Porting Uintah to Heterogeneous Systems

John K. Holmen
SCI Institute
University of Utah
Salt Lake City, USA
jholmen@sci.utah.edu

Damodar Sahasrabudhe
SCI Institute
University of Utah
Salt Lake City, USA
damodars@sci.utah.edu

Martin Berzins
SCI Institute
University of Utah
Salt Lake City, USA
mb@sci.utah.edu

ABSTRACT

The Uintah Computational Framework is being prepared to make portable use of forthcoming exascale systems, initially the DOE Aurora system through the Aurora Early Science Program. This paper describes the evolution of Uintah to be ready for such architectures. A key part of this preparation has been the adoption of the Kokkos performance portability layer in Uintah. The sheer size of the Uintah codebase has made it imperative to have a representative benchmark. The design of this benchmark and the use of Kokkos within it is discussed. This paper complements recent work with additional details and new scaling studies run 24*x* further than earlier studies. Results are shown for two benchmarks executing workloads representative of typical Uintah applications. These results demonstrate single-source portability across the DOE Summit and NSF Frontera systems with good strong-scaling characteristics. The challenge of extending this approach to anticipated exascale systems is also considered.

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*.

KEYWORDS

Asynchronous Many-Task Runtime System, Performance Portability, Parallelism and Concurrency, Portability, Software Engineering

ACM Reference Format:
John K. Holmen, Damodar Sahasrabudhe, and Martin Berzins. 2018. Porting Uintah to Heterogeneous Systems. In ,. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Forthcoming exascale systems pose new challenges for large-scale simulation codes. These challenges include understanding how to manage the increased concurrency, deep memory hierarchies, and heterogeneity of such systems. Additional challenges are posed by the increasing architectural diversity with systems such as the exascale DOE Aurora [1] and DOE Frontier [28] to feature proposed Intel- and AMD-based GPUs, respectively. This, however, complicates the design of today's software as only heterogeneous high performance computing (HPC) systems featuring NVIDIA-based GPUs are available among current Top 10 of November 2021's Top500 list [39].

While such challenges can also be met by wholesale refactoring or "hero-programming" efforts, it is desirable to make this process easier. One possible approach is to use asynchronous many-task (AMT) runtime systems to help manage the increased concurrency, deep memory hierarchies, and heterogeneity. Examples include Charm++ [21], HPX [20], Legion [3], PaRSEC [5], StarPU [2], and Uintah [4]. Such runtimes are advantageous for their ability to increase node-level parallelism through overdecomposition of an application into many tasks while also managing low-level system details necessary for efficient resource utilization behind-the-scenes.

Performance portability layers (PPL) offer another approach for addressing these challenges. Such layers allow developers to interact with multiple underlying programming models (e.g., CUDA, HIP, OpenMP, etc) through a single interface while also making efficient use of such underlying programming models. Examples include Kokkos [6], OCCA [25], RAJA [12], SYCL [33] and DPC++ [32]. There are many real-world applications using such approaches with, for example, applications using Kokkos collected on their GitHub [44]. A review of exascale challenges and performance portable programming models can be found in a recent survey [19]. While such layers ease exascale challenges for application developers, PPLs shift the "hero-programming" effort to infrastructure developers for those designing AMT runtimes due to the rapidly evolving state of PPLs, complexity of AMT runtimes, and potentially wide range of architectures and programming models to support.

One AMT runtime facing such challenges is the Uintah Computational Framework. Uintah addresses a complex suite of applications, is heterogeneous-capable, and has been ported to a diverse set of HPC systems. This paper considers how Uintah may be extended to exascale by complementing recent work in [11] and building on about a decade's worth of research in adaptive meshing, runtime systems, and radiation ray tracing. This paper also discusses successes and challenges that have arisen when preparing for heterogeneous systems.

The approach taken to prepare Uintah relies on Kokkos to meet challenges posed by diverse heterogeneous systems. The challenge for large codebases such as Uintah is understanding how to evaluate the success of such an approach at scale. One way to evaluate this success is to use two of the most challenging and representative parts of one of the most challenging Uintah applications of the last

PASC 22, PASC 22: ACM Symposium on Neural Gaze Detection, June 03–05, 2022, xxxx, YY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

decade. Uintah's performance is shown for these two benchmarks, which have been prepared for forthcoming exascale systems. This preparation has allowed us to achieve good strong-scaling to 24,576 NVIDIA V100 GPUs and 8,192 IBM POWER9 processors using MPI+Kokkos::OpenMP+Kokkos::CUDA on the DOE Summit system and to 8,192 Cascade Lake processors using MPI+Kokkos::OpenMP on the NSF Frontera system and to compare Uintah's performance across the two systems.

