A Heterogeneous MPI+PPL Task Scheduling Approach for Asynchronous Many-Task Runtime Systems

JOHN K. HOLMEN, SCI Institute
University of Utah, USA
DAMODAR SAHASRABUDHE, SCI Institute
University of Utah, USA
MARTIN BERZINS, SCI Institute
University of Utah, USA

Asynchronous many-task runtime systems and MPI+X hybrid parallelism approaches have shown promise for helping manage the increasing complexity of nodes in current and emerging high performance computing (HPC) systems, including those for exascale. The increasing architectural diversity, however, poses challenges for large legacy runtime systems emphasizing broad support for major HPC systems. Performance portability layers (PPL) have shown promise for helping manage this diversity. This paper describes a heterogeneous MPI+PPL task scheduling approach for combining these promising solutions with additional consideration for parallel third party libraries facing similar challenges to help prepare such a runtime for the diverse heterogeneous systems accompanying exascale computing. This approach is demonstrated using a heterogeneous MPI+Kokkos task scheduler and the accompanying portable abstractions [15] implemented in the Uintah Computational Framework, an asynchronous many-task runtime system, with additional consideration for hypre, a parallel third party library. Results are shown for two challenging problems executing workloads representative of typical Uintah applications. These results show performance improvements up to 4.4x when using this scheduler and the accompanying portable abstractions [15] to port a previously MPI-Only problem to Kokkos::OpenMP and Kokkos::CUDA to improve multi-socket, multi-device node use. Good strong-scaling to 1,024 NVIDIA V100 GPUs and 512 IBM POWER9 processor are also shown using MPI+Kokkos::OpenMP+Kokkos::CUDA at scale.

CCS Concepts: • **Computer systems organization** → *Heterogeneous (hybrid) systems*; • **Computing methodologies** → *Parallel computing methodologies*; • **Software and its engineering** → *Software development techniques*; • **Applied computing** → *Physical sciences and engineering*.

Additional Key Words and Phrases: Asynchronous Many-Task Runtime System, Performance Portability, Parallelism and Concurrency, Portability, Software Engineering

ACM Reference Format:
John K. Holmen, Damodar Sahasrabudhe, and Martin Berzins. 2021. A Heterogeneous MPI+PPL Task Scheduling Approach for Asynchronous Many-Task Runtime Systems. 1, 1 (March 2021), 15 pages. https://doi.org/10.1145/1122445.1122456

Authors' addresses: John K. Holmen, SCI Institute
University of Utah, Salt Lake City, Utah, USA, 84112, jholmen@sci.utah.edu; Damodar Sahasrabudhe, SCI Institute
University of Utah, Salt Lake City, Utah, USA, 84112, damodars@sci.utah.edu; Martin Berzins, SCI Institute
University of Utah, Salt Lake City, Utah, USA, 84112, mb@sci.utah.edu.

1 INTRODUCTION

The complexity of nodes anticipated in exascale systems poses new challenges for codes emphasizing large-scale simulation. In addition to understanding how to manage the increased concurrency, deep memory hierarchies, and heterogeneity to make efficient use of these nodes, one must also understand how to manage the increasing architectural diversity with systems such as the DOE Aurora [1] and DOE Frontier [2] to include Intel- and AMD-based GPUs, respectively. The latter, however, complicates preparation of such codes for exascale systems as only heterogeneous IBM- and NVIDIA-based systems, AMD- and NVIDIA-based system, and Intel- and NVIDIA-based systems are available among current heterogeneous high performance computing (HPC) systems on the Top 10 of November 2020's Top500 list [5].

Asynchronous many-task runtime systems and MPI+X hybrid parallelism approaches show promise for helping managing the increased concurrency, deep memory hierarchies, and heterogeneity. For such runtimes, this promise lies in their ability to increase node-level parallelism by overdecomposing an application into many tasks while also easing use of such nodes by offloading low-level details for making use of the underlying hardware to the runtime itself. Examples include Charm++ [22], HPX [21], Legion [7], PaRSEC [9], StarPU [6], and Uintah [8].

Performance portability layers (PPL) show promise for helping managing the architectural diversity. For such layers, 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 use of such nodes by offloading low-level details for making efficient use of the underlying programming models to the layer itself. Examples include Kokkos [11], OCCA [24], RAJA [17], and SYCL [23].

The combination of these promising solutions, however, pose 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 interoperability challenges, and large legacy runtimes, where the combination may require a significant investment.

These challenges are addressed here 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 implementation of 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 runtime 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 runtime infrastructure developers, this scheduler provides maintaining developers with a foundation for supporting wholy portable heterogeneous task scheduling in preparation for the diverse heterogeneous systems accompanying exascale computing.

