	(PPE)
Performance Portability Layer	(PPL)
Predictive Science Academic Alliance Program	(PSAAP)
POSIX Thread	(PThread)
Reverse Monte Carlo Ray-Tracing	(RMCRT)
Refinement Ratio	(RR)
Single Instruction Multiple Data	(SIMD)
Skylake	(SKX)
Sandy Bridge	(SNB)
Texas Advanced Computing Center	(TACC)

ACKNOWLEDGEMENTS

I would like to thank the CCMSC and those involved with Uintah, especially Alan Humphrey, John Schmidt, Brad Peterson, Damodar Sahasrabudhe, Dan Sunderland, Todd Harman, Jeremy Thornock, and Derek Harris. I would also like to thank my committee for their advice, and especially my advisor Martin Berzins, for his guidance, commitment to excellence, and encouraged continuous improvement throughout my research. Finally, I would like to thank my parents and my brother, Jacob, for their patience and support. Anything is possible with hard work and motivation.

Funding Acknowledgements

- This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375.
- This work was also supported by the Intel Parallel Computing Centers Program.
- This work was suppported by the University of Texas at Austin under Award Number(s) UTA19-001215.
- This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under contract DE-AC02-06CH11357.
- This research used resources of the Lawrence Livermore National Laboratory.
- This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.
- This research used resources of the Texas Advanced Computing Center, under Award Number(s) MCA08X004 - "Resilience and Scalability of the Uintah Software".
- This research used resources of the University of Utah Intel Parallel Computing Center.

CHAPTER 1

INTRODUCTION

The complexity of nodes anticipated in exascale systems poses new challenges for codes emphasizing large-scale simulations. The key features contributing to these challenges are deep memory hierarchies and increasing accelerator, core, and thread counts relative to traditional high-performance computing (HPC) systems. Figure 1.1 shows a 2014 example of what an exascale node may look like with anticipated nodes to be constructed similarly.

An example of such increases can be seen comparing the National Science Foundation (NSF) Stampede systems. Stampede 1 compute nodes featured two 8-core Intel Sandy Bridge processors with 2 threads per core and one 61-core Intel Knights Corner coprocessor with 4 threads per core. Stampede 2 compute nodes feature either two 24-core Intel Skylake

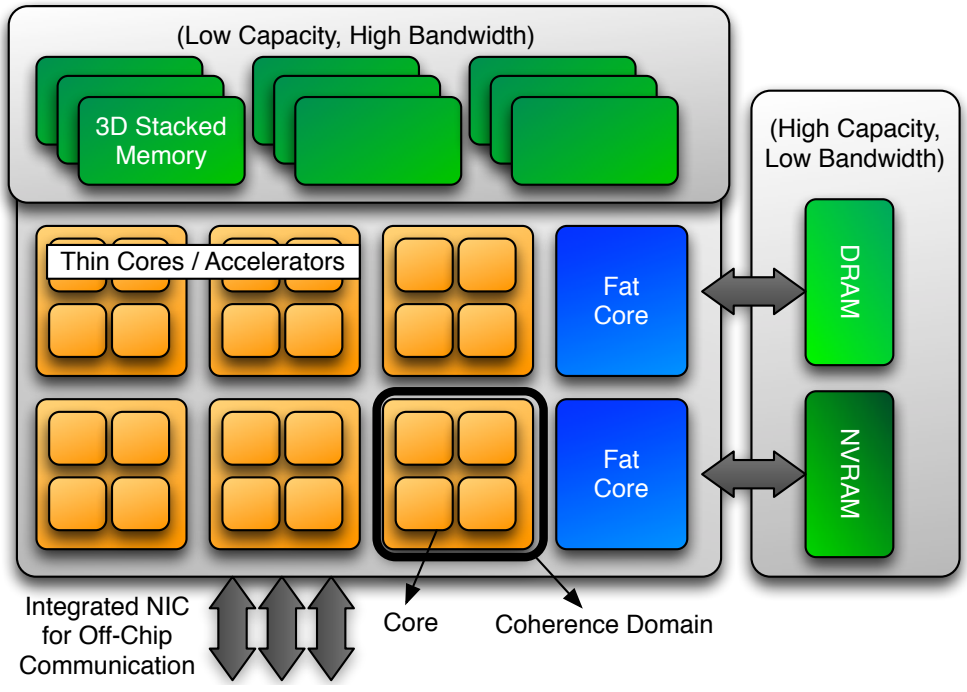

Fig. 1.1: Abstract machine model of an exascale node architecture [3].

processors with 2 threads per core or one 68-core Intel Knights Landing processor with 4 threads per core. Comparing node offerings, Intel Xeon core counts increased by $3x$ across systems, with the Intel Xeon Phi offering a $2x$ increase in per-core thread counts and up to $3.8x$ increases in core counts over the respective Intel Xeon.

Another example can be seen comparing the Department of Energy (DOE) Titan and DOE Summit systems. Titan compute nodes featured one 16-core AMD Opteron processor with 1 thread per core and one NVIDIA Kepler GPU. Summit compute nodes feature two 22-core IBM POWER9 processors with 4 threads per core and six NVIDIA Tesla GPUs. Comparing node offerings, core counts and per-core thread counts increased by $2.75x$ and $4x$, respectively, across systems with a $6x$ increase in per-node accelerator counts.

In addition to understanding how to manage such increases, one must also understand how to manage the increasing architectural diversity with additional consideration for novel emerging architectures. For example, the four systems mentioned feature AMD-, IBM-, Intel-, and NVIDIA-based architectures with exascale systems such as the DOE Aurora [5] and DOE Frontier [84] to include novel Intel- and AMD-based GPUs, respectively. The latter, however, complicates the preparation of codes for exascale systems as heterogeneous systems on the Top 10 of June 2021's Top500 list [109] are currently limited to NVIDIA-based GPUs. Two promising solutions for addressing these challenges are asynchronous many-task (AMT) runtime systems and performance portability layers (PPLs) [71].

1.1 Motivation

This dissertation is primarily motivated by the University of Utah's participation in the DOE / National Nuclear Security Administration (NNSA) Predictive Science Academic Alliance Program (PSAAP) II initiative. For this project, the University of Utah's Carbon Capture Multidisciplinary Simulation Center (CCMSC) has been using large-scale simulations to predict the performance of a commercial, 1200 MWe ultra-supercritical clean coal boiler developed by Alstom (GE) Power. These predictions support the design and evaluation of an existing boiler capable of providing power for nearly 1 million people. Figure 1.2 shows an example of such a power plant, which is approximately 90 meters tall.

CCMSC predictive boiler simulations have been made possible through the use of the reacting, large eddy simulation (LES)-based codes in the Uintah Computational Framework

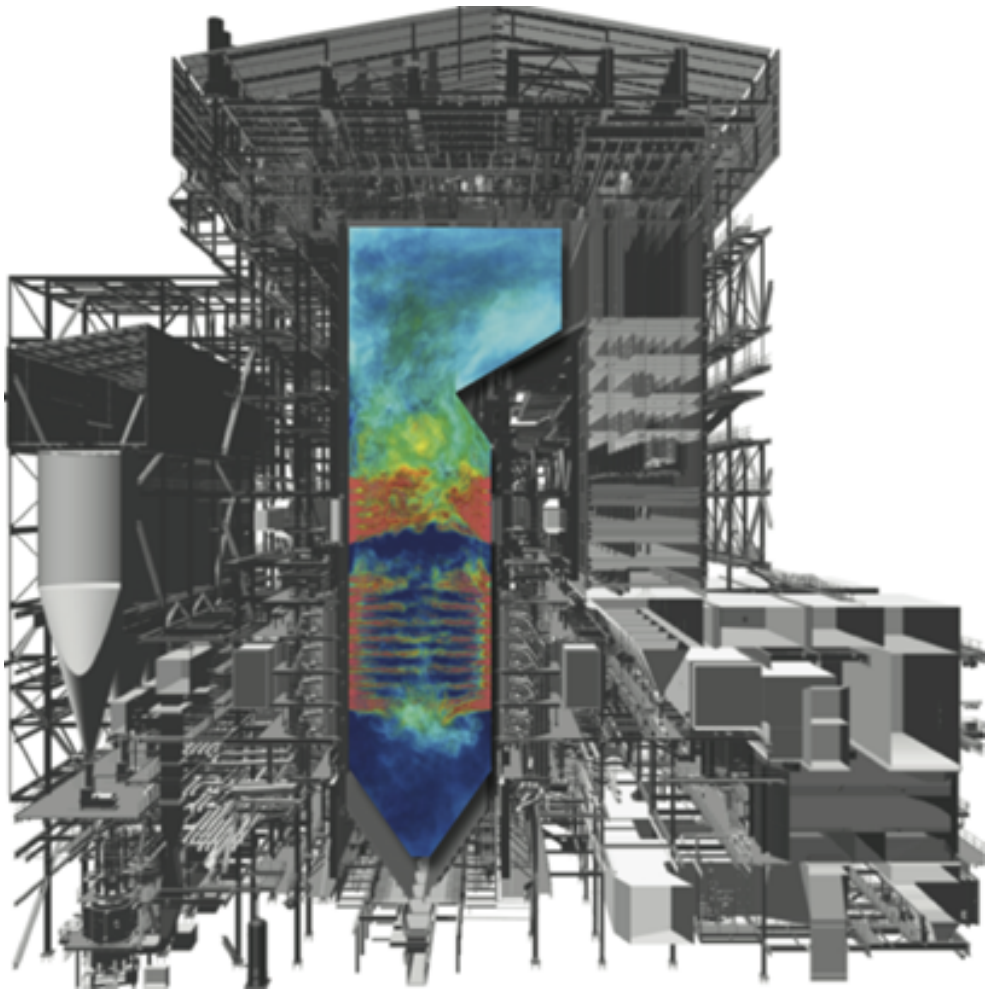

Fig. 1.2: GE Power's 1200 MWe ultra-supercritical clean coal boiler.

[18] and large HPC systems such as the NSF Stampede, DOE Mira, and DOE Titan systems. Intermediate simulations have used available HPC systems to simulate computational domains at lower resolutions for feasibility. Spatial and temporal requirements for target simulations produce problems 50 to 1,000 times larger than solved today and have been considered good exascale candidates. For example, approximately 9 trillion cells are needed to simulate such a boiler to 1-millimeter resolution.

The next phase of simulations aim to use the exascale DOE Aurora system through the Aurora Early Science Program. Aurora has been an early target system for the center, which was formed in 2014. Emphasis on performance portability is motivated by uncertainty surrounding which of the already diverse petascale systems will be available for intermediate simulations and uncertainty surrounding Aurora itself. For example, Aurora was initially

to feature the since discontinued Intel Xeon Phi and will now feature Intel Xe GPUs.

More broadly, this dissertation is motivated by the challenges reported [71] by the DOE's Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee for the Top Ten Exascale Research Challenges:

1. Energy Efficiency: Creating more energy-efficient circuit, power, and cooling technologies.

2. Interconnect Technology: Increasing the performance and energy efficiency of data movement.

3. Memory Technology: Integrating advanced memory technologies to improve both capacity and bandwidth.

4. Scalable System Software: Developing scalable system software that is power- and resilience-aware.

5. Programming Systems: Inventing new programming environments that express massive parallelism, data locality, and resilience.

6. Data Management: Creating data management software that can handle the volume, velocity and diversity of data that is anticipated.

7. Exascale Algorithms: Reformulating science problems and redesigning, or reinventing, their solution algorithms for exascale systems.

8. Algorithms for Discovery, Design, and Decision: Facilitating mathematical optimization and uncertainty quantification for exascale discovery, design, and decision making.

9. Resilience and Correctness: Ensuring correct scientific computation in the face of faults, reproducibility, and algorithm verification challenges.

10. Scientific Productivity: Increasing the productivity of computational scientists with new software engineering tools and environments.

Specific challenges motivating the research contributing to this dissertation include: "5. Programming Systems," "7. Exascale Algorithms," and "10. Scientific Productivity."

1.2 Target Architectures and Systems

The research contributing to this dissertation uses a variety of microarchitectures and major HPC systems to demonstrate approaches for addressing the exascale challenges related to increased concurrency, deep memory hierarchies, and architectural diversity. Microarchitectures examined include Intel Sandy Bridge, Intel Knights Corner, Intel Haswell, NVIDIA Maxwell, Intel Skylake, Intel Knights Landing, IBM POWER9, NVIDIA Volta, and Intel Cascade Lake. Systems examined include the NSF Stampede 1, NSF Stampede 2, DOE Titan, DOE Lassen, DOE Summit, and NSF Frontera. Broadly, this research targets microarchitectures comprising current and emerging major HPC systems with a future look towards the proposed DOE Aurora, DOE Frontier, and DOE El Capitan exascale systems.

1.3 Target Exascale Benchmarks

The research contributing to this dissertation uses a variety of intermediate benchmarks while working towards two exascale benchmarks that uniquely stress different portions of three individually ported codes: (1) Uintah's ARCHES turbulent combustion simulation component [106], (2) Uintah's standalone linear solver using Lawrence Livermore National Laboratory's hypre [34], and (3) Uintah's standalone reverse Monte-Carlo ray tracing (RMCRT) radiation model [54]. These codes are central to both CCMSC boiler simulations and subsequent combustion research.

The first problem, a helium plume, demonstrates the newly portable interoperability of (1) and (2) on a single-level structured grid. A key feature making this an essential problem for validating Uintah's heterogeneous MPI+Kokkos task scheduling approach is the large number of unique portable loops and variables in flight during execution. This helps ensure robustness due to the long and complex data dependency sequences generated by these loops (e.g., variables computed on the host, modified on the device, and later required on the host).

The second problem, a modified Burns and Christon benchmark [22], demonstrates newly portable interoperability of (1), (2), and (3) on a 2-level structured adaptive mesh refinement grid. A key feature making this an essential problem for validating Uintah's heterogeneous MPI+Kokkos task scheduling approach is the ability to simultaneously stress interoperability of ARCHES, hypre, and RMCRT while also stressing Uintah's adaptive

mesh refinement support. This helps ensure robustness due to the complex hand-offs taking place between these codes (e.g., shared data dependencies).

More details on both will be discussed in Section 3.5.1 and Section 3.5.4.

1.4 Thesis Statement

This research aims to demonstrate how a performance portability layer can be adopted in a large asynchronous many-task runtime system to achieve scalable performance portability for large-scale simulations on current and emerging HPC systems featuring diverse microarchitectures. This aim is achieved by indirectly adopting Kokkos, a performance portability layer, in a representative asynchronous many-task runtime system, Uintah, and extending Uintah's heterogeneous MPI+X task scheduling capabilities to support heterogeneous MPI+Kokkos task scheduling using Kokkos::OpenMP and Kokkos::CUDA on the host and device, respectively. This dissertation shows that it is possible to combine these promising solutions for exascale computing in a scalable manner for real-world applications with good strong scaling achieved across a many-core Intel Knights Landing system, a multicore Intel Cascade Lake system, and heterogeneous IBM POWER9 and NVIDIA Volta-based systems.

1.5 Dissertation Contributions

This research is believed to be helpful to others in the HPC community, for whom portability and scalability are also challenges, given trends among current and emerging HPC systems. In particular, this research will be of importance to those aiming to achieve scalable performance portability and evaluating performance portability layers such as Kokkos with lessons learned through this research helping to understand both adoption techniques and what performance and scalability is achievable in real-world applications. The broader impact of this research may ultimately include further adoption of Kokkos as means of portably preparing one's code for future architectures. Unique contributions resulting from this dissertation's research include:

1. Developing an MPI+PThreads task scheduling approach for many-core architectures and identifying that such architectures require greater attention to run configuration and domain decomposition as demonstrated by 10.1% performance differences across run configurations on Intel Sandy Bridge compared to performance differences up to

149.3% on Knights Corner [47].

2. Portably addressing domain decomposition challenges related to serial tasks by implementing an intermediate performance portability layer and underlying portable Kokkos-based data-parallel tasks. The resulting implementation achieved good strong scaling characteristics to 65,536 threads across 256 Knights Landing processors with node-level performance improvements up to 3.00x [48].

3. Portably addressing thread scalability challenges related to serial execution of portable Kokkos-based data-parallel tasks within an MPI process by implementing a task scheduler enabling parallel execution of portable tasks within an MPI process. The resulting implementation achieved good strong scaling characteristics to 442,368 threads across 1,728 Knights Landing processors with performance improvements up to 1.62x demonstrated at scale and little overhead added ($< 0.2\%$ per timestep) [49].

4. Detailed studies characterizing thread scalability on Intel and NVIDIA architectures with node-level performance improvements up to 2.63x demonstrated when more efficiently using a node with the newly implemented task scheduler [49].

5. Portably addressing heterogeneous execution challenges by implementing a task scheduler enabling simultaneous use of Kokkos on both host and device. The resulting implementation achieved good strong scaling characteristics to 8,192 IBM POWER9 processors and 24,576 NVIDIA V100 GPUs with performance improvements up to 4.4x when using this scheduler and the accompanying portable abstractions to port a previously MPI-Only problem to use both host and device [51].

6. Developing an approach for indirectly adopting a performance portability in large legacy codes [49] and a heterogeneous MPI+PPL task scheduling approach for asynchronous many-task runtime systems [51].

7. Designing two representative Uintah workloads capable of stressing Uintah's portable infrastructure in meaningful manners and suitable for use as an exascale benchmark [51].

1.6 Document Organization

The remainder of this dissertation is organized as follows: Chapter 2 provides an overview of related solutions showing promise for addressing exascale challenges. Chapter 3 provides an overview of the Uintah Computational Framework and relevant components. Chapter 4 describes Uintah's MPI+PThreads task scheduling approach and presents single-node and multi-node results. Chapter 5 describes Uintah's MPI+Kokkos::OpenMP task scheduling approach and presents single-node and multi-node results. Chapter 6 describes an approach for indirectly adopting a performance portability layer in a large code and presents single-node results. Chapter 7 describes a heterogeneous MPI+PPL task scheduling approach for asynchronous many-task runtime systems and presents multi-node results. Chapter 8 describes design guidelines for easing adoption of contributed approaches and Chapter 9 concludes this dissertation.

CHAPTER 2

RELATED EXASCALE SOLUTIONS

This chapter surveys solutions related to Uintah, Kokkos, and, broadly, the approaches discussed within this dissertation. Detailed overviews of Uintah and Kokkos can be found in Chapter 3 and Chapter 5, respectively. Given the emphasis on exascale computing challenges, related work is centered around software solutions from the Exascale Computing Project (ECP). These solutions target platform portability across the DOE Summit, Aurora, Frontier, and El Capitan systems in varying capacities. Details on software implementations used by all ECP application codes can be found in a recent survey [33]. Additionally, this chapter discusses related Uintah-specific efforts contributing to other dissertations.

2.1 Overview

Asynchronous many-task (AMT) runtime systems show promise for helping to manage the increased concurrency, deep memory hierarchies, and heterogeneity anticipated with exascale computing. This promise lies in their ability to increase node-level parallelism by overdecomposing an application into many tasks while also easing the use of such nodes by offloading low-level details for making use of the underlying hardware to the runtime itself. Uintah is one of many such runtimes. Other examples include Charm++ [64], DARMA [119], HPX [62], Legion [12], OCR [75], PaRSEC [20], STAPL [23], and StarPU [8]. Comparisons of these and other runtimes can be found in surveys [14, 68, 107] and recent Uintah dissertations [57, 89].

Uintah is also commonly classified among block-structured adaptive mesh refinement frameworks. Other examples include Athena++ [108], BoxLib [121] (superseded by AMReX [124]), Cactus [37], Enzo [86], and FLASH [19]. Comparisons of these and other frameworks can be found in a survey [30].

Hybrid parallelism approaches are commonly used by such codes emphasizing large-

scale simulation and show promise for helping manage the increased concurrency and deep memory hierarchies anticipated with exascale computing. Here, hybrid parallelism refers to the MPI+X programming model, where MPI is used for distributed memory parallelism and X (e.g., OpenMP) is used for shared memory parallelism. This promise lies in their ability to ease load balancing by shifting work among cores with shared memory rather than distributed memory and avoiding communication of data already on-node. For many-core and multicore systems, OpenMP and PThreads are commonly used. For GPU-based systems, CUDA and OpenCL are commonly used. Comparisons of these and other programming models can be found in surveys [24,31,73].

Among newer options for the "X" in MPI+X are performance portability layers (PPLs). Performance portability layers show promise for helping to manage the architectural diversity anticipated with exascale computing. This promise lies in their ability to interact with multiple underlying programming models (e.g., CUDA, HIP, OpenMP, etc.) through a single interface while also easing the use of such nodes by offloading low-level details for making efficient use of the underlying programming models to the layer itself. Kokkos [118] is one of many such layers. Other examples include DPC++ [98], HEMI [44], OCCA [76], RAJA [52], and SYCL [100]. Details on these and other layers can be found in a survey [61], comparative studies [41,42], and a recent Uintah dissertation [89].

Uintah is an early adopter of Kokkos with Uintah developers collaborating directly with Kokkos developers as a part of the University of Utah's participation in the DOE/NNSA's Predictive Science Academic Alliance Program (PSAAP) II initiative. This collaboration has resulted in bi-directional development efforts with developers working in each other's codebases. At Sandia National Laboratories, Kokkos has been integrated in Trilinos [46] and used in codes such as Albany [28], GenTen [95], HOMMEXX [15], LAMMPS [96], and SPARTA [36]. Examples of other codes investigating and/or adopting Kokkos include BabelStream [27], K-Athena [38], KARFS [97], NekMesh [32], and TeaLeaf [74]. A list of applications using Kokkos can be found on the Kokkos GitHub [117].

The sections to follow discuss ECP-related solutions.

2.2 AMReX

AMReX [122–124] is a C++ software framework that supports the development of block-structured adaptive mesh refinement (AMR) algorithms for solving systems of partial differential equations with complex boundary conditions on current and emerging architectures. For the ECP, the framework is used for applications including accelerator design, additive manufacturing, astrophysics, combustion, cosmology, multiphase flow, and wind energy. The precursor to AMReX is BoxLib [121] for which more details can be found in a block-structured AMR survey [30].

AMReX shares a similar design philosophy with Uintah in taking care to separate the design of data structures and basic operations from the algorithms that use the data structures. This split allows for flexibility in how application developers interact with the various levels of abstraction that are available. A key difference from Uintah is that abstractions are more compartmentalized and developers can choose, for example, to use only the AMReX data containers and iterators without higher-level functionality such as subcycling-in-time algorithms.

For performance portability, AMReX has implemented its own lightweight abstraction layer [123] to hide architecture details much like the intermediate portability layer proposed here. Rather than adopt Kokkos or RAJA, they've implemented their own *ParallelFor* looping constructs that take a C++ lambda-based loop body. For GPU, their loops map to CUDA, HIP, and DPC++ back-ends. For CPU, their loops map to C++ back-ends. A key difference between their layer is how OpenMP is handled. OpenMP is only used for vectorization inside of *ParallelFor* with parallelization over cells handled at the data iterator level outside of *ParallelFor*.

2.3 HPX

HPX [26, 45, 62, 63] is a C++ standard library for concurrency and parallelism that aims to achieve dynamic adaptive resource management and lightweight task scheduling in the context of a global address space. HPX fully conforms to the C++ ISO standards and implements the standardized concurrency and parallelism facilities with extensions to support distributed and data-flow programming cases. In doing so, it offers application developers the means to naturally use key AMT features such as overlapping communication and

computation and oversubscribing of execution resources.

The core of HPX's runtime is the thread manager, which maps lightweight HPX threads to kernel threads. As HPX threads don't require kernel calls, their goal is to use millions of threads per second on each core. This is paired with predefined scheduling policies and work-stealing to determine optimal task scheduling. Current policies are static, thread-local, and hierarchical. The static policy maintains one queue per core with no work-stealing. The thread-local policy maintains one queue per core and allows work-stealing from neighboring queues when cores run out of work. Lastly, the hierarchical policy allows for trees of queues to be created for cooperative use of queues. This model differs from Uintah in that Uintah targets few threads per core at most and uses per MPI process queues to distribute work across cores with implicit work-stealing.

For performance portability, HPX has adopted Kokkos with contributions made to both codebases. In Kokkos, an HPX back-end was created to offer Kokkos a well-established way of synchronizing, sequencing, and overlapping tasks. In HPX, HPX-specific executors and execution policies were created to dispatch parallel algorithms to Kokkos. The latter incorporates what is referred to as an HPX-Kokkos interoperability layer [26] in HPX that is similar to the intermediate portability layer proposed here. In the case of HPX, this layer is used to extend Kokkos parallel patterns (e.g., to return futures) and provide custom HPX parallel algorithms (e.g., hpx::for_loop) dispatched to Kokkos (Kokkos::parallel_for).

2.4 RAJA

RAJA [13,52,53] is a C++ library developed at Lawrence Livermore National Laboratory for providing software abstractions that insulate application source code from hardware architecture and programming model-specific implementation details. RAJA is similar to Kokkos in that it is also loop-based and uses C++ templating to accommodate multiple, interchangeable back-ends. Key concepts are (1) execution policies, (2) iteration spaces, and (3) traversal templates. (1) is a C++ type that specifies how a loop kernel will execute. (2) defines a set of loop indices for a kernel. (3) defines operations performed on a lambda loop body based on execution policy specialization and an iteration space object.

Both Kokkos and RAJA are key performance portability layers used by ECP applications. For 2020-2023, the libraries are supported through the same ECP project, 2.3.1.18

RAJA/Kokkos. A key goal of this project is to deliver back-end support for both the DOE Aurora and DOE Frontier systems that can be shared between Kokkos and RAJA. As of release 0.14.0, RAJA supports back-ends to Intel Threading Building Blocks, NVIDIA CUDA, OpenMP, OpenMP Target, and AMD HIP. OpenMP target offloading is a key focus for Aurora with efforts also underway to explore a SYCL back-end.

A key difference from Kokkos is that RAJA does not manage the placement of memory. RAJA views are similar to unmanaged Kokkos views and wrap a pointer to a block of memory. This was a key design decision to allow RAJA to focus on ease of expression and reducing the impact on application code in an effort to be non-invasive. For users looking to offload memory management, there are two associated libraries, Umpire [53] and CHAI [53], which cater to memory. Umpire aims to decouple memory operations such as copy and move from platform-specific memory spaces and offers portable memory management functions. CHAI aims to ensure that data used in RAJA kernels are in the proper memory space and provides a managed array abstraction to copy data as needed between memory spaces.

2.5 SYCL/DPC++

SYCL [100] is a single-source heterogeneous programming model built on C++ whose standard is maintained by Khronos Group. Similar to Kokkos and RAJA, SYCL offers a *parallel_for* loop construct. A key difference between models is that SYCL does not offer reduce or scan operations. Along similar lines, SYCL offers a buffer mechanism for memory rather than views. Buffers are data containers that can be read/written by kernel code and host code. SYCL automatically coordinates buffer data movement between host and device using an accessor mechanism that developers use to indicate when data is read or written.

SYCL has a variety of implementations with the most relevant to this research and the ECP being DPC++ [6, 98]. DPC++ is a SYCL implementation from Intel that uses Clang and LLVM [70] and is a part of the oneAPI project. This implementation extends SYCL functionality with new features such as unified shared memory, unnamed kernel lambdas, in-order queues, and reductions. Long-term goals for DPC++ are similar to those of Kokkos in that both aim to be a proving grounds for new functionality to be later incorporated into standards. Whereas Kokkos developers target the C++ standards, DPC++ developers target

the SYCL standard.

Of DPC++ extensions, unified shared memory and reductions bring DPC++ offerings closer to those of Kokkos and RAJA. Unified shared memory offers a pointer-based alternative to SYCL buffers that requires all devices to use a unified address space with the host. Supported allocation types are device, host, and shared. A key advantage of this offering over SYCL buffers is that shared allocations, which are accessible by both host and device, are meant to migrate data where it is being used without developer intervention. This eases the non-trivial effort of writing memory allocations to use SYCL's buffer mechanism.

The section to follow discusses related Uintah-specific efforts contributing to other dissertations.

2.6 Uintah Collaborations

The highly collaborative nature of the University of Utah's CCMSC has resulted in this dissertation progressing alongside several other dissertations using Uintah. As a result, this dissertation shares applications, experiments, and implementations with those of Alan Humphrey [57], Brad Peterson [89], and Damodar Sahasrabudhe [101]. However, there are distinctly different emphases and research directions in these theses.

Overlap with Alan Humphrey relates to the use of RMCRT to demonstrate how Uintah can be adapted to scale well on current petascale systems and emerging exascale systems. Humphrey's efforts primarily relate to Uintah's adaptive mesh refinement infrastructure and the directed acyclic graph used to represent computation and related data dependencies in RMCRT. This research differs in that it relates to Uintah's task scheduling infrastructure and extends the RMCRT implementations used to support portable execution through the Kokkos performance portability library.

Overlap with Brad Peterson relates to GPU task portability and GPU task scheduling. Specific overlap includes collaborative development efforts and experiments towards Chapter 6. Peterson's efforts primarily relate to individual task portability, Uintah's GPU task scheduling infrastructure, and adapting Kokkos' execution model to Uintah's execution model. This research makes use of Peterson's portable task infrastructure, Kokkos::CUDA task scheduling capabilities, and custom Kokkos implementation. This research differs in

that it emphasizes portability across large numbers of unique portable tasks and extends task scheduler capabilities to support, for example, read/write data and simultaneous use of Kokkos::OpenMP and Kokkos::CUDA rather than PThreads and Kokkos::CUDA. Among Chapter 6 results are select Maxwell-based results gathered by Peterson.

Overlap with Damodar Sahasrabudhe relates to heterogeneous task scheduling. Specific overlap includes collaborative development efforts and experiments towards Chapter 7. Sahasrabudhe's efforts primarily relate to scheduling third-party library tasks and extending Peterson's GPU task scheduler to support large-scale portable ARCHES simulations. This research makes use of the jointly developed GPU task scheduler as the foundation for Kokkos::CUDA capabilities in the task scheduler shown here. This research differs in that it emphasizes scheduling Uintah tasks and extends task scheduler capability to support simultaneous use of Kokkos::OpenMP and Kokkos::CUDA rather than PThreads and Kokkos::CUDA. Among Chapter 7 results are select Lassen, Summit, and Frontera results gathered jointly with Sahasrabudhe.

Further distinctions between these different but complementary efforts are made as they arise in individual chapters.

CHAPTER 3

THE UINTAH COMPUTATIONAL FRAMEWORK

3.1 Overview

The Uintah Computational Framework is an open-source asynchronous many-task (AMT) runtime system specializing in large-scale simulation of fluid-structure interaction problems. These problems are modeled by solving partial differential equations on structured adaptive mesh refinement grids. Uintah is based upon novel techniques for understanding a broad set of fluid-structure interaction problems. [17]

Uintah was initially developed by the University of Utah's Center for the Simulation of Accidental Fires and Explosions (C-SAFE), which was started in 1997 through the Department of Energy's Advanced Simulation and Computing program. C-SAFE focused on providing state-of-the-art, science-based tools for the numerical simulation of accidental fires and explosions with emphasis on handling and storage of highly flammable materials. The center's goal was to provide a software system in which fundamental chemistry and engineering physics are fully coupled with nonlinear solvers, optimization, computational steering, visualization, and experimental data verification. The resulting system, Uintah, was used to help evaluate the risks and safety issues associated with fires and explosions in accidents involving both hydrocarbon and energetic materials. The target simulation for this project was the heating of an explosive device placed in a large hydrocarbon pool fire and the subsequent deflagration explosion and blast wave [87].

In addition to projects such as C-SAFE and PSAAP II, Uintah has also experienced notable development as a part of student dissertations such as this one. Justin Luitjen's dissertation [72] research introduced adaptive mesh refinement support. Qingyu Meng's dissertation [82] research introduced dynamic task scheduling and several task schedulers. Alan Humphrey's dissertation [57] research introduced a scalable approach to radiation

modeling. Brad Peterson's dissertation [89] research introduced performant and portable GPU support. Damodar Sahasrabudhe's dissertation [101] research introduced a resiliency component and other solutions aimed at addressing the exascale challenges motivating this dissertation.

Uintah specializes in large-scale simulation and has been widely ported across a diverse set of leadership-class HPC systems. For multicore systems, good scaling characteristics have been demonstrated to 96K, 262K, 700K, and 700K cores on the NSF Stampede, DOE Titan, DOE Mira, and NSF Blue Waters systems, respectively [17, 54, 80]. For GPU-based systems, good strong scaling characteristics have been demonstrated to 16K GPUs [56] on the DOE Titan system. For many-core systems, good strong scaling characteristics have been demonstrated to 256 Knights Landing processors on the NSF Stampede system [48] using Uintah's preliminary MPI+Kokkos hybrid parallelism approach and 128 core groups on the National Research Center of Parallel Computer Engineering and Technology (NRCPC) Sunway TaihuLight system [120]. For standalone use of Kokkos::OpenMP in the task scheduler resulting from this research, good strong scaling has been shown to 1,728 Intel Knights Landing processors on the NSF Stampede 2 system [49]. For standalone use of Kokkos::CUDA in the task scheduler resulting from this research, good strong scaling has been shown to 64 NVIDIA K20X GPUs on the DOE Titan system [94] by Brad Peterson. For heterogeneous use of Kokkos::OpenMP and Kokkos::CUDA in the task scheduler resulting from this research, good strong scaling has been shown to 1,024 NVIDIA V100 GPUs and 512 IBM POWER9 processors on the DOE Lassen system [51].

Released in May of 2017, Uintah release 2.0.0 features four primary simulation components:

- *ARCHES*: This component targets the simulation of turbulent reacting flows with participating media radiation [106].
- *ICE*: This component targets the simulation of both low-speed and high-speed compressible flows [65].
- *MPM*: This component targets the simulation of multi-material, particle-based structural mechanics [111].

- *MPM-ICE*: This component corresponds to the combination of the ICE and MPM components for the simulation of fluid-structure interactions [40, 43].

For its boiler simulations, the CCMSC used the ARCHES turbulent combustion simulation component. ARCHES is a Large Eddy Simulation (LES) code described further in [106]. This code is second-order accurate in space and time and uses a low-Mach number, (M <0.3), variable density formulation to model heat, mass, and momentum transport in turbulent reacting flows. The LES algorithm used solves the filter, density-weighted, time-dependent coupled conservation equations for mass, momentum, energy, and particle moment equations.

A key idea maintained in Uintah is that application developers are isolated from infrastructure code. This is accomplished using an AMT-based approach to overdecompose application code into tasks and the computational domain into groups of individual cells, which tasks iterate over, to increase node-level parallelism. This approach is used to simplify development while easing the use of the underlying hardware for application developers. For application developers, this divide allows them to focus on writing loop-based tasks rather than building an understanding of low-level execution details (e.g., data access patterns, load balancing, task scheduling). For infrastructure developers, this divide allows for fine-tuning of such details to be managed in a central location, reducing the need for far-reaching changes across application code.

The topmost layer of Uintah, application code, consists of simulation components such as ARCHES, which has been the focus of Uintah's exascale computing goals. Application code is decomposed into individual tasks that correspond to, for example, physics routines that are executed on either the host or device. The resulting collection of tasks is compiled into a task graph and dynamically executed by the bottommost layer, infrastructure code, in an asynchronous out-of-order manner with implicit work-stealing using the underlying runtime system. Execution is managed by the task scheduler, which interacts with per-MPI process task queues to select and execute ready tasks (e.g., tasks with satisfied data dependencies). This dissertation's research focuses on both application code and infrastructure code.

3.2 Task Schedulers

In Uintah, the task scheduler component is responsible for computing task dependencies, determining the order of task execution, and ensuring that the correct inter-process communication is performed. The Uintah task scheduler compiles all of the tasks and variable dependencies into a task graph. Dependency edges are added between tasks based on the supplied variable dependencies. The computed dependency edges can be either internal or external. 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 specify required communication. The compilation process also combines external dependencies from the same source or to the same destination, thus coalescing messages [39].

Uintah has three primary task schedulers: (1) the MPI Scheduler, (2) the Dynamic MPI Scheduler, and (3) the Unified Scheduler. The MPI Scheduler features static task ordering and deterministic execution of tasks using MPI-only. The Dynamic MPI Scheduler features dynamic task scheduling with non-deterministic, out-of-order execution of tasks using MPI-only. The Unified Scheduler features dynamic task scheduling with non-deterministic, out-of-order execution of tasks using MPI+PThreads and MPI+PThreads+CUDA for GPU support. More details on Uintah's task schedulers can be found in a scheduler survey [79].

It has been shown that there is a substantial increase in MPI communication time at larger numbers of cores due to dependencies between computing tasks distributed to different nodes and Uintah's memory use associated with ghost cells and global meta-data [78]. This increase in MPI communication time becomes a barrier to scalability beyond $\mathcal{O}(100K)$ cores. Beyond these core counts, one has to employ Uintah's Unified Scheduler, which moves to a shared memory model on-node, to drastically reduce the memory footprint seen at high core counts with an MPI-only approach [39,77,78]. With the exception of an early prototype in Chapter 5, this dissertation's research focuses on extending Uintah's Unified Scheduler.