The work presented here complements recent work in [11], which documents early design details and initial experiments. The results shown here extend 24x beyond initial results. This paper builds on [11] by providing additional design details, demonstrating portability across leadership-class systems, explaining how linear algebra is used at scale, discussing porting successes and challenges, and describing next steps for exascale. While these results provide a convincing base for moving to GPU-based exascale machines, they also reinforce careful selection of run configurations and show that application code may still need to be refactored to improve node use.

The remainder of this paper is structured as follows. Section 2 provides an overview of AMT runtimes. Section 3 provides an overview of the Uintah Computational Framework. Section 4 describes the challenges and successes in preparing Uintah for heterogeneous systems and an overview of Uintah's runtime. Section 5 describes Uintah's target exascale benchmarks. Section 6 describes the systems used for strong-scaling studies and presents strong-scaling results gathered on the DOE Summit and NSF Frontera systems for two benchmarks executing workloads representative of typical Uintah applications. Section 7 describes next steps needed for Uintah's exascale preparation and Section 8 concludes this paper.

2 ASYNCHRONOUS MANY-TASK RUNTIME SYSTEMS

The use of asynchronous many task programming goes back to the 1960s [18, 42]. The idea behind such an approach is that a program consisting of a number of computational tasks may be specified as a task graph and that such a task graph has advantages that can be exploited to improve automated execution. Experience with AMT systems such as Uintah shows this to be particularly true if the task graph is complex and/or has behavior that may be hard to predict beforehand (e.g., due to communication delays). The theory for distributed memory cases is similar to that for multi-threaded cases [43]. Randomized or greedy schedulers that use work-stealing give surprisingly good results [43].

The AMT model relies on parallel slackness [43] in that the parallelism of the computation exceeds the physical parallelism of the processor on which the program runs. In the context of a standard message-passing parallel MPI program, this parallel slackness may take the form of multiple execution streams on a compute node whose number exceeds the physical on-node parallelism. In the case of Uintah, there are multiple MPI programs each of which is a task graph in itself. Even if each task graph is linear, there is the potential for hiding delays by switching to another task graph. The maximum benefit occurs when there is enough parallel slackness to hide external data transfers or, at the least, to ensure that such delays are reduced when more nodes are used as in strong scaling.

This characterization also exposes the challenge that in a simple enough code, providing that asynchronous overlapped communication execution may be introduced, the full machinery of an AMT may not be needed to achieve scalable execution and respectable performance may be achieved in a more straightforward manner. At the same time, however, when the behavior of the code becomes hard to predict (e.g., when using adaptive multi-physics, adaptive meshing, or adaptive multi-scale approaches), then unless a good scheduling algorithm is in place scalability may be problematic. A recent comparison of fixed execution and dynamic execution of a complex multi-physics task graph was undertaken by Humphrey and Berzins [14] in which a fully adaptive task execution approach delivered considerable speedups over a fixed execution pattern of the same problem with Uintah.

However, this does not automatically suggest that an AMT runtime is immediately the best approach as managing task execution on a variety of possible heterogeneous exascale machines appears, as will be shown, to have its own challenges. For even relatively simple codes, running on a heterogeneous CPU-GPU architecture may require decisions about which part of the code should be executed where to achieve the best execution time. It is these challenges that will be addressed in the context of the Uintah AMT framework.

3 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. Uintah is based upon novel techniques for understanding a broad set of fluid-structure interaction problems [4].

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.

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

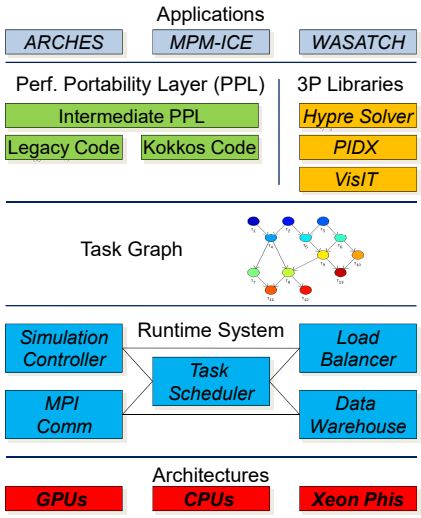

Figure 1: Uintah Software Architecture.

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.

As shown in Figure 1, 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. Uintah features four primary simulation components:

- *ARCHES*: This component targets the simulation of turbulent reacting flows with participating media radiation [38].
- *ICE*: This component targets the simulation of both low-speed and high-speed compressible flows [22].
- *MPM*: This component targets the simulation of multi-material, particle-based structural mechanics [40].
- *MPM-ICE*: This component corresponds to the combination of the ICE and MPM components for the simulation of fluid-structure interactions [7, 8].

Application code is then decomposed into individual tasks that correspond to, for example, physics routines that are executed on either the host or device. Application code can make use of Uintah's intermediate portability layer [10] or third party libraries. The resulting collection of tasks is then compiled into a task graph and dynamically executed by the runtime system in an asynchronous out-of-order manner with implicit work-stealing on the underlying hardware. 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).