This paper describes and 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 use of a parallel third party library, hypre [12], 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 [15] 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 [15] 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 [15] to port a previously MPI-Only problem to MPI+Kokkos::OpenMP+Kokkos::CUDA to improve multi-socket, multi-device 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.

The remainder of this paper is structured as follows. Section 2 provides an overview of the Uintah Computational Framework. Section 3 provides an overview of the Kokkos C++ library. Section 4 provides an overview of the resulting heterogeneous MPI+PPL task scheduling approach in the context of Uintah's heterogeneous MPI+Kokkos task scheduler with additional consideration for the parallel library hypre. Section 4.1 describes the large-scale simulations and Uintah MPI+X task schedulers informing this approach. Section 4.2 describes the algorithm central to this approach and making these demonstrations possible. Section 4.3 describes the interoperability challenges making these demonstrations difficult and informing this algorithm. Section 4.4 through 4.8 describe details informed by these demonstrations and related efforts for easing adoption of a heterogeneous MPI+PPL task scheduling approach in an asynchronous many-task runtime system. Paired with similar details [15] for easing indirect adoption of a performance portability layer in an asynchronous many-task runtime system, the two form the foundation for implementing both the scheduler and the accompanying portable abstractions [15] making these demonstrations possible. Section 5 demonstrates practical application of this approach through strong-scaling studies for two challenging problems using this scheduler to execute workloads representative of typical Uintah applications across a large-scale complex heterogeneous system. Section 6 suggests potential challenges moving forward and Section 7 concludes this paper.

2 THE UINTAH COMPUTATIONAL FRAMEWORK

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. This is accomplished through a variety of simulation components based on novel techniques for understanding a broad set of such problems. [8] The work presented here uses the ARCHES [35] turbulent combustion simulation component, which has been the focus of performance portability efforts and is a Large Eddy Simulation (LES) code used to model heat, mass, and momentum transport in turbulent reacting flows with participating media radiation.

Uintah emphasizes broad support for major HPC systems and has been been ported to a diverse set through its lifetime. Recent examples include the NSF Frontera, DOE Lassen, NSF Stampede 2, DOE Cori, DOE Theta, NRCPC Sunway TaihuLight, DOE Titan, NSF Stampede, and DOE Mira systems. For portable standalone use of Kokkos::OpenMP, good strong-scaling has been shown to 1,728 Intel Knights Landing processors on the NSF Stampede 2 system [15]. For portable standalone use of Kokkos::CUDA, good strong-scaling has been shown to 64 NVIDIA K20X GPUs on the DOE Titan system [30]. For portable standalone use of underlying third party libraries (i.e., hypre), good scaling has been shown to 512 Intel Knights Landing processors on the DOE Theta system [33] and NVIDIA V100 GPUs on the DOE Lassen system [34].

The work presented here extends past efforts by combining use of these individually stressed components to further inform and demonstrate the portable heterogeneous task scheduling approach described here. This approach has allowed for good strong-scaling to 1,024 NVIDIA V100 GPUs and 512 IBM POWER9 processors to be achieved using MPI+Kokkos::OpenMP+Kokkos::CUDA on the DOE Lassen system. This has been achieved while making portable simultaneous use of (1) the host, (2) the device, and (3) underlying third party libraries (i.e., hypre).

3 THE KOKKOS C++ LIBRARY

The Kokkos C++ library [11] is an open-source C++ programming model for writing single source performance portable code optimized for a diverse set of major HPC systems. Core abstractions include parallel execution patterns (e.g. *parallel_for*, *parallel_reduce*, *parallel_scan*) and data structures (e.g., Kokkos Views). This effort has since expanded into the Kokkos C++ Performance Portability Programming EcoSystem, which additionally offers a broad set of solutions for science and engineering applications (e.g., Kokkos-aware math kernels). Originally developed at Sandia National Laboratories, the team has also expanded with dedicated developers additionally at Argonne National Laboratory, Lawrence Berkeley National Laboratory, Los Alamos National Laboratory, Oak Ridge National Laboratory, and the Swiss National Supercomputing Centre. More details on Kokkos can be found on the Kokkos GitHub [3, 4].

Uintah is an early adopter of Kokkos with developers of each collaborating directly as a part of the University of Utah's participation in the DOE/NNSA's Predictive Science Academic Alliance Program (PSAAP) II initiative. In recent years, this adoption has proven worthwhile in easing Uintah's transition from many-core systems (e.g., the NSF Stampede 2 system) to multi-socket, multi-device heterogeneous systems (e.g., the DOE Lassen system) given the already familiar programming model. Uintah's Kokkos-related activities prior to the work presented here have been itemized in a recently published list [15]. Note, Kokkos back-ends to OpenMP and CUDA are referred to throughout this paper as Kokkos::OpenMP and Kokkos::CUDA, respectively.