Figure 3.1 shows the basic per-MPI process infrastructure forming the Unified Scheduler. The core of this infrastructure are the centrally located "CPU Core" and "GPU" ovals, which correspond conceptually to individual task executors. A task executor corresponds to the specific compute resources (e.g., cores) used to execute task executor logic responsible for task scheduling and execution. Task executors interact with task queues, process MPI sends

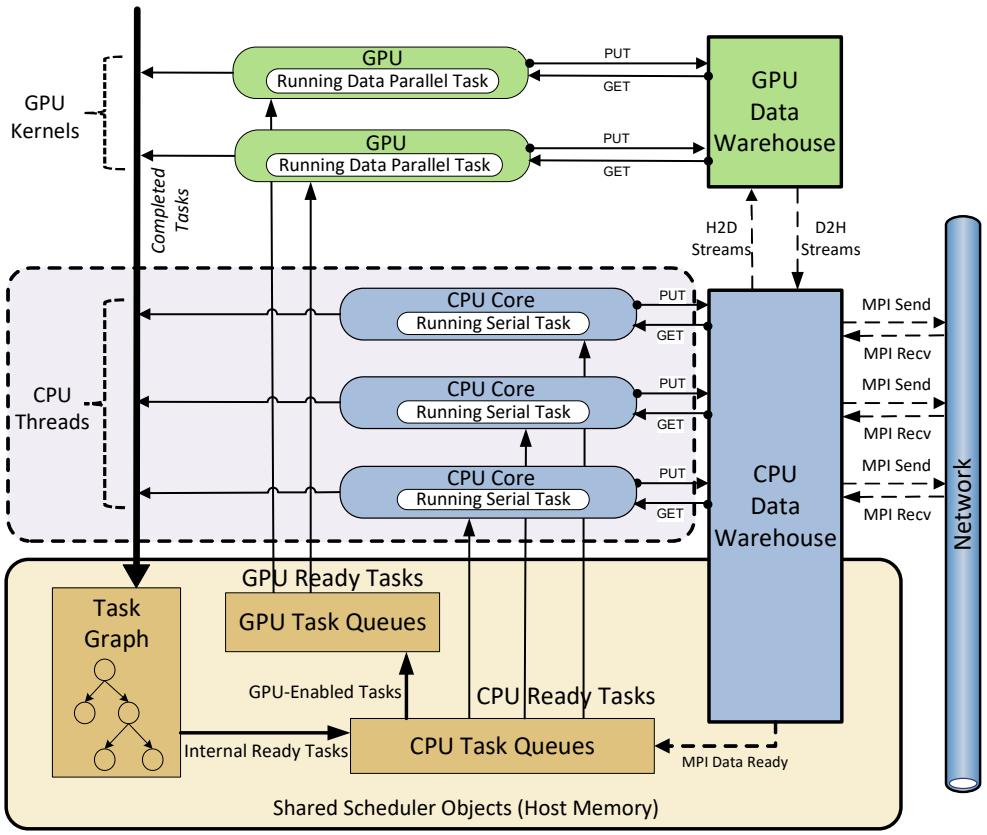

Fig. 3.1: Uintah's multi-threaded MPI scheduler [48].

and recvs, and share infrastructure components (e.g., regridder, load balancer, task graph, and data warehouse) within an MPI process and with read/write access to each using efficient, lock-free data structures.

Uintah adopted an MPI+X hybrid parallelism approach using the MPI+PThreads task scheduler [78] to overcome memory footprint limitations on the NSF Kraken and DOE Jaguar systems. Iterative efforts since have targeted extensions in four key areas: (1) support for heterogeneous systems [51, 55, 56, 79, 90–92] (2) support for many-core systems [47, 81], (3) portability of (1) and (2) [48–50, 93, 94, 113], and (4) support for third-party libraries using their own hybrid parallelism approaches [102, 103]. The research contributing to this dissertation targets (1) [51], (2) [47], and (3) [48–50, 93]. The scope and non-trivial nature of this collective effort has resulted in the implementation of several standalone task schedulers through the years.

The standalone task schedulers arrived at as a result of these extensions and available in Uintah today include: (1) a production-grade heterogeneous MPI+PThreads+CUDA

task scheduler, (2) an intermediate MPI+Kokkos::OpenMP task scheduler [48–50], (3) an intermediate heterogeneous MPI+PThreads+Kokkos::CUDA task scheduler, and (4) the heterogeneous MPI+Kokkos::OpenMP+Kokkos::CUDA task scheduler [51] demonstrated here. Schedulers (2) and (4) form the basis of discussion for Chapters 5 and 7, respectively, and are the focus of several of the author's publications [48–51]. For (2), (3), and (4), the production-grade task executor logic in (1) has been extended to support use of *Kokkos::parallel_for*, *Kokkos::parallel_reduce*, *Kokkos::View*, *Kokkos::Experimental::MasterLock*, and *Kokkos_Random* using the respective back-ends. Note, (2) and (3) have been strategically maintained, despite being replaceable by (4), for their invaluable ability to reduce complexity for development and debugging effort.

3.3 Simulation Domains

Uintah's tasks are scheduled and executed across three-dimensional structured grids of hexahedral cells configured by the user via an input file. Grids can consist of a single level of resolution or multiple levels of resolution when using Uintah's support for adaptive mesh refinement (AMR) [16]. For the latter, AMR can, for example, be used to resolve a region of interest at a finer resolution. In this case, a given level's cells are subdivided into smaller cells to generate the finer grid level.

At run-time, Uintah's grid is decomposed into a collection of patches, which consist of individual cells. Figure 3.2 provides an example of a patch. This patch consists primarily of coarse cells with a fine cell region of interest included in the top right corner. Local tasks, data variables, and particles reside on patches with four local tasks shown in the example. Also shown in this example is the layer of cells from neighboring patches, referred to as ghost cells, which are communicated via MPI for Uintah's stencil calculations.

Patches make possible Uintah's many-task model and are the primary unit of work in simulations. Tasks reside on patches and compute results for individual cells belonging to a given patch. Given this relationship, the number of patches in a simulation provides means of controlling the granularity of tasks in a simulation. Typical patch sizes used for production simulations include 10^3, 12^3, 16^3, and 32^3 [82]. As will be shown later in this dissertation, larger patches are needed for individual tasks to scale well across compute resources.

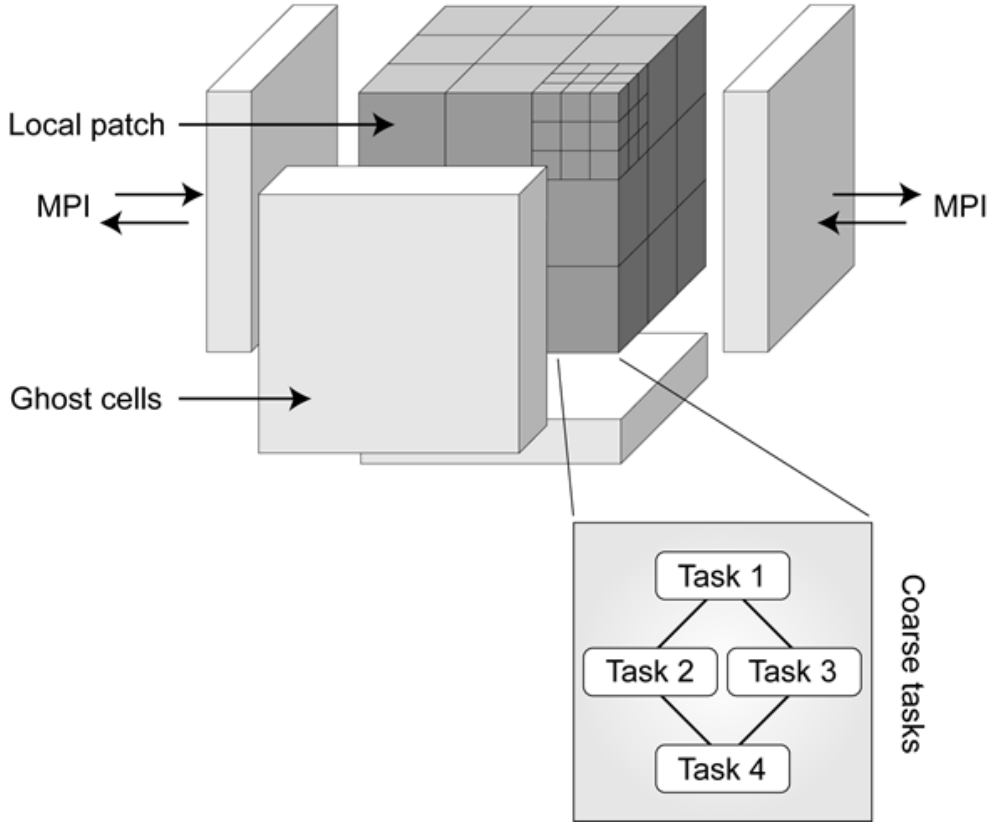

Fig. 3.2: Uintah patch.

3.4 Hypre

At every time sub-step, an implicit pressure projection is used as part of the low-Mach pressure formulation used by ARCHES [106]. This projection is formulated as a linear system and solved with the help of standalone linear solver packages. Currently, Uintah supports both hypre [34] and PETSc [11] for solving such systems.

For CCMSC predictive boiler simulations, hypre has been chosen as the target solver for its scalability [9,10]. Since, good scaling characteristics when using hypre in Uintah have been shown up to 512K cores [69,104]. For this reason, significant effort has also been put into optimizing Uintah's use of hypre for many-core- and GPU-based architectures [102,103] including extensions of hypre itself [101] to support the center's exascale goals.

Hypre offers a collection of scalable linear solvers for the large-scale solution of linear systems of equations on major HPC systems. The library features parallel multigrid methods for both structured and unstructured grids with emphasis on algebraic multigrid. In

preparation for heterogeneous and exascale computing, hypre has been ported to OpenMP, CUDA, Kokkos, and RAJA [35]. Note, hypre use for this dissertation uses the Conjugate Gradient method with the PFMG preconditioner based upon a Jacobi relaxation method for the structured multigrid approach.

3.5 Target Applications

This research uses 4 applications to demonstrate this dissertation's contributions. These applications are (1) a novel Reverse Monte-Carlo Ray Tracing (RMCRT)-based radiation model, (2) a char oxidation model, (3) a radiative particle property model, and (4) a helium plume problem. These applications have been chosen for their relevance to an important class of fluid-structure interaction and combustion problems such as the CCMSC target boiler problem, code complexity, and ability to stress key portions of Uintah application code and infrastructure code, including Uintah's task schedulers and AMR infrastructure. Summaries of these applications are provided in Sections 3.5.1 through 3.5.4.

3.5.1 Radiation Modeling

The CCMSC uses the ARCHES turbulent combustion simulation component for its predictive boiler simulations. In these simulations, radiation is the dominant mode of heat transfer and consumes a majority of compute time per timestep. At large scale, additional simulation challenges are faced due to the global, all-to-all nature of radiation [54].

ARCHES was initially developed using the parallel discrete ordinates method [67] and P1 approximation [66] to solve the radiative transport equation. Though scalable, this approach resulted in the solution of the associated sparse linear systems being the main computational cost for reacting flow simulations. To reduce this cost, attention has been given to potentially more efficient reverse Monte-Carlo ray tracing (RMCRT) methods [58, 112]. This has led to the development of a standalone RMCRT-based radiation model suitable for use within Uintah's simulation components [55].

With Monte-Carlo ray-tracing methods (forward or backward), two approaches are considered to parallelize the computation for structured grids: (1) parallelize by patch-based domain decomposition with local information only and pass ray information at patch boundaries via MPI, and (2) parallelize by patch-based domain decomposition with global information and reconstruct the domain for the quantities of interest on each node by

passing domain information via MPI [54]. For the CCMSC's predictive boiler simulations, the first approach becomes intractable due to the ray counts used and MPI communication costs required to pass ray information. These ray counts are orders of magnitude larger than the ray counts used to produce results presented within this dissertation, which were on the orders of 10s of millions to 10s of billions. In the second approach, the primary difficulty is efficiently constructing the global information for millions of cells in a spatially decomposed (patch-based) domain. With this approach, an all-to-all communication phase is incurred for the radiative properties across the computational domain.

Uintah has adopted the second RMCRT parallelization approach, providing a global reconstruction of the radiative properties on each node to enable local ray tracing. This model has since been (i) extended to support adaptive mesh refinement (AMR) [54], (ii) further adapted to run on GPUs at large-scale with this novel AMR approach [56], (iii) used to explore the performance on the Intel Knights Corner coprocessor [47], (iv) extended to support the Kokkos::OpenMP back-end and used to explore performance of Uintah's initial MPI+Kokkos hybrid parallelism approach at scale on the Intel Knights Landing processor [48], and (v) extended to support the Kokkos::Cuda back-end [94] and used to explore the performance of Uintah's resulting MPI+Kokkos hybrid parallelism approach [51]. Uintah offers multiple RMCRT-based radiation modeling approaches, ranging from a single-level approach to the AMR approach used in [54] and [56].

RMCRT uses random walks to model radiative heat transfer by tracing rays in reverse, *towards* their origin [58, 112]. During traversal, the amount of incoming radiative intensity absorbed by the emitter is computed to aid in solving the radiative transport equation. Figure 3.3 shows how a ray is traced backwards from **S** to the emitter, **E**, for single-level RMCRT in a structured grid. Figure 3.4 shows how ray traversal might be accomplished using a 3-level mesh coarsening scheme. Algorithm 3.1 and Algorithm 3.2 describe this process in pseudocode.

RMCRT lends itself to scalable parallelism by allowing multiple rays to be traced simultaneously at any given cell and/or timestep. Additionally, RMCRT eliminates the need to trace rays that may never reach an origin. However, RMCRT does not eliminate the global, all-to-all nature of radiation. Within Uintah, RMCRT has been parallelized by spatially decomposing the computational domain into patches and tracing rays within a

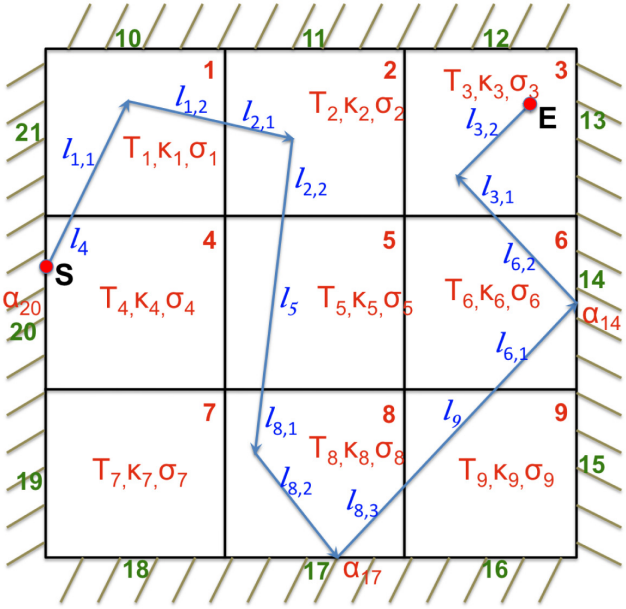

Fig. 3.3: Two-dimensional outline of reverse Monte-Carlo ray tracing for the single-level approach. [55].

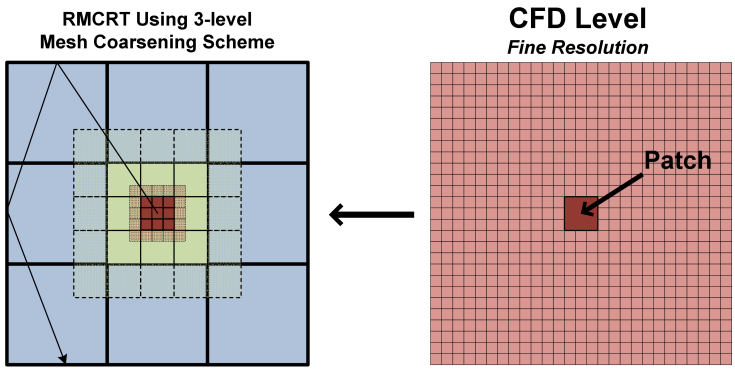

Fig. 3.4: Two-dimensional outline of reverse Monte-Carlo ray tracing for a 3-level mesh refinement approach, illustrating how rays from a fine-level patch (right) may be traced across a coarsened domain (left) [54].

given patch to termination.

RMCRT's communication, characterized in [54], grows quadratically as $\mathcal{O}(n^2)$ with respect to the problem size, where n corresponds to the number of communicating MPI processes. This is due to the all-to-all nature of radiation and each MPI process needing information about radiative properties across the entire domain for ray tracing. For non-fixed size problems, weak scaling is possible through the use of aggressive mesh refinement to reduce communication requirements as shown in past studies [57, 69, 104].

```
1: for all cells in a mesh patch do
2:     intensity_sum = 0;
3:     for all rays in a cell do
4:         find_ray_direction()
5:         find_ray_location()
6:         update_intensity_sum()
7:     compute divergence of heat flux
```

Algorithm 3.1: Ray Marching Pseudocode

```
1: initialize all ray marching variables
2: while intensity > threshold do
3:     while ray in domain do
4:         obtain per-cell coefficients
5:         march current ray to next cell
6:         update ray marching variables
7:         update ray location
8:         in_domain=cellType[curr]==-1
9:         compute optical thickness
10:        compute contribution of current cell to sumI
11: compute wall emissivity
12: compute intensity
13: compute sumI
```

Algorithm 3.2: Radiation Intensity Summation Pseudocode (update_intensity_sum())

A simple analysis of the two-level scheme of [59] has been given in [80] and 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 [116].

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 [80].

This research uses RMCRT to solve the Burns and Christon benchmark problem described in [22]. This problem exercises the radiation physics needed for predictive boiler simulations and the main features of Uintah's AMR support. Specifically, this problem calculates the radiative-flux divergence for each cell within the computational domain. An accuracy analysis verifying Uintah's RMCRT-based radiation model against the Burns and Christon benchmark problem can be found in [59]. More details on Uintah's RMCRT-based radiation model can be found in [54].

The results presented throughout this dissertation use a variety of RMCRT implementations solving the Burns and Christon benchmark:

- *Single-Level RMCRT:CPU*: This is the original implementation of single-level RMCRT written to use serial tasks.
- *Single-Level RMCRT:Kokkos*: This is a portable implementation of single-level RMCRT written for this research to use Kokkos-based data-parallel tasks.
- *2-Level RMCRT:CPU*: This is the original implementation of 2-level RMCRT written to use serial tasks.
- *2-Level RMCRT:GPU*: This is the original implementation of 2-level RMCRT written to use NVIDIA CUDA-based data-parallel tasks.
- *2-Level RMCRT:Kokkos*: This is a portable implementation of 2-level RMCRT written for this research to use Kokkos-based data-parallel tasks.

- *2-Level ARCHES:RMCRT:Kokkos*: This is a portable implementation of 2-level RMCRT written for this research to use Kokkos-based data-parallel tasks that also ties into key components of the ARCHES algorithm.

The most complex of these is ARCHES:RMCRT:Kokkos, which is used for experiments in Chapter 7 and modifies the ARCHES' Burns and Christon benchmark problem to incorporate a pressure solve, requiring the use of hypre. The resulting problem consists of 19 unique portable loops individually using up to 28 variables with complex interconnectedness. Underlying Kokkos functionality used among loops includes Kokkos::parallel_for, Kokkos::parallel_reduce, Kokkos::View, and Kokkos_Random. A key feature making this an important problem for validating Uintah's portable infrastructure is the ability to simultaneously stress interoperability of ARCHES, hypre, and RMCRT while also stressing Uintah's adaptive mesh refinement support. This is helpful for ensuring robustness due to the complex hand-offs taking place between these codes (e.g., shared data dependencies).

3.5.2 Char Oxidation Modeling

The char oxidation of a coal particle involves a complex set of physics. This set of physics includes mass transport of oxidizers from the bulk gas phase to the surface of the particle, diffusion of oxidizers into the pores of the particle, reaction of solid fuel with local oxidizers, and mass transport of the gas products back to the gas phase. As implemented within ARCHES, the char rate computes the rate of chemical conversion of solid carbon to gas products, the rate of heat produced by the reactions, and the rate of reduction of particle size [2]. These rates are used in the Direct Quadrature Method of Moments (DQMOM) [88], which subsequently affect the size, temperature, and fuel content of the particle field. For each snapshot of time simulated, an assumption of steady-state is made that produces a nonlinear set of coupled equations. This set of coupled equations is then solved pointwise at each cell within the computational domain using a Newton algorithm. The char model is the most expensive model evaluated during the time integration of physics within ARCHES.

Algorithm 3.3 provides an overview of the char oxidation model loop structure. The core loop refactored to use the Kokkos::OpenMP and Kokkos::Cuda back-ends is the for loop beginning at Line 3 of Algorithm 3.3. This loop features approximately 350 lines of code with a number of interior loops and Newton iterations within. Outside of the core loop,

```
1:  for all mesh patches do
2:      for all Gaussian quadrature nodes do
3:          for all cells in a mesh patch do                ▷ Core loop ported to Kokkos
4:              loop over reactions with an inner loop over reactions
5:              multiple loops over reactions
6:              loop over species
7:              loop over reactions with an inner loop over species
8:              for all Newton iterations do
9:                  multiple loops over reactions
10:                 multiple loops over reactions with inner loops over reactions
11:             loop over reactions
```

Algorithm 3.3: ARCHES Char Oxidation Model Loop Structure

there is a multiplier incurred by the number of Gaussian quadrature nodes, which results from the DQMOM approximation to the number density function. Inside of the core loop, there are a variety of multipliers incurred by the number of reactions and species computed. Additional complexity is introduced among these multipliers by the per-cell Newton iterations beginning at Line 8 of Algorithm 3.3. For example, the top bottleneck within the core loop has a multiplier of $GaussianQuadratureNodes * NewtonIterations * Reactions^2$ per cell.

3.5.3 Radiative Particle Property Modeling

Algorithm 3.4 provides an overview of the radiative particle property model loop structure. This loop features 3 lines of code and is one of the more representative loops in ARCHES. The resulting weighted properties are used to compute global radiative heat flux (e.g., to help understand heat flux profiles in large coal boilers).

3.5.4 Helium Plume Problems

The Taylor-Green vortex [21] is a well-known benchmark for validating CFD codes such as ARCHES. Though a simple incompressible flow problem, the Taylor-Green vortex

```
1:  for all mesh patches do
2:      for all cells in a mesh patch do
3:          apply a weight to a particle's absorption coefficient
4:          store the weighted coefficient for flow cells
5:          store a zero for non-flow cells
```

Algorithm 3.4: ARCHES Radiative Particle Property Model Loop Structure

neglects the effect of density variation, which is key for combustion applications. Helium plume problems are another incompressible flow problem helpful for validating CFD codes that account for density variations.

The helium plume has characteristics representative of a real fire and serves as an important ARCHES validation problem for the CCMSC's predictive boiler simulations. The validation of ARCHES using helium plumes is based on experimental data collected by O'Hern et al. [85] at Sandia National Laboratory's FLAME facility in Albuquerque, NM. Their solution exercises major components of the overall ARCHES algorithm, including the modeling of small, sub-grid turbulence scales. Additionally, the coupled problem combines the effects of fluid flow and turbulent scalar mixing for a full spectrum of length and time scales without introducing the complications of combustion reactions.

This problem requires the solution of the Navier-Stokes equations where the Navier-Stokes equations describe the spatio-temporal motion of a fluid and are given by

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \mathbf{u} = 0 \tag{3.1}$$

$$\frac{\partial \rho \mathbf{u}}{\partial t} = \mathbf{F} - \nabla p; \quad \mathbf{F} \equiv -\nabla \cdot \rho \mathbf{u}\mathbf{u} + \nu \nabla^2 \mathbf{u} + \rho g \tag{3.2}$$

Here, $\mathbf{u} = (u_x, u_y, u_z)$ is the velocity vector describing the speed of fluid particles in three orthogonal directions, ν is the kinematic viscosity - a fluid property that reflects its resistance to shearing forces, ρ is the fluid density, and p is the pressure.

The numerical solution of the Navier-Stokes equations requires evaluation of the pressure field while enforcing the continuity constraint given by (3.1). One standard approach for deriving an explicit equation for the pressure is to take the divergence of (3.2) and make use of (3.1) to act as the constraint. At the outset, one obtains a Poisson equation for the pressure.

$$\nabla^2 p = \nabla \cdot \mathbf{F} + \frac{\partial^2 \rho}{\partial t^2} \equiv R \tag{3.3}$$

Equation (3.3) is known as the pressure-Poisson-equation (PPE). Its solution requires the use of a solver such as hypre for large sparse systems of equations.

The computational scenario used for validating ARCHES consists of a $3m^3$ domain with a $1m$ opening that introduces the helium into a quiescent atmosphere of air with a co-flow of air. Velocity and density conditions are known at boundaries. The sides and top of the computational cube are modeled using pressure and outlet boundary conditions,

respectively. The outlet boundary condition allows the flow to leave the domain while the pressure conditions make it possible for airflow to enter (as driven by the buoyancy forces) through the side of the domain. Both the outlet and pressure conditions are driven by the resulting pressure field solution.

The problem used for this dissertation consists of 125 unique portable loops individually using up to 17 variables with complex interconnectedness. Underlying Kokkos functionality used among loops includes Kokkos::parallel_for, Kokkos::parallel_reduce, and Kokkos::View. A key feature making this an important problem for validating Uintah's portable infrastructure is the large number of unique portable loops and variables in flight during execution. This is helpful for ensuring robustness due to the long and complex data dependency sequences generated by these loops (e.g., variables computed on the host, modified on the device, and later required on the host).

CHAPTER 4

UINTAH'S MPI+PTHREADS TASK SCHEDULING APPROACH

4.1 Overview

From the center's start in 2014, the CCMSC has targeted the proposed DOE Aurora system for exascale boiler simulations as it was originally planned to be the United States' first exascale system. Early proposed versions of Aurora were to feature many-core architectures rather than the currently anticipated GPUs. Many-core architectures differ from multicore architectures in that they offer higher degrees of parallelism at the expense of, for example, slower clock speeds and single-thread performance. Examples include the Intel Xeon Phi and the Sunway SW26010. For this reason, such architectures have been examined as part of work contributing to several Uintah dissertations, including by Qingyu Meng [80,81], Damodar Sahasrabudhe [120], and the author [47]. This chapter examines Intel Xeon Phi performance in the context of Uintah's MPI+PThreads task scheduling approach. In doing so, this chapter captures work from a book chapter [47] by the author.

The Intel Xeon Phi is a many-core device based on Intel's Many Integrated Core (MIC) Architecture [60,99]. This architecture delivers high degrees of parallelism by offering up to 72 out-of-order cores featuring 4-way hyperthreading, 512-bit SIMD instructions, and sub-2 GHz clock speeds. The first-generation Xeon Phi coprocessor, Knights Corner, is a PCIe-based accelerator that requires cross-compilation and offers up to 61 cores. The second-generation Xeon Phi processor, Knights Landing, is a socket-based processor that is binary compatible with past generations of Intel processors and offers up to 72 cores.

Though easy to start using, the Xeon Phi poses new challenges for Uintah and others by requiring greater attention to data movement, thread scalability, and vectorization to achieve performance. Early studies exploring Uintah's performance on first-generation Xeon Phi coprocessors, Knights Corner, have helped demonstrate some of these challenges [80,81].

This chapter continues our evaluation of the Intel Knights Corner and explores the suitability of Uintah's MPI+PThreads task scheduling approach for the large per-node core counts found on many-core systems. This evaluation explores two key challenges relating to thread placement.

The first challenge relates to thread affinity patterns and less conventional thread management techniques for developers using the PThreads threading model. OpenMP eases thread placement by offering great control over where and how threads execute. Examples include Intel Compiler's *KMP_AFFINITY* flag to place threads and OpenMP loop scheduling parameters to determine how per-loop work is scheduled and executed. Uintah's Unified Scheduler, however, does not support OpenMP and implements an MPI+PThreads-based hybrid parallelism model requiring manual management of thread placement.

The second challenge relates to the Knights Corner's reserved core. On the 61-core Knights Corner coprocessor, the last-most physical core contains logical cores 241, 242, 243, and 0. Though */proc/cpuinfo* core id: 60 in practice, this physical core is commonly referred to as the 61^{st} core. The 61^{st} core is unique in that logical core 0 is reserved for the Xeon Phi operating system. Additionally, the 61^{st} core is also reserved for the offload daemon. While it is reportedly safe to use all 244 threads for native execution, it is unclear how to effectively manage the 61^{st} core when executing natively.

This chapter describes experiments evaluating several thread placement strategies with special attention to the 61^{st} core. These experiments have helped establish valuable baselines for future Uintah efforts. Perhaps more important, they have also provided valuable insight regarding eventual challenges and areas to address as we strive to achieve performance with the Xeon Phi. Specifically, single-node experiments have shown that the Intel Knights Corner requires greater attention to run configuration and domain decomposition as demonstrated by single-node results showing 10.1% performance differences across run configurations on Intel Sandy Bridge compared to performance differences up to 149.3% on Knights Corner. Multi-node experiments have shown that the Unified Scheduler's need to decompose a simulation domain into more, yet smaller, patches to support additional threads is not conducive to scalability, especially when problem sizes are limited by the Knights Corner memory footprint. When considering these results, it is important to

remember that we have examined native execution exclusively. When operating in offload mode, Intel's guidance is to refrain from using the reserved core as it actively supports offloading.

4.2 Scheduler Improvements

To support these evaluations, Uintah's Unified Scheduler was used as a foundation for implementing several thread affinity patterns. This dynamic scheduler features non-deterministic, out-of-order task execution at runtime. This is facilitated using a master thread and *nThreads*-1 task execution threads, where *nThreads* equals the number of threads launched at runtime. As-is, this scheduler manually binds threads from 0 to *nThreads*-1. Note, the master thread is also capable of executing tasks.

Patterns implemented for these experiments are itemized below:

- Compact: This pattern binds task execution threads incrementally across logical cores 1 through *n*t in a given physical core first and then across physical cores 1 through 61. This pattern is modeled after OpenMP's **KMP_AFFINITY = compact** with values of **61c,2t**, **61c,3t**, and **61c,4t** for the **KMP_PLACE_THREADS** environment variable.

- None: This pattern allows both the control and task execution threads to run anywhere among all 244 logical cores.

- Scatter: This pattern binds task execution threads incrementally across physical cores 1 through 60 first and then across logical cores 1 through *n*t in a given physical core. This pattern is modeled after OpenMP's **KMP_AFFINITY = scatter** with values of **60c,2t**, **60c,3t**, and **60c,4t** for the **KMP_PLACE_THREADS** environment variable. Note that, threads are spread across physical cores 1 through 60 only to support our exploration of the 61^{st} core.

- Selective: This affinity pattern binds the master thread to either logical core 240, 241, 242, 243, or 0 depending upon the values of *n*t and *nThreads*. Task execution threads are allowed to run anywhere among the logical cores preceding the control thread.

Note, *n*t corresponds to the number of threads per physical core.

Unlike the other patterns, the scatter affinity pattern requires more effort to implement. Figure 4.1 shows how to implement the scatter affinity pattern with PThreads. This example

```
void scatterAffinity( int threadID ) {

  int scatterPhysCores    = 61;
  int logCoresPerPhysCore = 4;
  int logCoreIndex        = 0;
  int physCoreIndex       = 0;
  int overallIndex        = 0;

  // Determine whether the thread will be bound to the 1st, 2nd,
  // 3rd, or 4th logical core in a given physical core
  logCoreIndex = floor(( threadID-1 ) / scatterPhysCores ) + 1;

  // Determine which physical core the thread will be bound to
  physCoreIndex = ( threadID - (( logCoreIndex-1 ) * scatterPhysCores ));

  // Determine the specific logical core the thread will be bound to
  overallIndex = logCoreIndex + ( physCoreIndex-1 ) * logCoresPerPhysCore;

  // Bind the thread to its corresponding logical core
  cpu_set_t mask;
  unsigned int len = sizeof( mask );
  CPU_ZERO( &mask );
  CPU_SET( overallIndex, &mask );
  sched_setaffinity( 0, len, &mask );
}
```

Fig. 4.1: PThreads-based implementation of the scatter affinity pattern.

assumes that each thread is uniquely identified by a *threadID* and calls **scatterAffinity()** to denote which logical core it is eligible to run on. Note, Figure 4.1 supports values of 0 through 243 for *threadID*, where *threadID* = 0 is mapped to logical core 0.

To enable exploration of the 61^{st} core with the *Compact*, *Scatter*, and *Selective* affinity patterns, multiple values of *nThreads* are used to increment the number of logical cores used on the 61^{st} core from 0 to *n*t. For example, a run configuration featuring *nThreads* = 180 and *n*t = 3 uses 0 logical cores on the 61^{st} core. With this in mind, the control thread is bound to the last logical core used by a given pattern. For example, a run configuration featuring *nThreads* = 180 and *n*t = 3 binds the control thread to logical core 240.

4.3 Single-Node Studies

The single-node experiments presented within this section solve the Burns and Christon benchmark problem described in [22] using single-level RMCRT:CPU and used for CPU- and GPU-based studies in [54] and [56], respectively. This problem exercises the radiation physics needed for predictive boiler simulations. Specifically, this problem calculates the radiative-flux divergence for each cell within the computational domain. An accuracy analysis verifying Uintah's RMCRT-based radiation model against the Burns and Christon

benchmark problem can be found in [59]. More details on Uintah's RMCRT-based radiation model can be found in Section 3.5.1 and [54].

These experiments were performed on a single-node machine using one MPI process and double-precision floating-point numbers. Host-side simulations were launched with 32 threads distributed among 16 physical cores. This was accomplished using two 8-core Intel Xeon E5-2680 processors in a dual-socket configuration. Coprocessor-side simulations were launched with as many as 244 threads distributed among 61 physical cores. This was accomplished using one 61-core 16 GB Intel Xeon Phi 7110P coprocessor.

Below are key parameters explored on the coprocessor-side:

- 3 physical core usage levels (2, 3, and 4 hardware threads per physical core)
  - For 2 hardware threads per physical core, 3 thread counts were used to allot 0-2 threads for the 61^{st} core (120-122 threads).
  - For 3 hardware threads per physical core, 4 thread counts were used to allot 0-3 threads for the 61^{st} core (180-183 threads).
  - For 4 hardware threads per physical core, 5 thread counts were used to allot 0-4 threads for the 61^{st} core (240-244 threads).
- 4 affinity patterns (*Compact*, *None*, *Scatter*, and *Selective* affinity)
- 4 mesh patch counts (facilitating ratios of 1, 2, 4, and 8 patches per thread)

Below are notes on simulation configuration:

- Simulation meshes are decomposed into mesh patches consisting of individual cells.
- Tasks are executed by threads, which are bound to logical cores.
- Tasks reside on mesh patches, which are computed serially using a single thread.
- Different threads may be used to compute tasks resident to a particular mesh patch.
- Tasks are assigned to idle threads without regard to the spatial locality of the mesh patch data that they access.
- Simulations were performed using a single-level 128^3 simulation mesh as this was the largest supported by the coprocessor.

- Radiation modeling calculations were performed over 10 consecutive timesteps.
- At each compute timestep, the simulation mesh was sampled using 100 rays per cell.
- Host-side simulations explored the use of 32 threads with the aforementioned affinity patterns and mesh patch counts.

4.3.1 Coprocessor-Side Results

Figure 4.2 visualizes results from the 192 simulations performed on the coprocessor-side. Below are notes on the figure for coprocessor-side results:

- Marks correspond to the average elapsed execution time per compute timestep (in seconds).
- Threads per core (TPC) corresponds to the number of hardware threads used per physical core.
- Patches per thread (PPT) corresponds to the ratio of mesh patches to threads. Note, each thread is not guaranteed to compute this number of mesh patches.
- Over 10 identical coprocessor-side simulations, there existed no more than a 4.29% difference in performance between two identical runs.

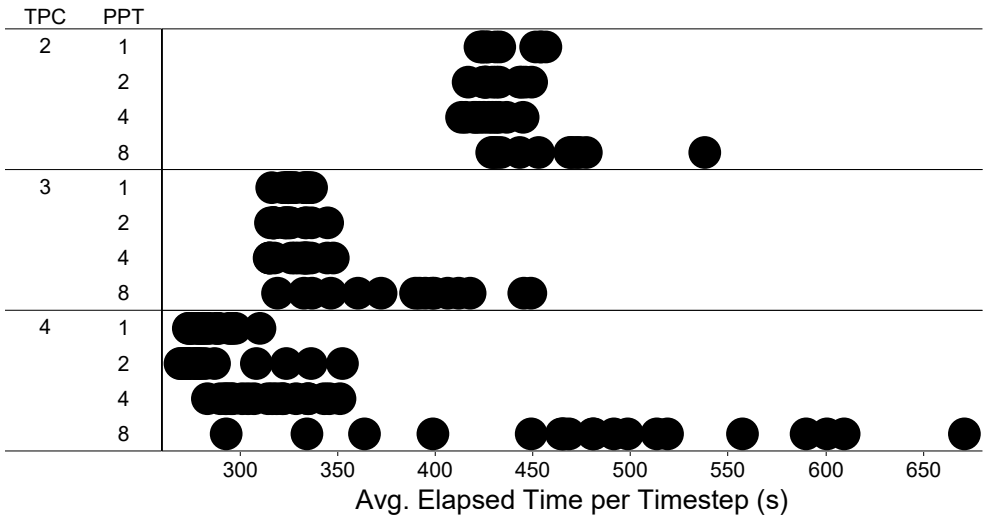

Fig. 4.2: Coprocessor-side results for single-level RMCRT:CPU

- At 2, 3, and 4 hardware threads per physical core, there existed 30.14%, 42.60%, and 149.33% differences in performance, respectively, between the fastest and slowest run configurations.

Note, a more in-depth analysis of the data examining reserved core use and thread affinity patterns can be found in the book chapter [47]. Such analysis is not included here as both thread affinity pattern and reserved core use had little impact on execution time during native execution. This is a result of Uintah's implicit work-stealing across threads and lack of regard to data locality when selecting work. Note, Intel guidance is to refrain from using the reserved core as it actively supports offloading.

4.3.2 Host-Side Results

Figure 4.3 visualizes results from the 16 simulations performed on the host-side. Below are notes on the figure for host-side results:

- Marks correspond to the average elapsed execution time per compute timestep (in seconds).
- Patches per thread (PPT) corresponds to the ratio of mesh patches to threads. Note, each thread is not guaranteed to compute this number of mesh patches.
- Over 10 identical host-side simulations, there existed no more than a 3.35% difference in performance between two identical runs.