For C-SAFE, the target problem was the heating of an explosive device placed in a large hydrocarbon pool fire and the subsequent deflagration explosion and blast wave [29]. Following this work, Uintah development focused on the solution of a problem that involved the detonation of a large amount of explosives in a semi-truck [4]. For the University of Utah's participation in the DOE / NNSA PSAAP II initiative, Uintah development focused on the solution of a problem using large-scale simulation to predict the performance of a commercial, 1200 MWe ultra-supercritical clean coal boiler developed by Alstom (GE) Power. These predictions supported the design and evaluation of an existing boiler capable of providing power for nearly 1 million people.

For predictive boiler simulations, the ARCHES turbulent combustion simulation component was used. ARCHES is a Large Eddy Simulation (LES) code described further in [38]. 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.

The important Uintah technical advancements are described in a series of student dissertations. Luitjen's dissertation [24] research introduced adaptive mesh refinement support. Meng's dissertation [26] research introduced dynamic task scheduling and several task schedulers. Humphrey's dissertation [13] research introduced a scalable approach to radiation modeling. Peterson's dissertation [30] research introduced performant and portable GPU support. Sahasrabudhe's dissertation [34] research introduced a resiliency component and other solutions aimed at addressing the exascale challenges motivating this work.

As a result of these advancements, Uintah has been widely ported and used for large-scale simulation 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 [4, 15, 27]. For GPU-based systems, good strong scaling characteristics have been demonstrated to 16K GPUs [17] 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 [9] and 128 core groups on the National Research Center of Parallel Computer Engineering and Technology (NRCPC) Sunway TaihuLight system [45]. For standalone use of Kokkos::OpenMP, good strong scaling has been shown to 1,728 Intel Knights Landing processors on the NSF Stampede 2 system [10]. For standalone use of Kokkos::CUDA, good strong scaling has been shown to 64 NVIDIA K20X GPUs on the DOE Titan system [31]. For heterogeneous use of Kokkos::OpenMP and Kokkos::CUDA, good strong scaling has been shown to 1,024 NVIDIA V100 GPUs and 512 IBM POWER9 processors on the DOE Lassen system [11].

4 UINTAH'S PORTING APPROACH

Sections 4.1 though 4.4 discuss key successes and challenges from Uintah's preparation for heterogeneous systems. Additional lessons learned can be found among details for easing indirect adoption of a performance portability layer in large legacy codes [10] and details for easing adoption of a heterogeneous MPI+PPL task scheduling approach in an asynchronous many-task runtime system [11].

4.1 Use of Kokkos for Performance Portability

Uintah's adoption of Kokkos began in 2014 [41] with a number of Kokkos-related activities [10] pursued since. These activities

have primarily focused on understanding where and how to add parallelism using Kokkos abstractions. As a result, many related efforts [9, 11, 31] have focused on understanding how Uintah's existing task scheduling approaches change in the context of Kokkos. More details on Uintah's current heterogeneous MPI+PPL task scheduling approach can be found in a recent paper [11].

One success of Uintah's use of Kokkos has been the indirect adoption of this performance portability layer [10], which has proved beneficial in three main ways. Indirect adoption 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. The latter has been particularly helpful for allowing application developers to focus on science rather than learning new programming models. More details on this indirect adoption approach can be found in a recent paper [10].

One challenge for Uintah lies in its working use of Kokkos prototypes that arose from early Kokkos adoption and the fast pace at which both Kokkos and Uintah development moved. An example of this is Uintah's support for the asynchronous execution of Kokkos::CUDA kernels, which currently relies on modifications to Kokkos itself. Rather than adopt Kokkos' latest solution, however, early prototypes were preserved and development paths diverged. Long-term, a better balance needs to be maintained between rapid prototypes and production-grade solutions.

4.2 Use of MPI Endpoints for Third Party Library Support

Uintah uses the Hypre linear solver library to solve pressure equations with billions of variables generated during simulations. Traditionally, Uintah has used Hypre in an MPI-Only setting using one MPI process per CPU core with success scaling to 192k cores [37]. One of the major lessons learned when moving to many-core architectures has been how to make efficient use of Hypre with MPI+OpenMP using MPI Endpoints (EP) and other optimizations.

One challenge for Uintah's early use of Hypre with MPI+OpenMP were 3x to 8x slowdowns over MPI-Only on Intel Knights Landing (KNL) nodes. This was a result of thousands of OpenMP parallel for loops being executed with very small workloads. Such workloads were unable to justify synchronization costs at the end of every such loop.