The work presented here extends past efforts by (1) completing implementation of a new task scheduler supporting simultaneous use of nested Kokkos::OpenMP and Kokkos::CUDA and (2) continuing to incrementally refactor loops in ARCHES to additionally support use of Kokkos::CUDA. Unique loops refactored to date total in the 100s and range in size from single-digit lines of code to 100s of lines of code. This progress has allowed for individual ARCHES problems to offload up to two orders of magnitude more unique portable loops to the device using Kokkos::CUDA. This is a notable first among Uintah's Kokkos-related activities as ARCHES previously lacked GPU support with recent case studies [15] limited to a single unique portable loop. Note, there is a domain decomposition dependent multiplier on unique loops not reflected in counts above.

4 UINTAH'S HETEROGENEOUS MPI+KOKKOS TASK SCHEDULER

This section provides an overview of the resulting heterogeneous MPI+PPL task scheduling approach in the context of Uintah's heterogeneous MPI+Kokkos task scheduler with additional consideration for the parallel library hypre.

4.1 History

The heterogeneous MPI+PPL task scheduling approach described throughout this work is informed through application in practice across some of the world's largest of HPC systems through the years. In addition to examples in Section 2, other examples of large-scale demonstrations of Uintah's task scheduling capabilities include runs achieving good scaling to 16,384 GPUs, 700K cores, and 700K cores on the DOE Titan, DOE Mira, and NSF Blue Waters systems [8, 20]. For these capabilities, we've also been fortunate to have the opportunity to participate in early user access periods to help stress early-life and rapidly maturing major HPC systems such as through the Stampede Knights Landings Early Science Program and Aurora Early Science Program. For these reasons, the task executor logic forming the core of Uintah's heterogeneous MPI+Kokkos task scheduler has come about naturally through the years.

Uintah adopted an MPI+X hybrid parallelism approach using the foundational MPI+PThreads task scheduler [25] 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 [19, 20, 26, 28, 29, 31] (2) support for many-core systems [13, 27], (3) portability of (1) and (2) [14, 15, 30, 32, 36], and (4) support for third party libraries using their own hybrid parallelism approaches [33, 34]. The scope and non-trivial nature of this collective effort has resulted in 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, (3) an intermediate MPI+PThreads+Kokkos::CUDA task scheduler, and (4) the heterogeneous MPI+Kokkos::OpenMP+Kokkos::CUDA task scheduler demonstrated here. 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 existence of (4), for their invaluable ability to reduce complexity for development and debugging effort.

4.2 Core Algorithm

Algorithm 1 provides an overview of the task executor logic forming the core of the heterogeneous MPI+Kokkos task scheduler. Task executor logic is executed by task executors, which correspond to the specific compute resources (e.g., cores) used to process actions itemized in Algorithm 1. Task executors are exclusive to MPI process and operate largely independent of one another aside from coordination for data integrity. Execution begins serially at Line 1. Execution is passed to Kokkos at Line 2 to launch multiple disjoint task executors using *Kokkos::OpenMP::partition_master* to achieve 2 levels of portable parallelism within an MPI process. The top level (i.e., task scheduling level) of parallelism determines how many simultaneously executing task executors are used in an MPI process (i.e., 1 to many). The bottom level (i.e., individual task execution level) of parallelism determines how many compute resources are used by a given task executor (i.e., 1 to many). Together, the two dictate how many tasks are capable of been 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 the action forming the state machine in Lines 3 to 15 (e.g., to determine how many or few task executors are, say, initiating host-device data transfers). In practice, these values are specified using OpenMP environment variables (e.g., *OMP_NUM_THREADS*) with care taken to ensure that task executors are distributed across a node in a disjoint manner (e.g., using *OMP PLACES=threads* and *OMP PROC BIND=spread*). More details on Uintah's use of *Kokkos::OpenMP::partition_master* be found in a recent technical report [16].