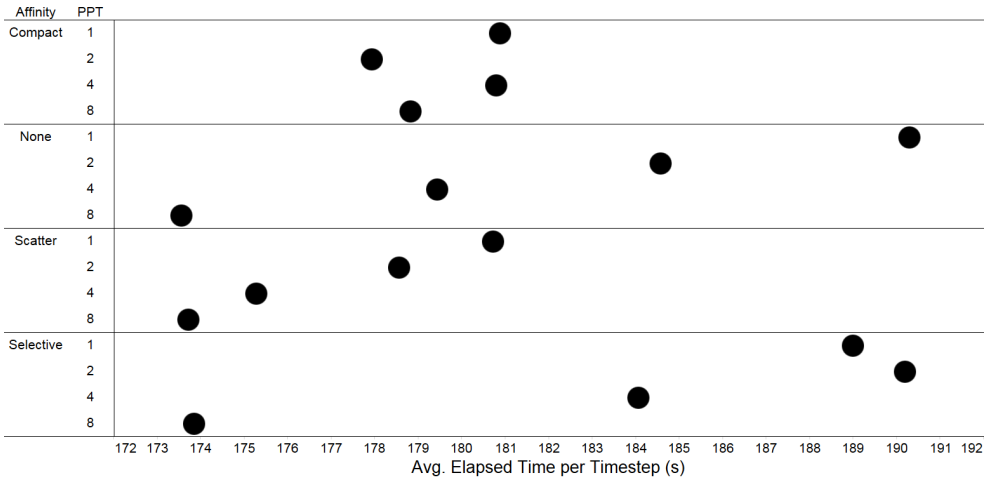

Fig. 4.3: Host-side results for single-level RMCRT:CPU using each affinity pattern.

- There existed a 10.14% difference in performance between the fastest and slowest run configurations.

4.3.3 Further Analysis

Addressing comparisons between architectures first, the two Xeon processors outperformed the single Xeon Phi coprocessor. Specifically, there existed a 39.43% difference in performance between the fastest run configurations for each architecture. Regarding accuracy, simulation results computed by each architecture were identical to one another up to a relative tolerance of 1E-15.

Given that this has been a naive port of our CPU-based algorithm, these results are encouraging as they leave ample opportunity to shift performance in favor of the coprocessor. Having not yet adequately pursued such optimizations, effective memory management and vectorization are believed to be the factors attributing to these differences. Supporting this conclusion, version 15.0 compiler optimization reports and experimentation with simpler vectorization approaches (e.g., SIMD directives) suggest that little, if any, vectorization is being achieved. Further, predominantly 100% core usage during compute timesteps suggests that thread-level parallelism is sufficient.

Turning to observations, performance disparities among coprocessor-based results deserve attention. As more hardware threads per physical core were utilized, the difference in performance between fastest and slowest run configurations increased. This is likely attributed to the sharing of the 512 KB per core L2 cache among four hardware threads. Though it offered better run times, the use of more hardware threads per physical core further divided the amount of L2 cache available to a given thread. This resulted in increased sensitivity to simulation mesh decomposition.

Returning to the question motivating this chapter, our fastest run configurations utilized the 61^{st} core. Further, performance was not profoundly impacted when explicitly oversubscribing the coprocessor operating system thread. For similar algorithms, this suggests that the use of the 61^{st} core may be both forgiving and capable of offering modest performance improvements. Lastly, the overarching takeaway from these native execution-based experiments is that no one thread placement strategy dominated performance. For similar algorithms, this suggests that time may be best spent by first pursuing more favorable areas

of optimization.

4.4 Multi-Node Studies

The strong scaling studies presented within this section also include solution of the Burns and Christon benchmark problem using single-level RMCRT:CPU. Aside from domain decomposition subtleties to be discussed, experiments have been run as in [54] and [56] with results averaged over 7 consecutive timesteps. The absorption coefficient was initialized per [22] with a uniform temperature field. For each thread count, 100 rays were used to compute the radiative-flux divergence for each cell.

These results have been gathered on the NSF Stampede system [114]. This system featured two 8-core Xeon E5-2680 Sandy Bridge processors and one Intel Xeon Phi SE10P Knights Corner coprocessor on a PCIe card connected to its Sandy Bridge host. With this in mind, each problem size explored fits within the 8 GB memory footprint of the Knights Corner.

These studies emphasize strong scaling due to the fixed target problem that the CCMSC aims to simulate at exascale. As such, weak scaling is not addressed due to the nature of communication growth for this problem, which has been characterized in [54]. Specifically, communication grows quadratically as $\mathcal{O}(n^2)$ with respect to the problem size, where n corresponds to the number of communicating MPI processes. This is due to the all-to-all nature of radiation and each MPI process needing information about the entire domain to trace rays throughout the domain. For non-fixed size problems, weak scaling is possible through the use of aggressive mesh refinement to reduce communication requirements as shown in past studies [57, 69, 104].

Here, strong scaling refers to the subdivision of a fixed size problem to support increasing node counts. A fixed number of patches was maintained per node and their size reduced to create additional patches for additional nodes. Patches were sized to enforce 1 patch per hardware thread.

As a whole, simulations were launched using 1 MPI process per Knights Corner node. Within an MPI process, threads were launched in multiples of 60 to ease domain decomposition. PThreads were launched without thread affinity.

Figure 4.4 shows strong scaling of single-level RMCRT:CPU. This implementation

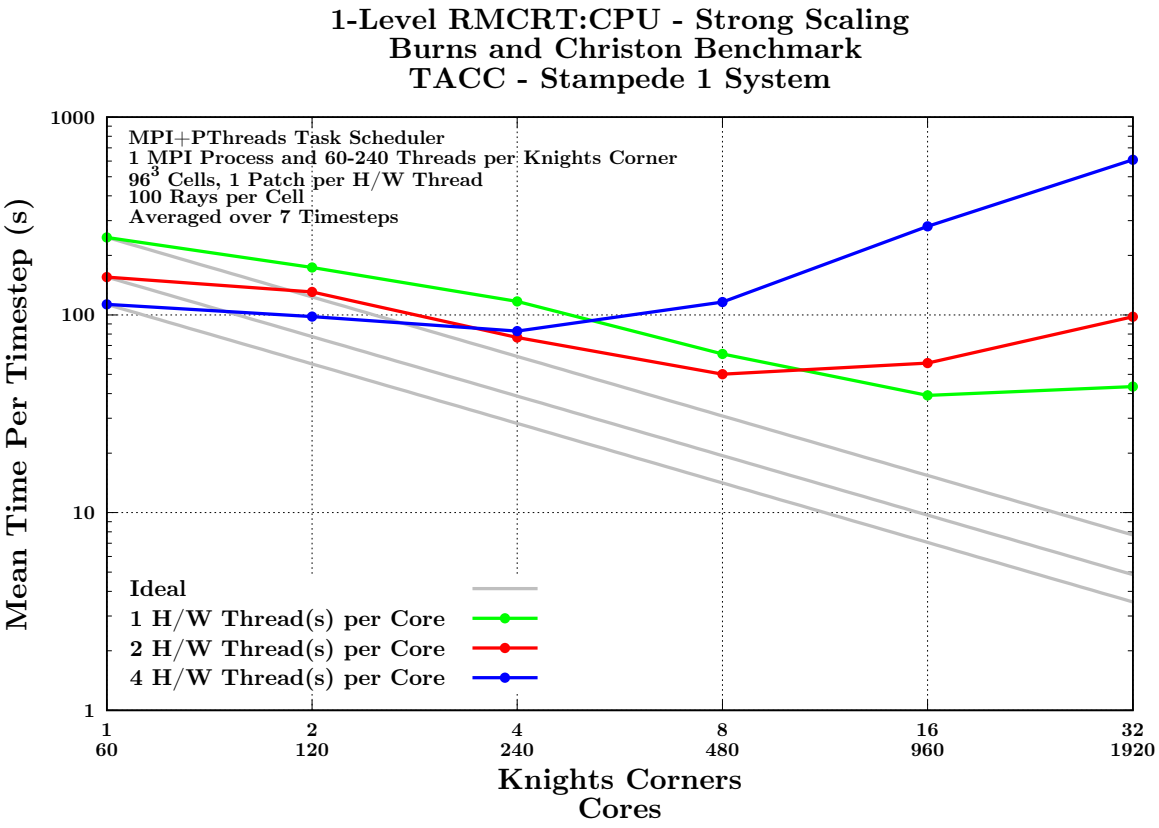

Fig. 4.4: Strong scaling results to 32 nodes for single-level RMCRT:CPU with parallel execution of serial tasks on Stampede's Knights Corner coprocessors.

features parallel execution of serial tasks within an MPI process. This figure presents results for three thread counts (60, 120, and 240 threads per MPI process to utilize 1, 2, and 4 hardware threads per core, respectively). For each thread count, a problem size of 96^3 cells was utilized.

4.4.1 Further Analysis

These results demonstrate that it is difficult for Uintah's RMCRT-based radiation model to scale across Stampede's Knight Corner coprocessors. This is attributed to the Unified Scheduler's use of serial tasks and the coprocessor's 8GB memory footprint limitation. To run with more threads per core, patch sizes must be halved to accommodate each doubling of the total threads used. However, this is not conducive to scalability as patch sizes for 1 thread per core experiments already struggle to hide communication.

To help overcome this scalability barrier, Uintah's task scheduling approach needs to shift from using serial tasks to data-parallel tasks. Data-parallel tasks eliminate the need to use more, yet smaller, patches to accommodate more compute resources (e.g., cores, threads). Such tasks allow users to use more compute resources per patch to avoid potentially unwanted reductions in patch size. This is desirable for architectures featuring shared per-core resources as it allows tasks to make cooperative use of such resources (e.g., caches).

4.5 Summary

This work has helped improve our understanding of how to run well with Uintah on many-core devices such as the Intel Xeon Phi. Specifically, it has evaluated suitability of Uintah's MPI+PThreads task scheduling approach for the large per-node core counts found on many-core systems. This evaluation explored two key challenges: (1) thread placement when using the PThreads threading model, and (2) how to manage use of the reserved core.

This evaluation has been made possible by implementing several thread affinity patterns modeled after OpenMP thread affinity patterns. Uintah's capabilities have been shown for a challenging radiation problem using these thread affinity patterns to execute a workload central to CCMSC predictive boiler simulations. Single-node experiments have shown that the Intel Knights Corner requires greater attention to run configuration and domain decomposition as demonstrated by single-node results showing 10.1% performance

differences across run configurations on Intel Sandy Bridge compared to performance differences up to 149.3% on Knights Corner. Multi-node experiments have shown that the Unified Scheduler's need to decompose a simulation domain into more, yet smaller, patches to support additional threads is not conducive to scalability, especially when problem sizes are limited by the Knights Corner memory footprint.

The understanding of performance established here helps define an appropriate direction as we prepare Uintah for many-core systems. Next steps include extending Uintah's MPI+X task scheduling approach to support data-parallel tasks in a portable manner using the Kokkos C++ library to adopt OpenMP. For Uintah's Aurora Early Science Program efforts, this will allow for cooperative use of shared per-core resources and address scalability barriers exposed by this work. For Uintah's emphasis on maintaining broad support for major HPC systems, this will adopt OpenMP in a portable manner to avoid code bifurcation when porting Uintah to other systems requiring potentially different programming models. As a part of this, emphasis will be placed on evaluating Kokkos for widespread adoption through Uintah.

CHAPTER 5

AN MPI+KOKKOS::OPENMP TASK SCHEDULING APPROACH

5.1 Overview

The DOE Aurora system was initially to feature upwards of 50,000 nodes based on Intel's since canceled third-generation Xeon Phi processor, Knights Hill. For this reason, the CCMSC placed heavy emphasis on understanding how to run well and scale on second-generation Intel Xeon Phi-based systems such as the DOE Cori [83], NSF Stampede 2 [115], and DOE Theta [4] systems before the discontinuation of the product line. Cori is a 30 petaflop system featuring 9,688 Intel Knights Landing Xeon Phi nodes and 2,388 Intel Haswell Xeon Haswell nodes. Stampede 2 is an 18 petaflop system featuring 4,200 Intel Knights Landing Xeon Phi nodes and 1,736 Intel Skylake Xeon nodes. Theta is an 11.7 petaflop system featuring 4,392 Intel Knights Landing Xeon Phi nodes. This chapter explores the adoption of OpenMP [25] through a performance portability layer to support such systems and address serial task limitations described in Chapter 4. In doing so, this chapter captures work from a conference paper [48], technical report [50], and workshop paper by the author [49].

OpenMP is commonly recommended to achieve the high levels of parallelism needed to make performant use of the large core and thread counts offered by the Intel Xeon Phi [60,99]. For this reason and ease of use mentioned in Chapter 4, OpenMP was chosen as the target programming model used to port Uintah to many-core systems. Adopting OpenMP itself, however, is problematic for Uintah's 1-2 million line codebase due to emphasis on maintaining broad support for major HPC systems. Such adoption would require multiple implementations of a given task to support both many-core and heterogeneous systems for which NVIDIA CUDA-based data-parallel tasks have already been introduced in Uintah at small-scale [55,56].

With this in mind, the portability of a codebase is becoming more important due to the variety of architectures being introduced in current and emerging high-performance computing (HPC) systems. The Top 10 of November 2021's Top500 list [110] includes ARM-based systems, heterogeneous systems with NVIDIA-based GPUs, Sunway-based systems, Intel Xeon-based systems, and Intel Xeon Phi-based systems. Further, the proposed DOE Aurora, DOE Frontier, and DOE El Capitan exascale systems are to feature Intel and AMD GPUs. Such variety complicates programming model selection for codebases looking to maintain long-term portability across major HPC systems when adopting a new programming model. Such is the case for Uintah when using shared memory programming models to transition from serial tasks to data-parallel tasks.

Programming model selection is simplified using a performance portability layer (PPL). Such layers provide abstractions (e.g., parallel loop statements) that allow developers to use a single interface to interact with multiple underlying programming models (e.g., CUDA, OpenCL, OpenMP, etc.) through PPL-specific back-ends. This approach eases the adoption of multiple programming models by reducing the amount of duplicated code and the knowledge required of underlying programming models by offloading low-level implementation details to the performance portability layer. Such an approach is desirable for Uintah to avoid multiple programming model adoption efforts, as adopting one alone requires a substantial investment.

This chapter addresses the challenge of programming model selection when converting Uintah's serial tasks to data-parallel tasks using the Kokkos [31,118] performance portability layer. Here, Kokkos is adopted directly in application code as a part of early prototypes evaluating the layer's suitability for porting Uintah to many-core systems. Within an MPI process, the underlying Kokkos back-ends allow data-parallel tasks to run on multicore-, many-core-, and GPU-based architectures for tasks with portable code. For many-core systems, this adoption has helped overcome the scalability barrier identified in Chapter 4 and relating to strict domain decomposition requirements. When using a single level of OpenMP-based parallelism within an MPI process, this has allowed for good strong scaling to 256 Knights Landing processors for a challenging radiation problem. When using two levels of OpenMP-based parallelism within an MPI process, this has allowed for good strong scaling to 442,368 threads across 1,728 Knights Landing processors to be achieved

for a challenging radiation problem with performance improvements up to 1.62x and little overhead added ($< 0.2\%$ per timestep).

5.2 The Kokkos C++ Library

The Kokkos C++ library [31, 118] is an open-source C++ programming model initially developed at Sandia National Laboratories for writing portable, thread-scalable code optimized for a diverse set of architectures supported in major HPC systems. This programming model is part of the Kokkos C++ Performance Portability Programming EcoSystem, which additionally provides developers with Kokkos-aware algorithms, math kernels, and tools. Kokkos is one of many programming models offering a single interface to multiple underlying programming models (e.g., CUDA, OpenCL, OpenMP, etc.). Examples of similar programming models include DPC++ [98], OCCA [76], RAJA [52], and SYCL [100].

A key idea among performance portability layers is the use of back-ends to manage execution and memory in a portable manner. In the case of Kokkos, these back-ends are mapped to abstractions providing developers with portable parallel execution patterns (e.g. *parallel_for*, *parallel_reduce*, *parallel_scan*) and related data structures (e.g., Kokkos Views). These fundamental abstractions allow Kokkos to manage both where and how: (1) patterns are executed, and (2) data is stored and accessed. Note, Kokkos back-ends to OpenMP and CUDA are referred to throughout this dissertation as Kokkos::OpenMP and Kokkos::CUDA, respectively.

Figure 5.1 shows an example of a Kokkos parallel pattern. The high-level idea is that developers write functors or lambda expressions to be placed inside a parallel pattern. These are customizable with additional parameters capable of tuning the number of threads used, execution patterns, and scheduling policies. Behind the scenes, C++ templating and compile-time conditional branching is used to avoid run-time branching and compile loops for the target back-end(s) specified by the user.

```
parallel_for( n, KOKKOS_LAMBDA( int i )
BODY
);
```

Fig. 5.1: Simplified syntax for Kokkos parallel pattern from Figure 5 in a recent evaluation [42].

Parallel patterns are supplemented by Kokkos Views, which are arrays of zero or more dimensions. In a similar way to parallel patterns, C++ templating is used to manage how Views are stored. For example, View layouts can be column-major, row-major, strided, tiled, etc. Further, Views can be either managed, where Kokkos handles reference counting and automatic deallocation, or unmanaged, where raw pointers are wrapped. In Uintah, unmanaged Kokkos Views are used as Uintah manages its own memory.

Uintah has adopted Kokkos to extend its codebase in a portable manner to multicore-, many-core-, and GPU-based systems. Specifically, to: (1) avoid code bifurcation when extending Uintah to accelerators and many-core devices with CUDA and OpenMP, respectively, (2) use a single interface to interact with multiple underlying programming models, and (3) offload low-level implementation details to Kokkos. This adoption has also allowed for a reduction in the gap between development time and our ability to run on newly introduced systems. For these advantages, Kokkos is believed to play a critical role in preparing Uintah for future HPC systems.

Uintah is an early adopter of Kokkos with Uintah developers collaborating directly with Kokkos developers as a part of the University of Utah's participation in the DOE/NNSA's Predictive Science Academic Alliance Program (PSAAP) II initiative. This collaboration has resulted in bi-directional development efforts with developers working in each other's codebases. At Sandia National Laboratories, Kokkos has been integrated in Trilinos [46] and used in codes such as Albany [28], GenTen [95], HOMMEXX [15], LAMMPS [96], and SPARTA [36]. Examples of other codes investigating and/or adopting Kokkos include BabelStream [27], K-Athena [38], KARFS [97], NekMesh [32], and TeaLeaf [74]. A list of applications using Kokkos can be found on the Kokkos GitHub [117].

5.3 Scheduler Improvements

To support this evaluation, Uintah task schedulers were modified in two phases. The first phase used Uintah's Dynamic MPI Scheduler as a foundation. The resulting scheduler adopted Kokkos in a manner using a single level of OpenMP-based parallelism to execute Kokkos-based data-parallel tasks. This approach adds parallelism inside tasks but not across multiple tasks. That is, this scheduler is limited to executing a single Kokkos parallel pattern using *nThreads* at a time within an MPI process. In this case, the key challenge

for adoption related to identifying where and when to initialize and finalize Kokkos with respect to per-process task execution. Figure 5.2 shows an example of what this task execution model looks like in the context of Uintah's multi-threaded MPI scheduler and Intel Knights Landing. Here, 1 data-parallel task (i.e., Task 1 in blue) is executed at a time using 64 threads.

The second phase used Uintah's Unified Scheduler as a foundation and, ultimately, established the OpenMP-based host-side capabilities of the heterogeneous MPI+Kokkos task scheduling approach to be discussed in Chapter 7. The resulting scheduler adopted Kokkos in a manner using two levels of OpenMP-based parallelism to execute Kokkos-based data parallel tasks. This approach adds parallelism inside tasks and across multiple tasks. That is, this scheduler is capable of executing from 1 to *nThreads* Kokkos parallel patterns using *nThreads* to 1 thread, respectively, at a time within an MPI process. In this case, Kokkos adoption was less intuitive and required new Kokkos functionality (Kokkos partitioning) to support Uintah's use case. Here, the key challenge was understanding how to replace

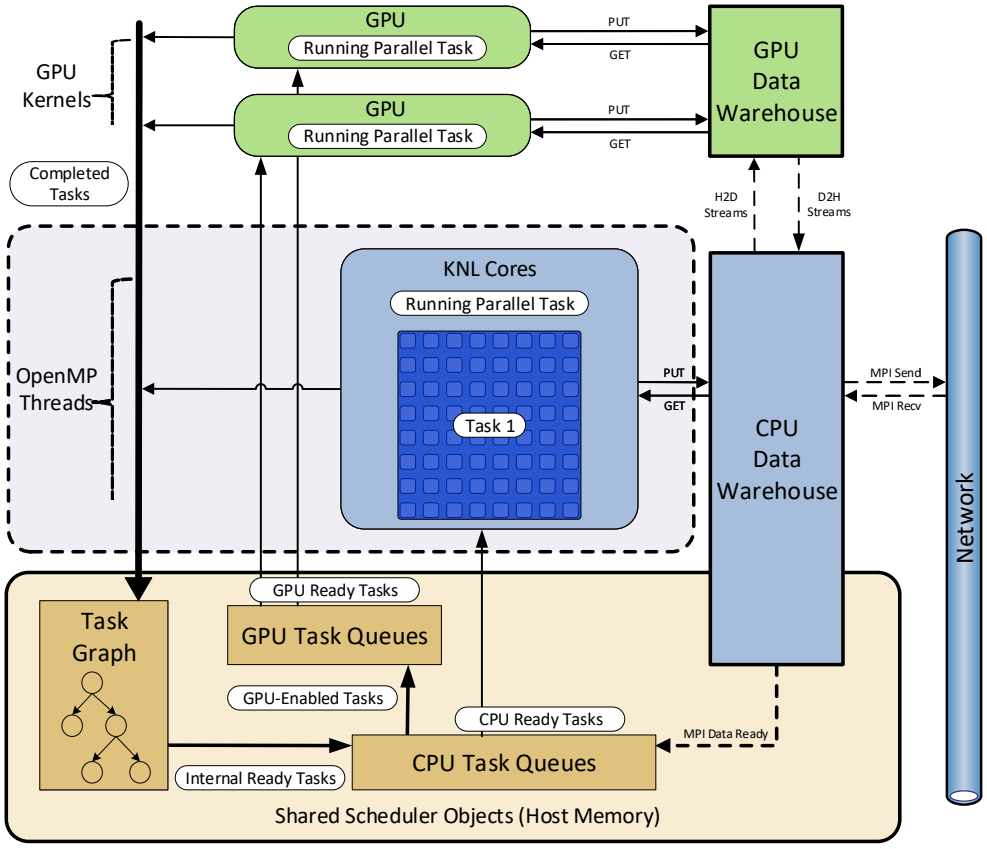

Fig. 5.2: Serial execution of data-parallel tasks.

Uintah's hundreds of lines of PThreads-based thread pool management code with Kokkos partitioning. The former implemented a master-slave thread driver that manually manages thread state (e.g., active, exit, inactive), thread affinity, and coordinates serial task executors. The latter will be described in further detail in Section 5.3.1. Figure 5.3 shows an example of what this task execution model looks like in the context of Uintah's multi-threaded MPI scheduler and Intel Knights Landing. Here, 4 data-parallel tasks (i.e., Task 1, 2, 3, and 4 in blue, green, red, and yellow, respectively) are executed at a time using 16 threads each. Note, run configurations are flexible and able to accommodate any variation of simultaneously executing tasks (e.g., 2 data-parallel tasks at a time using 32 threads each).

5.3.1 Kokkos Partitioning

Uintah's use of two levels of OpenMP-based parallelism has been made possible by partitioning functionality added to Kokkos via *partition_master*. This functionality allows a Kokkos execution space instance to be subdivided into multiple instances. Multiple

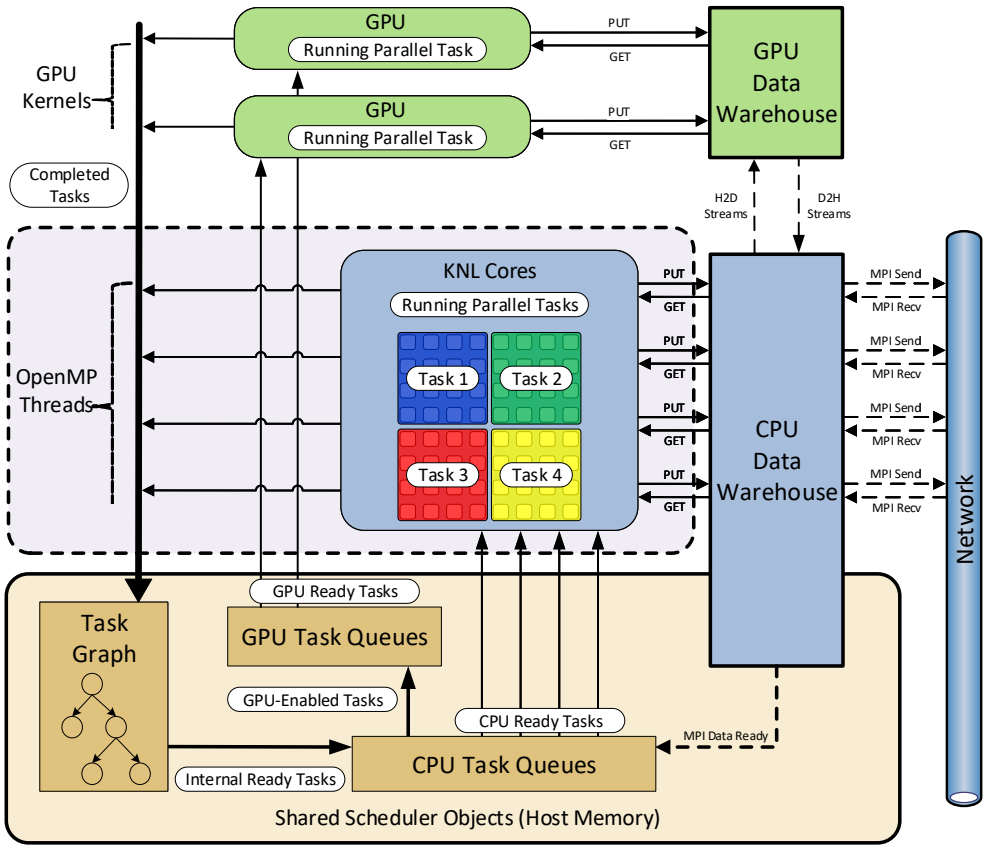

Fig. 5.3: Parallel execution of data-parallel tasks.

execution space instances allow for multiple Kokkos parallel patterns to be run simultaneously in parallel with one another. In the context of Uintah, an execution space instance corresponds to a task executor (i.e., the specific compute resources (e.g., cores) used to execute task executor logic responsible for task scheduling and execution and described further in Section 3.2). Similar to CUDA support for GPU asynchrony [94], the introduction of *partition_master* marks another instance of Uintah's needs as an AMT runtime system helping drive Kokkos development.

Uintah's adoption of *partition_master* required only a few lines of new code once the existing PThreads-based infrastructure was removed. Among infrastructure that *partition_master* replaced is a master-slave thread driver that manually manages thread state (e.g., active, exit, inactive), thread affinity, and coordinates serial task executors. Figure 5.4 shows how *partition_master* has been used within Uintah. This code is called on a per-timestep basis and has replaced hundreds of lines of thread pool management code within Uintah's MPI+PThreads task scheduler [78]. At the start of a timestep, *partition_master* uses OpenMP to subdivide the original execution space instance into multiple instances. During a timestep, each instance calls *runTasks()* to select and execute all tasks for a given timestep. At the end of a timestep, *partition_master* restores the original execution space instance.

When using *partition_master* with multiple execution space instances, care must be taken to ensure that a node is fully utilized. Specifically, thread placement becomes critical as it is easy to launch overlapping instances inadvertently. Three OpenMP environment variables are important for using *partition_master*: *OMP_NESTED*, *OMP_PLACES*, and *OMP_PROC_BIND*. The *OMP_NESTED* environment variable enables nested parallelism

```
auto task_worker = [&] ( int partition_id, int num_partitions ) {
  runTasks(); // runTasks is an existing function within Uintah
};

// Each partition executes task_worker
Kokkos::OpenMP::partition_master( task_worker
                                , num_partitions
                                , threads_per_partition );
```

Fig. 5.4: Code listing illustrating Uintah-based code required to enable parallel execution of newly-written Kokkos-based data-parallel tasks and existing serial tasks within an MPI process.

and must be set to true to allow for multiple execution space instances within an MPI process. The *OMP_PLACES* and *OMP_PROC_BIND* environment variables manage thread placement. For best performance, Kokkos recommends use of *threads* and *spread* for *OMP_PLACES* and *OMP_PROC_BIND*, respectively. This combination of environment variables spreads a set of threads as evenly as possible among places where each place corresponds to a single hardware thread on the target machine. In particular, *spread* is critical for ensuring that task executors are placed disjointly across a node.

Figure 5.5 shows an example of properly placing two task executors, t1 and t2 in red and blue, respectively, across a node using *OMP_PLACES=threads* and *OMP_PROC_BIND=spread*. Here, red and blue correspond to the resources used by each task executor. With this placement, tasks executed by individual task executors do not overlap one another and utilize all cores.

Figure 5.6 shows an example of improperly placing two task executors, t1 and t2 in red and blue, respectively, across a node using *OMP_PLACES=threads* and *OMP_PROC_BIND=close*. Here, red and blue correspond to the resources used by each task executor with purple and white corresponding to oversubscribed resources and unused resources, respectively. With this placement, tasks executed by individual task executors overlap one another and do not utilize all cores.

An analysis of *partition_master* overheads will be discussed in Section 5.5.

5.4 Loop Refactoring

With task schedulers implemented, the single-level and 2-level RMCRT variants described in Section 3.5.1 and char oxidation model described in Section 3.5.3 were then ported to support the use of the Kokkos::OpenMP back-end. For these ports, Kokkos parallel patterns were used directly in application code to ease rapid prototyping. Given the emphasis on OpenMP, ports were fairly straightforward and primarily involved converting legacy data structures to Kokkos views. Other examples include creating temporary variables for class members to be passed into Kokkos parallel patterns.

Algorithm 5.1 and Algorithm 5.2 provide an overview of the RMCRT algorithm described further in Section 3.5.1. The core loop refactored to use the Kokkos::OpenMP back-end features approximately 500 lines of code with a number of interior loops. Inside

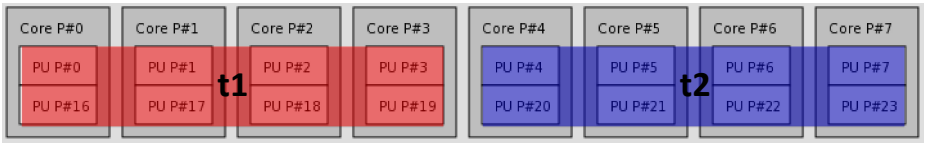

Fig. 5.5: Disjointly placed task executors, fully utilizing a node with *OMP_PLACES=threads* and *OMP_PROC_BIND=spread*. Red and blue regions are resources used by a given task executor.

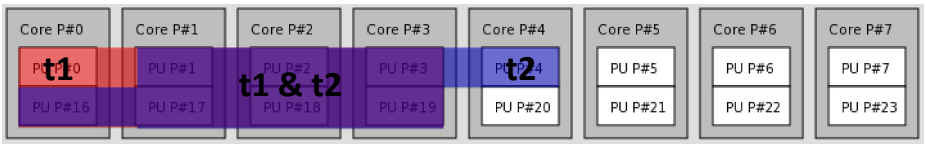

Fig. 5.6: Oversubscribed task executors, under-utilizing a node with *OMP_PLACES=threads* and *OMP_PROC_BIND=close*. Red and blue regions are resources used by a given task executor. Purple is where resources are oversubscribed. White is where resources are unused.

```
1: for all cells in a mesh patch do
2:     intensity_sum = 0;
3:     for all rays in a cell do
4:         find_ray_direction()
5:         find_ray_location()
6:         update_intensity_sum()                    ▷ Core loop ported to Kokkos
7:     compute divergence of heat flux
```

Algorithm 5.1: Ray Marching Pseudocode

```
1: initialize all ray marching variables
2: while intensity > threshold do
3:     while ray in domain do
4:         obtain per-cell coefficients
5:         march current ray to next cell
6:         update ray marching variables
7:         update ray location
8:         in_domain=cellType[curr]==-1
9:         compute optical thickness
10:        compute contribution of current cell to sumI
11: compute wall emissivity
12: compute intensity
13: compute sumI
```

Algorithm 5.2: Radiation Intensity Summation Pseudocode (update_intensity_sum())

of the core loop, there are a variety of multipliers incurred by the number of rays cast per cell and the number of cells that each ray is traced across. For example, the core loop has a multiplier of $100 RaysPerCell * CellsTracedAcross$ for the results presented here.

Algorithm 5.3 provides an overview of the char oxidation model described further in Section 3.5.3. The core loop refactored to use the Kokkos::OpenMP back-end is the for loop beginning at Line 3 of Algorithm 5.3. This loop features approximately 350 lines of code with a number of interior loops and Newton iterations within. Outside of the core loop, there is a multiplier incurred by the number of Gaussian quadrature nodes, which results from the DQMOM approximation to the number density function. Inside of the core loop, there are a variety of multipliers incurred by the number of reactions and species computed. Additional complexity is introduced among these multipliers by the per-cell Newton iterations beginning at Line 8 of Algorithm 5.3. For example, the top bottleneck within the core loop has a multiplier of $GaussianQuadratureNodes * NewtonIterations * Reactions^2$ per cell.

5.5 Single-Node Studies

This section presents results from experimental studies solving the char oxidation model within ARCHES. The results presented within this section used the following implementation of CharOx:

- *CharOx:Kokkos*: This is a new implementation of the char oxidation model written for this research to use Kokkos-based data-parallel tasks.

```
1:  for all mesh patches do
2:      for all Gaussian quadrature nodes do
3:          for all cells in a mesh patch do          ▷ Core loop ported to Kokkos
4:              loop over reactions with an inner loop over reactions
5:              multiple loops over reactions
6:              loop over species
7:              loop over reactions with an inner loop over species
8:              for all Newton iterations do
9:                  multiple loops over reactions
10:                 multiple loops over reactions with inner loops over reactions
11:             loop over reactions
```

Algorithm 5.3: ARCHES Char Oxidation Model Loop Structure

SNB-based results have been gathered on a node featuring two 2.7 GHz Intel Xeon E5-2680 Sandy Bridge processors with 8 cores (2 threads per core) per processor and 64 GB of RAM. Simulations were launched using 1 MPI process per node. Run configurations were selected to use the extent of each node. Per-timestep timings correspond to timings for execution of a timestep as a whole. Results have been averaged over 7 consecutive timesteps. Additional details on problem setup and run configuration are discussed in individual result paragraphs.

Tables 5.1 and 5.2 show *partition_master* overheads incurred at the start and end of a timestep, respectively. These tables present SNB-based results gathered using the CharOx:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells) and various combinations of OpenMP wait policies. Tasks were executed using 16 task executors with 2 threads per task executor via 1 MPI process and 32 OpenMP threads. The *OMP_WAIT_POLICY* environment variable decides whether threads spin (active) or yield (passive) while they are waiting. *OMP_WAIT_POLICY* defaults to yielding (passive). The *KMP_BLOCKTIME* environment variable sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping. *KMP_BLOCKTIME* defaults to 200 milliseconds.

Table 5.1: Dual-socket start-of-timestep partition_master overheads across OpenMP wait policies for CharOx:Kokkos on Intel Sandy Bridge.

START-OF-TIMESTEP PARTITION_MASTER OVERHEAD - in microseconds (% of execution) - SNB

OMP_WAIT_POLICY	KMP_BLOCKTIME	16 - 16^3 Patches	16 - 32^3 Patches	16 - 64^3 Patches
unspecified	unspecified	117.19 (0.0202%)	122.48 (0.0138%)	135.05 (0.0026%)
passive	0	103.27 (0.0190%)	98.05 (0.0117%)	102.14 (0.0020%)
passive	infinite	112.69 (0.0182%)	123.02 (0.0132%)	95.57 (0.0017%)
active	infinite	108.12 (0.0173%)	100.46 (0.0108%)	105.94 (0.0019%)

Table 5.2: Dual-socket end-of-timestep partition_master overheads across OpenMP wait policies for CharOx:Kokkos on Intel Sandy Bridge.

END-OF-TIMESTEP PARTITION_MASTER OVERHEAD - in microseconds (% of execution) - SNB

OMP_WAIT_POLICY	KMP_BLOCKTIME	16 - 16^3 Patches	16 - 32^3 Patches	16 - 64^3 Patches
unspecified	unspecified	900.88 (0.1555%)	900.55 (0.1016%)	34.44 (0.0007%)
passive	0	59.38 (0.0109%)	55.33 (0.0066%)	113.98 (0.0022%)
passive	infinite	38.73 (0.0063%)	41.10 (0.0044%)	1197.01 (0.0216%)
active	infinite	9342.95 (1.4979%)	10190.70 (1.0974%)	7555.71 (0.1328%)

5.5.1 Further Analysis

Results in Table 5.1 and Table 5.2 suggest that the overheads incurred when using *partition_master* are negligible in the context of this problem. Further, the default OpenMP wait policies are sensible for Uintah's MPI+Kokkos hybrid parallelism approach. For the default OpenMP wait policies, start-of-timestep *partition_master* overheads account for 0.0202%, 0.0138%, and 0.0026% of the elapsed time per timestep for 16^3, 32^3, and 64^3 patches, respectively. For the default OpenMP wait policies, end-of-timestep *partition_master* overheads account for 0.1555%, 0.1016%, and 0.0007% of the elapsed time per timestep for 16^3, 32^3, and 64^3 patches, respectively. Together, these *partition_master* overheads account for 0.1757%, 0.1154%, and 0.0033% of the elapsed time per timestep for 16^3, 32^3, and 64^3 patches, respectively, for default OpenMP wait policies.