Early performance was improved by implementing a custom lightweight parallel_for in Hypre. This lightweight parallel_for used atomic primitives and busy-waiting to achieve lock-free thread synchronization to improve MPI+OpenMP performance by 1.5x to 3.9x. However, these improvements alone were not able to meet or exceed MPI-Only performance.

One success for Uintah's use of Hypre with MPI+OpenMP was adopting MPI Endpoints (EP) and making a few other optimizations to not only avoid slowdowns but achieve 2.4x speedups over MPI-Only performance on 256 KNL nodes. In the MPI EP approach, every team of OpenMP threads acts independently as if it is a separate MPI rank (i.e., an MPI endpoint) making calls to Hypre. Smaller team sizes reduce synchronization overhead and can exploit data parallelism on many-core processors. This approach allows multiple threads to participate in MPI communication instead of only the single main thread as done by the original OpenMP implementation. More details on this use of Hypre and MPI EP can be found in recent papers [35, 36, 46].

4.3 Concurrent Development

The simultaneous development of challenging computational application models and a runtime system capable of enabling scalable large-scale parallelism on a compressed timescale has only been possible through "hero-programming" efforts carried out by a multi-disciplinary team of research scientists, and students whose backgrounds range from Computer Science and Electrical Engineering to Chemical Engineering and Mechanical Engineering. These developers are generally either: (1) application developers focused on large-scale science or (2) infrastructure developers focused on supporting the science.

One success of Uintah's concurrent multi-disciplinary development has been good collaboration between application and infrastructure developers. This has led to the simplification of Uintah's portable abstractions to ease use for non-CS developers. An example are application developer-guided variable accessors for Uintah's portable per-grid data structures. More broadly, such collaboration has made possible very large-scale runs using INCITE allocations and much applications-driven Computer Science research for large-scale problems run on major HPC systems.

One challenge for Uintah's concurrent multi-disciplinary development has been striking a balance between application developer focus on ease of use and infrastructure developer focus on rapid performance portability prototypes. For example, application developers have stored input-generated variables in vectors with different naming conventions that have made debugging difficult, especially for race conditions. Another example is tasks being written to cater to the science, rather than the hardware, with arithmetic intensities making efficient resource utilization difficult. Conversely, infrastructure developers have implemented custom solutions for GPU asynchrony in Kokkos [31] but were unable to maintain pace with Kokkos as such features were formally implemented. Another example is infrastructure documentation seeing few updates during development cycles. Nevertheless, insights provided from one group to the other have been essential for the solution of large-scale production-grade applications on major HPC systems.

4.4 Uintah Runtime System

Uintah uses the AMT model to increase node-level parallelism and exploit the related task graph. Once built, Uintah's task graph is executed in an asynchronous out-of-order manner with implicit work-stealing using Uintah's 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) and with data warehouses (DW-House) for CPU and GPU. Figure 2 shows the logic forming Uintah's heterogeneous task scheduler.

In Figure 2, parallel lines connecting steps (1) through (9) correspond to individual task executors. Here, a task executor corresponds to the resources assigned for individual task execution (e.g., cores). Typically, Uintah simulations are run with many task executors using few threads per task executor rather than few task

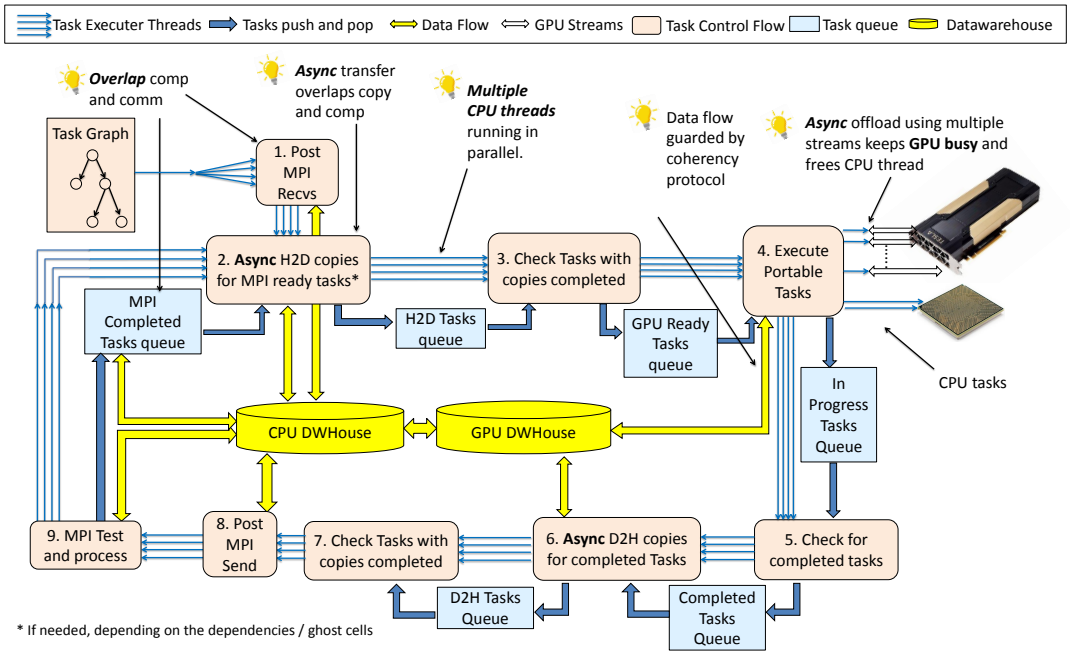

Figure 2: Uintah's heterogeneous task scheduling logic.