Algorithm 1 Task Executor Logic.

```
1:  while tasks to compute do
2:      launch multiple disjoint task executors
3:      while tasks to compute do
4:          // TASK QUEUE STATE MACHINE
5:          mark tasks with MPI complete as “pending ghosts”
6:          initiate H2D/D2H transfers for “pending ghosts”
7:          query pending H2D/D2H transfers
8:          decrement dependency count for complete transfers
9:          mark tasks with all dependencies complete as “ready”
10:         break for “ready” hypre task
11:         mark W and R/W dependencies as “invalid“
12:         launch “ready” task on host or device
13:         mark W and R/W dependencies as “valid”
14:         initiate MPI sends for completed tasks
15:         process pending MPI receives
16:     end while
17:     exit multiple disjoint task executors
18:     launch hypre task using single task executor
19: end while
```

The logic in Lines 3 to 15 corresponds to a task queue-based state machine used to coordinate the non-trivial interactions taking place while tasks are in flight. In practice, this state machine consists of a series of individual task queues specific to each action itemized in Line 4 through 14. For example, Line 4 corresponds to a task queue collecting all tasks that have been identified as having MPI complete and are awaiting ghost cells to arrive from another task. After an individual task has been processed for a given state's action it is pushed to the next queue. For example at Line 4, after a given task has been identified as having MPI complete and awaiting ghost cells it is pushed to the next queue, Line 5, where there host-device data movement coordinating the gathering of these ghost cells is initiated. In practice, an individual task executor repeatedly iterates over this state machine looking for work to process.

4.3 Interoperability Challenges

Figure 1 shows the basic interactions that this state machine coordinates at the per-MPI process level immediately surrounding the centrally locally “CPU Core” and “GPU” ovals, which correspond conceptually to individual task executors. Additional complexity not shown in Figure 1 include hand-offs between Uintah infrastructure and: (1) simulation components (e.g., the ARCHES turbulent combustion simulation component used here), (2) standalone models supporting (1) (e.g., a standalone reverse Monte-Carlo tracing (RMCRT) radiation model used here), (3) standalone third party libraries supporting (1) (e.g., a standalone linear solver using LLNL's hypre used here), and (4) third party libraries used inside (1), and (5) structured adaptive mesh refinement grid support (e.g., single-level and 2-level used here).

The combinatorially increasing nature of these interactions and others is the key factor increasing the complexity of these interoperability demonstrations. This is attributed to the coordination of (1) host-device data movement and (2) task granularity taking place both between and among hand-offs. This is complicated further by additional multipliers that are incurred based upon how (1) a domain is decomposed and (2) a task is written. That is, 1 to many tasks coordinating data for 1 to many variables used among 1 to many portable loops in a given task.

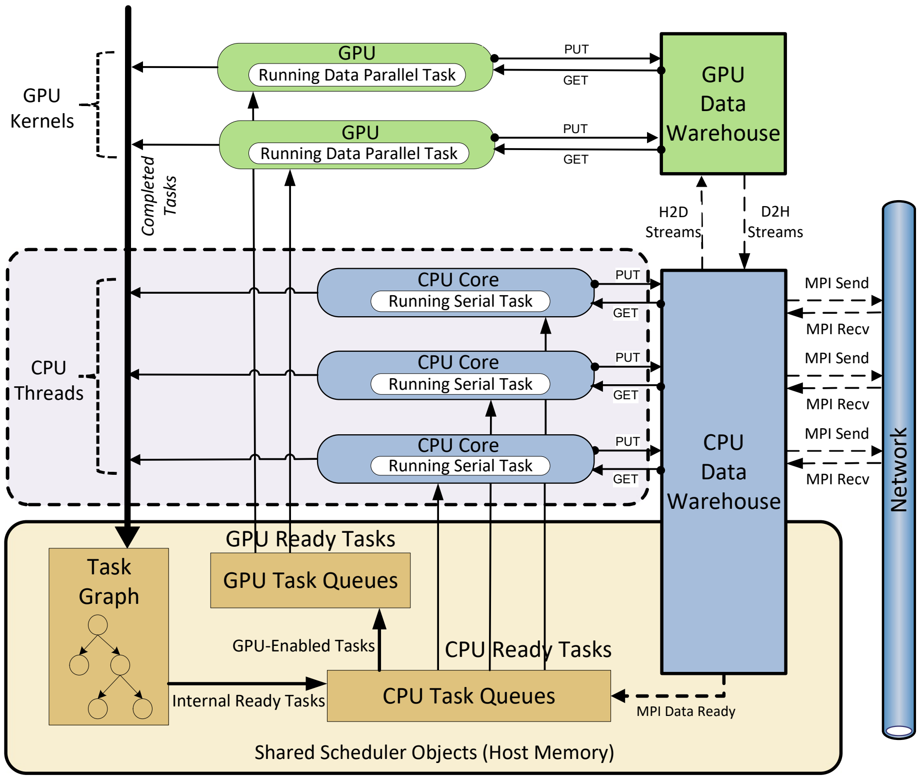

Fig. 1. Uintah's multi-threaded MPI scheduler [14].