5.6 Multi-Node Studies

The strong scaling studies presented within this chapter solve the Burns and Christon benchmark problem described in [22] and used for recent CPU- and GPU-based studies in [54] and [56], respectively. This problem exercises the radiation physics needed for predictive boiler simulations and the main features of Uintah's AMR support. Specifically, this problem calculates the radiative-flux divergence for each cell within the computational domain. An accuracy analysis verifying Uintah's RMCRT-based radiation model against the Burns and Christon benchmark problem can be found in [59]. More details on Uintah's RMCRT-based radiation model can be found in Section 3.5.1 and [54].

The results presented within this section used the following implementations of RMCRT:

- *Single-Level RMCRT:CPU*: This is an existing implementation of single-level RMCRT written to use serial tasks.
- *Single-Level RMCRT:Kokkos*: This is an implementation of single-level RMCRT written for this research to use Kokkos-based data-parallel tasks.
- *2-Level RMCRT:CPU*: This is an existing implementation of 2-level RMCRT written to use serial tasks.
- *2-Level RMCRT:GPU*: This is an existing implementation of 2-level RMCRT written to use NVIDIA CUDA-based data-parallel tasks.

- *2-Level RMCRT:Kokkos*: This is an implementation of 2-level RMCRT written for this research to use Kokkos-based data-parallel tasks.

The results presented within this section used the following implementations of Uintah's multi-threaded MPI scheduler:

- *Parallel Execution of Serial Tasks*: This is an existing implementation of Uintah's Unified Scheduler that executes serial tasks with parallelism across tasks using one level of PThreads-based parallelism.

- *Serial Execution of Data-Parallel Tasks*: This is an implementation extending Uintah's Dynamic MPI Scheduler to execute Kokkos-based data-parallel tasks with parallelism inside tasks but not across tasks using one level of OpenMP-based parallelism.

- *Parallel Execution of Data-Parallel Tasks*: This is an implementation rewriting Uintah's Unified Scheduler to use *partition_master* to execute Kokkos-based data-parallel tasks with parallelism inside tasks and across tasks using two levels of OpenMP-based parallelism.

Aside from domain decomposition subtleties discussed later in this section, experiments have been run as in [54] and [56] with results averaged over 7 consecutive timesteps. The absorption coefficient was initialized per [22] with a uniform temperature field. For single-level RMCRT simulations, 100 rays were used to compute the radiative-flux divergence for each cell. For 2-level RMCRT simulations, 100 rays were used to compute the radiative-flux divergence for each cell on the fine level.

With the exception of GPU-based results, these results have been gathered on the KNL Upgrade of the NSF Stampede system [114]. This portion of Stampede features the Intel Xeon Phi 7250 Knights Landing processor and offers a variety of memory and cluster mode configurations. These studies were conducted on Knights Landing processors configured for *Cache-Quadrant* mode. With this in mind, each problem size explored fits within the 16 GB memory footprint of MCDRAM.

These studies emphasize strong scaling due to the fixed target problem that the CCMSC aims to simulate at large scale. As such, weak scaling is not addressed due to the nature of communication growth for this problem, which has been characterized in [54]. Specifically,

communication grows quadratically as $\mathcal{O}(n^2)$ with respect to the problem size, where n corresponds to the number of communicating MPI processes. This is due to the all-to-all nature of radiation and each MPI process needing information about the entire domain to trace rays throughout the domain. For non-fixed size problems, weak scaling is possible through the use of aggressive mesh refinement to reduce communication requirements as shown in past studies [57, 69, 104].

Here, strong scaling refers to the subdivision of a fixed size problem to support increasing node counts. A fixed number of patches was maintained per node and their size reduced to create additional patches for additional nodes. For simulations using serial tasks, patches were sized to enforce 1 patch per hardware thread. For simulations using data-parallel tasks, patches were sized to enforce 1 patch per MPI process unless noted otherwise. For 2-level RMCRT simulations, coarse-level patch sizes were fixed and fine-level patch sizes were reduced as described above.

As a whole, simulations were launched using 1 MPI process per Knights Landing node. Within an MPI process, threads were launched in multiples of 64 to ease domain decomposition. For simulations using serial tasks, PThreads were launched without thread affinity. For simulations using data-parallel tasks, OpenMP threads were launched via Kokkos using *OMP_PLACES=threads* and *OMP_PROC_BIND=spread*. Additional details on problem setup and run configuration are discussed in individual result paragraphs.

5.6.1 Serial Execution of Data-Parallel Tasks

Figure 5.7 shows strong scaling of single-level RMCRT:Kokkos. This implementation features serial execution of data-parallel tasks within an MPI process. This figure presents results for three problem sizes (128^3, 256^3, and 512^3 cells). For each problem size, MPI processes were launched with 256 threads to utilize 4 hardware threads per core.

Figure 5.8 shows strong scaling of single-level RMCRT:CPU. This implementation features parallel execution of serial tasks within an MPI process. This figure presents results for three thread counts (64, 128, and 256 threads per MPI process to utilize 1, 2, and 4 hardware threads per core, respectively). For each thread count, a problem size of 128^3 cells was utilized. To enable comparisons, this plot also features single-level RMCRT:Kokkos results from Figure 5.7 for the corresponding problem size and node counts.

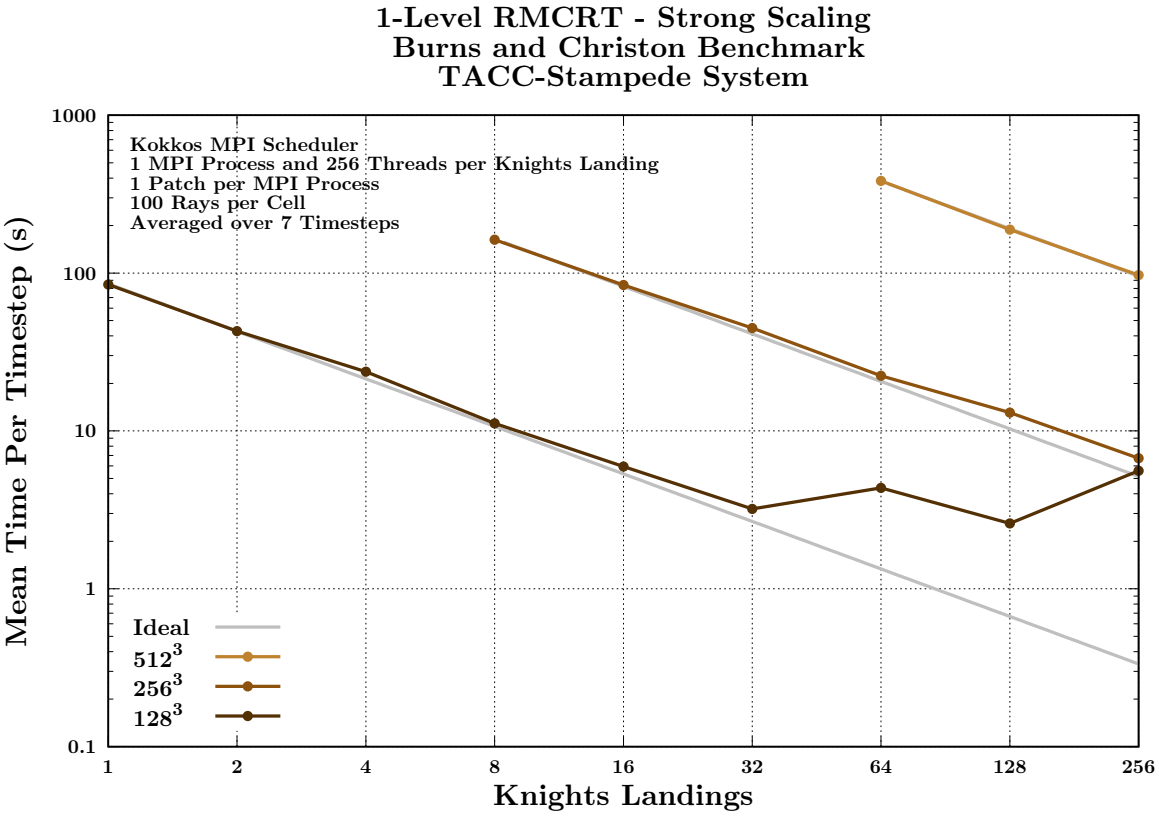

Fig. 5.7: Strong scaling results to 256 nodes for single-level RMCRT:Kokkos with serial execution of data-parallel tasks on Stampede's Knights Landing processors.

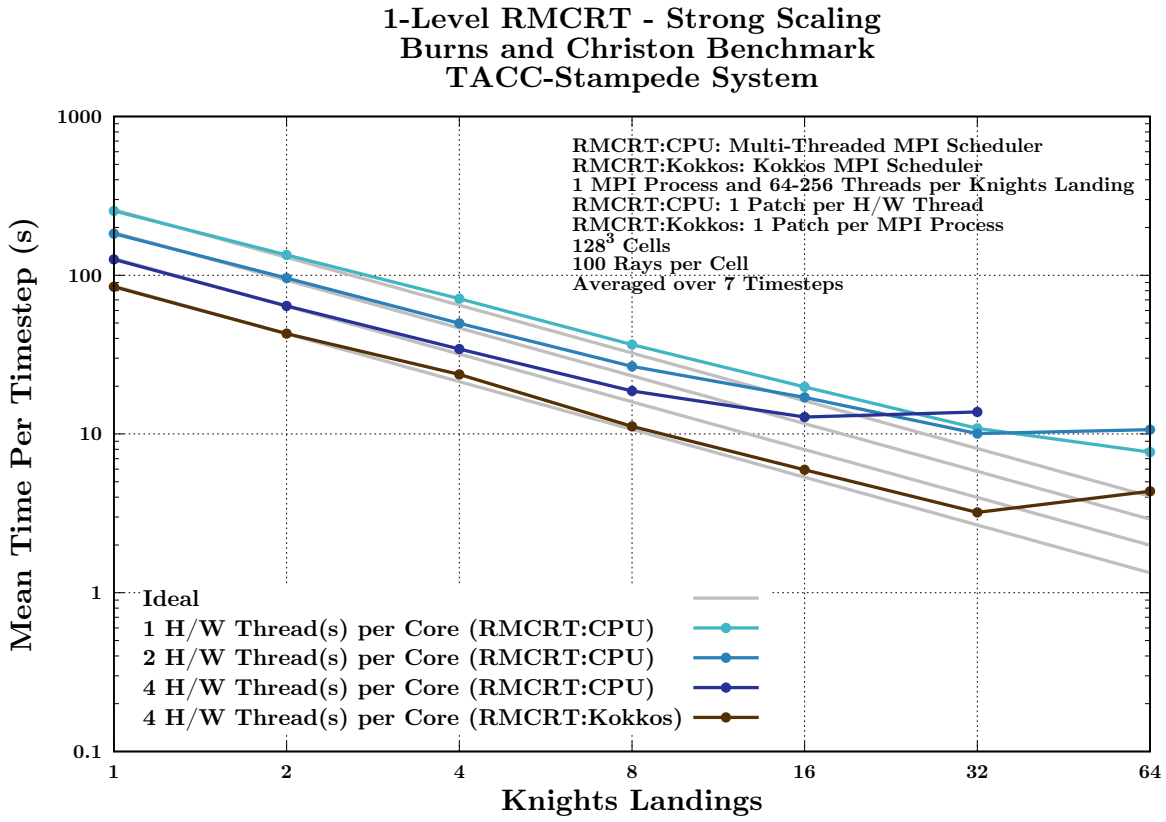

Fig. 5.8: Strong scaling results to 64 nodes for single-level RMCRT:CPU with parallel execution of serial tasks and single-level RMCRT:Kokkos with serial execution of data-parallel tasks on Stampede's Knights Landing processors.

Figure 5.9 shows strong scaling of 2-level RMCRT:CPU. This implementation features parallel execution of serial tasks within an MPI process. This figure presents results for three problem sizes (128^3, 256^3, and 512^3 cells on the fine mesh with 32^3, 64^3, and 128^3 cells on the coarse mesh, respectively). For each problem size, MPI processes were launched with 256 threads to utilize 4 hardware threads per core. To enable comparisons, this plot also features prior 2-level RMCRT:GPU results gathered on the DOE NVIDIA Tesla K20X-based Titan system for the corresponding problem sizes and node counts by Alan Humphrey [56]. More details on 2-level RMCRT:GPU can be found in Section 3.5.1 and [56].

Figure 5.10 shows strong scaling of 2-level RMCRT:Kokkos. This implementation features serial execution of data-parallel tasks within an MPI process. This figure presents results for two problem sizes (128^3 and 256^3 cells on the fine mesh with 32^3 and 128^3 cells on the coarse mesh, respectively). For each problem size, MPI processes were launched with 256 threads to utilize 4 hardware threads per core. To enable comparisons, this plot also features a portion of 2-level RMCRT:CPU results from Figure 5.9 for the corresponding problem sizes and node counts. For 2-level RMCRT:Kokkos, fine-level patches were sized to enforce 8 fine-level patches per MPI process.

5.6.2 Parallel Execution of Data-Parallel Tasks

Figure 5.11 shows strong scaling of 2-level RMCRT:Kokkos. This implementation features parallel execution of data-parallel tasks within an MPI process. This figure presents results for a problem featuring 768^3 cells on the fine mesh and 192^3 cells on the coarse mesh. For this problem, the fine mesh was decomposed into 110,592 patches with 16^3 cells per patch. Results are presented for three run configurations (1, 4, and 32 task executor(s) with 256, 64, and 8 threads per task executor, respectively, via 1 MPI process and 256 OpenMP threads).

5.6.3 Further Analysis

Results presented within Figure 5.8 demonstrate that as more hardware threads were utilized per core, node-level performance increased at the expense of reduced strong scaling efficiency. This is attributed to the strict domain decomposition requirements imposed by the serial task execution model. Though it improved node-level performance, the utilization of more threads per core required immediate reductions in patch size to accommodate

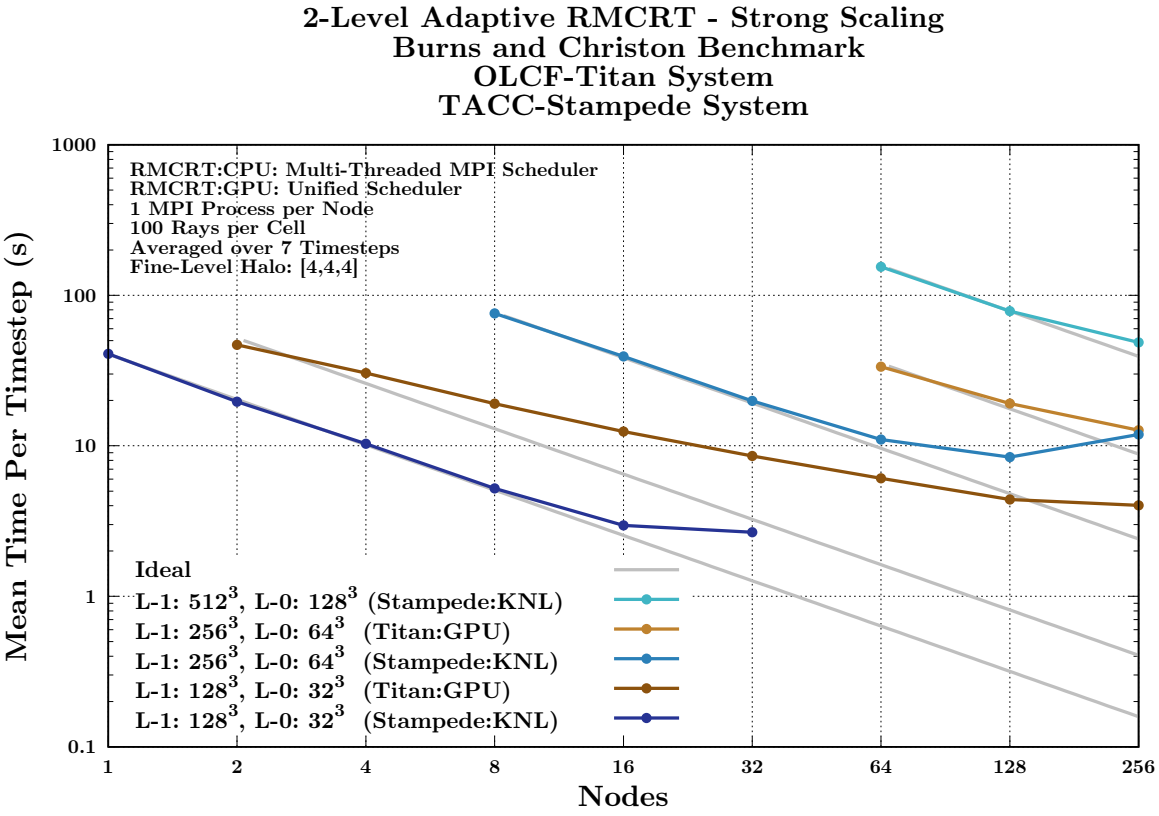

Fig. 5.9: Strong scaling results to 256 nodes for 2-level RMCRT:CPU with parallel execution of serial tasks on Stampede's Knights Landing processors and 2-level RMCRT:GPU with parallel execution of data-parallel tasks on Titan's K20X GPUs.

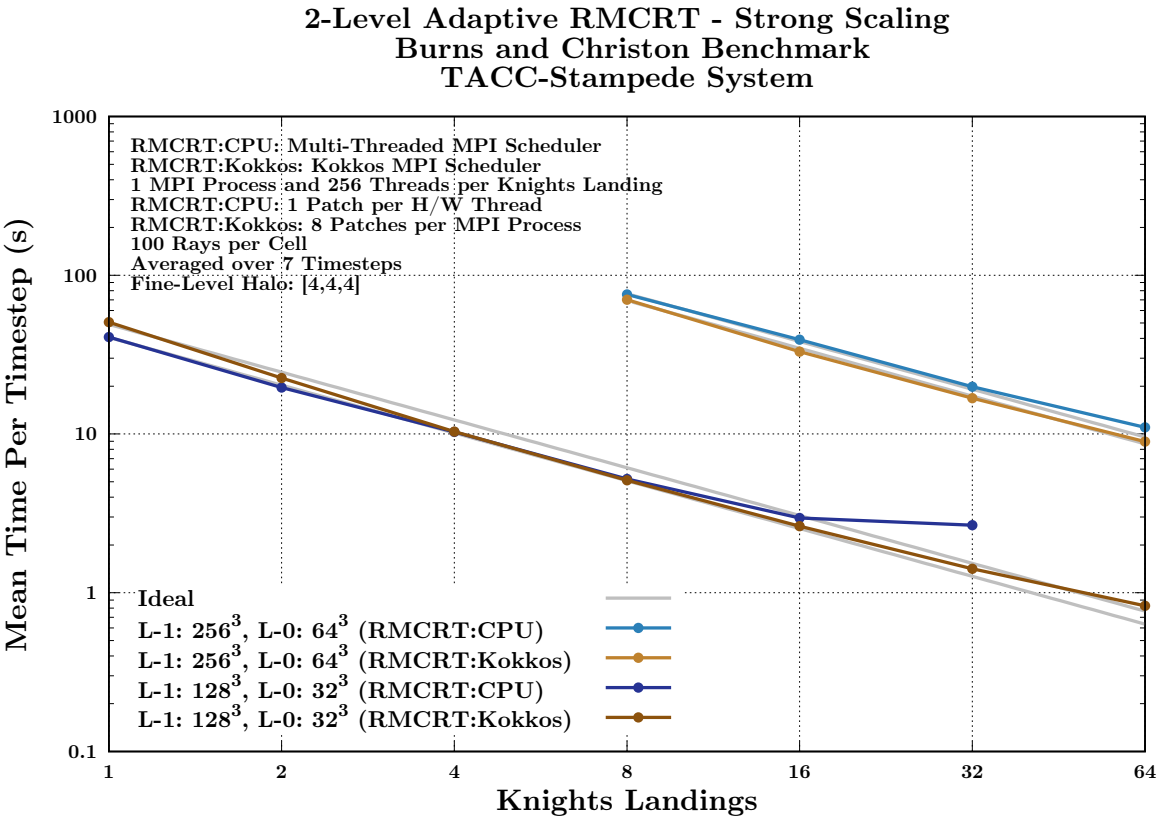

Fig. 5.10: Strong scaling results to 64 nodes for 2-level RMCRT:Kokkos with serial execution of data-parallel tasks and 2-level RMCRT:CPU with parallel execution of serial tasks on Stampede's Knights Landing processors.

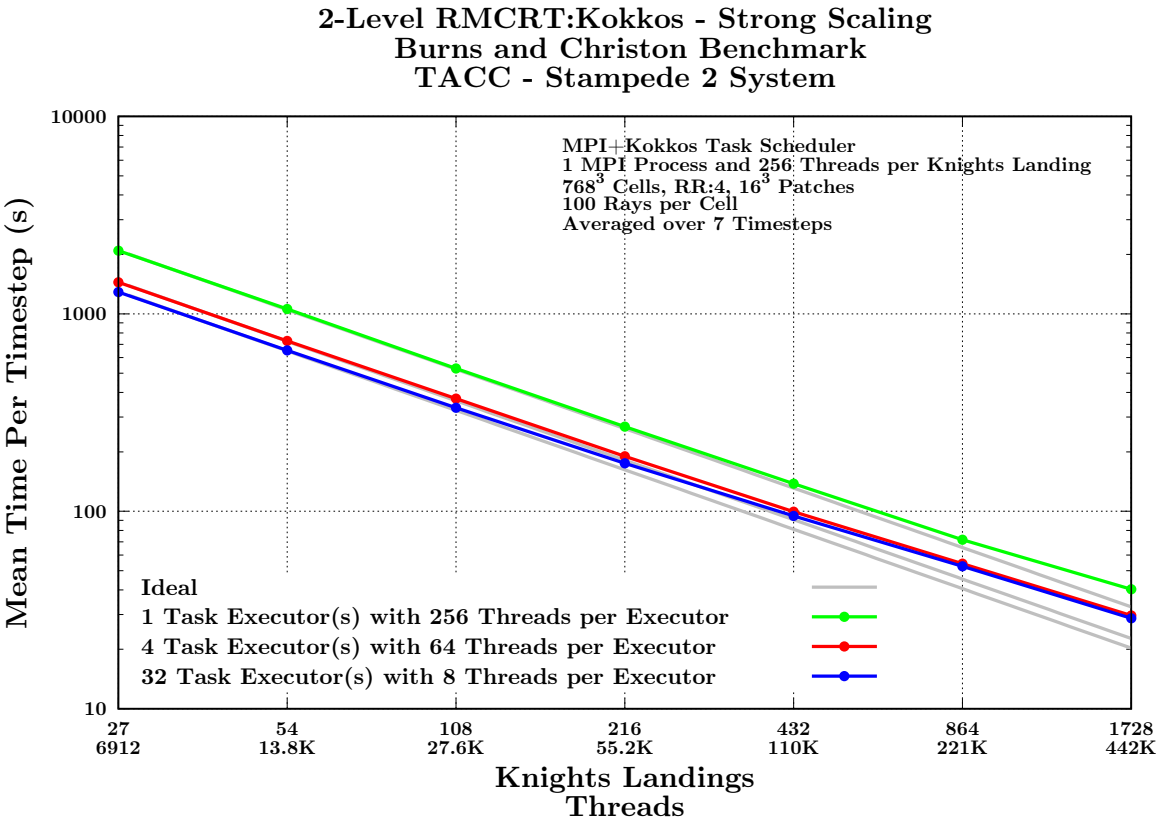

Fig. 5.11: Strong scaling results to 1728 nodes for 2-level RMCRT:Kokkos with parallel execution of data-parallel tasks on Stampede 2's Knights Landing processors.

additional threads within an MPI process. This expedited the breakdown of scalability, which is attributed to per-patch computation no longer sufficing to hide communication.

While this approach has suited CPU-based architectures well [54,78], these observations suggest that serial tasks are undesirable for large-scale simulations on many-core systems with additional support for this provided by Chapter 4. Supporting this conclusion, single-level RMCRT:Kokkos results included within Figure 5.8 suggest that we have overcome the scalability barrier posed by strict domain decomposition requirements through the use of data-parallel tasks. This has been achieved with an accompanying improvement in node-level performance, which is believed to be attributed to improvements in microarchitecture utilization enabled via data-parallel tasks.

Comparing results presented within Figure 5.7 to those within Figure 5.9, single-level RMCRT:Kokkos exhibited strong scaling characteristics comparable to those of 2-level RMCRT:CPU. This is encouraging as the AMR approach utilized within 2-level RMCRT:CPU enabled strong scaling characteristics to 262K CPU cores on the DOE Titan system that were previously unattainable via single-level RMCRT:CPU [54]. Further, node-level performance on Stampede's Knights Landing processors outperformed that of Titan's K20X GPUs. Though the K20X is dated, this is encouraging as Titan is one of the largest systems that we utilized at the time. For upcoming Theta and Aurora simulations, this suggests a potential for improving boiler performance predictions through the use of finer mesh resolutions and/or more simulated time.

Results presented in Figure 5.11 demonstrate that as more, yet smaller, task executors were used per node, node-level performance increased at the expense of reductions in strong scaling efficiency. This is attributed to thread scalability within individual task executors. For 16^3 patches, individual tasks are executed more efficiently when using fewer threads per task executor, resulting in more quickly executing tasks. This expedited the breakdown of scalability, which is attributed to computation no longer sufficing to hide communication. More efficient use of a node has allowed to speedups up to 1.62x and 1.40x to be achieved at 27 and 1728 nodes, respectively, over the use of 1 task executor with 256 threads per task executor within an MPI process.

Perhaps more important, these results demonstrate that it is possible to achieve good strong scaling characteristics to 442,368 threads across 1728 Knights Landing processors

using this MPI+Kokkos::OpenMP task scheduling approach. This is encouraging as it suggests a potential for reducing the number of per-node MPI processes by a factor of up to the number of cores/threads per node in comparison to an MPI-only approach. This is advantageous for many-core systems where the number of MPI processes required to utilize increasingly larger per-node core/thread counts becomes intractable.

5.7 Summary

This work has helped advance Uintah's preparedness for large-scale simulations on many-core systems. Perhaps more important, it has improved our readiness for CCMSC simulations on the DOE Theta and Aurora systems. Such readiness promotes more productive use of our Aurora Early Science Program allocation when predicting boiler performance for the PSAAP II project.

These advancements have been made possible by the direct adoption of Kokkos within Uintah and the extension of Uintah's task scheduler to support MPI+Kokkos::OpenMP. Though already supported for GPU-based architectures, Kokkos back-ends enable data-parallel tasks for CPU- and MIC-based architectures. These data-parallel tasks have helped overcome the MIC-specific scalability barrier pertaining to strict domain decomposition requirements identified in Chapter 4. The resulting flexibility in domain decomposition and run configuration allows Uintah to accommodate larger thread counts within an MPI process and offers greater control over the balance between local and global communication. This has been accomplished by allowing MPI processes to utilize fewer, yet larger, patches, improving our ability to hide communication.

These results offer encouragement as we prepare for the widespread adoption of Kokkos throughout Uintah. Next steps include creating Uintah-specific portable abstractions to indirectly adopt Kokkos throughout Uintah and ARCHES as a whole. For Uintah's Aurora Early Science Program efforts, this will encourage the adoption of Kokkos in a manner that provides flexibility should another performance portability layer or programming model be needed for eventual exascale systems.

CHAPTER 6

AN APPROACH FOR INDIRECTLY ADOPTING KOKKOS

6.1 Overview

Ports in Chapter 5 relied on direct adoption of Kokkos in application code. Direction adoption of a performance portability layer, however, poses challenges for large pre-existing codebases that may need to preserve legacy code and/or adopt other programming models in the future (e.g., a new programming model needed to support a novel architecture). This is a result of reliance on the performance portability layer to provide support for new underlying programming models. Such challenges are complicated in large pre-existing codebases where multiple refactors are not feasible and even one refactor may require a significant investment. Such is the case for Uintah, which aims to preserve legacy code in addition to using Kokkos to support the use of OpenMP and CUDA.

This chapter captures work from a workshop paper by the author [49] and addresses this challenge using an indirect adoption approach that introduces a framework-specific portability layer between the application developer and the adopted performance portability layer. Figure 6.1 shows an example of such an intermediate layer in the context of Uintah. Much like how a performance portability layer eases investment in multiple programming models, a framework-specific intermediate portability layer is needed to ease investment in a performance portability layer.

The goal of this intermediate layer is for application developers to, hopefully, need only adopt the layer once to support current and future interfaces to underlying programming models. For application developers, this layer allows for easy adoption of underlying programming models without requiring knowledge of low-level implementation details. For infrastructure developers, this layer allows for easy addition, removal, and tuning of interfaces behind the scenes in a single location, reducing the need for far-reaching changes

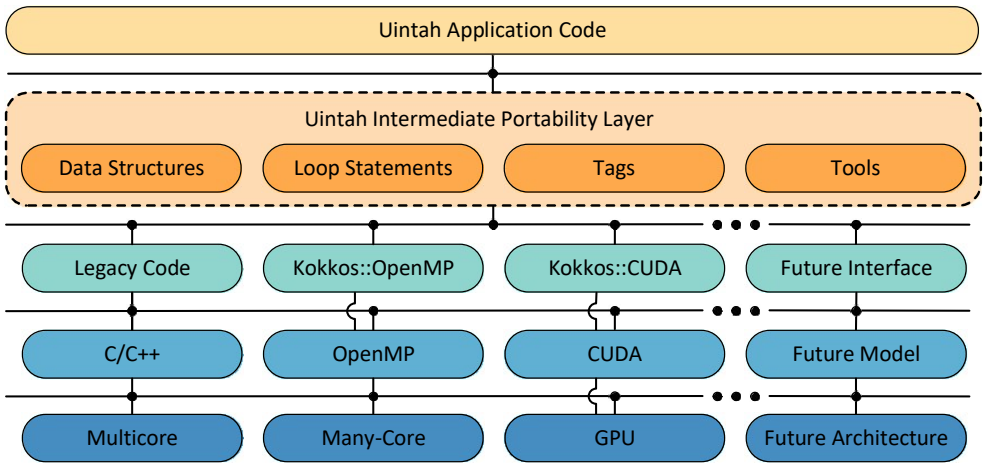

Fig. 6.1: Structure of Uintah's intermediate portability layer [50].

across application code.

This chapter describes the implementation of such a framework-specific portability layer used to address performance portability layer limitations for legacy code. This intermediate layer consists of three components: (1) loop-level support providing application developers with framework-specific abstractions (e.g., generic parallel loop statements) that map to interface-specific abstractions (e.g., PPL-specific parallel loop statements), (2) application-level support that includes a tagging system to identify which interfaces are supported by a given loop, and (3) build-level support that includes selective compilation of loops to allow for incremental refactoring and simultaneous use of multiple underlying programming models for heterogeneous HPC systems.

This design is informed by the CCMSC's multi-year Kokkos adoption effort adding portable support for OpenMP and CUDA in a complex real-world application, ARCHES. This ongoing effort has been non-trivial due to the codebase:

1. consisting of 1-2 million lines of complex code,
2. maintaining a divide between application code, where framework-specific abstractions are needed, and infrastructure code, where interface-specific abstractions are implemented,
3. having hundreds of pre-existing loops to port in application code,
4. being under active development with many contributors, and

5. having a pre-existing userbase to support.

The resulting approach aims to ease performance portability layer adoption in similar codebases and help improve legacy code support and long-term portability for future architectures and programming models. Though adoption has been limited to Kokkos, high-level ideas associated with this approach are broad enough to apply to performance portability layers offering similar parallel loop statements such as RAJA. Note, the author formally defined this layer with many others having helped to implement portions of this layer. Section 6.2 provides a comprehensive overview of contributors.

To demonstrate Kokkos capabilities, three case studies using this approach are examined for challenging calculations modeling the char oxidation of coal particles and radiative heat transfer in large-scale combustion simulations predicting the performance of a next-generation, 1000 MWe ultra-supercritical clean coal boiler and a related simple, yet representative, radiative particle property model loop. These case studies show the portable use of OpenMP and CUDA via Kokkos across multicore-, many-core-, and GPU-based nodes using a single implementation. The associated refactors have allowed for performance improvements up to 2.7x when refactoring for portability and 2.6x when more efficiently using a node to be achieved at the node level.

6.2 State of Uintah's Kokkos Adoption

The non-trivial nature of Uintah's adoption of Kokkos has required a number of small-scale case studies and refactors [48,50,51,93,94,113]. These individual efforts validate the use of Kokkos:

(1) in simple representative settings outside of Uintah, (2) in simple isolated portions of Uintah, (3) in complex isolated portions of Uintah, (4) in complex far-reaching portions of Uintah, and (5) at scale. This incremental approach has been critical for ensuring the continued feasibility and success of the effort given the high levels of investment required of Uintah when adopting a performance portability layer. Refactoring the most complex code early on has been key to identifying challenges quickly and refining best practices to simplify refactors moving forward. Such an approach is important for this and similar codebases where suitability must be evaluated before far-reaching adoption and significant investment in a performance portability layer.

Though the author has led Uintah's adoption of Kokkos, several other Uintah developers contributed to this effort. Key contributors include Daniel Sunderland (D.Su.), Brad Peterson (B.P.), Damodar Sahasrabudhe (D.Sa.), Jeremy Thornock (J.T.), Derek Harris (D.H.), and Oscar Díaz-Ibarra (O.D.-I.). Uintah's Kokkos-related activities to date are itemized below, with initials included for key contributors.

- Single-node case studies exploring the use of Kokkos parallel patterns with Kokkos::OpenMP in a simple standalone example outside of Uintah's simulation components [93] (D.Su., J.K.H)
- Single-node case studies exploring the use of Kokkos parallel patterns and unmanaged Kokkos views with Kokkos::OpenMP and Kokkos::CUDA in a mock runtime system representative of Uintah and the ARCHES simulation component [113] (D.Su., B.P.)
- Implementation of Uintah-specific abstractions to provide portable interfaces to Kokkos parallel patterns and unmanaged Kokkos views (D.Su., J.K.H., B.P., D.Sa., D.H.)
- Refactoring of a challenging standalone radiative heat transfer calculation outside of Uintah's simulation components to support the use of Kokkos::OpenMP [48] (J.K.H.)
- Multi-node case studies exploring the use of Kokkos::OpenMP at scale on the NSF Stampede 2 system using the refactored radiative heat transfer calculation [48] (J.K.H.)
- Incremental refactoring of loops in ARCHES to support the use of Kokkos::OpenMP *(ongoing)* (J.T., D.H., O.D.-I., J.K.H., B.P., D. Sa.)
- Implementation of a portable random number generator adopting Kokkos_Random functionality [94] (B.P.)
- Refactoring of the radiative heat transfer calculation to additionally support the use of Kokkos::CUDA [94] (B.P., J.K.H.)
- Extension of Kokkos itself to support asynchronous execution of Kokkos parallel patterns [94] (B.P.)
- Extension of Uintah's Unified Scheduler to support execution of individual Kokkos::CUDA tasks [94] (B.P.)

- Multi-node case studies exploring the use of Kokkos::OpenMP at scale on the DOE Theta system and use of Kokkos::CUDA at scale on the DOE Titan system using the refactored radiative heat transfer calculation [94] (B.P., J.K.H.)
- Implementation of a new task scheduler adopting Kokkos partitioning functionality to support the use of nested Kokkos::OpenMP [50] (J.K.H.)
- Implementation of portable synchronization primitives based on Kokkos MasterLock functionality (J.K.H.)
- Implementation of a task tagging system to allow for selective compilation of loops across Kokkos back-ends [50] (B.P., J.K.H.)
- Extension of Uintah's task tagging system throughout ARCHES [50] (J.K.H., B.P.)
- Refactoring of a challenging combustion loop modeling the char oxidation of coal particles in ARCHES to support the use of Kokkos::OpenMP and Kokkos::CUDA [50] (J.K.H)
- Refactoring of a simple loop weighting radiative particle properties in ARCHES to support the use of Kokkos::OpenMP and Kokkos::CUDA (D.H.)
- Single-node case studies exploring the use of nested Kokkos::OpenMP and Kokkos:: CUDA at the loop-level using the refactored radiative heat transfer calculation, refactored char oxidation model, and refactored simple loop [50] (J.K.H., B.P.)
- Multi-node case studies exploring the use of nested Kokkos::OpenMP at scale on the NSF Stampede 2 system using the refactored radiative heat transfer calculation [50] (J.K.H.)
- Incremental refactoring of loops in ARCHES to additionally support the use of Kokkos::CUDA *(ongoing)* (J.K.H., D. Sa., B.P., D.H.)
- Extension of Uintah's Unified Scheduler to support execution of multiple Kokkos:: CUDA tasks [51] (D.Sa., J.K.H.)
- Implementation of a new task scheduler to support simultaneous use of nested Kokkos::OpenMP and Kokkos::CUDA [51] (J.K.H.)

- Multi-node case studies exploring simultaneous use of nested Kokkos::OpenMP and Kokkos::CUDA using an ARCHES-based radiative heat transfer calculation and helium plume problem [51] (J.K.H., D.Sa.)

This progress has been achieved using the following Kokkos functionality [1]:

- Kokkos::parallel_for
- Kokkos::parallel_reduce (min and sum reductions)
- Kokkos::View (unmanaged)
- Kokkos::OpenMP::partition_master
- Kokkos::Experimental::MasterLock
- Kokkos_Random

The indirect performance portability layer adoption approach that has been informed by this progress was formally defined by the author in a workshop paper [50].

6.3 Uintah's Intermediate Portability Layer

A fundamental abstraction shared among several performance portability layers is the parallel loop statement. This abstraction is key to providing access to multiple underlying programming models through a single interface. Though exact syntax and implementation details vary, parallel loop statements generally rely upon an iteration range and a loop body defined as a C++ lambda or functor. Figure 6.2 shows an example of simplified syntax for Kokkos and RAJA parallel loop statements. Note, more discussion on similarities and

```
// Kokkos
parallel_for( n, KOKKOS_LAMBDA( int i )
BODY
);

// RAJA
forall<thread_exec>( 0, n, [=]( Index_type i )
BODY
);
```

Fig. 6.2: Simplified syntax for Kokkos and RAJA parallel loop statements from Figure 5 in a recent evaluation [42].

differences among modern C++ parallel programming models, including Kokkos, RAJA, and SYCL, can be found in a recent evaluation [42] and comparative analysis [41].