executors using many threads per task executor. This is done to avoid performance penalties for tasks that don't scale well. Once MPI receives have been posted, execution iterates over steps (2) through (9) until all tasks have been executed.

A key limitation of Uintah's current heterogeneous 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. Steps (2) and (6) make use of cudaMemcpyAsync for asynchronous host-to-device (H2D) and device-to-host (D2H) transfers. Steps (3), (5), and (7) make 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.

5 UINTAH EXASCALE TARGET BENCHMARKS

The evolution of Uintah's runtime system for anticipated exascale systems requires suitable evaluation benchmarks. The benchmarks described here come from the PSAAP II large-scale simulation used to predict the performance of a commercial, 1200 MWe ultra-supercritical clean coal boiler developed by Alstom (GE) Power.

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

However, the full boiler problem is presently challenging given the sheer complexity of the code, the timescales, and the demanding nature of the application. For this reason, a representative benchmark that exercises the core components of the boiler problem was constructed. This benchmark consists of two representative problems. For feasibility, this benchmark does not feature some of ARCHES most complex loops, such as those in CharOx, which has been previously evaluated in [10]. Though not included here, we've established how to run well with such loops [10] and can revisit such analysis when moving to more complex problems.

Strong-scaling studies in Section 6.3 will use the resulting benchmark problems to uniquely stress different portions of three individually ported codes: (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. 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 [13, 23, 37]. For (2), hypre's preconditioned conjugate gradient (PCG) solver was used with a parallel semi-coarsening multigrid solver as the preconditioner. For RMCRT, weak-scaling is possible through use of aggressive mesh refinement to reduce communication requirements. For both problems, single-source implementations with underlying support for legacy serial loops and Kokkos-based data parallel loops for Kokkos::OpenMP and Kokkos::CUDA were used. Additionally, a modified version of Kokkos implementing recently published techniques [31] for improving GPU-based performance of Kokkos was used.

5.1 Turbulent Combustion Helium Plume Problems

The first problem, a helium plume, demonstrates portable interoperability of (1) and (2) on a single-level structured grid. For this problem, the hypre pressure solve consumes 50-70% of execution time.

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[37]. 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 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.

5.2 Radiation Modeling

The second problem, a modified Burns and Christon benchmark, demonstrates portable interoperability of (1), (2), and (3) on a 2-level structured adaptive mesh refinement grid. For this problem, the hypre pressure solve consumes sub-5% of execution time.

Uintah's 2-level reverse Monte-Carlo ray tracing (RMCRT) radiation model [15] 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 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.

6 STRONG-SCALING STUDIES

Strong-scaling studies make use of both the DOE Summit and NSF Frontera systems. Long-term, Uintah also aims to support the exascale DOE Frontier, Aurora, and El Capitan systems. These systems are similar to Summit in that each is a heterogeneous system with multiple GPUs per node. The DOE Aurora system features Intel-based GPUs. The DOE El Capitan and Frontier systems feature AMD-based GPUs. For this reason, performance portability is important for easing transitions between systems.

6.1 Summit

The DOE Summit is a 200 petaflop system maintained at Oak Ridge National Laboratory's Leadership Computing Facility (OLCF). As of November 2021, Summit ranks at #2 on the Top500 list [39]. Summit consists of 4,608 compute nodes featuring two IBM POWER9 processors with 22 cores each, six NVIDIA Volta V100 GPUs with 80 streaming multiprocessors and 16 GB of HBM2 each, and 512 GB of DDR4 per node. In total, Summit features 27,648 GPUs and 202,752 CPU cores. The system has a peak power consumption of 13 MW and is interconnected by a Mellanox Infiniband EDR interconnect.

6.2 Frontera

The NSF Frontera is a 38 petaflop system maintained at the University of Texas at Austin's Texas Advanced Computing Center (TACC). As of November 2021, Frontera ranks at #13 on the Top500 list [39]. Frontera consists of 8,008 compute nodes featuring two Intel Xeon Platinum 8280 processors with 28 cores each and 192 GB of DDR4 per node. In total, Frontera features 448,448 CPU cores. The system has a peak power consumption of 6 MW and is interconnected by a Mellanox Infiniband HDR interconnect.