For coordinating host-device data movement, the key challenge lies in data integrity. That is, the need to identify both (1) where data resides and (2) when that data can be used. For example, is data computed by an RMCRT task on a given simulation grid mesh level on the device side ready to be handed off to ARCHES to be modified on a different simulation grid mesh level. This is further complicated by data required across tasks having potentially different ghost cell quantities. This results in scenarios where, for example, data residing on the device-side may require re-allocation to accommodate data being transferred from the host-side but with more ghost cells required.

For coordinating task granularity, the key challenge lies in the sizing and re-sizing of simulation mesh patches at hand-off boundaries. This is done to, for example, cater domain decomposition approaches favored by an interconnected third party library. For example, the ability to combine simulation mesh patches into a single larger mesh patch as a part of the hand-off used here and captured in Line 10 and 18 to achieve performance with hypre on the device-side.

For these reasons, this coordination required a non-trivial effort to refine and inform the resulting task executor logic described here and necessary for these interoperablity demonstrations. Challenges hidden among the complexity of this interoperability are discussed in Sections [?] though 4.8. These sections offer suggestions of things to consider when addressing these challenges with Uintah-specific examples provided to ease adoption of a heterogeneous MPI+PPL task scheduling approach in an asynchronous many-task runtime system. Paired with similar details [15] for easing indirect adoption of a performance portability layer in an asynchronous many-task runtime system, the two form the foundation for implementing both the scheduler and the accompanying portable abstractions [15] making these demonstrations possible.

4.4 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 [10] to accommodate third party libraries performing best with an MPI-only parallelism approach.

4.5 Granularity of Host-Device Data Movement

A key benefit of the asynchronous many-task model is 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., to avoid host-device data movement altogether).

4.6 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., valid on the host, invalid on the device, awaiting ghost cell data, etc), (2) allocating memory to accommodate the largest number of ghost cells used across tasks for a given variable, (3) restricting 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 persistent use of either the host or device for a given task.

4.7 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 having 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.

4.8 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) ghost cell and neighboring data requirements for each variable, (4) data movement status for each task, and (5) task execution order.

5 STRONG-SCALING STUDIES

The results presented in this section used two problems to uniquely stress different portions of three individually ported codes key to CCMSC combustion simulations: (1) Uintah's ARCHES turbulent combustion simulation component, (2) Uintah's standalone linear solver using LLNL's hypre, and (3) Uintah's standalone reverse Monte-Carlo tracing (RMCRT) radiation model. The first, a helium plume problem, demonstrates newly portable interoperability of (1) and (2) on a single-level structured grid. The second, a modified Burns and Christon benchmark problem, demonstrates 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 [34] for improving GPU-based performance of hypre was used. Note for both, a modified version of Kokkos implementing recently published techniques [30] for improving GPU-based performance of Kokkos was used.

Helium plume problems play 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[**?**]. This complex problem consists of 125 unique portable loops individually using up to 10s of variables with complex interconnectedness. Underlying Kokkos functionality used among loops include 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 numbers 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 counts above.

Uintah's 2-level reverse Monte-Carlo ray tracing (RMCRT) radiation model [18] plays 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. This complex problem consists of 19 unique portable loops individually using up to 10s of variables with complex interconnectedness. Underlying Kokkos functionality used among loops include 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 counts above.

To demonstrate scalability of the resuling 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 that fit in the 64 GB per-node memory footprint of HBM2. Note, demonstration of host-only capabilities related to this approach are left to recently published results [15] gathered on the NSF Stampede 2 system.

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. Note, a patch is the collection of cells executed by a loop. 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.

Figure 2 shows strong-scaling results across DOE Lassen nodes 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 [34]. 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 larger patch sizes do not provide enough patches for scaling across nodes. Use of MPI+Kokkos::OpenMP+Kokkos::CUDA to improve multi-socket, multi-device 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.

Figure 3 shows strong-scaling results across DOE Lassen nodes 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 [34]. Note, MPI-Only comparisons are not made here as the global, all-to-all nature of the radiation model used by this problem necessitates use of MPI+X hybrid parallelism [18].

Results presented in Figure 2 and 3 show that it is possible to achieve good strong-scaling to 1,024 NVIDIA V100 GPUs and 512 IBM POWER9 processors using MPI+Kokkos::OpenMP+Kokkos::CUDA. 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.