For OpenMP and CUDA themselves, the parallel loop statements and other abstractions offered by Kokkos have worked well in Uintah for the advantages discussed in Section 5.2. The high levels of investment required of Uintah when adopting Kokkos, however, have discouraged the direct adoption of these PPL-specific abstractions. Specifically, direct adoption of Kokkos throughout Uintah has been avoided to:

1. allow for legacy code to be preserved,
2. eliminate reliance on Kokkos to provide support for new underlying programming models,
3. simplify abstractions provided for application developers, and
4. ease re-work should implementation changes or a different performance portability layer be needed.

The approach taken to indirectly adopt Kokkos within Uintah uses an intermediate portability layer to provide Uintah-specific abstractions that interact with underlying programming models through various interfaces (e.g., implementing Kokkos-specific abstractions behind-the-scenes). These framework-specific interfaces allow for pre-existing code to be preserved when adopting Kokkos and, in theory, provide easy means of adopting other programming models should Kokkos not yet support one needed for a novel architecture. To date, Uintah's interfaces map Uintah-specific abstractions to:

1. legacy code,
2. Kokkos-specific abstractions for Kokkos::OpenMP, and
3. Kokkos-specific abstractions for Kokkos::CUDA.

Note, individual interfaces are used for Kokkos::OpenMP and Kokkos::CUDA to ease selective compilation of loops and provide more control over the implementation and execution of loops.

Table 6.1 shows the individual components that form Uintah's intermediate portability layer. Specifically, this intermediate layer consists of three components: (1) loop-level sup-

Table 6.1: Components of Uintah's intermediate portability layer.

Level	Component
Loop[a]	Generic Loop Statements Mapped to Multiple Execution Schemes
	Generic Data Structures Mapped to Multiple Data Structures
Application[b]	Arbitrary Tags to Manage Interfaces to Programming Models
	Arbitrary Execution Spaces to Manage Execution Schemes
	Arbitrary Memory Spaces to Manage Data Structures
	Portable Tools (e.g., Locks, Random Number Generators)
Build[c]	Preprocessor Macros to Manage Multiple Build Configurations
	Build-Specific Tags to Manage Selective Compilation of Loops

port providing application developers with framework-specific abstractions (e.g., generic parallel loop statements) that map to interface-specific abstractions (e.g., PPL-specific parallel loop statements), (2) application-level support that includes a tagging system to identify which interfaces are supported by a given loop, and (3) build-level support that includes selective compilation of loops to allow for incremental refactoring and simultaneous use of multiple underlying programming models for heterogeneous HPC systems.

Figure 6.3 shows an example of how these components are implemented using the *Uintah::parallel_for*. This is a framework-specific parallel loop statement modeled after the approach used by performance portability layers. Similar to performance portability layer goals, this abstraction aims to provide application developers with a single loop statement that allows for easy adoption of underlying programming models without requiring knowledge of low-level implementation details (e.g., for Kokkos). In practice, this approach has worked well for application developers used to writing serial loops in Uintah with little parallel programming experience. Note, this particular abstraction was developed in collaboration with Brad Peterson.

The *Uintah::parallel_for* parameter list differs slightly from previous examples in that it includes an additional parameter and requires 3-dimensional indexing. The *executionObject*

```
// Uintah
parallel_for( executionObject
            , iterationRange
            , LAMBDA( int i, int j, int k )
BODY
);
```

Fig. 6.3: Uintah's framework-specific abstraction for parallel_for.

parameter is a templated object used to pass non-portable objects and additional parameters (e.g., CUDA streams, CUDA blocks per loop, template parameters used to manage paths of execution, etc.) into portable loops for use behind-the-scenes in interfaces to underlying programming models. Three-dimensional indexing is used to ease legacy code support and is mapped to 1-dimensional indexing, as needed, behind the scenes. This is managed with the help of *iterationRange*, which is an object used to pass iteration range indices into portable loops. Note, *LAMBDA* is a generic macro for managing lambda capture clauses and CUDA annotation (e.g., *__device__*) in a manner similar to that used by Kokkos and RAJA (e.g., KOKKOS_LAMBDA, RAJA_DEVICE, RAJA_HOST_DEVICE).

Behind-the-scenes, preprocessor macros and template metaprogramming are used to manage paths of execution for Uintah's interfaces to underlying programming models in a single location. For example, a *Uintah::parallel_for* is executed using a *Kokkos::parallel_for* optimized for CUDA when Uintah is built with Kokkos::CUDA. This behind-the-scenes management is key to easily adding, removing, and tuning interfaces (e.g., to change how a *Kokkos::parallel_for* iterates over work items or, in theory, add support for another performance portability layer's parallel loop statement). Details on Uintah's use of preprocessor macros for managing multiple build configurations and custom paths of execution can be found in Chapter 8.

A notable component of the layer that builds on this use of preprocessor macros and template metaprogramming are the arbitrary tags used to manage interfaces to programming models. Tags are used by application developers to indicate which back-end(s) are supported by a given task. For example, each Uintah task uses defines for *UINTAH_CPU_TAG*, *KOKKOS_OPENMP_TAG*, and *KOKKOS_CUDA_TAG*. Commenting a given tag disables execution via the related execution space (e.g., Kokko::Cuda). Uncommented tags are mapped to the corresponding execution and memory space behind-the-scenes (e.g., #define KOKKOS_OPENMP_TAG Kokkos::OpenMP COMMA Kokkos::HostSpace). Additional details on task tagging can be found in Brad Peterson's dissertation [89]. Note, the Uintah's task tagging system has been extended throughout ARCHES by the author as a part of this dissertation's research.

Another notable component of the layer is Uintah's MasterLock class. The MasterLock class is used to provide a general abstraction for synchronization primitives. Currently,

std:mutex and OpenMP-based omp_lock_t primitives are supported. This class has been helpful for avoiding the mixing of primitives and ensuring that all locks are paired with the appropriate type to avoid deadlock. At run-time, either all std:mutex or all omp_lock_t primitives are used based upon which scheduler is used.

6.4 Loop Refactoring

With Uintah's intermediate portability later implemented, several key loops were then ported to support the use of both the Kokkos::OpenMP and Kokkos::CUDA back-end. Loops ported include single-level and 2-level RMCRT variants (Algorithm 5.2 from Chapter 5), a char oxidation model (Algorithm 5.3 from Chapter 5), and radiative particle property model (Algorithm 6.1). RMCRT variants and the char oxidation model were chosen due to being among the most complex loops in Uintah and, thus, used to identify challenges sooner. The radiative property model was chosen due to being a more straightforward loop representative of typical ARCHES loops. The key challenge for these ports related to support for Kokkos::CUDA and primarily involved removing C/C++ functionality that did not have CUDA equivalents. Examples include replacing the use of *std::vector* inside of loops with 1-dimensional arrays of doubles, hard-coding short virtual functions inside of loops, and replacing *std::string*comparisons inside of loops with integer-based comparisons.

6.5 Single-Node Studies

Results in Section 6.5.1 and Section 6.5.2 were gathered in collaboration with Brad Peterson as part of the author's workshop paper [50]. In the sections to follow, Peterson collected Maxwell-based GPU results for RMCRT in Section 6.5.1 and CharOx in Section 6.5.2.

```
1: for all mesh patches do
2:     for all cells in a mesh patch do
3:         apply a weight to a particle's absorption coefficient
4:         store the weighted coefficient for flow cells
5:         store a zero for non-flow cells
```

Algorithm 6.1: ARCHES Radiative Particle Property Model Loop Structure

6.5.1 Radiation Modeling Results

This section presents results from experimental studies solving the Burns and Christon benchmark problem described in [22]. Past Uintah-based studies solving this problem on CPU-, GPU-, and KNL-based systems can be found in [48], [54], [56], and [94]. The studies presented here have been run as in past studies.

The results presented within this section used the following implementations of 2-level RMCRT:

- *2-Level RMCRT:CPU*: This is an existing implementation of 2-level RMCRT written to use serial tasks.
- *2-Level RMCRT:GPU*: This is an existing implementation of 2-level RMCRT written to use CUDA-based data-parallel tasks.
- *2-Level RMCRT:Kokkos*: This is an existing implementation of 2-level RMCRT written to use Kokkos-based data-parallel tasks. This implementation previously supported the Kokkos::OpenMP back-end and has been refactored to support the Kokkos::Cuda back-end as a part of this work.

SNB-based results have been gathered on a node featuring two 2.7 GHz Intel Xeon E5-2680 Sandy Bridge processors with 8 cores (2 threads per core) per processor and 64 GB of RAM. HSW-based results have been gathered on a node featuring four 2.5 GHz Intel Xeon E7-8890 v3 Haswell processors with 18 cores (2 threads per core) per processor and 2,976 GB of RAM. MAX-based results have been gathered on a node featuring a Maxwell-based NVIDIA GeForce GTX Titan X GPU with 12 GB of RAM. SKX-based results have been gathered on a node featuring one 2.7 GHz Intel Xeon Gold 6136 Skylake processor with 12 cores (2 threads per core) per processor and 256 GB of RAM. KNL-based results have been gathered on a node featuring one 1.3 GHz Intel Xeon Phi 7210 Knights Landing processor configured for *Flat-Quadrant* mode with 64 cores (4 threads per core) and 96 GB of RAM.

Simulations were launched using 1 MPI process per node. Run configurations were selected to use the extent of each node. Per-timestep timings correspond to timings for execution of a timestep as a whole. Results have been averaged over 7 consecutive timesteps. Additional details on problem setup and run configuration are discussed in individual result paragraphs.

Table 6.2 depicts cross-architecture comparisons for 2-level RMCRT. This table presents SNB-, HSW-, MAX-, SKX-, and KNL-based results for three 2-level RMCRT implementations using a problem featuring 128^3 cells on the fine mesh and 32^3 cells on the coarse mesh. Results are presented for three fine-mesh configurations (512, 64, and 8 patches with 16^3, 32^3, and 64^3 cells per patch, respectively). These results demonstrate the portability of a single codebase across many-core, multicore, and GPU-based architectures. Further, these results suggest that no performance has been lost when moving to 2-Level RMCRT:Kokkos. Best run configurations correspond to the fastest running run configuration among all possible run configurations. For SNB-based results, best run configurations with 2L-RMCRT:Kokkos have allowed for speedup up to 1.48x and 1.71x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported run configurations using the existing non-Kokkos implementation. For HSW-based results, best run configurations with 2L-RMCRT:Kokkos have allowed for speedup up to 1.53x to be achieved for 16^3 patches, respectively, over previously supported run configurations using the existing non-Kokkos implementation.

Table 6.2: Single-node per-timestep timings comparing 2-level RMCRT performance across Intel Sandy Bridge, NVIDIA GTX Titan X, Intel Skylake, and Intel Knights Landing. Same Configuration indicates the use of the same run configuration as the existing non-Kokkos implementation. Best Configuration indicates the use of the best run configuration enabled by additional flexibility introduced when adopting Kokkos. (X) indicates an impractical patch count for a run configuration using the full node. (*) indicates the use of 2 threads per core. (**) indicates the use of 4 threads per core.

PER-TIMESTEP TIMINGS - in seconds (x speedup) - SNB/HSW/MAX/SKX/KNL

Architecture	Implementation	512 - 16^3 Patches	64 - 32^3 Patches	8 - 64^3 Patches
Dual Sandy Bridge	2L-RMCRT:CPU	51.57* (-)	71.69 (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	36.30* (1.42x)	55.49 (1.29x)	X (-)
Best Configuration	2L-RMCRT:Kokkos	34.96* (1.48x)	42.03* (1.71x)	60.55* (-)
Quad Haswell	2L-RMCRT:CPU	12.21* (-)	X (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	8.81* (1.39x)	X (-)	X (-)
Best Configuration	2L-RMCRT:Kokkos	8.00* (1.53x)	9.94* (-)	14.91* (-)
Maxwell	2L-RMCRT:GPU	32.08 (-)	46.58 (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	25.88 (1.24x)	36.66 (1.27x)	X (-)
Best Configuration	2L-RMCRT:Kokkos	19.96 (1.61x)	25.60 (1.82x)	43.63 (-)
Skylake	2L-RMCRT:CPU	40.19* (-)	61.90 (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	32.41* (1.24x)	46.93 (1.32x)	X (-)
Best Configuration	2L-RMCRT:Kokkos	28.00* (1.44x)	36.97* (1.67x)	59.71* (-)
Knights Landing	2L-RMCRT:CPU	57.93** (-)	102.11 (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	43.82** (1.32x)	80.99 (1.26x)	X (-)
Best Configuration	2L-RMCRT:Kokkos	29.17** (1.99x)	38.78** (2.63x)	60.45** (-)

For Maxwell-based results, best run configurations with 2L-RMCRT:Kokkos have allowed for speedup up to 1.61x and 1.82x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported run configurations using the existing non-Kokkos implementation. For SKX-based results, best run configurations with 2L-RMCRT:Kokkos have allowed for speedup up to 1.44x and 1.67x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported run configurations using the existing non-Kokkos implementation. For KNL-based results, best run configurations with 2L-RMCRT:Kokkos have allowed for speedup up to 1.99x and 2.63x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported run configurations using the existing non-Kokkos implementation. For SNB, HSW, SKX, and HSW, these results suggest that it is advantageous to use all threads within a core.

6.5.2 Char Oxidation Modeling Results

This section presents results from experimental studies solving the char oxidation model within ARCHES.

The results presented within this section used the following implementations of CharOx:

- *CharOx:CPU*: This is an existing implementation of the char oxidation model written to use serial tasks.
- *CharOx:Kokkos*: This is a new implementation of the char oxidation model written to use Kokkos-based data-parallel tasks. This implementation has been refactored to support the Kokkos::OpenMP and Kokkos::Cuda back-ends as a part of this work.

Results have been gathered on the same nodes used for radiation modeling and described in Section 6.5.1. Simulations were launched using 1 MPI process per node. SNB-, HSW-, SKX-, and KNL-based problems used 1 patch per core with the exception of per-loop throughput timings. MAX-based problems used 16 patches with the exception of per-loop throughput timings. Note, a patch is the collection of cells assigned to a task executor. Run configurations were selected to use the extent of each node. Per-loop timings correspond to timings for the *Uintah::parallel_for* itself. Per-timestep timings correspond to timings for execution of a timestep as a whole. Results have been averaged over 7 consecutive timesteps. Additional details on problem setup and run configuration are discussed in individual result paragraphs.

Table 6.3 depicts incremental performance improvements achieved when refactoring the char oxidation model. This table presents SNB-based results gathered using the CharOx:CPU implementation for three patch sizes (16^3, 32^3, and 64^3 cells) at various steps of the refactor. Tasks were executed using 16 task executors with 1 thread per task executor via 16 MPI processes. Step 0 corresponds to the original serial loop. Step 1 corresponds to refactoring the loop to use the *Uintah::parallel_for* interface described in Section 6.3. Step 2 corresponds to replacing the use of *std::vector* inside of the loop with 1-dimensional arrays of doubles. Step 3 corresponds to replacing temporary object construction inside of the loop with 2-dimensional arrays of doubles. Step 4 corresponds to hard-coding short virtual functions inside of the loop. Step 5 corresponds to refactoring data warehouse variables to use the *Uintah::KokkosView3* interface. Step 6 corresponds to replacing *std::string* comparisons inside of the loop with integer-based comparisons. Step 7 corresponds to restructuring the loop to improve data warehouse variable access patterns. These results demonstrate that performance is a by-product of refactoring for portability.

Table 6.4 depicts cross-architecture comparisons for char oxidation modeling. This table presents SNB-, HSW-, MAX-, SKX-, and KNL-based results gathered using the CharOx:Kokkos implementation with the Kokkos::OpenMP, Kokkos::Cuda, and Kokkos:: OpenMP back-ends, respectively. For SNB-based results, tasks were executed using 16 task executors with 1 thread per task executor via 1 MPI process and 16 OpenMP threads. For HSW-based results, tasks were executed using 72 task executors with 1 thread per task executor via 1 MPI process and 72 OpenMP threads. For MAX-based results, tasks were executed using 1 CUDA stream and 16 CUDA blocks per loop with 256 CUDA threads

Table 6.3: Dual-socket per-loop timings at various steps of the CharOx:CPU refactor on Intel Sandy Bridge. Note, refactor steps are cumulative.

PER-LOOP TIMINGS - in milliseconds (x speedup) - SNB

CharOx:CPU Refactor Step	16^3 Patch	32^3 Patch	64^3 Patch
0: Original serial loop	17.87 (-)	141.80 (-)	1132.46 (-)
1: Using Uintah::parallel_for	19.19 (0.93x)	142.06 (1.00x)	1147.99 (0.99x)
2: No std::vector in loops	11.74 (1.52x)	93.72 (1.51x)	752.62 (1.50x)
3: No temporary object construction in loops	10.96 (1.63x)	88.50 (1.60x)	709.80 (1.60x)
4: No virtual functions in loops	9.75 (1.83x)	78.55 (1.81x)	634.25 (1.79x)
5: Using unmanaged Kokkos views	10.18 (1.76x)	78.61 (1.80x)	633.02 (1.79x)
6: No std::string in loops	9.16 (1.95x)	73.35 (1.93x)	591.37 (1.91x)
7: Improved memory access patterns	6.73 (2.66x)	55.19 (2.57x)	444.64 (2.55x)

Table 6.4: Single-node per-timestep loop throughput timings comparing CharOx:Kokkos performance across Intel Sandy Bridge, Intel Haswell, NVIDIA GTX Titan X, Intel Skylake, and Intel Knights Landing. (X) indicates an impractical patch count for a run configuration using the full node. (-) indicates a problem size that does not fit on the node.

PER-TIMESTEP LOOP THROUGHPUT - in milliseconds - SNB/HSW/MAX/SKX/KNL

16^3 Patches per Node	Dual Sandy Bridge	Quad Haswell	Maxwell	Skylake	Knights Landing
16	34.60	X	9.76	28.26	X
32	69.29	X	20.71	42.59	X
64	138.49	X	41.79	84.74	117.41
128	277.08	71.93	76.69	155.89	230.96
256	554.89	140.10	150.93	312.36	461.85
512	1108.88	278.82	-	618.63	915.99
1024	2219.71	526.15	-	1234.12	1878.02
2048	4444.84	1019.46	-	2437.63	3706.70
4096	-	2000.71	-	4859.79	7356.10
8192	-	4041.47	-	9868.60	-

per block and 255 registers per thread. For SKX-based results, tasks were executed using 12 task executors with 1 thread per task executor via 1 MPI process and 12 OpenMP threads. For KNL-based results, tasks were executed using 64 task executors with 4 threads per task executor via 1 MPI process and 256 OpenMP threads. Results are presented for nine patch counts (16, 32, 64, 128, 256, 512, 1024, 2048, and 4096 patches with 16^3 cells per patch). These results demonstrate the portability of a single codebase across many-core, multicore, and GPU-based architectures. While the GPU outperforms both CPU and KNL, this is achieved at the expense of problem size restrictions.

Table 6.5 depicts SNB-based OpenMP thread scalability within a task executor for char oxidation modeling. This table presents SNB-based results gathered using the CharOx:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, tasks were executed using 1, 2, 4, 8, and 16 task executor(s) with 16, 8, 4, 2, and 1 thread(s) per task executor, respectively, via 1 MPI process and 16 OpenMP threads. For 2 threads per core runs, tasks were executed using 1 and 16 task executor(s) with 32 and 2 threads per task executor, respectively, via 1 MPI process and 32 OpenMP threads. These results demonstrate that it is possible to achieve good loop-level scalability across dual-socket Sandy Bridge. When identifying optimal run configurations, this suggests that task execution times may vary little across variations of task executor counts and sizes. Comparing 1 core per loop, 1 thread per loop timings to Step 7 timings

Table 6.5: Dual-socket per-loop thread scalability in a task executor for CharOx:Kokkos on Intel Sandy Bridge. All speedups are referenced against 1 core per loop, 1 thread per loop timings. (*) indicates the use of 2 threads per core for an individual loop. (**) indicates the use of 2 sockets for an individual loop.

PER-LOOP SCALABILITY - in milliseconds (x speedup) - SNB					
Total Threads	Cores per Loop	Threads per Loop	16^3 Patch	32^3 Patch	64^3 Patch
32*	1	2	7.36 (0.94x)	50.38 (1.11x)	426.66 (1.10x)
16	1	1	6.90 (-)	55.88 (-)	469.20 (-)
16	2	2	4.38 (1.58x)	29.34 (1.90x)	239.76 (1.96x)
16	4	4	2.54 (2.72x)	15.13 (3.69x)	120.42 (3.90x)
16	8	8	1.54 (4.48x)	7.51 (7.44x)	60.28 (7.78x)
16	16**	16	0.48 (14.38x)	3.72 (15.02x)	30.62 (15.32x)
32*	16**	32	0.41 (16.83x)	3.24 (17.25x)	26.66 (17.60x)

in Table 6.3 suggests that no performance has been lost when moving to CharOx:Kokkos. The use of additional OpenMP threads within a task executor has allowed for speedups up to 16.83x, 17.25x, and 17.60x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 16 cores with 2 threads per core over the use of 1 core and 1 thread per loop. These results suggest that 2 threads per core can be used when enough per-core work is provided. Best per-loop timings achieve 43.59x, 43.77x, and 42.48x speedups for 16^3, 32^3, and 64^3 cells, respectively, over the use of 1 Sandy Bridge core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 6.3).

Table 6.6 depicts HSW-based OpenMP thread scalability within a task executor for char oxidation modeling. This table presents HSW-based results gathered using the CharOx:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, tasks were executed using 1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 36, and 72 task executor(s) with 72, 36, 24, 18, 12, 9, 8, 6, 4, 3, 2, and 1 thread(s) per task executor, respectively, via 1 MPI process and 72 OpenMP threads. For 2 threads per core runs, tasks were executed using 1 and 72 task executor(s) with 144 and 2 threads per task executor, respectively, via 1 MPI process and 144 OpenMP threads. These results demonstrate that it can be difficult to achieve good loop-level scalability across quad-socket Haswell. When identifying optimal run configurations, this suggests that task execution times may vary little across variations of task executor counts and sizes. Comparing 1 core per loop, 1 thread per loop timings to Step 7 timings in Table 6.3 suggests that no performance has been lost when moving to CharOx:Kokkos. The use of additional OpenMP

Table 6.6: Quad-socket per-loop thread scalability in a task executor for CharOx:Kokkos on Intel Haswell. All speedups are referenced against 1 core per loop, 1 thread per loop timings. (*) indicates the use of 2 threads per core for an individual loop. (**) indicates the use of 2 sockets for an individual loop. (***) indicates the use of 4 sockets for an individual loop.

PER-LOOP SCALABILITY - in milliseconds (x speedup) - HSW

Total Threads	Cores per Loop	Threads per Loop	16^3 Patch	32^3 Patch	64^3 Patch
144*	1	2	7.25 (0.97x)	50.66 (1.13x)	468.97 (1.18x)
72	1	1	7.05 (-)	57.40 (-)	553.95 (-)
72	2	2	6.33 (1.11x)	37.74 (1.52x)	276.12 (2.01x)
72	3	3	3.97 (1.78x)	26.26 (2.19x)	185.99 (2.98x)
72	4	4	3.43 (2.06x)	20.52 (2.80x)	137.48 (4.03x)
72	6	6	2.51 (2.81x)	13.79 (4.16x)	94.63 (5.85x)
72	8	8	2.16 (3.26x)	10.48 (5.48x)	73.98 (7.49x)
72	9	9	1.93 (3.65x)	9.48 (6.05x)	64.68 (8.56x)
72	12	12	1.51 (4.67x)	7.61 (7.54x)	50.63 (10.94x)
72	18	18	1.00 (7.05x)	5.16 (11.12x)	34.20 (16.20x)
72	24**	24	0.94 (7.50x)	4.25 (13.51x)	26.15 (21.18x)
72	36**	36	1.02 (6.91x)	3.81 (15.07x)	18.75 (29.54x)
72	72***	72	0.23 (30.65x)	2.17 (26.45x)	13.62 (40.67x)
144*	72***	144	0.40 (17.63x)	3.20 (17.94x)	27.68 (20.01x)

threads within a task executor has allowed for speedups up to 30.65x, 26.45x, and 40.67x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 72 cores with 1 thread per core over the use of 1 core and 1 thread per loop. These results suggest that 2 threads per core can be used when enough per-core work is provided. Best per-loop timings achieve 77.70x, 65.35x, and 83.15x speedups for 16^3, 32^3, and 64^3 cells, respectively, over the use of 1 Sandy Bridge core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 6.3).

Table 6.7 depicts SKX-based OpenMP thread scalability within a task executor for char oxidation modeling. This table presents SKX-based results gathered using the CharOx:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, tasks were executed using 1, 2, 3, 4, 6, and 12 task executor(s) with 12, 6, 4, 3, 2, and 1 thread(s) per task executor, respectively, via 1 MPI process and 12 OpenMP threads. For 2 threads per core runs, tasks were executed using 1 and 12 task executor(s) with 24 and 2 threads per task executor, respectively, via 1 MPI process and 24 OpenMP threads. These results demonstrate that it is possible to achieve good loop-level scalability across single-socket Skylake. When identifying optimal run

Table 6.7: Single-socket per-loop thread scalability in a task executor for CharOx:Kokkos on Intel Skylake. All speedups are referenced against 1 core per loop, 1 thread per loop timings. (*) indicates the use of 2 threads per core for an individual loop.

PER-LOOP SCALABILITY - in milliseconds (x speedup) - SKX					
Total Threads	Cores per Loop	Threads per Loop	16^3 Patch	32^3 Patch	64^3 Patch
24*	1	2	3.32 (0.85x)	22.78 (1.12x)	163.99 (1.20x)
12	1	1	2.81 (-)	25.60 (-)	196.86 (-)
12	2	2	2.03 (1.38x)	15.06 (1.70x)	103.77 (1.90x)
12	3	3	1.40 (2.01x)	10.13 (2.53x)	74.14 (2.66x)
12	4	4	1.01 (2.78x)	7.77 (3.29x)	54.89 (3.59x)
12	6	6	0.74 (3.80x)	5.20 (4.92x)	37.54 (5.24x)
12	12	12	0.29 (9.69x)	2.24 (11.43x)	18.63 (10.57x)
24*	12	24	0.22 (12.77x)	1.72 (14.88x)	14.09 (13.97x)

configurations, this suggests that task execution times may vary little across variations of task executor counts and sizes. The use of additional OpenMP threads within a task executor has allowed for speedups up to 12.77x, 14.88x, and 13.97x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 12 cores with 2 threads per core over the use of 1 core and 1 thread per loop. These results suggest that 2 threads per core can be used when enough per-core work is provided. Best per-loop timings achieve 81.23x, 82.44x, and 80.37x speedups for 16^3, 32^3, and 64^3 cells, respectively, over the use of 1 Skylake core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 6.3).

Table 6.8 depicts KNL-based OpenMP thread scalability within a task executor for char oxidation modeling. This table presents KNL-based results gathered using the CharOx:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, tasks were executed using 1, 2, 4, 8, 16, 32, and 64 task executor(s) with 64, 32, 16, 8, 4, 2, and 1 thread(s) per task executor(s), respectively, via 1 MPI process and 64 OpenMP threads. For 2 threads per core runs, tasks were executed using 1 and 64 task executor(s) with 128 and 2 threads per task executor, respectively, via 1 MPI process and 128 OpenMP threads. For 4 threads per core runs, tasks were executed using 1 and 64 task executor(s) with 256 and 4 threads per task executor, respectively, via 1 MPI process and 256 OpenMP threads. These results demonstrate that it can be difficult to achieve good loop-level scalability across Knights Landing. When identifying optimal run configurations, this suggests that task execution times may vary across variations of task executor counts and sizes. As a result, the use of more, yet smaller,

Table 6.8: Single-socket per-loop thread scalability in a task executor for CharOx:Kokkos on Intel Knights Landing. All speedups are referenced against 1 core per loop, 1 thread per loop timings. (*) indicates the use of 2 threads per core for an individual loop. (**) indicates the use of 4 threads per core for an individual loop.

PER-LOOP SCALABILITY - in milliseconds (x speedup) - KNL

Total Threads	Cores per Loop	Threads per Loop	16^3 Patch	32^3 Patch	64^3 Patch
256**	1	4	23.44 (1.17x)	150.83 (1.44x)	1232.83 (1.47x)
128*	1	2	23.48 (1.17x)	160.96 (1.35x)	1343.58 (1.35x)
64	1	1	27.36 (-)	216.79 (-)	1812.62 (-)
64	2	2	18.02 (1.52x)	114.52 (1.89x)	903.23 (2.01x)
64	4	4	9.55 (2.86x)	59.26 (3.66x)	459.39 (3.95x)
64	8	8	4.84 (5.65x)	31.18 (6.95x)	232.40 (7.80x)
64	16	16	2.62 (10.44x)	17.76 (12.21x)	122.78 (14.76x)
64	32	32	1.64 (16.68x)	10.55 (20.55x)	62.99 (28.78x)
64	64	64	0.63 (43.43x)	4.63 (46.82x)	30.79 (58.87x)
128*	64	128	0.59 (46.37x)	3.31 (65.50x)	23.57 (76.90x)
256**	64	256	1.59 (17.21x)	5.08 (42.68x)	27.19 (66.66x)

task executors have the potential to improve node utilization. The use of additional OpenMP threads within a task executor has allowed for speedups up to 46.37x, 65.50x, and 76.90x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 64 cores with 2 threads per core over the use of 1 core and 1 thread per loop. These results suggest that up to 4 threads per core can be used when enough per-core work is provided. Best per-loop timings achieve 30.29x, 42.84x, and 48.05x speedups for 16^3, 32^3, and 64^3 cells, respectively, over the use of 1 Sandy Bridge core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 6.3).

Table 6.9 and Table 6.10 depict MAX-based CUDA block and thread scalability for char oxidation modeling. These tables present MAX-based results gathered using the

Table 6.9: Single-device performance for varying quantities of CUDA blocks per loop for CharOx:Kokkos on NVIDIA GTX Titan X using 256 CUDA threads per block. All speedups are referenced against 1 block per loop timings.

PER-LOOP SCALABILITY - in milliseconds (x speedup) - MAX

CUDA Blocks per Loop	16^3 Patch	32^3 Patch	64^3 Patch
1	2.80 (-)	18.57 (-)	147.59 (-)
2	1.47 (1.90x)	9.59 (1.94x)	77.58 (1.90x)
4	0.80 (3.50x)	5.50 (3.38x)	43.99 (3.36x)
8	0.48 (5.83x)	3.17 (5.86x)	25.57 (5.77x)
16	0.36 (7.78x)	2.29 (8.11x)	18.99 (7.77x)
24	0.28 (10.00x)	1.92 (9.67x)	13.88 (10.63x)

Table 6.10: Single-device performance for varying quantities of CUDA threads per CUDA block for CharOx:Kokkos on NVIDIA GTX Titan X using 4 blocks per loop. All speedups are referenced against 128 threads per block timings.

PER-LOOP SCALABILITY - in milliseconds (x speedup) - MAX			
CUDA Threads per CUDA Block	16^3 Patch	32^3 Patch	64^3 Patch
128	1.35 (-)	9.09 (-)	71.91 (-)
192	1.14 (1.18x)	7.12 (1.28x)	55.24 (1.30x)
256	0.80 (1.69x)	5.50 (1.65x)	43.99 (1.63x)

CharOx:Kokkos implementation with the Kokkos::Cuda back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For Table 6.9, tasks were executed using 1 CUDA stream and 1, 2, 4, 8, 16, and 24 CUDA block(s) per loop with 256 CUDA threads per block and 255 registers per thread. For Table 6.10, tasks were executed using 1 CUDA stream and 4 CUDA blocks per loop with 128, 192, and 256 CUDA threads per block and 255 registers per thread. The use of additional CUDA blocks per loop has allowed for speedups up to 10.00x, 9.67x, and 10.63x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using up to 24 blocks per loop over the use of 1 block per loop. The use of additional CUDA threads per CUDA block has allowed for speedups up to 1.69x, 1.65x, and 1.63x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 256 threads per block over the use of 128 threads per block. Best per-loop timings achieve 63.82x, 73.85x, and 81.59x speedups for 16^3, 32^3, and 64^3 cells, respectively, over the use of 1 Sandy Bridge core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 6.3).

6.5.3 Radiative Particle Properties Modeling Results

This section presents results from experimental studies solving the radiative particle properties model within ARCHES.

The results presented within this section used the following implementations of RadProp:

- *RadProp:CPU*: This is an existing implementation of the radiative particle properties model written to use serial tasks.
- *RadProp:Kokkos*: This is a new implementation of the radiative particle properties model written to use Kokkos-based data-parallel tasks.

Results have been gathered on the same nodes used for char oxidation modeling and described in Section 6.5.1. Simulations were launched using 1 MPI process per node. Run configurations were selected to use the extent of each node. Per-timestep timings correspond to timings for execution of a timestep as a whole. Results have been averaged over 7 consecutive timesteps. Additional details on problem setup and run configuration are discussed in individual result paragraphs.

Table 6.11 depicts incremental performance improvements achieved when refactoring the radiative particle properties model. This table presents SNB-based results gathered using the RadProp:CPU implementation for three patch sizes (16^3, 32^3, and 64^3 cells) at various steps of the refactor. Tasks were executed using 16 task executors with 1 thread per task executor via 16 MPI processes. Step 0 corresponds to the original serial loop. Step 1 corresponds to refactoring the loop to use the *Uintah::parallel_for* interface described in Section 6.3. These results demonstrate that performance is a by-product of refactoring for portability.

Table 6.12 depicts cross-architecture comparisons for radiative particle properties modeling. This table presents SNB-, HSW-, MAX-, SKX-, and KNL-based results gathered using the RadProp:Kokkos implementation with the Kokkos::OpenMP, Kokkos::Cuda, and Kokkos::OpenMP back-ends, respectively. For SNB-based results, tasks were executed using 16 task executors with 1 thread per task executor via 1 MPI process and 16 OpenMP threads. For HSW-based results, tasks were executed using 72 task executors with 1 thread per task executor via 1 MPI process and 72 OpenMP threads. For MAX-based results, tasks were executed using 1 CUDA stream and 16 CUDA blocks per loop with 256 CUDA threads per block and 255 registers per thread. For SKX-based results, tasks were executed using 12 task executors with 1 thread per task executor via 1 MPI process and 12 OpenMP threads. For KNL-based results, tasks were executed using 64 task executors with 4 threads per task

Table 6.11: Dual-socket per-loop timings at various steps of the RadProp:CPU refactor on Intel Sandy Bridge. Note, refactor steps are cumulative.

PER-LOOP TIMINGS - in microseconds (x speedup) - SNB

RadProp:CPU Refactor Step	16^3 Patch	32^3 Patch	64^3 Patch
0: Original serial loop	78.67 (-)	591.28 (-)	4574.66 (-)
1: Using Uintah::parallel_for	17.80 (4.42x)	119.90 (4.93x)	973.45 (4.70x)

Table 6.12: Single-node per-timestep loop throughput timings comparing RadProp:Kokkos performance across Intel Sandy Bridge, Intel Haswell, NVIDIA GTX Titan X, Intel Skylake, and Intel Knights Landing. (X) indicates an impractical patch count for a run configuration using the full node. (-) indicates a problem size that does not fit on the node.

PER-TIMESTEP LOOP THROUGHPUT - in microseconds - SNB/HSW/MAX/SKX/KNL

16^3 Patches per Node	Dual Sandy Bridge	Quad Haswell	Maxwell	Skylake	Knights Landing
16	307.91	-	-	354.08	X
32	614.95	-	-	540.19	X
64	1234.04	-	-	1089.02	1435.31
128	2470.63	540.37	-	1991.16	2865.38
256	4925.28	1064.93	-	3972.65	5717.83
512	9840.19	2116.74	-	7793.70	11382.00
1024	19650.43	3946.78	-	15685.47	22809.94
2048	-	7661.01	-	31083.13	45686.11
4096	-	15605.85	-	62035.25	-

executor via 1 MPI process and 256 OpenMP threads. Results are presented for nine patch counts (16, 32, 64, 128, 256, 512, 1024, 2048, and 4096 patches with 16^3 cells per patch). These results demonstrate the portability of a single codebase across many-core, multicore, and GPU-based architectures. While the GPU outperforms both CPU and KNL, this is achieved at the expense of problem size restrictions.

Table 6.13 depicts SNB-based OpenMP thread scalability within a task executor for radiative particle properties modeling. This table presents SNB-based results gathered using the RadProp:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, tasks were executed using 1, 2, 4, 8, and 16 task executor(s) with 16, 8, 4, 2, and 1 thread(s) per task executor, respectively,

Table 6.13: Dual-socket per-loop thread scalability in a task executor for RadProp:Kokkos on Intel Sandy Bridge. All speedups are referenced against 1 core per loop, 1 thread per loop timings. (*) indicates the use of 2 threads per core for an individual loop. (**) indicates the use of 2 sockets for an individual loop.