6.3 Scaling Experiments

For the DOE Summit and NSF Frontera systems, these studies explored varying domain decomposition approaches using problems sized to fill Summit's 64 GB per-node memory footprint of HBM2. Note, additional demonstration of portable capabilities can be found among MPI+Kokkos::OpenMP results [10] gathered on the NSF Stampede 2 system and MPI+Kokkos::OpenMP+Kokkos::CUDA results [11] gathered on the DOE Lassen system.

For Summit, 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. For Frontera, simulations were launched using both 1 MPI process per node to execute loops using 28 cores per loop and 56 MPI processes per node to execute loops using 1 core per loop.

The simulation domain is decomposed into a collection of patches, which are distributed across MPI processes. Here, a patch refers to the collection of cells executed by a loop. The modified Burns and Chrison benchmark problem also uses adaptive mesh refinement (AMR) to coarsen/refine patches [16]. Domain decomposition and, thus, patch size is user-specified at run-time and remains fixed throughout the simulation.

Problems were sized to provide each MPI process with at least 1 patch in all data points shown. Note, larger patch sizes result in fewer patches being available to distribute across MPI processes and results in fewer strong-scaling data points. Reported per-timestep timings measure the simultaneous execution of all loops in a given timestep. Results have been averaged over 7 consecutive timesteps.

Figure 3 shows strong-scaling results across DOE Summit nodes 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 problem sizes,

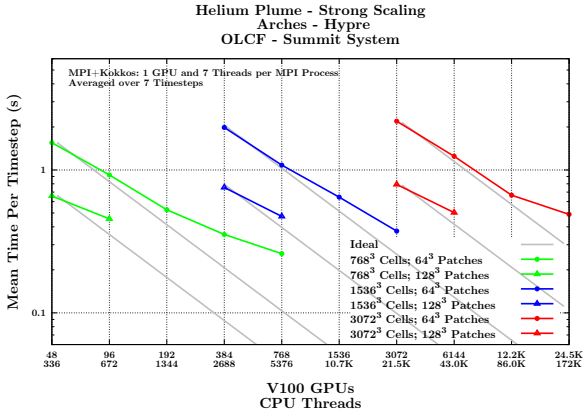

Figure 3: Helium plume run to 24,576 V100 GPUs and 8,192 POWER9 processors using MPI+Kokkos.

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 [36].

Figure 4 shows strong-scaling results across DOE Summit nodes for the modified Burns and Christon benchmark problem on a 2-level structured adaptive mesh refinement grid. Results were

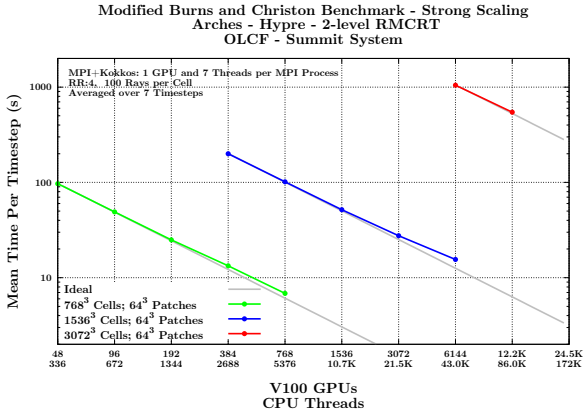

Figure 4: Modified Burns and Christon benchmark run to 12,288 V100 GPUs and 4,096 POWER9 processors using MPI+Kokkos.

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 fine mesh patch size (64^3 cells per fine mesh patch). Note for all problem 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 [36]. For MPI+CUDA comparisons, see related MPI+CUDA and MPI+Kokkos::CUDA comparisons [10, 31] gathered on a single node and the DOE Titan system, respectively.

Figure 5 shows strong-scaling results across NSF Frontera nodes for the helium plume problem on a single-level structured grid. Results were gathered using MPI+Kokkos::OpenMP for three problem

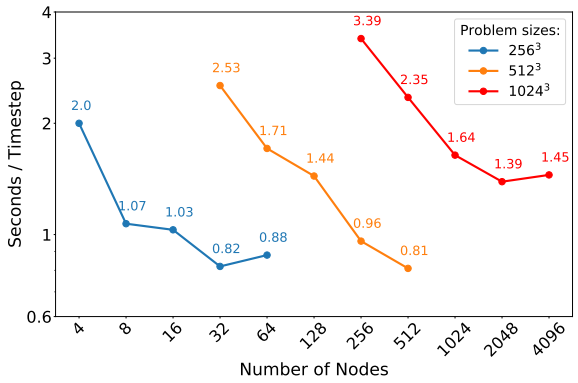

Figure 5: Helium plume run to 8,192 Cascade Lake processors using MPI+Kokkos.

sizes (256^3, 512^3, and 1024^3 cells) and one patch size (32^3 cells per patch). It should be noted that the relatively poor scaling performance is due to lock contention on shared variable access. This is attributed to imbalance in the ratio between patch/task count and simultaneously executing tasks. Specifically, the run configuration results in too few simultaneously executing tasks.