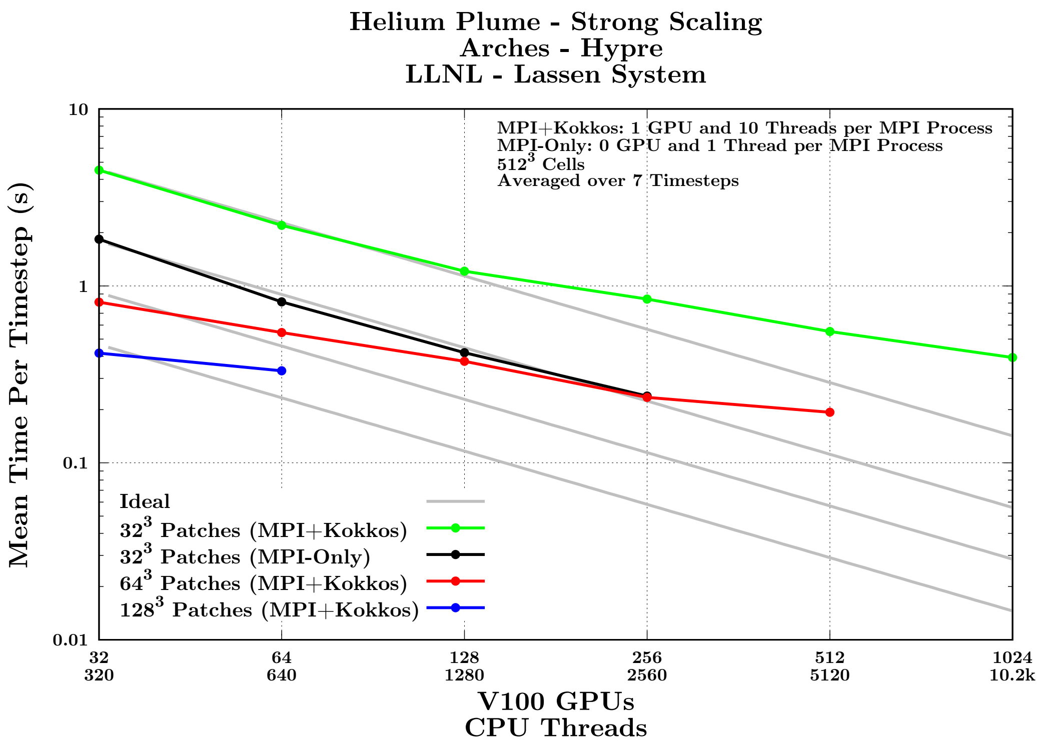

Fig. 2. Helium plume run to 1,024 V100 GPUs and 512 POWER9 processors.

Results presented in Figure 3 show that for a computation-dominant problem it is possible to achieve good strong-scaling across multi-socket, multi-device nodes using an asynchronous many-task model. Results presented in Figure 2 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 2, 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 suggests 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 2 to 64^3 patch and 128^3 patch MPI+Kokkos results in Figure 2 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 [15] to port legacy serial loops to OpenMP and CUDA via Kokkos. This is encouraging as the wholesale refactoring of ARCHES loops to additionally support 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 100s of files comprising this simulation component. This is advantageous as it allows application developers to expedite scientific efforts.