PER-LOOP SCALABILITY - in microseconds (x speedup) - SNB

Total Threads	Cores per Loop	Threads per Loop	16^3 Patch	32^3 Patch	64^3 Patch
32*	1	2	223.88 (0.46x)	784.69 (1.01x)	5413.83 (1.19x)
16	1	1	102.43 (-)	793.02 (-)	6427.33 (-)
16	2	2	78.16 (1.31x)	499.11 (1.59x)	3549.99 (1.81x)
16	4	4	59.46 (1.72x)	259.76 (3.05x)	1852.52 (3.47x)
16	8	8	33.29 (3.08x)	115.95 (6.84x)	818.76 (7.85x)
16	16**	16	19.79 (5.18x)	74.41 (10.66x)	443.11 (14.51x)
32*	16**	32	19.00 (5.39x)	59.12 (13.41x)	356.52 (18.03x)

via 1 MPI process and 16 OpenMP threads. For 2 threads per core runs, tasks were executed using 1 and 16 task executor(s) with 32 and 2 threads per task executor, respectively, via 1 MPI process and 32 OpenMP threads. These results demonstrate that it is possible to achieve good loop-level scalability across dual-socket Sandy Bridge. When identifying optimal run configurations, this suggests that task execution times may vary little across variations of task executor counts and sizes. Comparing 1 core per loop, 1 thread per loop timings to Step 7 timings in Table 6.11 suggests that no performance has been lost when moving to RadProp:Kokkos. The use of additional OpenMP threads within a task executor has allowed for speedups up to 5.39x, 13.41x, and 18.03x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 16 cores with 2 threads per core over the use of 1 core and 1 thread per loop. These results suggest that 2 threads per core can be used when enough per-core work is provided. Best per-loop timings achieve 43.15x, 43.83x, and 42.48x speedups for 16^3, 32^3, and 64^3 cells, respectively, over the use of 1 Sandy Bridge core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 6.11).

Table 6.14 depicts HSW-based OpenMP thread scalability within a task executor for radiative particle properties modeling. This table presents HSW-based results gathered

Table 6.14: Quad-socket per-loop thread scalability in a task executor for RadProp:Kokkos on Intel Haswell. All speedups are referenced against 1 core per loop, 1 thread per loop timings. (*) indicates the use of 2 threads per core for an individual loop. (**) indicates the use of 2 sockets for an individual loop. (***) indicates the use of 4 sockets for an individual loop.

PER-LOOP SCALABILITY - in microseconds (x speedup) - HSW					
Total Threads	Cores per Loop	Threads per Loop	16^3 Patch	32^3 Patch	64^3 Patch
144*	1	2	340.39 (0.28x)	1125.72 (0.64x)	5883.01 (1.04x)
72	1	1	96.30 (-)	722.08 (-)	6093.75 (-)
72	2	2	232.19 (0.41x)	1103.73 (0.65x)	4943.64 (1.23x)
72	3	3	190.59 (0.51x)	770.78 (0.94x)	3432.86 (1.78x)
72	4	4	188.96 (0.51x)	552.55 (1.31x)	2675.64 (2.28x)
72	6	6	170.94 (0.56x)	356.99 (2.02x)	1783.82 (3.42x)
72	8	8	153.67 (0.63x)	284.09 (2.54x)	1296.63 (4.70x)
72	9	9	142.91 (0.67x)	247.60 (2.92x)	1121.73 (5.43x)
72	12	12	163.14 (0.59x)	235.99 (3.06x)	904.26 (6.74x)
72	18	18	137.55 (0.70x)	184.18 (3.92x)	596.30 (10.22x)
72	24**	24	178.87 (0.54x)	174.76 (4.13x)	545.12 (11.18x)
72	36**	36	187.03 (0.51x)	192.18 (3.76x)	496.12 (12.28x)
72	72***	72	18.79 (5.13x)	88.31 (8.18x)	257.35 (23.68x)
144*	72***	144	26.96 (3.58x)	140.62 (5.13x)	305.45 (19.95x)

using the RadProp:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, tasks were executed using 1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 36, and 72 task executor(s) with 72, 36, 24, 18, 12, 9, 8, 6, 4, 3, 2, and 1 thread(s) per task executor, respectively, via 1 MPI process and 72 OpenMP threads. For 2 threads per core runs, tasks were executed using 1 and 72 task executor(s) with 144 and 2 threads per task executor, respectively, via 1 MPI process and 144 OpenMP threads. These results demonstrate that it can be difficult to achieve good loop-level scalability across quad-socket Haswell. When identifying optimal run configurations, this suggests that task execution times may vary little across variations of task executor counts and sizes. Comparing 1 core per loop, 1 thread per loop timings to Step 7 timings in Table 6.11 suggests that no performance has been lost when moving to RadProp:Kokkos. The use of additional OpenMP threads within a task executor has allowed for speedups up to 5.13x, 8.18x, and 23.68x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 72 cores with 1 thread per core over the use of 1 core and 1 thread per loop. These results suggest that 2 threads per core can be used when enough per-core work is provided. Best per-loop timings achieve 77.32x, 65.23x, and 83.13x speedups for 16^3, 32^3, and 64^3 cells, respectively, over the use of 1 Sandy Bridge core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 6.11).

Table 6.15 depicts SKX-based OpenMP thread scalability within a task executor for radiative particle properties modeling. This table presents SKX-based results gathered using the RadProp:Kokkos implementation with the Kokkos::OpenMP back-end for three

Table 6.15: Single-socket per-loop thread scalability in a task executor for RadProp:Kokkos on Intel Skylake. All speedups are referenced against 1 core per loop, 1 thread per loop timings. (*) indicates the use of 2 threads per core for an individual loop.

PER-LOOP SCALABILITY - in microseconds (x speedup) - SKX					
Total Threads	Cores per Loop	Threads per Loop	16^3 Patch	32^3 Patch	64^3 Patch
24*	1	2	85.70 (0.69x)	533.64 (0.85x)	3808.82 (0.94x)
12	1	1	59.13 (-)	455.37 (-)	3581.34 (-)
12	2	2	61.30 (0.96x)	418.98 (1.09x)	2179.17 (1.64x)
12	3	3	31.55 (1.87x)	397.69 (1.15x)	1463.91 (2.45x)
12	4	4	25.99 (2.28x)	256.55 (1.77x)	1096.30 (3.27x)
12	6	6	95.03 (0.62x)	163.83 (2.78x)	743.24 (4.82x)
12	12	12	15.18 (3.90x)	68.13 (6.68x)	352.61 (10.16x)
24*	12	24	14.32 (4.13x)	61.06 (7.46x)	328.25 (10.91x)

patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, tasks were executed using 1, 2, 3, 4, 6, and 12 task executor(s) with 12, 6, 4, 3, 2, and 1 thread(s) per task executor, respectively, via 1 MPI process and 12 OpenMP threads. For 2 threads per core runs, tasks were executed using 1 and 12 task executor(s) with 24 and 2 threads per task executor, respectively, via 1 MPI process and 24 OpenMP threads. These results demonstrate that it is possible to achieve good loop-level scalability across single-socket Skylake. When identifying optimal run configurations, this suggests that task execution times may vary little across variations of task executor counts and sizes. The use of additional OpenMP threads within a task executor has allowed for speedups up to 4.13x, 7.46x, and 10.91x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 12 cores with 2 threads per core over the use of 1 core and 1 thread per loop. These results suggest that 2 threads per core can be used when enough per-core work is provided. Best per-loop timings achieve 82.23x, 82.61x, and 80.35x speedups for 16^3, 32^3, and 64^3 cells, respectively, over the use of 1 Skylake core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 6.11).

Table 6.16 depicts KNL-based OpenMP thread scalability within a task executor for radiative particle properties modeling. This table presents KNL-based results gathered using the RadProp:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, tasks were executed using

Table 6.16: Single-socket per-loop thread scalability in a task executor for RadProp:Kokkos on Intel Knights Landing. All speedups are referenced against 1 core per loop, 1 thread per loop timings. (*) indicates the use of 2 threads per core for an individual loop. (**) indicates the use of 4 threads per core for an individual loop.

PER-LOOP SCALABILITY - in microseconds (x speedup) - KNL

Total Threads	Cores per Loop	Threads per Loop	16^3 Patch	32^3 Patch	64^3 Patch
256**	1	4	992.09 (0.48x)	3215.50 (1.16x)	20491.30 (1.45x)
128*	1	2	719.39 (0.66x)	3303.79 (1.13x)	22400.40 (1.32x)
64	1	1	478.18 (-)	3728.15 (-)	29660.60 (-)
64	2	2	431.42 (1.11x)	2572.35 (1.45x)	16909.00 (1.75x)
64	4	4	287.66 (1.66x)	1444.63 (2.58x)	8257.03 (3.59x)
64	8	8	264.37 (1.81x)	648.64 (5.75x)	4111.10 (7.21x)
64	16	16	200.43 (2.39x)	409.10 (9.11x)	2109.31 (14.06x)
64	32	32	187.86 (2.55x)	314.80 (11.84x)	1144.52 (25.96x)
64	64	64	34.64 (13.80x)	146.72 (25.41x)	757.80 (39.14x)
128*	64	128	39.87 (11.99x)	214.89 (17.35x)	1013.38 (29.27x)
256**	64	256	124.55 (3.84x)	526.49 (7.08x)	2070.74 (14.32x)

1, 2, 4, 8, 16, 32, and 64 task executor(s) with 64, 32, 16, 8, 4, 2, and 1 thread(s) per task executor(s), respectively, via 1 MPI process and 64 OpenMP threads. For 2 threads per core runs, tasks were executed using 1 and 64 task executor(s) with 128 and 2 threads per task executor, respectively, via 1 MPI process and 128 OpenMP threads. For 4 threads per core runs, tasks were executed using 1 and 64 task executor(s) with 256 and 4 threads per task executor, respectively, via 1 MPI process and 256 OpenMP threads. These results demonstrate that it can be difficult to achieve good loop-level scalability across Knights Landing. When identifying optimal run configurations, this suggests that task execution times may vary across variations of task executor counts and sizes. As a result, the use of more, yet smaller, task executors have the potential to improve node utilization. The use of additional OpenMP threads within a task executor has allowed for speedups up to 13.80x, 25.41x, and 39.14x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 64 cores with 1 thread per core over the use of 1 core and 1 thread per loop. These results suggest that up to 4 threads per core can be used when enough per-core work is provided. Best per-loop timings achieve 30.11x, 42.84x, and 48.04x speedups for 16^3, 32^3, and 64^3 cells, respectively, over the use of 1 Sandy Bridge core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 6.11).

6.6 Foreseable Challenges

The approach presented here is a starting point for easing the adoption of a performance portability layer in large legacy codebases. Foreseeable challenges include understanding how to: (1) use third party libraries using a performance portability layer in a codebase using a performance portability layer (e.g., using hypre in Uintah while using Kokkos in both hypre and Uintah), (2) manage increasing configurability (e.g., multiple tuneable run-time parameters across host and device), (3) make informed use of underlying programming models (e.g., using the device only when advantageous for a given loop), and (4) efficiently manage parallel execution and memory across multiple underlying programming models.

6.7 Summary

This work has helped improve Uintah's portability to current and future architectures and programming models while also preserving support for pre-existing code. Specifically, it has shown an approach for indirectly adopting a performance portability layer to

help improve legacy code support and long-term portability in a large legacy codebase. Kokkos capabilities have been shown when using this approach to make portable use of Kokkos::OpenMP and Kokkos::CUDA across multicore-, many-core-, and GPU-based nodes using a single implementation for the case studies examined. At the node-level, performance improvements up to 2.7x when refactoring for portability and 2.6x when more efficiently using a node have been achieved. At scale, good strong-scaling to 442,368 threads across 1,728 Knights Landing processors has been achieved using MPI+Kokkos.

These advancements have been made possible by the introduction of a framework-specific portability layer between Uintah's application code and Kokkos. This intermediate layer consists of three components: (1) loop-level support providing application developers with framework-specific abstractions (e.g., generic parallel loop statements) that map to interface-specific abstractions (e.g., PPL-specific parallel loop statements), (2) application-level support that includes a tagging system to identify which interfaces are supported by a given loop, and (3) build-level support that includes selective compilation of loops to allow for incremental refactoring and simultaneous use of multiple underlying programming models for heterogeneous HPC systems. This layer provides application developers with easy to use portable abstractions while allowing maintaining developers to easily add, remove, and tune interfaces to underlying programming models in a single location with fewer far-reaching changes across application code.

The portability and performance improvements shown here offer encouragement as we extend more of Uintah to heterogeneous HPC systems using Kokkos::OpenMP and Kokkos::CUDA. Next steps include furthering our understanding of Kokkos use across host and device simultaneously on heterogeneous IBM- and NVIDIA-based systems with multiple sockets and devices per node. As a part of this, emphasis will again be placed on long-term portability and managing simultaneous use of host and device in a portable manner with upcoming systems such as the future Intel-based DOE Aurora and AMD-based DOE Frontier in mind.

CHAPTER 7

A HETEROGENEOUS MPI+KOKKOS TASK SCHEDULING APPROACH

7.1 Overview

The complexity of nodes anticipated in exascale systems poses new challenges for codes emphasizing large-scale simulation. Asynchronous many-task runtime systems and MPI+X hybrid parallelism approaches show promise for helping manage the increased concurrency, deep memory hierarchies, and heterogeneity. Performance portability layers (PPL) show promise for helping manage the architectural diversity. The combination of these promising solutions, however, poses challenges for asynchronous many-task runtime systems. This is a result of the rapid and varying rates of development that the other solutions are experiencing while trying to maintain pace with current and emerging HPC systems. Such challenges are complicated further for runtimes using third-party libraries, whose developers are facing similar challenges, and large legacy runtimes, where the combination may require a significant investment.

This chapter captures work from a conference paper by the author [51] and addresses these challenges using a heterogeneous MPI+PPL task scheduling approach for combining these solutions with additional consideration for parallel third party libraries to help prepare such a runtime for the diverse heterogeneous systems accompanying exascale computing. The goal of this approach is the implementation of a heterogeneous MPI+PPL task scheduler achieving scalable adaptive execution of individually portable tasks making simultaneous use of both the host and device through a performance portability layer on complex heterogeneous nodes. For application developers, this scheduler allows for easy means of improving complex heterogeneous node use without requiring extensive knowledge of low-level details for making efficient use of the underlying hardware and programming models. For infrastructure developers, this scheduler provides the foundation

for a wholly portable heterogeneous MPI+PPL task scheduling approach. Here, wholly portable refers to an approach using portable abstractions for task scheduling in addition to portable abstractions inside individual tasks.

This chapter demonstrates this approach using a heterogeneous MPI+Kokkos task scheduler implemented in the Uintah Computational Framework, an asynchronous many-task runtime system, with additional consideration for hypre, a parallel library. This design is informed by a Kokkos adoption effort adding portable support for OpenMP and CUDA in both (1) a complex real-world application requiring the use of a parallel third-party library, hypre [34], and (2) an asynchronous many-task runtime system, Uintah. This ongoing effort has been non-trivial due to the codebase: (1) consisting of 1-2 million lines of complex code, (2) maintaining a divide between application code, where accompanying portable abstractions [49] are used, and infrastructure code, where task scheduling logic is maintained, (3) having 100s of pre-existing loops to port in application code, (4) being under active development with many contributors, and (5) having a pre-existing userbase to support. The resulting approach aims to ease scheduler implementation in similar runtimes while also helping to prepare Uintah for early use of the DOE Aurora system through the Aurora Early Science Program. Though implementation has been limited to support for NVIDIA GPUs, high-level ideas associated with this approach are broad enough to apply to other GPUs.

To demonstrate task scheduling capabilities, strong scaling studies using this scheduler with accompanying portable abstractions [49] are examined for two challenging problems executing workloads representative of typical Uintah applications. These strong scaling studies show heterogeneous use of OpenMP and CUDA via Kokkos across multi-socket, multi-device nodes using a single source implementation. The associated refactors have allowed for performance improvements up to $4.4x$ to be achieved when using this scheduler and the accompanying portable abstractions [49] to port a previously MPI-Only problem to MPI+Kokkos::OpenMP+Kokkos::CUDA to improve complex heterogeneous node use. At scale, the use of MPI+Kokkos::OpenMP+Kokkos::CUDA has allowed for good strong scaling to 1,024 NVIDIA V100 GPUs and 512 IBM POWER9 processors to be achieved.

7.2 Scheduler Improvements

To support this work, Uintah's Unified Scheduler was extended with the help of Damodar Sahasrabudhe (D.Sa.) to address several shortcomings preventing large-scale ARCHES simulations using the latest portable infrastructure. Shortcomings addressed are itemized below, with initials included for key contributors.

1. The scheduler required all tasks to use the same number of halo cells to avoid excessive synchronizations. (D.Sa., J.K.H.)
2. The scheduler supported "read-only" and "write-only" dependencies between tasks but not "read-write" dependencies. (J.K.H., D.Sa.)
3. Scheduler logic featured multiple race conditions. (D.Sa.)
4. Hypre tasks were not GPU portable. (D.Sa.)
5. Uintah's parallelism model did not match hypre's most performant. (D.Sa., J.K.H.)
6. Legacy tasks were not portable. (J.K.H., D.Sa.)
7. The scheduler was based on PThreads and limited to execution of serial tasks on the host-side. (J.K.H.)

To address (7), the device-side support from this effort was combined with the host-side support from Chapter 5's task scheduler. Additional details on (1) through (6) can be found in Damodar Sahasrabudhe's dissertation [101].

Algorithm 7.1 provides an overview of the resulting task executor logic forming the core of the heterogeneous MPI+Kokkos task scheduler. Execution begins serially at Line 1. Execution is passed to Kokkos at Line 2 to launch multiple disjoint task executors within an MPI process using *Kokkos::OpenMP::partition_master* to achieve 2 levels of portable parallelism. Individual task executors execute Algorithm 7.1 largely independent of one another until all tasks are computed. Note, synchronization is used as needed to ensure data integrity (e.g., when selecting a task to execute at Line 10).

The logic in Lines 3 to 16 corresponds to a task queue-based state machine used to coordinate data dependencies and non-trivial interactions taking place while tasks are in flight. This state machine consists of a series of individual task queues specific to each

```
1: while tasks to compute do
2:     launch multiple disjoint per-MPI process task executors
3:     while tasks to compute do
4:         // TASK QUEUE STATE MACHINE
5:         mark tasks with MPI complete as “pending halo cells”
6:         initiate host-to-device/device-to-host transfers for “pending halo cells”
7:         query pending host-to-device/device-to-host transfers
8:         decrement dependency count for completed transfers
9:         mark tasks with all dependencies complete as “ready”
10:        select a “ready” task and break if a hypre task
11:        mark write-only and read-write dependencies as “invalid“
12:        launch “ready” task on host or device
13:        mark write-only and read-write dependencies as “valid”
14:        initiate MPI sends for completed tasks
15:        process pending MPI receives
16:    exit multiple disjoint per-MPI process task executors
17:    launch hypre task using single per-MPI process task executor on host or device
```

Algorithm 7.1: Task Executor Logic.

action itemized in Line 5 through 15. Note for Line 5 and 6, halo cells exist as individually allocated and transferred groups of cells independent of per-loop cell data. More details on Uintah's automatic halo management can be found in a recent article [92]. Note for Line 12 and 18, host or device execution is determined by the application developer using a task tagging system implemented as a part of the portable abstractions [49] that accompany this task scheduler.

For example, Line 5 corresponds to a task queue collecting all tasks that have been identified as having MPI complete that are awaiting halo cells to arrive from another task (e.g., an earlier task that modified halo cell data). After a given task has been identified as having MPI complete and awaiting halo cells, it is pushed to the next queue, Line 6, where host-device data movement coordinating the gathering of these halo cells is initiated. Each individual task executor repeatedly iterates over this state machine looking for work to process and implicitly stealing work from one another until all tasks are computed.

Uintah's use of *Kokkos::OpenMP::partition_master* determines (1) how many simultaneously executing task executors are used in an MPI process (i.e., 1 to many) and (2) how many compute resources are used by a given task executor (i.e., 1 to many). Together, the two dictate how many tasks may be executed across compute resources at a given time (e.g., 1 task across 10 cores vs. 10 tasks across 1 core each). This is a key detail for improving node use as it provides means of controlling granularity on not only task execution itself but actions forming the state machine in Lines 3 to 16 (e.g., to determine how many or few

task executors are, say, initiating host-device data transfers).

In practice, individual task executors are bound to specific compute resources (e.g., cores) and *MPI_THREAD_MULTIPLE* is used to coordinate data dependencies across a shared per-MPI process data warehouse. Binding is accomplished using a combination of OpenMP environment variables (e.g., *OMP_NUM_THREADS*) and Uintah-specific run configuration parameters (e.g., specifying the number of CUDA threads per block). More details on Uintah's use of *Kokkos::OpenMP::partition_master* be found in a recent technical report [50]. Note, use of *MPI_THREAD_MULTIPLE*, where multiple threads may call MPI with no restrictions, has proven difficult when trying to coordinate data dependencies across a number of task executors. To ease coordination, an *MPI_THREAD_FUNNELED*-based approach, where only the main thread will make MPI calls, has been proposed by Alan Humphrey. The key difference between the two is a transition from using a single shared data warehouse across tasks to a per-task data warehouse model. The latter eases coordination as it eliminates the need to manage data dependencies across tasks.

A key limitation of Uintah's current heterogeneous MPI+Kokkos task scheduler is that it makes use of raw CUDA, which must be replaced with portable alternatives for forthcoming exascale systems. Specifically, cudaStream, cudaMemcpyAsync, and cudaStreamQuery are used. Line 6 makes use of cudaMemcpyAsync for asynchronous host-to-device (H2D) and device-to-host (D2H) transfers. Line 7 makes use of cudaStreamQuery to check the status of transfers. While limited, this use was unavoidable due to the maturity of Kokkos at the time of early adoption.

7.3 Target Exascale Benchmarks

The results presented in this chapter used two problems previously described in Section 3.5.1 and Section 3.5.4 to uniquely stress different portions of three individually ported codes: (1) Uintah's ARCHES turbulent combustion simulation component [106], (2) Uintah's standalone linear solver using Lawrence Livermore National Laboratory's hypre [34], and (3) Uintah's standalone reverse Monte-Carlo ray tracing (RMCRT) radiation model [54]. These codes are central to both CCMSC boiler simulations and subsequent combustion research. Note, demonstrations of weak-scaling for these codes can be found in past studies [57, 69, 104]. For RMCRT, weak scaling is possible through the use of aggressive

mesh refinement to reduce communication requirements.

The first problem, a helium plume, demonstrates the newly portable interoperability of (1) and (2) on a single-level structured grid. The second problem, a modified Burns and Christon benchmark, demonstrates the newly portable interoperability of (1), (2), and (3) on a 2-level structured adaptive mesh refinement grid. For both problems, newly ported single source implementations with underlying support for legacy serial loops and Kokkos-based data-parallel loops for Kokkos::OpenMP and Kokkos::CUDA were used. Note for (2), a modified version of hypre implementing recently published techniques [103] for improving GPU-based performance of hypre was used. Note for both, a modified version of Kokkos implementing recently published techniques [94] for improving GPU-based performance of Kokkos was used.

As described in Section 3.5.4, helium plume problems played a key role in CCMSC efforts for their ability to validate ARCHES using problems with characteristics representative of a real fire but without introducing the complexities of combustion [104]. The problem used here consists of 125 unique portable loops individually using up to 17 variables with complex interconnectedness. Underlying Kokkos functionality used among loops includes Kokkos::parallel_for, Kokkos::parallel_reduce, and Kokkos::View. A key feature making this an important problem for validating Uintah's heterogeneous MPI+Kokkos task scheduler is the large number of unique portable loops and variables in flight during execution. This is helpful for ensuring robustness due to the long and complex data dependency sequences generated by these loops (e.g., variables computed on the host, modified on the device, and later required on the host). Note, there are domain decomposition and run configuration dependent multipliers on unique loops not reflected in the counts above. More details on the helium plume problem can be found in Section 3.5.4.

As described in Section 3.5.1, Uintah's 2-level reverse Monte-Carlo ray tracing (RMCRT) radiation model [54] also played a key role in CCMSC boiler simulations, where radiation is the dominant mode of heat transfer. The problem used here modifies the ARCHES' Burns and Christon benchmark problem to incorporate a pressure solve, requiring use of hypre, and consists of 19 unique portable loops individually using up to 28 variables with complex interconnectedness. Underlying Kokkos functionality used among loops includes Kokkos::parallel_for, Kokkos::parallel_reduce, Kokkos::View, and Kokkos_Random. A

key feature making this an important problem for validating Uintah's heterogeneous MPI+Kokkos task scheduler is the ability to simultaneously stress interoperability of ARCHES, hypre, and RMCRT while also stressing Uintah's adaptive mesh refinement support. This is helpful for ensuring robustness due to the complex hand-offs that take place between these codes (e.g., shared data dependencies). Note, there are domain decomposition and run configuration dependent multipliers on unique loops not reflected in the counts above. More details on the radiation problem can be found in Section 3.5.1.

7.4 Multi-Node Results

Results in Section 7.4.1 through Section 7.4.3 were gathered in collaboration with Damodar Sahasrabudhe as part of the joint conference paper [51].

7.4.1 Lassen Results

To demonstrate scalability of Uintah's heterogeneous MPI+Kokkos task scheduler, strong scaling studies were performed on the DOE Lassen system. This system features two IBM POWER9 processors with 22 cores (4 SMT threads per core) per processor, four Volta-based NVIDIA Tesla V100 GPUs with 5,120 CUDA cores and 16 GB of HBM2 per GPU, and 256 GB of DDR4 per node. For both problems, these studies explored varying domain decomposition approaches using problems sized to fill the 64 GB per-node memory footprint of HBM2. Note, additional demonstration of host-only capabilities related to this approach can be found in Chapter 5.

For MPI+Kokkos, simulations were launched using 4 MPI processes per node. Within an MPI process, 10 OpenMP threads were used to simultaneously launch and execute loops across: (1) 10 cores using 1 core and 1 SMT thread per loop for Kokkos::OpenMP and (2) 1 V100 using 1 CUDA stream and 256 CUDA blocks per loop for Kokkos::CUDA with 256 CUDA threads per block. For MPI-Only, simulations were launched using 40 MPI processes per node to execute loops using 1 core and 1 SMT thread per loop.

Problems were sized to provide each MPI process with at least 1 patch in all data points shown. Here, a patch refers to the collection of cells executed by a loop. Note, larger patch sizes result in fewer patches being available to distribute across MPI processes. Data points with fewer than 1 patch per MPI process were omitted from figures. Reported per-timestep timings measure the simultaneous execution of all loops across both the host

and device in a given timestep. Results have been averaged over 7 consecutive timesteps. Additional details on problem setup and run configuration are discussed in individual result paragraphs.

Figure 7.1 shows strong scaling results for the helium plume problem on a single-level structured grid. Results were gathered using MPI+Kokkos::OpenMP+Kokkos::CUDA for a problem featuring 512^3 cells for three patch sizes (32^3, 64^3, and 128^3 cells per patch). Note for all patches sizes, individual patches were combined to a single patch when passed to hypre for execution using CUDA. This is done to allow Uintah to make performant use of hypre as recently demonstrated in [103]. To enable comparisons to how this problem would traditionally have been run using ARCHES, results were also gathered using Uintah's MPI-Only task scheduler for a problem featuring 512^3 cells for one patch size (32^3 cells per patch). Note, a single patch size is used here as the MPI-Only task scheduler does not support running this problem with 64^3 and 128^3 patches due to Uintah's 1 patch per MPI process requirement. Use of MPI+Kokkos::OpenMP+Kokkos::CUDA to improve complex heterogeneous node use has allowed for speedups up to 2.3x and 4.4x to be achieved for 64^3 and 128^3 cells, respectively, over MPI-Only.

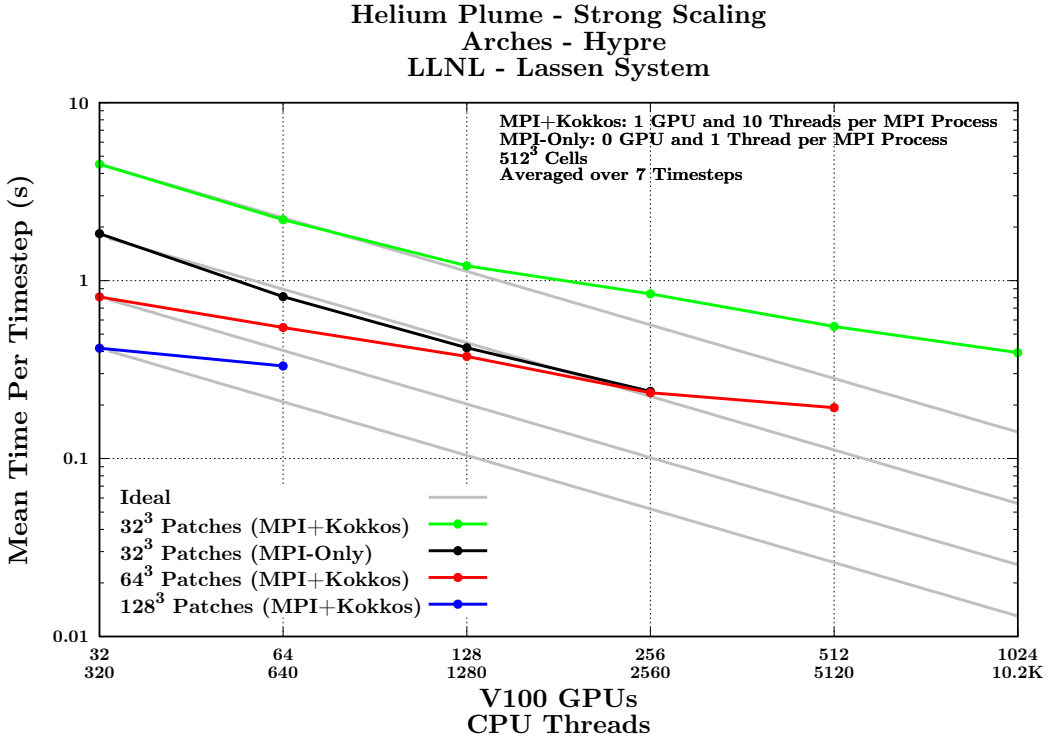

Fig. 7.1: Helium plume run to 1,024 V100 GPUs and 512 POWER9 processors.

Figure 7.2 shows strong scaling results for the modified Burns and Christon benchmark problem on a 2-level structured adaptive mesh refinement grid. Results were gathered using MPI+Kokkos::OpenMP+Kokkos::CUDA for a problem featuring 512^3 cells on the fine mesh and 128^3 cells on the coarse mesh for three fine-mesh patch sizes (32^3, 64^3, and 128^3 cells per fine mesh patch). Note for all patches sizes, individual patches were combined to a single patch when passed to hypre for execution using CUDA. This is done to allow Uintah to make performant use of hypre as recently demonstrated in [103]. Note, MPI-Only comparisons are not made here as the global, all-to-all nature of the radiation model used by this problem necessitates the use of MPI+X hybrid parallelism [54]. For MPI+CUDA comparisons, see related MPI+CUDA and MPI+Kokkos::CUDA comparisons [49, 94] gathered on a single node and the DOE Titan system, respectively.

7.4.2 Summit Results

To demonstrate scalability of Uintah's heterogeneous MPI+Kokkos task scheduler, strong scaling studies were performed on the DOE Summit system. This system features

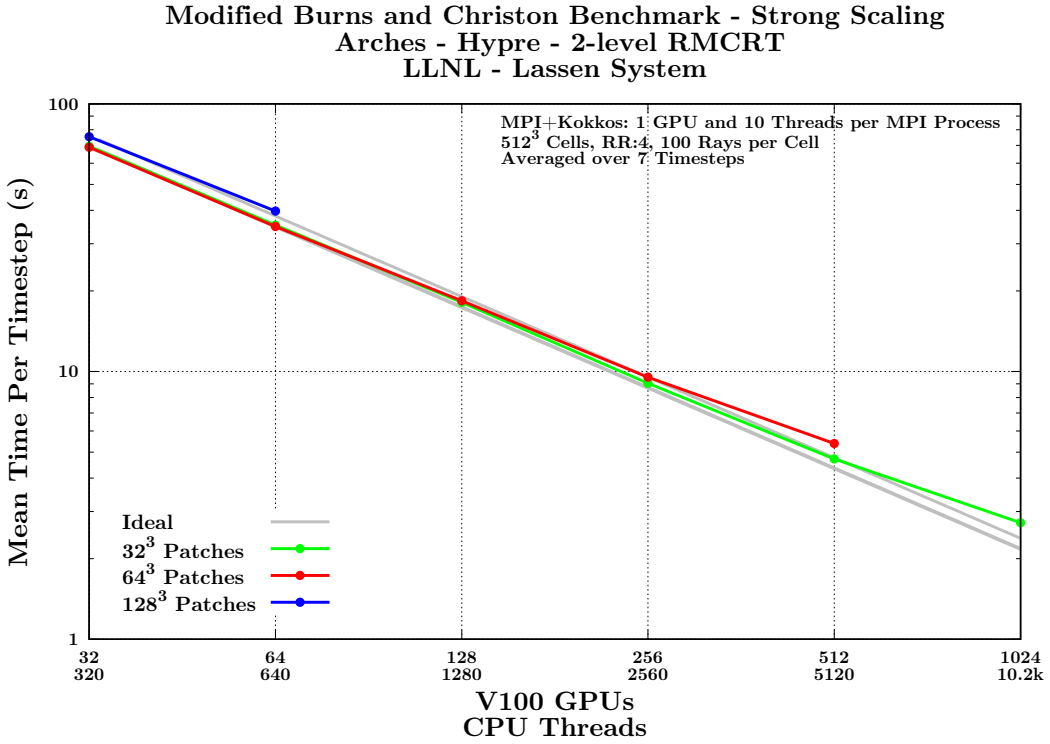

Fig. 7.2: Modified Burns and Christon benchmark run to 1,024 V100 GPUs and 512 POWER9 processors.

two IBM POWER9 processors with 22 cores (4 SMT threads per core) per processor, six Volta-based NVIDIA Tesla V100 GPUs with 5,120 CUDA cores and 16 GB of HBM2 per GPU, and 512 GB of DDR4 per node. For both problems, these studies explored varying domain decomposition approaches using problems sized to fill the 96 GB per-node memory footprint of HBM2. Note, additional demonstration of host-only capabilities related to this approach can be found in Chapter 5.

For MPI+Kokkos, simulations were launched using 6 MPI processes per node. Within an MPI process, 7 OpenMP threads were used to simultaneously launch and execute loops across: (1) 7 cores using 1 core and 1 SMT thread per loop for Kokkos::OpenMP and (2) 1 V100 using 1 CUDA stream and 256 CUDA blocks per loop for Kokkos::CUDA with 256 CUDA threads per block.

Problems were sized to provide each MPI process with at least 1 patch in all data points shown. Here, a patch refers to the collection of cells executed by a loop. Note, larger patch sizes result in fewer patches being available to distribute across MPI processes. Data points with fewer than 1 patch per MPI process were omitted from figures. Reported per-timestep timings measure the simultaneous execution of all loops across both the host and device in a given timestep. Results have been averaged over 7 consecutive timesteps. Additional details on problem setup and run configuration are discussed in individual result paragraphs.

Figure 7.3 shows strong scaling results for the helium plume problem on a single-level structured grid. Results were gathered using MPI+Kokkos::OpenMP+Kokkos::CUDA for three problem sizes (768^3, 1536^3, and 3072^3 cells) and two patch sizes (64^3 and 128^3 cells per patch). Note for all patches sizes, individual patches were combined to a single patch when passed to hypre for execution using CUDA. This is done to allow Uintah to make performant use of hypre as recently demonstrated in [103].

Figure 7.4 shows strong scaling results for the modified Burns and Christon benchmark problem on a 2-level structured adaptive mesh refinement grid. Results were gathered using MPI+Kokkos::OpenMP+Kokkos::CUDA for three problem sizes (768^3, 1536^3, and 3072^3 cells on the fine mesh with 192^3, 384^3, and 768^3 cells on the coarse mesh, respectively) and one patch size (64^3 cells per patch). Note for all patches sizes, individual patches were combined to a single patch when passed to hypre for execution using CUDA. This is done

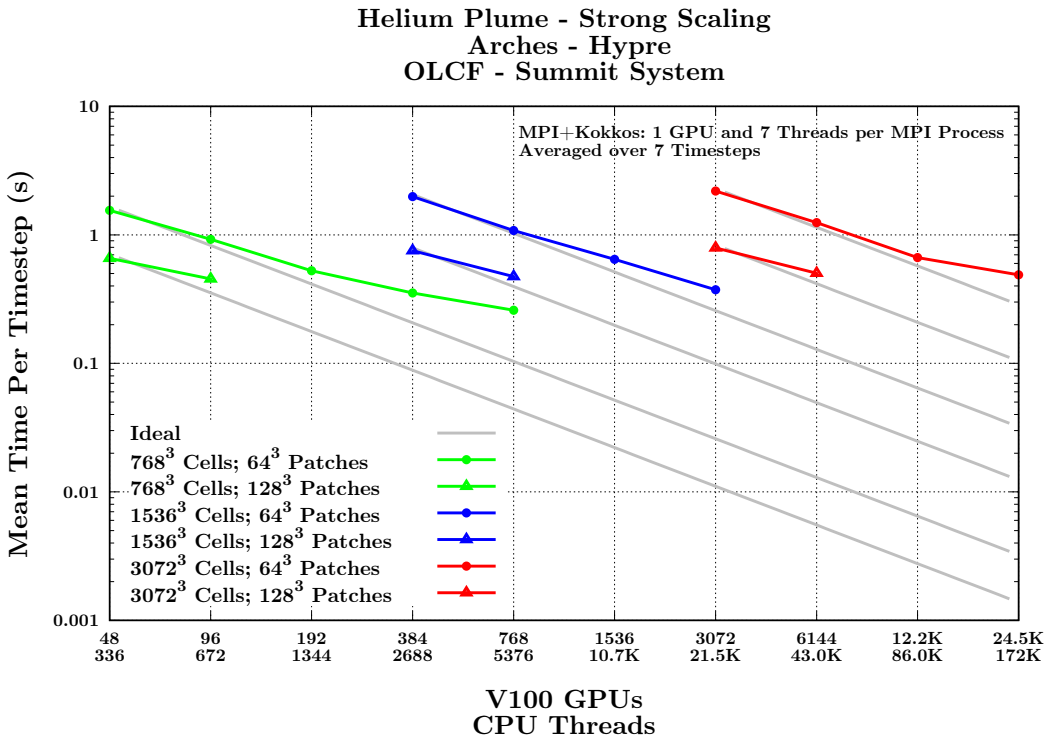

Fig. 7.3: Helium plume run to 24,576 V100 GPUs and 8,192 POWER9 processors.