Figure 6 shows strong-scaling results across NSF Frontera nodes for the helium plume problem on a single-level structured grid. Results were gathered using MPI+Only for three problem sizes

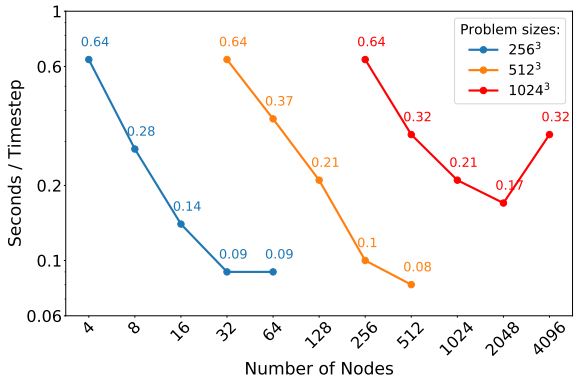

Figure 6: Helium plume run to 8,192 Cascade Lake processors using MPI-Only.

(256^3, 512^3, and 1024^3 cells) and one patch size (32^3 cells per patch). Note, this problem uses Uintah's dynamic MPI task scheduler and executes individual task serially. As such, this does not use Uintah's Kokkos support.

Figure 7 shows strong-scaling results across NSF Frontera 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 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 fine mesh patch sizes (32^3 cells per fine mesh patch).

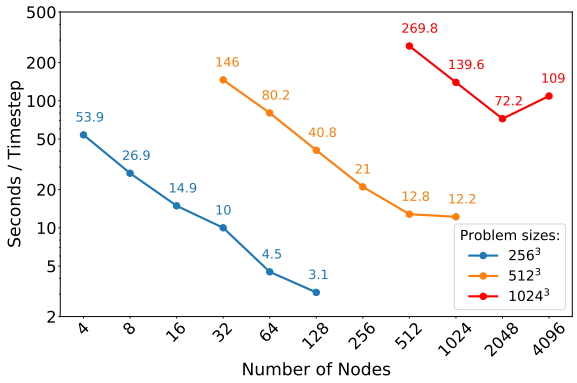

Figure 7: Modified Burns and Christon benchmark run to 8,192 Cascade Lake processors using MPI+Kokkos.

Results presented in Figure 3 show that good strong-scaling to 24,576 GPUs is possible using MPI+Kokkos at scale. This quantity is 24*x* greater than past results [11] gathered on the DOE Lassen system, which demonstrated good strong-scaling with MPI+Kokkos to 1,024 GPUs. This capability increase has been achieved using a single-source implementation and changing only the size of the simulation domain (i.e., to provide more patches to accommodate more GPUs). This is encouraging as it demonstrates the ease with which the asynchronous many-task model can be used to scale out across larger systems. This suggests that Uintah is well-prepared for forthcoming exascale systems as the increase in GPU quantities is anticipated to be on the order of a single-digit multiplier. However, a substantial effort is needed to port Uintah to other underlying programming models supported by exascale systems.

Results presented in Figure 4 show that for a compute-dominant problem it is possible to achieve good strong-scaling across heterogeneous nodes using an asynchronous many-task model. Results presented in Figure 3 show that for a communication-dominant problem it can be difficult to achieve performance across heterogeneous 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 3, 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 heterogeneous nodes. Though performance improvements are achievable when using the full node, performance reductions are also possible when overdecomposing a simulation domain too far.

Comparing results in Figure 5 to Figure 6 shows that it's possible for an MPI-only approach to outperform MPI+Kokkos. This is not unexpected given the nature of the loops in the helium plume problem. This is attributed to the run configuration used with the MPI+Kokkos scheduler and individual tasks being run across multiple cores. Specifically, the many lightweight tasks forming the helium plume problem scale poorly across cores as similarly seen in [10]. Though a poor run configuration was selected for MPI+Kokkos runs, this result is encouraging as the runs were performed by a new user who was able to make quick use of the new scheduler. This outcome reinforces that care must be taken when selecting a task scheduler and run configuration to ensure that efficient use of a node is made. For example, earlier experiments [36] showed that the larger patches (e.g., 64^3 or 128^3) are required to provide enough GPU workload to justify data transfer between CPU and GPU. However, larger patches result in fewer patches to distribute across MPI processes. For this reason, CPU-based simulations, especially those using the MPI-only approach, are run with smaller patches (e.g.,16^3 or 32^3) to provide enough work for cores.