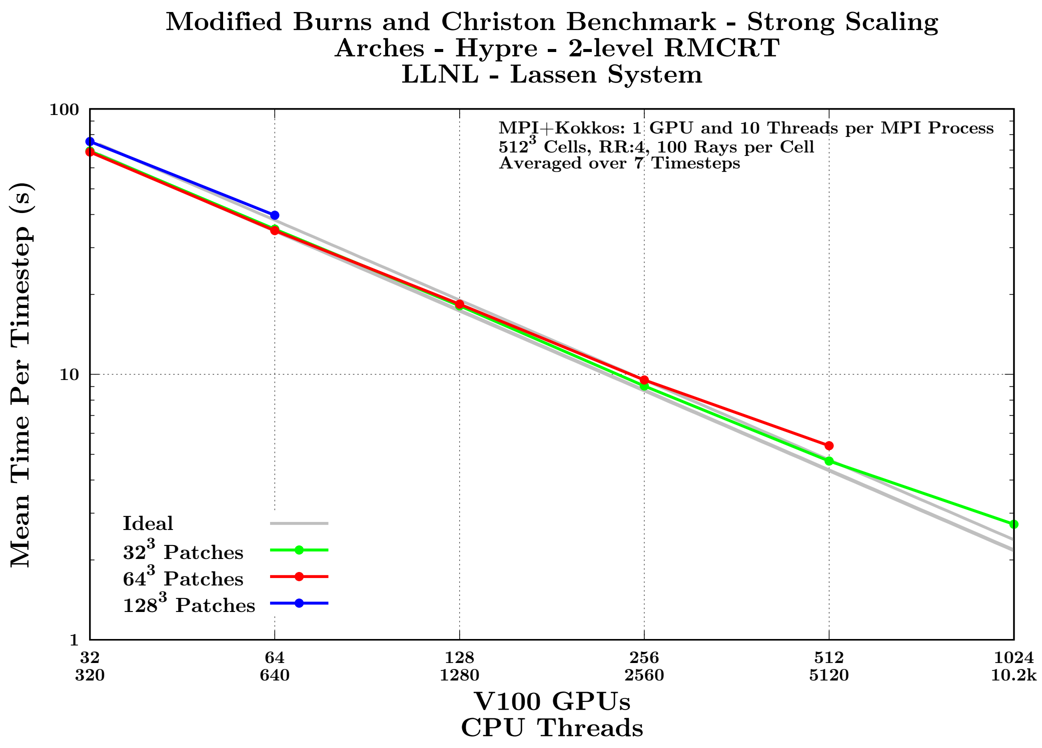

Fig. 3. Modified Burns and Christon benchmark run to 1,024 V100 GPUs and 512 POWER9 processors.

6 FORESEEABLE 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) execute tasks using parallel third party libraries among tasks using a performance portability layer, and (4) automate placement of portable tasks in host and device task queues (e.g., to make informed use of the device).

7 CONCLUSIONS AND FUTURE WORK

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 [15] to execute workloads representative of typical Uintah applications across complex 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 [15] to port a previously MPI-Only problem to Kokkos::OpenMP and Kokkos::CUDA to improve multi-socket, multi-device node use. At scale, good strong-scaling to 1,024 NVIDIA V100 GPUs and 512 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 [15] 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 heterogenerous MPI+Kokkos task scheduling.

ACKNOWLEDGMENTS

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. This research used resources of the Lawrence Livermore National Laboratory. Support for J. K. Holmen and D. Sahasrabudhe comes from the Intel Parallel Computing Centers Program and the TACC Intel Graphics Center. We would like to thank the ALCF for the opportunity to participate in the Aurora Early Science Program as an early user of Aurora. Additionally, we would like to thank all of those involved with the CCMSC and Uintah past and present, with special thanks to Brad Peterson, Jeremy Thornock, Derek Harris, Oscar Díaz-Ibarra, and Todd Harman for their Kokkos-related ARCHES efforts.

REFERENCES

[1] 2019. Aurora. https://aurora.alcf.anl.gov/.

[2] 2019. Frontier. https://www.olcf.ornl.gov/frontier/.

[3] 2019. Kokkos: The C++ Performance Portability Programming Model Wiki. https://github.com/kokkos/kokkos/wiki.

[4] 2019. Tutorials for the Kokkos C++ Performance Portability Programming EcoSystem. https://github.com/kokkos/kokkos-tutorials.

[5] 2020. November 2020 - TOP500 Supercomputer Sites. https://top500.org/lists/top500/2020/11/.

[6] CÃľdric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-AndrÃľ Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[7] M Bauer, S Treichler, E Slaughter, and A. Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *Proceedings of the international conference on high performance computing, networking, storage and analysis*. IEEE Computer Society Press, 66.

[8] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, , and C. Wight. 2016. Extending the Uintah Framework through the Petascale Modeling of Detonation in Arrays of High Explosive Devices. *SIAM Journal on Scientific Computing* 38, 5 (2016), 101–122.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering* 15, 6 (Nov 2013), 36–45.

[10] James Dinan, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. 2013. Enabling MPI interoperability through flexible communication endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting*. 13–18.

[11] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202 – 3216.

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

[13] J. K. Holmen, A. Humphrey, and M. Berzins. 2015. Chapter 13 - Exploring Use of the Reserved Core. In *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, J. Reinders and J. Jeffers (Eds.). Vol. 2. Morgan Kaufmann, Boston, MA, USA, 229 – 242.

[14] J K Holmen, A Humphrey, D Sunderland, and M Berzins. 2017. Improving Uintah's Scalability Through the Use of Portable Kokkos-Based Data Parallel Tasks. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact* (New Orleans, LA, USA) *(PEARC17)*. ACM, New York, NY, USA, Article 27, 27:1–27:8 pages.