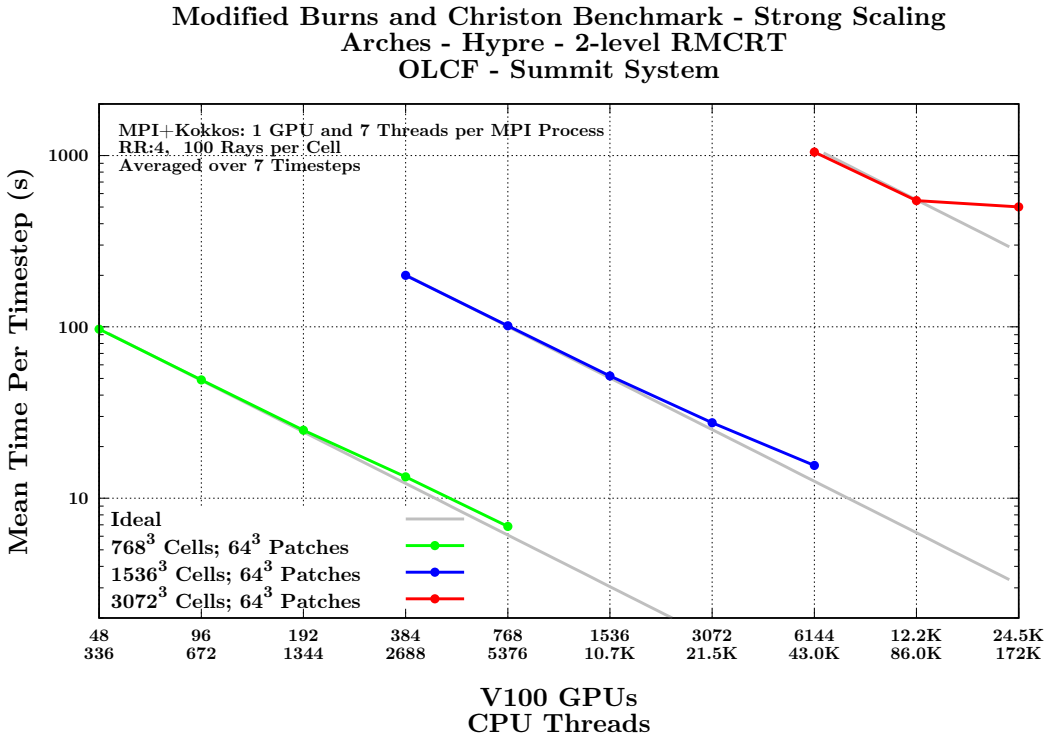

Fig. 7.4: Modified Burns and Christon benchmark run to 24,576 V100 GPUs and 8,192 POWER9 processors.

to allow Uintah to make performant use of hypre as recently demonstrated in [103].

7.4.3 Frontera Results

To demonstrate scalability of Uintah's heterogeneous MPI+Kokkos task scheduler, strong scaling studies were performed on the NSF Frontera system. This system features two Intel Cascade Lake processors with 28 cores per processor and 192 GB of DDR4 per node. For both problems, these studies explored varying domain decomposition approaches using problems sized similar to DOE Lassen runs. Note, additional demonstration of host-only capabilities related to this approach can be found in Chapter 5.

For MPI+Kokkos, simulations were launched using 1 MPI processes per node. Within an MPI process, 56 OpenMP threads were used to simultaneously launch and execute loops across: (1) 56 cores using 28 cores per loop for Kokkos::OpenMP. For hypre, both 14 endpoints with 4 threads per endpoint and 8 endpoints with 7 threads per endpoint were used to provide at least 1 patch per endpoint.

Problems were sized to provide each MPI process with at least 1 patch in all data points shown. Here, a patch refers to the collection of cells executed by a loop. Note, larger patch sizes result in fewer patches being available to distribute across MPI processes. Data points with fewer than 1 patch per MPI process were omitted from figures. Reported per-timestep timings measure the simultaneous execution of all loops across both the host and device in a given timestep. Results have been averaged over 7 consecutive timesteps. Additional details on problem setup and run configuration are discussed in individual result paragraphs.

Figure 7.5 shows strong scaling results for the helium plume problem on a single-level structured grid. Results were gathered using MPI+Kokkos::OpenMP for three problem sizes (256^3, 512^3, and 1024^3 cells) and one patch sizes (32^3 cells per patch). Note for all problem sizes, hypre was used with 14 endpoints and 4 threads per endpoint during initial scaling data points and with 8 endpoints and 7 threads per endpoint during final scaling data points to provide at least 1 patch per endpoint. This is done to allow Uintah to make performant use of hypre as recently demonstrated in [103].

Figure 7.6 shows strong scaling results for the modified Burns and Christon benchmark problem on a 2-level structured adaptive mesh refinement grid. Results were gathered

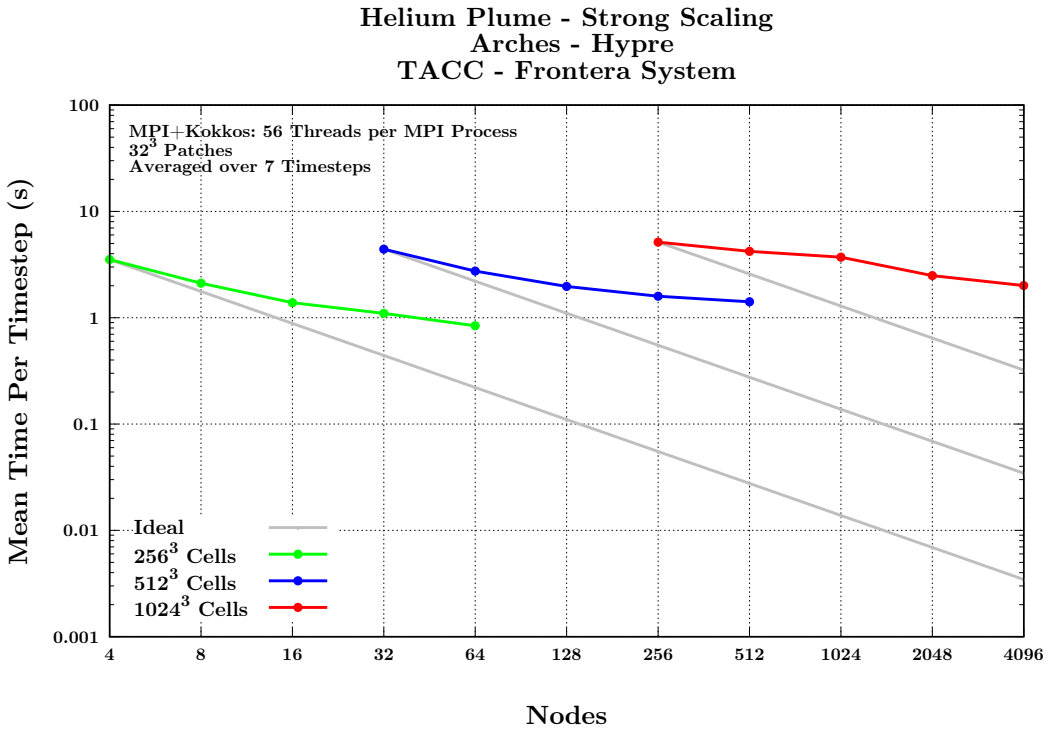

Fig. 7.5: Helium plume run to 8,192 Cascade Lake processors.

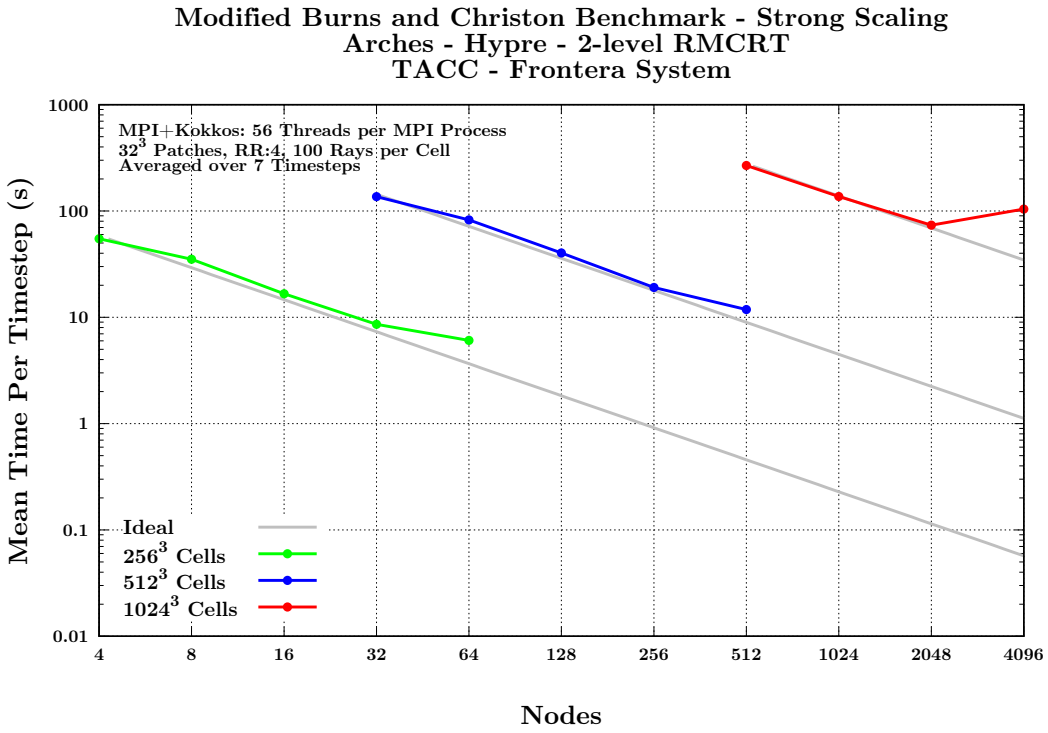

Fig. 7.6: Modified Burns and Christon benchmark run to 8,192 Cascade Lake processors.

using MPI+Kokkos::OpenMP for three problem sizes (256^3, 512^3, and 1024^3 cells on the fine mesh with 64^3, 128^3, and 256^3 cells on the coarse mesh, respectively) and one patch size (32^3 cells per patch). Note for all problem sizes, hypre was used with 14 endpoints and 4 threads per endpoint during initial scaling data points and with 8 endpoints and 7 threads per endpoint during final scaling data points to provide at least 1 patch per endpoint. This is done to allow Uintah to make performant use of hypre as recently demonstrated in [103].

7.4.4 Further Analysis

Results in Figure 7.1 through 7.6 show that good strong scaling to 24,576 GPUs and 8,192 processors is possible using MPI+Kokkos. This is encouraging as it suggests a potential for this heterogeneous MPI+Kokkos task scheduler to reduce the gap between development time and our ability to run on heterogeneous systems requiring other underlying programming models. This is advantageous for expediting Uintah's ability to support forthcoming exascale systems such as the Intel-based DOE Aurora and AMD-based DOE Frontier.

Results in Figure 7.2 show that for a compute-dominant problem, it is possible to achieve good strong scaling across multi-socket, multi-device nodes using an asynchronous many-task model. Results in Figure 7.1 show that for a communication-dominant problem, it can be difficult to achieve performance across multi-socket, multi-device nodes using an asynchronous many-task model. This is not unexpected given the additional overheads (i.e., for data movement) incurred between the host and device on such nodes. As shown in Figure 7.1, offloading fewer, yet larger, patches to the device can be used to improve node-level performance at the expense of reductions in strong scaling efficiency for communication-dominant problems. These results suggest that care must be taken when using an asynchronous many-task model on multi-socket, multi-device nodes. Though performance improvements are achievable when using the full node, performance reductions are also possible when overdecomposing a simulation domain too far.

Comparing 32^3 patch MPI-Only results in Figure 7.1 to 64^3 patch and 128^3 patch MPI+Kokkos results in Figure 7.1 shows that it is possible for Uintah application developers to improve node-level performance with relative ease using this scheduler and the accompanying portable abstractions [49] to port legacy serial loops to OpenMP and CUDA via Kokkos. This is encouraging as the wholesale refactoring of ARCHES loops to additionally

support the use of Kokkos::OpenMP and Kokkos::CUDA has been largely naive with ample opportunity to improve performance. The ease with which this has been achieved is attributed to Kokkos abstractions aligning well with Uintah's loop-based asynchronous many-task model. For ARCHES, this has spared application developers having to, for example, learn CUDA and write individual kernels for the triple-digit files comprising this simulation component. This is advantageous as it allows application developers to expedite scientific efforts.

7.5 Foreseable Challenges

The approach presented here is a starting point for achieving portable heterogeneous MPI+PPL task scheduling in an asynchronous many-task runtime system. Foreseeable challenges include understanding how to: (1) coordinate host-device data movement in a portable manner, (2) efficiently coordinate host-device data movement in the context of an asynchronous many-task model, (3) coordinate MPI in a simpler manner (e.g., using *MPI_THREAD_FUNNELED* to allow only the main thread to make MPI calls), (4) execute tasks using parallel third party libraries among tasks using a performance portability layer, and (5) automate the decision of when to use the host or device for portable tasks (e.g., to make informed use of the device).

7.6 Summary

This work has helped improve Uintah's portability to complex heterogeneous systems. Specifically, it has shown an approach for heterogeneous MPI+PPL task scheduling to help prepare an asynchronous many-task runtime system for the diverse heterogeneous systems accompanying exascale computing. This approach combines three individual solutions offering promise for making efficient use of the complex nodes anticipated in these systems: (1) asynchronous many-task runtime systems, (2) MPI+X hybrid parallelism approaches, and (3) performance portability layers. This is done with additional consideration for parallel third-party libraries facing similar challenges related to (2) and (3).

This approach has been demonstrated using a heterogeneous MPI+Kokkos task scheduler implemented in the Uintah Computational Framework, an asynchronous many-task runtime system, with additional consideration for hypre, a parallel third party library. Kokkos capabilities have been shown for two challenging problems using this scheduler

and the accompanying portable abstractions [49] to execute workloads representative of typical Uintah applications across multi-socket, multi-device nodes while making heterogeneous use of OpenMP and CUDA via Kokkos with a single source implementation. Performance improvements up to 4.4x have been achieved when using this scheduler and the accompanying portable abstractions [49] to port a previously MPI-Only problem to Kokkos::OpenMP and Kokkos::CUDA to improve complex heterogeneous node use. At scale, good strong-scaling to 24,576 NVIDIA V100 GPUs and 8,192 IBM POWER9 processors has been achieved using MPI+Kokkos::OpenMP+Kokkos::CUDA.

The portability and performance improvements shown here offer encouragement as we prepare Uintah for the diverse heterogeneous systems accompanying exascale computing. Next steps include extending Uintah's intermediate portability layer [49] to support Kokkos' default host and device spaces to make quicker use of underlying programming models. For Uintah's Aurora Early Science Program efforts, this will improve the speed with which we can support Kokkos::OpenMPTarget for running on the Intel GPUs anticipated in the DOE Aurora system. For Uintah's emphasis on maintaining broad support for major HPC systems, this will allow the speed with which we can implement, refine, and extend Uintah's more formal support for future major HPC systems. As a part of this, emphasis will be placed on generalizing CUDA-specific code used in Uintah's heterogeneous MPI+Kokkos task scheduler to achieve more portable heterogeneous MPI+Kokkos task scheduling.

CHAPTER 8

LESSONS LEARNED

8.1 Overview

This chapter collects lessons learned while designing the heterogeneous MPI+PPL task scheduling approach for asynchronous many-task runtime systems [51] (Chapter 7) and the indirect performance portability layer adoption approach for large legacy codes [49] (Chapter 6). As a whole, these lessons offer insights into opportunities to ease implementation of both the task scheduler [51] and the accompanying portable abstractions [49] making scalability demonstrations in Chapter 7 possible. These lessons are informed by the CCMSC's multi-year Kokkos adoption effort adding portable support for Kokkos::OpenMP and Kokkos::CUDA in a complex real-world application and representative asynchronous many-task runtime system, Uintah. This ongoing effort has been non-trivial due to the codebase:

1. consisting of 1-2 million lines of complex code,
2. maintaining a divide between application code, where framework-specific abstractions are needed, and infrastructure code, where interface-specific abstractions are implemented,
3. having hundreds of pre-existing loops to port in application code,
4. being under active development with many contributors, and
5. having a pre-existing userbase to support.

An example of the non-trivial nature of the codebase can be seen in Uintah's input-driven variable creation. Global variables shared across a simulation domain are initially specified by name in an XML-based input file by the application developer. At run-time, input files are parsed to obtain variable names, which are commonly assigned to a string-based

temporary variable in application code. These string-based temporary variables are then used to register the global variable with the runtime system. Once registered, these same strings are then used to access the associated data in application code and commonly assigned to a temporary variable of the associated data type.

Figure 8.1 shows an example of how variable names may differ in application code. Here, the runtime system knows this variable as "volFraction" (e.g., for debugging output). However, application code uses this variable as "vol_fraction" (e.g., for calculations). Such naming conventions are problematic as they make it difficult to debug and isolate issues to a given file or line of code. This is a result of the variant known to the runtime system being searchable among inputs alone, while the variant known to application code may be used throughout countless tasks or files.

The primary lesson learned is that performance portability is difficult and non-trivial for existing asynchronous many-task runtime systems. This is a result of the many-task model inherently having data dependencies and tasks in abundance with potentially long and complex data dependency sequences, which, for example, greatly complicate debugging. While porting may appear straightforward, the combinatorially increasing data dependency sequences to consider as task graphs grow and flexibility in where and how tasks may execute make it easy to introduce unanticipated changes for a given back-end(s) and requires great attention to detail when porting to ensure correctness across all back-ends. Performance portability is further complicated for runtimes with (1) existing application code, where application developers may have an abundance of over-engineered and/or non-portable code, and (2) problems that execute different subsets of partially overlapping portable loops, where fixes to one problem may break another problem.

High-level lessons learned include:

- Collaborate Closely: Maintaining a divide between application code and infrastructure code with separate developers for each makes close collaboration critical for improving productivity. Porting efforts related to this research were slowed by, for

```
std::string m_volFraction_name = "volFraction";
register_variable(m_volFraction_name, ...);
auto vol_fraction = tsk_info->getField<...>(m_volFraction_name);
```

Fig. 8.1: Multiple naming conventions for a single shared variable, volFraction.

example, application developers over-engineering solutions in application code that posed barriers to portability.

- Favor Simplicity: It is difficult to support a wide range of underlying programming models with a single source implementation. Porting efforts related to this research were slowed by, for example, having to replace unnecessarily complex data structures with plain-old-data variants in application code.

- Offer Flexibility: Decomposing problems into a collection of tasks results in a wide variety of performance characteristics across tasks. Performance improvements related to this research were more easily achieved by, for example, using the flexibility offered in where and how tasks are run to improve node use for individual problems.

- Standardize Code: The interconnectedness of data dependencies and tasks when using a many-task model makes standardized code (e.g., formatting, naming conventions, whitespace) critical for improving productivity. Far-reaching changes and debugging efforts related to this research were slowed by, for example, inconsistent naming conventions used across different files.

- Test Often: It is easy to introduce unanticipated changes in portable code due to emphasis on executing a single source implementation in many different ways. Early efforts related to this research were made unnecessarily complicated by, for example, allowing application developers to port code without proper test coverage.

- Work Incrementally: Decomposing problems into a collection of tasks results in a wide variety of ways in which individual tasks can be combined to produce individual problems. Porting efforts related to this research were slowed by, for example, starting to port code using unnecessarily complex problems executing large numbers of tasks.

The sections to follow capture detailed loop-level, application-level, build-level, and general lessons learned resulting from this effort.

8.2 Loop-Level Lessons

8.2.1 Portable Code Inside of Portable Abstractions

A key benefit of performance portability layers is their ability to execute a single source implementation in many different ways. This, however, is not guaranteed by adopting a portable abstraction itself. This is complicated by underlying programming models supporting code to different extents (e.g., convenience mechanisms). Understanding what can and cannot be done in portable loops is helpful for ensuring successful compilation and execution across multiple underlying programming models. This is eased by keeping code in portable loops as simple as possible.

Examples of changes needed in pre-existing loops to make portable use of OpenMP and CUDA via Kokkos in Uintah include:

1. eliminating the use of C++ standard library classes and functions that do not have CUDA equivalents,
2. copying object data members into local variables to pass them into a portable loop, and
3. eliminating allocation of host memory in portable loops.

8.2.2 Implementation of Portable Loops

Portable loop abstractions bring with them implementation challenges independent of whether they are adopted directly or indirectly. This is a result of great flexibility in where and how execution and memory are managed. This is complicated by pre-existing serial loops where parallel execution and thread safety need not be accounted for. Thinking through the implementation and execution of portable loops is important for improving loop-level performance and scalability.

Implementation and execution details found helpful when adopting Kokkos in Uintah include:

1. ensuring that portable loops are written in a thread-safe manner,
2. ensuring that portable loops are provided with enough work items to iterate over in parallel (e.g., at least as many work items as there are OpenMP threads),
3. using lambdas instead of functors (e.g., to avoid duplication of long parameter lists),

4. considering how to structure portable loops (e.g., 1D, 3D, etc.),
5. considering how portable loops iterate over work items (e.g., individually or in groups),
6. considering how portable loops utilize underlying hardware (e.g., cores, caches, etc.),
7. exploring configurability of underlying programming models (e.g., OpenMP loop scheduling parameters), and
8. adding run-time parameters to manage execution (e.g., OpenMP threads per loop, CUDA blocks per loop, etc.).

For (1), tools such as Archer [7], Intel Inspector, and ThreadSanitizer [105] are helpful for identifying data races.

8.3 Application-Level Lessons

8.3.1 Portable Code Outside of Portable Abstractions

A key benefit of performance portability layers is their ability to reduce the amount of duplicated code in an application. This, however, applies only to the portable abstractions adopted. In practice, application code extends beyond portable abstractions (e.g., in large legacy codebases). Looking for opportunities to apply portable techniques used by performance portability layers elsewhere in application code is important for improving long-term portability and code maintainability.

Examples encountered when adopting Kokkos in Uintah include using behind-the-scenes preprocessor macros and template metaprogramming to add portable support for:

1. arbitrary tags to manage interfaces to underlying programming models (e.g., for selective compilation of loops),
2. arbitrary execution spaces to manage loop execution schemes,
3. arbitrary memory spaces to manage data structures, and
4. an object to pass interface-specific needs into portable loops (e.g., CUDA streams).

8.3.2 Portable Tools for Application Code

Commonly used tools (e.g., C++ standard library convenience mechanisms) pose portability challenges when using multiple underlying programming models. This is a result of underlying programming models supporting such tools to different extents. This is complicated by pre-existing loops using non-portable tools (e.g., in large legacy codebases). Thinking through which portable tools to support before the far-reaching adoption of a performance portability layer is important for avoiding unexpected refactors.

Portable tools found helpful when adopting Kokkos in Uintah include portable:

1. vector containers,
2. synchronization mechanisms,
3. random number generation, and
4. mechanisms for simultaneously executing portable loops.

In Kokkos, portable options for (1), (3), and (4) include *Kokkos::Vector*, *Kokkos_Random*, and *Kokkos::OpenMP::partition_master*, respectively. For (2), a Uintah-specific abstraction based on *Kokkos::Experimental::MasterLock* was implemented to avoid mixing use of *std::mutex* and *omp_lock_t*. Details on Uintah's use of *Kokkos::OpenMP::partition_master* can be found in a recent technical report [50].

8.3.3 Support for Tasks Using Third Party Libraries

Third-party libraries pose interoperability challenges when used in a parallel codebase. This is a result of each supporting parallelism in potentially different capacities (e.g., MPI-Only vs. MPI+X hybrid parallelism). This is complicated by differing rates of development and preferred models of execution (e.g., MPI+X to improve node use for a global, all-to-all algorithm). Thinking through how to accommodate tasks using third-party libraries is important for avoiding performance and thread-safety issues.

Examples found helpful when adopting a heterogeneous MPI+PPL task scheduling approach in Uintah include:

1. implementing individual interfaces to underlying programming models used by third-party libraries,

2. modifying third-party libraries to more closely align parallelism approaches,
3. separating execution of tasks using third-party libraries from other tasks, and
4. using MPI endpoints [29] to accommodate third-party libraries performing best with an MPI-only parallelism approach.

8.3.4 Granularity of Host-Device Data Movement

A key benefit of the asynchronous many-task model is the adaptive execution of tasks. Limitless adaptive execution, however, does not guarantee performance on heterogeneous systems where data movement between the host and device must be considered. This is complicated by lightweight tasks whose running time may not justify associated overheads. Thinking through mechanisms for managing the granularity of data movement is helpful for ensuring performance across broad applications.

Examples found helpful when adopting a heterogeneous MPI+PPL task scheduling approach in Uintah include supporting the ability to:

1. group individual patches into a single larger patch,
2. group individual tasks into a single larger task,
3. specify the number of task executors used on the host to launch kernels on the device, and
4. restrict portable tasks from being able to execute on the device (i.e., leaving a portable task on the host to avoid host-device data movement altogether).

8.3.5 Coordination of Host-Device Data Movement

Heterogeneous systems pose thread-safety challenges when using both the host and the device. This is a result of data associated with an individual task being able to reside on (1) the host, (2) the device, or (3) both the host and the device. This is complicated by the asynchronous many-task model, where data dependencies and tasks are in abundance with potentially long and complex data dependency sequences. Looking for opportunities to ease the coordination of data movement (e.g., identifying where data resides, whether data is ready to use, etc.) is important for avoiding data integrity issues.

Examples found helpful when adopting a heterogeneous MPI+PPL task scheduling approach in Uintah include supporting the ability to:

1. using bit sets and boolean logic to monitor data movement status (e.g., to identify when data is valid on the host, invalid on the device, awaiting halo cells, etc. and similar to a cache coherence protocol),
2. allocating memory to accommodate the largest number of halo cells used across tasks for a given variable,
3. restricting the use of underlying programming models to one per task (i.e., to avoid data residing in multiple locations for tasks using multiple parallel patterns), and
4. enforcing continuous use of either the host or device for a given task.

8.4 Build-Level Lessons

8.4.1 Support for Multiple Build Configurations

Adoption of multiple underlying programming models requires careful consideration of new build configurations and paths of execution. This is complicated by heterogeneous HPC systems requiring the simultaneous use of multiple underlying programming models (e.g., OpenMP and CUDA) to fully utilize a heterogeneous compute node. Thinking through how to manage current and future build configurations before the far-reaching adoption of a performance portability layer is important for avoiding unexpected refactors.

Recurring paths of execution encountered when adopting Kokkos in Uintah include:

1. code needed for the underlying programming model independent of the performance portability layer (e.g., OpenMP locks),
2. code needed for the performance portability layer independent of the underlying programming model(s) (e.g., Kokkos Views), and
3. code needed for the performance portability layer dependent upon the underlying programming model(s) (e.g., Uintah-specific abstractions for Kokkos::OpenMP).

Consistent use of standardized preprocessor macros simplifies the management of such paths. An example of a macro definition for (1) is *HAVE_<BACK-END>*, which is

defined when the application picks up the underlying programming model. An example of a macro definition for (2) is *HAVE_<PPL>*, which is defined when the application picks up the performance portability layer. An example of a macro definition for (3) is *<APP>_ENABLE_<PPL>_<BACK-END(S)>*, which is defined when both the application and the performance portability layer pick up the underlying programming model(s). Note, preprocessor macros helpful for identifying when Kokkos itself picks up the underlying programming model(s) can be found in *kokkos/core/src/Kokkos_Macros.hpp*.

When using multiple underlying programming models, preprocessor macro logic to support (3) becomes complicated quickly. This posed unexpected challenges requiring additional refactors when adding support for Kokkos::OpenMP and Kokkos::CUDA in the same Uintah build. The use of preprocessor macros explicitly identifying code specific to such builds (e.g., *UINTAH_ENABLE_KOKKOS_OPENMP_CUDA*) is helpful for simplifying logic and readability. Note, heterogeneous builds can be simplified using the *nvcc_wrapper* Linux shell script found on the Kokkos GitHub.

8.4.2 Selective Compilation of Portable Loops

The use of portable abstractions across multiple loops poses challenges when adding support for additional underlying programming models. This is a result of every portable loop having to properly support the newly adopted programming model to avoid breaking builds. It is not feasible, however, to refactor all loops at once (e.g., to remove non-portable code) after portable abstractions have been widely adopted throughout a codebase.

This challenge has been addressed using a tagging system that allows application developers to individually identify each loop's supported interfaces. These tags are used to ensure that loops are compiled for only the respective underlying programming models that are supported to avoid breaking builds. This allows for incremental refactoring on a loop-by-loop basis when adding support for additional programming models, eliminating the need to refactor all loops at once. This approach also simplifies the isolation of problematic code by allowing loops to be easily enabled/disabled across programming models when debugging.

Such a tagging system has been implemented in Uintah using preprocessor macros and template metaprogramming. At compile-time, a portable loop is compiled for all interfaces

identified as being currently supported by application developers using macro-based tags (e.g., KOKKOS_OPENMP_TAG). Behind-the-scenes, provided tags are mapped to their respective underlying execution and memory spaces (e.g., Kokkos::OpenMP and Kokkos::HostSpace). At run-time, the portable loop is executed by one of the supported underlying programming models based upon build-specific paths of execution and Uintah-specific template parameters (e.g., ExecSpace and MemSpace). Uintah-specific template parameters extend those used by Kokkos to allow for other non-Kokkos execution spaces and memory spaces to be supported (e.g., Uintah::Legacy and Uintah::HostSpace to preserve legacy code). This approach is used to improve the long-term portability of execution and memory spaces for future interfaces and underlying programming models.

8.5 General Lessons

8.5.1 Far-Reaching Test Coverage and Regular Testing

Far-reaching test coverage and regular testing of a codebase is critical for easing the adoption of a performance portability layer. This is a result of it being inherently easy to introduce unanticipated changes in portable code due to emphasis on executing a single source implementation in many different ways. This is complicated by combinatorially increasing scenarios to consider for newly introduced underlying programming models, build configurations, and run configurations. Test scenarios found helpful when adopting Kokkos in Uintah include testing:

1. each underlying programming model,
2. relevant combinations of underlying programming models,
3. each build configuration,
4. serial execution of portable loops, and
5. parallel execution of portable loops.

Such testing is crucial for codebases where multiple tests execute different subsets of partially overlapping portable loops, which makes it easier to introduce unanticipated changes.

8.5.2 Intermediate Portability Layers

Portability layers require a significant investment from large legacy codebases. This is a result of such codebases having existing code to maintain and users to support. Intermediate portability layers ease this investment by allowing legacy code to be preserved, underlying portability layers (e.g., Kokkos, RAJA) to be centrally managed, and adoption of an underlying portability layer to take place incrementally. An example of an intermediate portability layer can be found in Chapter 6.

8.5.3 Standardization of Adopted Portability Layers

The far-reaching adoption of a portability layer poses code maintainability and debugging challenges. This is a result of portable loops relying upon each other for successful compilation and execution. Standardization of newly adopted portability layers eases these challenges by improving searchability to simplify far-reaching changes (e.g., to add support for a new interface) and debugging (e.g., to quickly identify all code using a given interface). An example of standardization applied to Uintah's intermediate portability layer includes consistent formatting, naming conventions, and whitespace.

8.5.4 Living Best Practices

Pre-existing loops pose refactoring challenges when incrementally adopting a performance portability layer in a large legacy codebase. This is a result of not knowing both what and how much non-portable code is used in loops before adoption. This is complicated by loops having different barriers to portability. Maintaining a living document collecting best practices and portability barriers from past refactors helps simplify refactors moving forward.

8.5.5 Incremental Case Studies

Carefully selected case studies are helpful for evaluating a performance portability layer before far-reaching adoption and significant investment. Two types of case studies used to ease the adoption of Kokkos in Uintah include those examining:

1. the most complex code and

2. simple representative code.

Case studies using (1) identified challenges quickly and informed best practices. Examples of (1) include loops with complicated nesting hierarchies, extensive use of C/C++ functionality, complex data structures, etc. Case studies using (2) evaluated the representative performance of more typical portable loops and refined best practices. Examples of (2) include loops with recurring patterns throughout the codebase, simple math, etc. Results from past Uintah case studies can be found in a technical report [50] and other publications [48,49,51,93,94,113].

8.5.6 Run Configuration Parameters

The asynchronous many-task model requires careful consideration of run configuration parameters. This is a result of applications being decomposed into a number of individual tasks that is typically much larger than the number of compute resources (e.g., cores). This is complicated by combinatorially increasing configurations to consider as MPI processes span more hardware and individual tasks with different arithmetic intensities, running times, and scalability. Thinking through where and how to manage run configuration parameters is important for ease of use and code maintainability. An example found helpful when adopting a heterogeneous MPI+PPL task scheduling approach in Uintah has been to apply a global run configuration across tasks based on user-defined run configuration parameters with defaults applied when not provided.

8.5.7 Thread Scalability

A key benefit of adding loop-level parallelism is the ability to make cooperative use of compute resources (e.g., cores, caches, etc.). Limitless scaling, however, does not guarantee performance as thread scalability must be considered. This is complicated by loops having different scaling characteristics. Thinking through how many threads to use per loop and how many loops to execute simultaneously is important for improving performance. An example found helpful when adopting a heterogeneous MPI+PPL task scheduling approach in Uintah has been to explore performance of all available run configurations to identify the optimal run configuration for a given problem.

8.5.8 Debugging Output

Debugging output is critical for improving productivity when implementing portable heterogeneous tasks. This is a result of the inherent interconnectedness of data dependencies and tasks when using an asynchronous many-task model. This is complicated by combinatorially increasing data dependency sequences to consider as task graphs grow and flexibility in where and how tasks may execute. Offering debugging output to monitor the flow of data movement and task execution is helpful for improving productivity. Examples found helpful when adopting a heterogeneous MPI+PPL task scheduling approach in Uintah include output identifying:

1. the underlying programming model used by each task,
2. variables computed, modified, and/or required by each task,
3. halo cell and neighboring data requirements for each variable,
4. data movement status for each task, and
5. task execution order.

8.5.9 Differing Rates of Development

Use of multiple third-party libraries poses development challenges when codes are under active development. This is a result of each moving at different rates of development. This is complicated by each having different resource constraints. Thinking through how to maintain pace across codes is important for improving productivity. Examples found helpful when adopting a heterogeneous MPI+PPL task scheduling approach in Uintah include:

1. avoiding custom patches,
2. merging code regularly,
3. using pull requests, and
4. collaborating closely.

8.6 Summary

These lessons learned have helped ease the adoption of a heterogeneous MPI+PPL task scheduling approach in Uintah [51] (Chapter 7) and ease indirect adoption of a performance portability layer in Uintah [49] (Chapter 6). As a whole, these lessons offer insights into opportunities to ease implementation of both the task scheduler [51] and the accompanying portable abstractions [49] making scalability demonstrations in Chapter 7 possible. These lessons are informed by the CCMSC's multi-year Kokkos adoption effort adding portable support for Kokkos::OpenMP and Kokkos::CUDA in a complex real-world application and representative asynchronous many-task runtime system, Uintah.

While many are general software engineering best practices, use of an intermediate portable layer is becoming common practice when adopting a performance portability layer. Uintah's indirect adoption of Kokkos through such a layer has made it possible to (1) preserve legacy code for existing users, (2) reduce reliance on Kokkos for new underlying programming models, and (3) further simplify portable abstractions for non-CS application developers. Similar approaches are taken by AMReX [122–124] and HPX [26, 45, 62, 63]. For HPX, indirect adoption is also used to extend Kokkos parallel pattern functionality (e.g., to return futures).

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

This dissertation has helped improve Uintah's node use, legacy code support, and long-term portability to complex heterogeneous systems. Specifically, it has helped address challenges relating to the complexity of nodes anticipated in exascale systems. These challenges include understanding how to manage the increased concurrency, deep memory hierarchies, and heterogeneity to make efficient use of such nodes, as well as understanding how to manage the increasing architectural diversity with systems such as the DOE Aurora [5] and DOE Frontier [84] to include Intel- and AMD-based GPUs, respectively. Two promising solutions were explored for addressing these challenges: (1) asynchronous many-task runtime systems and (2) performance portability layers.

This research aimed to demonstrate how a performance portability layer can be adopted in a large asynchronous many task runtime system to achieve scalable performance portability for large-scale simulations on current and emerging HPC systems featuring diverse microarchitectures. This aim has been achieved by indirectly adopting Kokkos, a performance portability layer, in Uintah, a representative asynchronous many-task runtime system, and extending Uintah's heterogeneous MPI+X task scheduling capabilities to support heterogeneous MPI+Kokkos task scheduling using Kokkos::OpenMP and Kokkos::CUDA on the host and device, respectively. This dissertation shows that it is possible to combine these promising solutions for exascale computing in a scalable manner for real-world applications with good strong scaling achieved across Intel Knights Landing- and Intel Cascade Lake-based many-core systems and IBM POWER9 and NVIDIA Volta-based heterogeneous systems. The portability and performance improvements shown throughout this dissertation offer encouragement as we prepare Uintah for the diverse heterogeneous systems accompanying exascale computing, specifically for DOE Aurora runs through the Aurora Early Science Program.

For CCMSC goals, this dissertation's research marks notable progress towards the end goal of an exascale-capable runtime system. This has been made possible through incremental improvements to Uintah's MPI+X task scheduling approach. Paired with the use of a performance portability layer, these advancements lay the foundation for a wholly portable task scheduling approach allowing one to use exascale systems when they arrive. It must be noted that this progress is the result of several collaborative efforts among the CCMSC, most notably with Brad Peterson and Damodar Sahsrabudhe.