[15] J. K. Holmen, B. Peterson, and M. Berzins. 2019. An Approach for Indirectly Adopting a Performance Portability Layer in Large Legacy Codes. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)* (Denver, CO, USA). 36–49. https://doi.org/10.1109/P3HPC49587.2019.00009

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

[17] R. D Hornung and J. A. Keasler. 2014. *The RAJA portability layer: overview and status.* Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

[18] A. Humphrey, T. Harman, M. Berzins, and P. Smith. 2015. A Scalable Algorithm for Radiative Heat Transfer Using Reverse Monte Carlo Ray Tracing. In *High Performance Computing*, Julian M. Kunkel and Thomas Ludwig (Eds.). Lecture Notes in Computer Science, Vol. 9137. Springer International Publishing, 212–230.

[19] A. Humphrey, Q. Meng, M. Berzins, and T. Harman. 2012. Radiation Modeling Using the Uintah Heterogeneous CPU/GPU Runtime System. In *Proceedings of the first conference of the Extreme Science and Engineering Discovery Environment (XSEDE'12)*. Association for Computing Machinery.

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

[21] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (Eugene, OR, USA) *(PGAS '14)*. ACM, New York, NY, USA, Article 6, 11 pages.

[22] L. V Kale and S. Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications* (Washington, D.C., USA) *(OOPSLA '93)*. ACM, New York, NY, USA, 91–108.

[23] Ronan Keryell, Maria Rovatsou, and Lee Howes. 2019. Khronos Group SYCL 1.2.1 Specification. https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf.

[24] David S Medina, Amik St-Cyr, and Tim Warburton. 2014. OCCA: A unified approach to multi-threading languages. *arXiv preprint arXiv:1403.0968* (2014).

[25] Q. Meng, M. Berzins, and J. Schmidt. 2011. Using Hybrid Parallelism to improve memory use in Uintah. In *Proceedings of the TeraGrid 2011 Conference* (Salt Lake City, Utah). ACM.

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

[27] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. 2013. Preliminary Experiences with the Uintah Framework on Intel Xeon Phi and Stampede. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery (XSEDE 2013)* (San Diego, California). 48:1–48:8.

[28] B. Peterson, H. Dasari, A. Humphrey, D. Sunderland, J. Sutherland, T. Saad, and M. Berzins. 2016. Reducing Overhead in the Uintah Framework to Support Short-Lived Tasks on GPU-Heterogeneous Architectures. Submitted *International Journal of Parallel Programming, 2016* (2016).

[29] Brad Peterson, Harish Dasari, Alan Humphrey, James Sutherland, Tony Saad, and Martin Berzins. 2015. Reducing Overhead in the Uintah Framework to Support Short-lived Tasks on GPU-heterogeneous Architectures. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing* (Austin, Texas) *(WOLFHPC '15)*. ACM, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/2830018.2830023

[30] Brad Peterson, Alan Humphrey, John K. Holmen, Todd Harman, Martin Berzins, Dan Sunderland, and H. Carter Edwards. 2018. Demonstrating GPU code portability and scalability for radiative heat transfer computations. *Journal of Computational Science* 27 (2018), 303 – 319. https://doi.org/10.1016/j.jocs.2018.06.005

[31] B. Peterson, A. Humphrey, J. Schmidt, and M. Berzins. 2017. Addressing Global Data Dependencies in Heterogeneous Asynchronous Runtime Systems on GPUs. In *Third International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. IEEE Press.

[32] B. Peterson, N. Xiao, J. K. Holmen, S. Chaganti, A. Pakki, J. Schmidt, D. Sunderland, A. Humphrey, and M. Berzins. 2015. *Developing Uintah's Runtime System For Forthcoming Architectures.* Technical Report. SCI Institute.

[33] Damodar Sahasrabudhe and Martin Berzins. 2020. Improving Performance of the Hypre Iterative Solver for Uintah Combustion Codes on Manycore Architectures Using MPI Endpoints and Kernel Consolidation. In *Computational Science – ICCS 2020*. Springer International Publishing, Cham, 175–190.

[34] Damodar Sahasrabudhe, Rohit Zambre, Aparna Chandramowlishwaran, and Martin Berzins. 2021. Optimizing the hypre solver for manycore and GPU architectures. *Journal of Computational Science* 49 (2021), 101279. https://doi.org/10.1016/j.jocs.2020.101279

[35] P. J. Smith, R.Rawat, J. Spinti, S. Kumar, S. Borodai, and A. Violi. 2003. Large eddy simulations of accidental fires using massively parallel computers. In *16th AIAA Computational Fluid Dynamics Conference*. 3697.

[36] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins. 2016. An Overview of Performance Portability in the Uintah Runtime System Through the Use of Kokkos. In *Proceedings of the Second Internationsl Workshop on Extreme Scale Programming Models and Middleware* (Salt Lake City, Utah) *(ESPM2)*. IEEE Press, Piscataway, NJ, USA, 44–47.