Contributions and future work are summarized in subsequent sections.

9.1 Uintah's MPI+PThreads Task Scheduling Approach

Chapter 4 examined Intel Xeon Phi performance in the context of Uintah's MPI+PThreads task scheduling approach. This evaluation explored several thread placement strategies with special attention to the Intel Knights Corner's 61^{st} core. Single-node experiments have shown that many-core architectures such as the Intel Knights Corner require greater attention to run configuration and domain decomposition as demonstrated by 10.1% performance differences across run configurations on Intel Sandy Bridge compared to performance differences up to 149.3% on Knights Corner [47]. Multi-node experiments have shown that Uintah's MPI+PThreads task scheduling approach's need to decompose a simulation domain into more, yet smaller, patches to support additional threads is not conducive to scalability, especially when problem sizes are limited by the Knights Corner memory footprint.

9.2 An MPI+Kokkos::OpenMP Task Scheduling Approach

Chapter 5 described two approaches taken to portably address domain decomposition challenges related to Uintah's use of serial tasks. The first approach implemented a task scheduler enabling serial execution of data-parallel tasks within an MPI process. This implementation achieved good strong scaling characteristics to 65,536 threads across 256 Knights Landing processors with node-level performance improvements up to $3.00x$ [48] for a key algorithm refactored to use Kokkos. Though this approach addressed domain decomposition challenges, it also identified thread scalability challenges related to serial

execution of portable Kokkos-based data-parallel tasks within an MPI process.

The second approach implemented a task scheduler enabling parallel execution of data-parallel tasks within an MPI process. This implementation achieved good strong scaling characteristics to 442,368 threads across 1,728 Knights Landing processors with performance improvements up to $1.62x$ demonstrated at scale and little overhead added ($< 0.2\%$ per timestep) [49] for a key algorithm refactored to use Kokkos. This approach addressed thread scalability challenges by allowing fewer threads to be used per task for cases when tasks struggle to scale across large core counts. Ultimately, this approach formed the foundation for the OpenMP-based host-side capabilities of the heterogeneous MPI+Kokkos task scheduling approach in Chapter 7.

9.3 An Approach for Indirectly Adopting Kokkos

Chapter 6 described an approach for indirectly adopting a performance portability layer to help improve legacy code support and long-term portability in a large legacy codebase. The goal of this intermediate layer is for application developers to, hopefully, need only adopt the layer once to support current and future interfaces to underlying programming models.

For application developers, this layer allows for easy adoption of underlying programming models without requiring knowledge of low-level implementation details. For infrastructure developers, this layer allows for easy addition, removal, and tuning of interfaces behind the scenes in a single location, reducing the need for far-reaching changes across application code. This layer has made possible node-level performance improvements up to $2.63x$ in Uintah when porting key loops to the associated parallel patterns to more efficiently using a node [49].

9.4 A Heterogeneous MPI+Kokkos Task Scheduling Approch

Chapter 7 described an approach taken to portably address Uintah's heterogeneous task execution needs. This approach implemented a task scheduler enabling simultaneous use of Kokkos on both host and device. The resulting implementation achieved good strong scaling characteristics to 8,192 IBM POWER9 processors and 24,576 NVIDIA V100 GPUs with performance improvements up to $4.4x$ when using this scheduler and the

accompanying portable abstractions to port a previously MPI-Only problem to use both host and device [51]. Completion of this scheduler in collaboration with Damodar Sahasrabudhe marks a significant step forward for the CCSMC as a whole for the many individual efforts through the years that have worked towards this exascale computing goal.

9.5 Lessons Learned

Chapter 8 provided lessons learned that have helped ease the adoption of a heterogeneous MPI+PPL task scheduling approach in Uintah [51] (Chapter 7) and ease indirect adoption of a performance portability layer in Uintah [49] (Chapter 6). As a whole, these lessons offer insights into opportunities to ease implementation of both the task scheduler [51] and the accompanying portable abstractions [49] making scalability demonstrations in Chapter 7 possible. These lessons are the result of careful reflection on Uintah's Kokkos adoption effort as a whole.

9.6 Future Work

9.6.1 Generic Host and Device Support

Uintah's intermediate portability layer is limited to the use of Kokkos::OpenMP and Kokkos::CUDA back-ends for the host and device, respectively. While this eased rapid development, hard-coded use of Kokkos back-ends hindered Uintah's long-term portability to exascale systems requiring different back-ends for the device (e.g., Kokkos::OpenMPTarget). This hard coding was unavoidable due to the complexity of Uintah's infrastructure, use of raw CUDA for GPU-specific paths of execution, and maturity of Kokkos at the time of early adoption.

This proposed direction suggests extending Uintah's intermediate portability layer to additionally support Kokkos' default host and device execution spaces. This may be achievable by extending Uintah's task tagging system to support two new tags for default spaces and porting both application code and infrastructure code using tags to support the newly added tags. As a part of this, Uintah's support for multiple build configurations would also need to be extended to ensure that standardized preprocessor macros used throughout Uintah to manage back-end specific paths of execution are managed correctly. Such an extension has the potential to improve the speed with which Uintah can run on new systems requiring different underlying programming models (e.g., Kokkos::OpenMPTarget

for the Intel-based GPUs to appear in the DOE Aurora system).

The key challenge for this proposed direction will be understanding how to generalize Uintah's data structures and related infrastructure to support arbitrary memory spaces. This adds new complexity due to Uintah's many convenience mechanisms for managing data (e.g., specialized data types) and use of raw CUDA for GPU support. A risk associated with this direction is that mixing hard-coded use of back-ends and default back-ends makes specialized paths of execution unintuitive for the user (i.e., application developers). Accomplishing this proposed direction will pave the way for faster prototyping when new underlying programming models are available for use in Kokkos (e.g., to help stress test Kokkos itself).

9.6.2 Portable Heterogeneous MPI+PPL Task Scheduling

Uintah's heterogeneous MPI+Kokkos task scheduler is not yet wholly portable. While individual tasks themselves are portable, task executor logic used to schedule and execute tasks makes use of raw CUDA (e.g., cudaStream, cudaMemcpyAsync, and cudaStreamQuery). This use was unavoidable due to the functionality used and maturity of Kokkos at the time of early adoption.

This proposed direction suggests generalizing task executor logic in Uintah's heterogeneous MPI+Kokkos task scheduler to allow other back-ends to be used on the device (e.g., Kokkos::OpenMPTarget). This may be achievable using Kokkos instance functionality to replace the use of cudaStream objects. As a part of this, portable alternatives for initiating asynchronous host-to-device transfers and checking if a transfer is complete are also needed to replace use of cudaMemcpyAsync and cudaStreamQuery, respectively. Such generalization has the potential to improve the speed with which Uintah can run on new systems requiring different underlying programming models (e.g., Kokkos::OpenMPTarget for the Intel-based GPUs to appear in the DOE Aurora system).

The key challenge for this proposed direction will be identifying portable abstractions suitable for scheduling and executing portable tasks across a diverse set of underlying programming models. This adds new complexity for scenarios where underlying programming models offer unique abstractions not found in others. A risk associated with this proposed direction is that underlying programming models are too dissimilar and portable

abstractions are not feasible. Accomplishing this proposed direction will pave the way for defining AMT-related portable abstractions for task scheduling and execution (e.g., to help refine Kokkos' HPX functionality).

9.6.3 Improving Per-MPI Process Task Throughput

The MPI+Kokkos task scheduling approaches introduced by this work are limited to using the same number of threads for all tasks executed by a given problem. While this simplifies task scheduling, it lends itself to poor use of cores and threads when executing serial tasks (i.e., non-Kokkos) and/or parallel tasks (e.g., Kokkos::OpenMP) with loops that do not scale well across the designated number of threads. This inefficiency is unavoidable due to the diverse mix of tasks and loops within Uintah and its applications. For example, ARCHES has approximately 500 loops ranging from 1 line of code to approximately 800 lines of code with an average of 20 lines of code.

This proposed direction suggests extending the MPI+Kokkos task scheduling approach to support using either 1 thread or multiple threads to compute tasks executed by a given problem. This may be achievable by using Kokkos partitioning functionality recursively in task executors to sub-divide a group of threads designated for executing 1 task at a time with n threads per task into a group of threads designated for executing n tasks at a time with 1 thread per task. Such an extension has the potential to improve node use by avoiding idle threads within task executors when executing a serial task and reducing performance penalties when executing a parallel task with loops that do not scale.

The key challenge for this proposed direction will be understanding how to indicate OpenMP thread placement when recursively using Kokkos partitioning functionality. This adds new complexity when groups of threads may feature different levels of nesting and requires new task scheduling logic to differentiate and selectively execute tasks. A risk associated with this proposed direction is that the cost of additional scheduling logic may outweigh performance improvements offered by improved task throughput. Accomplishing this proposed direction will pave the way for more intricate thread group management approaches (e.g., dynamic sizing of thread groups based on thread scalability).

9.7 Summary

In summary, this dissertation demonstrates that the proposed approaches for improving asynchronous many-task runtime node use work well across the largest of HPC systems that we have access to (i.e., Summit). Though much work remains to enable wholly portable task scheduling, we know what needs to be done and have articulated a clear path forward among future work discussed here. As a part of this, a substantial yet well-understood refactor of the code is needed to support new back-ends (e.g., Kokkos::OpenMPTarget) for future exascale systems.

APPENDIX

PUBLICATIONS

This section provides a listing of all publications to date. Section A.1 includes publications central to this dissertation. Section A.2 includes other publications pursued alongside this dissertation.

A.1 Related Publications

1. **J.K. Holmen**. D. Sahasrabudhe. M. Berzins. "Porting Uintah to Heterogeneous Systems". Submitted. 2021.

2. **J.K. Holmen**. D. Sahasrabudhe. M. Berzins. "A Heterogeneous MPI+PPL Task Scheduling Approach for Asynchronous Many-Task Runtime Systems". In *Proceedings of the Practice & Experience in Advanced Research Computing (PEARC) Conference Series*. 2021. DOI: 10.1145/3437359.3465581.

3. **J.K. Holmen**, B. Peterson, M. Berzins. "An Approach for Indirectly Adopting a Performance Portability Layer in Large Legacy Codes". In *Proceedings of the International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 2019. DOI: 10.1109/P3HPC49587.2019.00009.

4. **J.K. Holmen**, B. Peterson, A. Humphrey, D. Sunderland, O.H. Diaz-Ibarra, J.N. Thornock, M. Berzins. "Portably Improving Uintah's Readiness for Exascale Systems Through the Use of Kokkos". Technical Report UUSCI-2019-001, SCI Institute. 2019.

5. B. Peterson, A. Humphrey, **J.K. Holmen**, T. Harman, M. Berzins, D. Sunderland, H.C. Edwards. "Demonstrating GPU Code Portability and Scalability for Radiative Heat Transfer Computations". In *Journal of Computational Science*. 2018. DOI: 10.1016/j.jocs.2018.06.005.

6. **J.K. Holmen**, A. Humphrey, D. Sunderland, M. Berzins. "Improving Uintah's Scalability Through the Use of Portable Kokkos-Based Data Parallel Tasks". In *Proceedings of the Practice & Experience in Advanced Research Computing (PEARC) Conference Series*. 2017. DOI: 10.1145/3093338.3093388. **Best Paper Nominee**.

7. **J.K. Holmen**, A. Humphrey, M. Berzins. "Exploring Use of the Reserved Core". In *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, J. Reinders, J. Jeffers, Ed. Morgan Kaufmann. 2015. DOI: 10.1016/b978-0-12-803819-2.00010-0.

8. B. Peterson, N. Xiao, **J.K. Holmen**, S. Chaganti, A. Pakki, J. Schmidt, D. Sunderland, A. Humphrey, M. Berzins. "Developing Uintah's Runtime System for Forthcoming Architectures". Refereed Paper for *the Workshop on Runtime Systems for Extreme Scale Programming Models and Architectures (RESPA)*. 2015.

A.2 Other Publications

1. **J.K. Holmen**, D. Sahasrabudhe, M. Berzins, A. Bardakoff, T. J. Blattner, W. Keyrouz. "Uintah+Hedgehog: Combining Parallelism Models for End-to-End Large-Scale Simulation Performance". *Refereed Paper for the Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. 2021.

2. Z.A. Bookey, **J.K. Holmen**, J.J. Hu. "Kokkos-Based Structured Grid Multigrid Solver". In *Center for Computing Research at Sandia National Laboratories Summer Proceedings*, Ed. J.B. Carleton, Ed. M.L. Parks. 2016.

REFERENCES

[1] "Kokkos C++ Performance Portability Programming EcoSystem: The programming model - parallel execution and memory abstraction," 2019, https://github.com/kokkos/kokkos.

[2] W. P. Adamczyk, B. Isaac, J. Parra-Alvarez, S. T. Smith, D. Harris, J. N. T. amd Minmin Zhoub, P. J. Smith, and R. Zmuda, "Application of LES-CFD for predicting pulverized-coal working conditions after installation of NOx control system," *Energy*, vol. 160, pp. 693–709, Oct. 2018.

[3] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. Kelly *et al.*, "Abstract machine models and proxy architectures for exascale computing," in *2014 Hardware-Softw. Co-Des. High Perform. Comput.*, Nov. 17, 2014, pp. 25–32.

[4] Argonne Leadership Computing Facility, "Theta," 2017, https://www.alcf.anl.gov/alcf-resources/theta.

[5] ——, "Aurora," 2019, https://aurora.alcf.anl.gov/.

[6] B. Ashbaugh, A. Bader, J. Brodman, J. Hammond, M. Kinsner, J. Pennycook, R. Schulz, and J. Sewall, "Data parallel C++ enhancing SYCL through extensions for productivity and performance," in *Proc. Int. Workshop OpenCL*, Apr. 27–29, 2020, pp. 1–2.

[7] S. Atzeni, G. Gopalakrishnan, Z. Rakamaric, D. H. Ahn, I. Laguna, M. Schulz, G. L. Lee, J. Protze, and M. S. Müller, "ARCHER: Effectively spotting data races in large openmp applications," in *2016 IEEE Int. Parallel Distrib. Proc. Symp. (IPDPS)*, May 23–27 2016, pp. 53–62.

[8] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier, "StarPU: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput.*, vol. 23, no. 2, pp. 187–198, Nov 2011.

[9] A. H. Baker, R. D. Falgout, T. Gamblin, T. V. Kolev, M. Schulz, and U. M. Yang, "Scaling algebraic multigrid solvers: On the road to exascale," in *Competence High Perform. Comput.* Jun. 22–24, 2010, pp. 215–226.

[10] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling hypre's multigrid solvers to 100,000 cores," in *High-Performance Scientific Computing: Algorithms and Applications*. M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, and F. Saied, Eds. London, UK: Springer, 2012, pp. 261–279.

[11] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc web page," 2021, https://petsc.org/.

[12] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Nov. 10–16 2012, p. 66.

[13] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM Int. Workshop Perform. Portability Productiv. HPC (P3HPC)*, Nov. 22 2019, pp. 71–81.

[14] J. Bennett, R. Clay, G. Baker, M. Gamell, D. Hollman, S. Knight, H. Kolla, G. Sjaardema, N. Slattengren, K. Teranishi, J. Wilke, M. Bettencourt, S. Bova, K. Franko, P. Lin, R. Grant, S. Hammond, S. Olivier, L. Kale, N. Jain, E. Mikida, A. Aiken, M. Bauer, W. Lee, E. Slaughter, S. Treichler, M. Berzins, T. Harman, A. Humphrey, J. Schmidt, D. Sunderland, P. McCormick, S. Gutierrez, M. Schulz, A. Bhatele, D. Boehme, P. Bremer, and T. Gamblin, "ASC ATDM level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms," Sandia National Laboratories, Albuquerque, NM, USA, Tech. Rep. SAND-2015-8312662088, Sep. 1, 2015.

[15] L. Bertagna, M. Deakin, O. Guba, D. Sunderland, A. M. Bradley, I. K. Tezaur, M. A. Taylor, and A. G. Salinger, "HOMMEXX 1.0: A performance-portable atmospheric dynamical core for the energy exascale earth system model," *Geosci. Model Develop.*, vol. 12, no. 4, pp. 1423–1441, Apr. 2019.

[16] M. Berzins, "Status of release of the Uintah Computational Framework," Scientific Computing and Imaging Institute, Salt Lake City, UT, USA Tech. Rep. UUSCI-2012-001, Apr. 15, 2012.

[17] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, and C. Wight, "Extending the Uintah framework through the petascale modeling of detonation in arrays of high explosive devices," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. 101–122, Oct. 2016.

[18] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. Wight, and J. Peterson, "Uintah: A scalable framework for hazard analysis," in *Proc. 2010 TeraGrid Conf.*, Aug. 2-5, 2010, p. 3.

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

[20] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Comput. Sci. Eng.*, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[21] M. Brachet, D. Meiron, S. Orszag, B. Nickel, R. Morf, and U. Frisch, "Small-scale structure of the Taylor-Green vortex," *J. Fluid Mech.*, vol. 130, no. 41, p. 1452, May 1983.

[22] S. Burns and M. Christon, "Spatial domain-based parallelism in large-scale, participating-media, radiative transport applications," *Numeri. Heat Transf. A Appl*, vol. 31, no. 4, pp. 401–421, Jun. 1997.

[23] A. Buss, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, "STAPL: Standard template adaptive parallel library," in *Proc. 3rd Annu. Haifa Exp. Syst. Conf.*, May 24-26 2010, p. 14.

[24] P. Czarnul, J. Proficz, and K. Drypczewski, "Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems," *Sci. Program.*, vol. 2020, 4176794, Jan. 2020.

[25] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan.-Mar. 1998.

[26] G. Daiß, M. Simberg, A. Reverdell, J. Biddiscombe, T. Pollinger, H. Kaiser, and D. Pflüger, "Beyond fork-join: Integration of performance portable Kokkos kernels with HPX," in *2021 IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, Jun. 17-21, 2021, pp. 377–386.

[27] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via BabelStream," *Int. J. Comput. Sci. Eng.*, vol. 17, no. 3, pp. 247–262, Oct. 2018.

[28] I. Demeshko, J. Watkins, I. K. Tezaur, O. Guba, W. F. Spotz, A. G. Salinger, R. P. Pawlowski, and M. A. Heroux, "Toward performance portability of the Albany finite element analysis code using the Kokkos library," *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 2, pp. 332–352, Mar. 2019.

[29] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling MPI interoperability through flexible communication endpoints," in *Proc. 20th Eur. MPI Users' Group Meeting*, Sep. 15-18, 2013, pp. 13–18.

[30] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, B. O'Shea, E. Schnetter, B. V. Straalen, and K. Weide, "A survey of high level frameworks in block-structured adaptive mesh refinement packages," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3217–3227, Dec. 2014.

[31] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202 – 3216, Dec. 2014.

[32] J. Eichstädt, M. Green, M. Turner, J. Peiró, and D. Moxey, "Accelerating high-order mesh optimisation with an architecture-independent programming model," *Comput. Phys. Commun.*, vol. 229, pp. 36 – 53, Aug. 2018.

[33] T. M. Evans, A. Siegel, E. W. Draeger, J. Deslippe, M. M. Francois, T. C. Germann, W. E. Hart, and D. F. Martin, "A survey of software implementations used by application codes in the exascale computing project," *Int. J. High Perform. Comput. Appl.*, Jan. 2022.

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

[35] R. Falgout, R. Li, B. Sjogreen, L. Wang, and U. Yang, "Porting hypre to heterogeneous computer architectures: Strategies and experiences," *Parallel Comput.*, vol. 108, p. 102840, Dec. 2021.

[36] M. A. Gallis, J. R. Torczynski, S. J. Plimpton, D. J. Rader, and T. Koehler, "Direct simulation Monte Carlo: The quest for speed," *AIP Conf. Proc.*, vol. 1628, no. 27, pp. 27–36, Feb. 2015.

[37] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, and J. Shalf, "The Cactus framework and toolkit: Design and applications," in *High Perform. Comput. Comput. Sci. (VECPAR 2002)*, J. M. L. M. Palma, A. A. Sousa, J. Dongarra, and V. Hernández, Eds., Jun. 26-28 2003, pp. 197–227.

[38] P. Grete, F. W. Glines, and B. W. O'Shea, "K-Athena: A performance portable structured grid finite volume magnetohydrodynamics code," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 1, pp. 89–97, May 2019.

[39] J. Guilkey, T. Harman, J. Luitjens, J. Schmidt, J. Thornock, J. D. de St. Germain, S. Shankar, J. Peterson, C. Brownlee, C. Reid, T. Saad, J. Beckvermit, A. Humphrey, and B. Peterson, "Uintah user guide," 2021, http://uintah-build.sci.utah.edu/trac/chrome/site/user_guide.pdf.

[40] J. E. Guilkey, T. Harman, A. Xia, B. Kashiwa, and P. McMurtry, "An Eulerian-Lagrangian approach for large deformation fluid structure interaction problems, Part 1: Algorithm development," *WIT Trans. Built Environ.*, vol. 71, May 2003.

[41] J. R. Hammond, M. Kinsner, and J. Brodman, "A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C++ applications," in *Proc. Int. Workshop OpenCL (IWOCL'19)*, May 13-15, 2019, pp. 15:1–15:2.

[42] J. R. Hammond and T. G. Mattson, "Evaluating data parallelism in C++ using the parallel research kernels," in *Proc. Int. Workshop OpenCL (IWOCL'19)*, May, 13-15 2019, pp. 14:1–14:6.

[43] T. Harman, J. E. Guilkey, B. Kashiwa, J. Schmidt, and P. McMurtry, "An Eulerian-Lagrangian approach for large deformation fluid structure interaction problems, Part 2: Multi-physics simulations within a modem computational framework," *WIT Trans. Built Environ.*, vol. 71, May 2003.

[44] M. Harris, "Hemi: Simpler, more portable CUDA C++," 2017, http://harrism.github.io/hemi/.

[45] T. Heller, P. Diehl, Z. Byerly, J. Biddiscombe, and H. Kaiser, "HPX–an open source C++ standard library for parallelism and concurrency," in *Proc. OpenSuCo*, vol. 5, Nov. 17, 2017.

[46] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, Sep. 2005.

[47] J. K. Holmen, A. Humphrey, and M. Berzins, "Exploring use of the reserved core," in *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches*, J. Reinders and J. Jeffers, Eds. Boston, MA, USA: Morgan Kaufmann, 2015, vol. 2, pp. 229 – 242.

[48] J. K. Holmen, A. Humphrey, D. Sunderland, and M. Berzins, "Improving Uintah's scalability through the use of portable Kokkos-based data parallel tasks," in *Proc. Pract. Exp. Adv. Res. Comput. 2017 Sustain. Success Impact*: *PEARC17*, Jul. 9-13 2017, pp. 27:1–27:8.

[49] J. K. Holmen, B. Peterson, and M. Berzins, "An approach for indirectly adopting a performance portability layer in large legacy codes," in *2019 IEEE/ACM Int. Workshop Perform. Portability Productiv. HPC (P3HPC)*, Nov. 22 2019, pp. 36–49.

[50] J. K. Holmen, B. Peterson, A. Humphrey, D. Sunderland, O. H. Diaz-Ibarra, J. N. Thornock, and M. Berzins, "Portably improving Uintah's readiness for exascale systems through the use of Kokkos," SCI Institute, Salt Lake, City, UT, USA, Tech. Rep. UUSCI-2019-001, May 17, 2019.

[51] J. K. Holmen, D. Sahasrabudhe, and M. Berzins, "A heterogeneous MPI+PPL task scheduling approach for asynchronous many-task runtime systems," in *Proc. Pract. and Experience in Adv. Res. Comput. 2021 Sustainability, Success and Impact (PEARC21)*, Jul 18-22 2021.

[52] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, USA, Tech. Rep., LLNL-TR-661403, Sep. 24, 2014.

[53] R. Hornung and D. Beckingsale, "ECP SP project 2.3.1.06-STPM08-RAJA," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, USA, Tech. Rep., Sep. 20, 2019.

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

[55] A. Humphrey, Q. Meng, M. Berzins, and T. Harman, "Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system," in *Proc. Conf. Extreme Sci. Eng. Disc. Environ. (XSEDE'12)*, Jul. 16-20 2012, pp. 1–8.

[56] A. Humphrey, D. Sunderland, T. Harman, and M. Berzins, "Radiative heat transfer calculation on 16384 GPUs using a reverse Monte Carlo ray tracing approach with adaptive mesh refinement," in *2016 IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 23-27, 2016, pp. 1222–1231.

[57] A. Humphrey, "Scalable asynchronous many-task runtime solutions to globally coupled problems," Ph.D. dissertation, School Comput., Univ. Utah, Salt Lake City, UT, USA, 2019.

[58] I. Hunsaker, "Parallel distributed, reciprocal Monte Carlo radiation in coupled, large eddy combustion simulations," Ph.D. dissertation, Dept. Chem. Eng., Univ. Utah, Salt Lake City, UT, USA, 2013.

[59] I. Hunsaker, T. Harman, J. Thornock, and P. Smith, "Efficient parallelization of RMCRT for large scale LES combustion simulations," in *20th AIAA Comput. Fluid Dyn. Conf.*, Jun. 27-30 2011, p. 3770.

[60] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Boston, MA, USA: Morgan Kaufmann Publishers Inc., 2013.

[61] A. Johnson, "Area exam: General-purpose performance portable programming models for productive exascale computing," University of Oregon, Eugene, OR, USA. Area Exam Report. Jun. 5-6, 2020.

[62] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proc. 8th Int. Conf. Partitioned Global Address Space Program. Models*, Oct. 6-10 2014, pp. 6:1–6:11.

[63] H. Kaiser, P. Diehl, A. S. Lemoine, B. A. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller *et al.*, "HPX-the C++ standard library for parallelism and concurrency," *J. Open Source Softw.*, vol. 5, no. 53, p. 2352, Sep. 2020.

[64] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," *ACM SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, Oct. 1993.

[65] B. Kashiwa and E. Gaffney, "Design basis for CFDLIB, Tech. Rep. LA-UR-03-1295," Los Alamos National Laboratory, Los Alamos, NM, USA, 2003.

[66] G. Krishnamoorthy, R. Rawat, and P. J. Smith, "Parallelization of the P-1 radiation model," *Numer. Heat Transf. B Fundam.*, vol. 49, no. 1, pp. 1–17, Jan. 2006.

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

[68] A. Kulkarni and A. Lumsdaine, "A comparative study of asynchronous many-tasking runtimes: Cilk, Charm++, ParalleX and AM++," 2019, *arXiv:1904.00518*.

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

[70] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Int. Symp. Code Gener. Optim. 2004: CGO 2004.* Mar. 20-24, 2004, pp. 75–86.

[71] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra *et al.*, "DOE Advanced Scientific Computing Advisory Subcommittee (ASCAC) report: Top ten exascale research challenges," USDOE Office of Science (SC)(United States), Washington, DC, USA, Tech. Rep., Feb. 10, 2014.

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

[73] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An evaluation of emerging many-core parallel programming models," in *Proc. 7th Int. Workshop Program. Models Appl. Multicores Manycores*, Mar. 12-16 2016, pp. 1–10.

[74] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Assessing the performance portability of modern parallel programming models using TeaLeaf," *Concurr. Comput. Pract. Exp.*, vol. 29, no. 15, p. e4117, Aug. 2017.

[75] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, and M. Lee, "The Open Community Runtime: A runtime system for extreme scale computing," in *2016 IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 13-15, 2016, pp. 1–7.

[76] D. S. Medina, A. St-Cyr, and T. Warburton, "OCCA: A unified approach to multi-threading languages," 2015, *arXiv:1403.0968*.

[77] Q. Meng and M. Berzins, "Scalable large-scale fluid-structure interaction solvers in the Uintah framework via hybrid task-based parallelism algorithms," *Concurr. Comput. Pract. Exp.*, vol. 26, no. 7, pp. 1388–1407, May 2014.

[78] Q. Meng, M. Berzins, and J. Schmidt, "Using hybrid parallelism to improve memory use in Uintah," in *Proc. TeraGrid 2011 Conf.*, Jul. 18-21, 2011, pp. 1–8.

[79] Q. Meng, A. Humphrey, and M. Berzins, "The Uintah framework: A unified heterogeneous task scheduling and runtime system," in *High Perform. Comput., Netw., Storage and Anal. (SCC), 2012 SC Companion:*, Nov 2012, pp. 2441–2448.

[80] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Investigating applications portability with the Uintah DAG-based runtime system on PetaScale supercomputers," in *Proc. SC13 Int. Conf. High Perform. Comput. Netw. Storage Anal.*, Nov. 17-21, 2013, pp. 96:1–96:12.

[81] ——, "Preliminary experiences with the Uintah framework on Intel Xeon Phi and Stampede," in *Proc. Conf. Extreme Sci. Eng. Disc. Environ. Gateway Disc. (XSEDE 2013)*, Jul. 22-25, 2013, pp. 48:1–48:8.

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

[83] National Energy Research Scientific Computing Center, "Cori," 2020, https://www.nersc.gov/systems/cori/.

[84] Oak Ridge Leadership Computing Facility, "Frontier," 2019, https://www.olcf.ornl.gov/frontier/.

[85] T. O'hern, E. Weckman, A. Gerhart, S. Tieszen, and R. Schefer, "Experimental study of a turbulent buoyant helium plume," *J. Fluid Mech.*, vol. 544, pp. 143–171, Dec. 2005.

[86] B. O'Shea, G. Bryan, J. Bordner, M. Norman, T. Abel, R. Harkness, and A. Kritsuk, "Introducing Enzo, an AMR cosmology application," in *Adaptive Mesh Refinement - Theory and Applications*, T. Plewa, T. Linde, and V. G. Weirs, Eds., vol. 41. Berlin, Germany: Springer-Verlag, 2005, pp. 341–350.

[87] S. Parker, "A component-based architecture for parallel multi-physics PDE simulation," *Future Gener. Comput. Syst.*, vol. 22, no. 1-2, pp. 204–216, Jan. 2006.

[88] J. Pedel, J. N. Thornock, S. T. Smith, and P. J. Smith, "Large eddy simulation of polydisperse particles in turbulent coaxial jets using the direct quadrature method of moments," *Int. J. Multiph. Flow*, vol. 63, pp. 23–38, Jul. 2014.

[89] B. Peterson, "Portable and performant GPU/heterogeneous asynchronous many-task runtime system," Ph.D. dissertation, School Comput., Univ. Utah, Salt Lake City, UT, USA, 2019.

[90] B. Peterson, H. Dasari, A. Humphrey, J. Sutherland, T. Saad, and M. Berzins, "Reducing overhead in the Uintah framework to support short-lived tasks on GPU-heterogeneous architectures," in *Proc. 5th Int. Workshop Domain-Specific Lang. High-Level Frameworks High Perform. Comput.*, Nov. 15, 2015, pp. 4:1–4:8.

[91] B. Peterson, A. Humphrey, J. Schmidt, and M. Berzins, "Addressing global data dependencies in heterogeneous asynchronous runtime systems on GPUs," in *Third Int. Workshop Extreme Scale Program. Models Middleware*, Nov. 12-17, 2017, pp. 1–8.

[92] B. Peterson, A. Humphrey, D. Sunderland, J. Sutherland, T. Saad, H. Dasari, and M. Berzins, "Automatic halo management for the Uintah GPU-heterogeneous asynchronous many-task runtime," *Int. J. Parallel Program.*, vol. 47, no. 5/6, pp. 1086–1116, Dec. 2018.

[93] B. Peterson, N. Xiao, J. K. Holmen, S. Chaganti, A. Pakki, J. Schmidt, D. Sunderland, A. Humphrey, and M. Berzins, "Developing Uintah's runtime system for forthcoming architectures," SCI Institute, Salt Lake City, UT, USA, Tech. Rep.

[94] B. Peterson, A. Humphrey, J. K. Holmen, T. Harman, M. Berzins, D. Sunderland, and H. C. Edwards, "Demonstrating GPU code portability and scalability for radiative heat transfer computations," *J. Comput. Sci.*, vol. 27, pp. 303–319, Jul. 2018.

[95] E. Phipps and T. Kolda, "Software for sparse tensor decomposition on emerging computing architectures," *SIAM J. Sci. Comput.*, vol. 41, no. 3, pp. C269–C290, Jun. 2019.

[96] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1 – 19, Mar. 1995.

[97] F. E. H. Pérez, N. Mukhadiyev, X. Xu, A. Sow, B. J. Lee, R. Sankaran, and H. G. Im, "Direct numerical simulations of reacting flows with detailed chemistry using many-core/GPU acceleration," *Comput. Fluids*, vol. 173, pp. 73 – 79, Sep. 2018.

[98] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems Using C++ and SYCL*. Berlin, Germany: Springer Nature, 2021.

[99] J. Reinders, J. Jeffers, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming Knights Landing Edition.* Boston, MA, USA: Morgan Kaufmann Publishers Inc., 2016.

[100] M. Rovatsou, L. Howes, and R. Keryell, "Khronos Group SYCL 2020 Specification," 2019, https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf.

[101] D. Sahasrabudhe, "Enhancing asynchronous many-task runtime systems for next-generation architectures and exascale supercomputers," Ph.D. dissertation, School Comput., Univ. Utah, Salt Lake City, UT, USA, 2021.

[102] D. Sahasrabudhe and M. Berzins, "Improving performance of the hypre iterative solver for Uintah combustion codes on manycore architectures using MPI endpoints and kernel consolidation," in *Comput. Sci. ICCS 2020,* V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, J. Teixeira, Eds., Cham, Switzerland: Springer International Publishing, 2020, pp. 175–190.

[103] D. Sahasrabudhe, R. Zambre, A. Chandramowlishwaran, and M. Berzins, "Optimizing the hypre solver for manycore and GPU architectures," *J. Comput. Sci.*, vol. 49, p. 101279, Feb. 2021.

[104] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland, "Large scale parallel solution of incompressible flow problems using Uintah and hypre," in *2013 13th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput. (CCGrid)*, May 13-16, 2013, pp. 458–465.

[105] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data race detection in practice," in *Proc. Workshop Binary Instrum. Appl.*, Dec. 12, 2009, pp. 62–71.

[106] P. J. Smith, R.Rawat, J. Spinti, S. Kumar, S. Borodai, and A. Violi, "Large eddy simulations of accidental fires using massively parallel computers," in *16th AIAA Comput. Fluid Dyn. Conf.*, Jun. 23-26 2003, p. 3697.

[107] T. Sterling, M. Anderson, and M. Brodowicz, "A survey: Runtime software systems for high performance computing," *Supercomput. Front. Innov.*, vol. 4, no. 1, pp. 48–68, Feb. 2017.

[108] J. M. Stone, K. Tomida, C. J. White, and K. G. Felker, "The Athena++ adaptive mesh refinement framework: Design and magnetohydrodynamic solvers," *Astrophys. J. Suppl. Ser.*, vol. 249, no. 1, p. 4, Jun. 2020.

[109] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "June 2021 | TOP 500," 2021, https://top500.org/lists/top500/2021/06/.

[110] ——, "November 2021 | TOP 500," 2021, https://top500.org/lists/top500/2021/11/.

[111] D. Sulsky, Z. Chen, and H. L. Schreyer, "A particle method for history-dependent materials," *Comput. Meth. Appl. Mech. Eng.*, vol. 118, no. 1-2, pp. 179–196, Sep. 1994.

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

[113] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins, "An overview of performance portability in the Uintah runtime system through the use of Kokkos," in *Proc. Second Int. Workshop Extreme Scale Program. Models Middleware*, Nov. 18, 2016, pp. 44–47.

[114] Texas Advanced Computing Center, "Stampede," 2017, https://www.tacc.utexas.edu/systems/stampede.

[115] ——, "Stampede 2," 2020, https://www.tacc.utexas.edu/systems/stampede2.

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

[117] C. Trott, "Apps using Kokkos," 2018, https://github.com/kokkos/kokkos/issues/1950.

[118] C. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. A. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 4, pp. 805–817, Jul. 2021.

[119] J. Wilke, D. Hollman, N. Slattengren, J. Lifflander, H. Kolla, F. Rizzi, K. Teranishi, and J. Bennett, "DARMA 0.3. 0-alpha specification," Sandia National Laboratory (SNL-CA), Livermore, CA, USA, Tech. Rep. SAND-2016-5397, 2016.

[120] Z. Yang, D. Sahasrabudhe, A. Humphrey, and M. Berzins, "A preliminary port and evaluation of the Uintah AMT runtime on Sunway TaihuLight," in *9th IEEE Int. Workshop Parallel Distrib. Sci.Eng. Comput. (PDSEC 2018)*, May 21-25, 2018, pp. 1006–1015..

[121] W. Zhang, A. S. Almgren, M. Day, T. Nguyen, J. Shalf, and D. Unat, "BoxLib with tiling: An AMR software framework," *SIAM J. Sci. Comput.*, vol. 38, no. 5, 2016.

[122] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, M. Katz, A. Myers, T. Nguyen, A. Nonaka, M. Rosso, S. Williams, and M. Zingale, "AMReX: A framework for block-structured adaptive mesh refinement," *J. Open Source Softw.*, vol. 4, no. 37, p. 1370, May 2019.

[123] W. Zhang, A. Myers, K. Gott, A. Almgren, and J. Bell, "AMReX: Block-structured adaptive mesh refinement for multiphysics applications," *Int. J. High Perform. Comput. Appl.*, vol. 35, no. 6, Nov. 2021.

[124] M. Zingale, A. S. Almgren, M. G. B. Sazo, V. E. Beckner, J. B. Bell, B. Friesen, A. M. Jacobs, M. P. Katz, C. M. Malone, A. J. Nonaka, D. E. Willcox, and W. Zhang, "Meeting the challenges of modeling astrophysical thermonuclear explosions: Castro, Maestro, and the AMReX astrophysics suite," *J. Phys. Conf. Ser.*, vol. 1031, no. 1, p. 012024, May 2018.