Comparing 8 node runs in Figure 3 to Figure 6, Summit nodes perform approximately 11.5x faster for the helium plume problem when accounting for the larger problem size on Summit. Comparing 8 node runs in Figure 4 to Figure 7, Summit nodes performance approximately 7.5x faster for the modified Burns and Christon benchmark when accounting for the larger problem size on Summit. This is in line with theoretical peak performance across systems. Note, care must be taken when selecting a scheduler and run configuration to ensure that efficient use of a node is made.

7 MOVING UINTAH TO EXASCALE

A key limitation of Uintah's current exascale preparation is that its portable GPU infrastructure depends on use of the Kokkos::CUDA back-end. This is a result of incrementally porting Uintah's extensive pre-existing CUDA-based infrastructure to Kokkos rather than designing new infrastructure with other back-ends in mind. Immediate next steps for continuing Uintah's exascale preparation are in three areas: (1) updating Uintah's use of hypre and Kokkos, (2) improving portability of Uintah's heterogeneous MPI+PPL task scheduling approach, and (3) generalizing Uintah's intermediate portability layer.

7.1 Updating Third Party Library Use

Uintah relies on dated hypre and Kokkos releases. Use of fixed stable releases has helped keep Uintah's Kokkos::CUDA-related research moving quickly and made easy preserving custom modifications to both libraries. An example of such modifications are Uintah's updates to Kokkos for GPU asynchrony [31], which pre-dated Kokkos instances. Reliance on dated releases, however, has been problematic when attempting to move to other Kokkos back-ends as updates are now non-trivial. This effort aims to update Uintah's use of hypre and Kokkos to the latest releases and removing reliance on custom modifications where possible.

7.2 Portable Task Scheduling

Uintah's heterogeneous MPI+Kokkos task scheduler is not 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). While limited, this use was unavoidable due to the functionality used and maturity of Kokkos at the time of early adoption.

This effort aims to generalize 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. 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 effort 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 effort is that underlying programming models are too dissimilar and portable abstractions are not feasible. Accomplishing this will pave the way for defining AMT-related portable abstractions for task scheduling and execution (e.g., to help refine Kokkos' HPX functionality).

7.3 Generalizing Uintah's Portability Layer

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 effort aims to extend 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 effort 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 effort 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 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).

7.4 Task Granularity

In Uintah, task granularity is left to the discretion of the application developer, who typically comes from a non-CS background. Tasks are often written to cater to the science rather than the underlying hardware. ARCHES, for example, consists primarily of low arithmetic intensity loops featuring single-digit lines of code. It is difficult to make good use of OpenMP with such such low intensity loops and this calls for a refactoring of application code.

8 CONCLUSIONS

This study has helped demonstrate Uintah's preparedness for the diverse heterogeneous systems accompanying exascale computing. In summary, the work described here provides a foundation for Uintah's portable exascale use. While exascale ports will be lengthy and challenging for such a large code, subsequent ports will hopefully be easier long-term.

This preparedness has been made possible by use of the asynchronous many-task model, our integration of Kokkos within Uintah, and use of MPI endpoints for third party libraries. Scaling capabilities have been shown for two benchmarks using Uintah's MPI+Kokkos scheduler [11] and the accompanying portable abstractions [10] to execute workloads representative of typical Uintah applications. 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 shown here offers encouragement as we prepare to make portable use of the DOE Aurora system. Next steps include generalizing CUDA-specific code used in Uintah's heterogeneous MPI+Kokkos task scheduler to achieve more portable task scheduling for the Intel- and AMD-based GPUs anticipated in forthcoming exascale systems. As a part of this, emphasis will be placed on extending Uintah's intermediate portability layer [10] to support Kokkos' default host and device spaces to make quicker use of underlying programming models. Another important step will be to refactor ARCHES loops in the helium plume code and elsewhere to increase their computational intensity.

ACKNOWLEDGMENTS

This material is based upon work originally supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. This research used resources of the Oak Ridge National Laboratory through support of the DOE Aurora project and the NSF Texas Advanced Computing Center. Support for J. K. Holmen and D. Sahasrabudhe comes from the University of Texas at Austin under Award Number(s) UTA19-001215 and a gift from the Intel Parallel Computing Centers Program. We would like to thank all 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 Kokkos-related ARCHES efforts.

REFERENCES

[1] Argonne Leadership Computing Facility. 2019. Aurora. https://aurora.alcf.anl.gov/.

[2] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier. 2011. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2 (2011), 187–198.

[3] 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.

[4] 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.

[5] 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.

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

[7] J. E. Guilkey, T. Harman, A. Xia, B. Kashiwa, and P. McMurtry. 2003. An Eulerian-Lagrangian approach for large deformation fluid structure interaction problems, Part 1: algorithm development. *WIT Transactions on The Built Environment* 71 (2003).

[8] T. Harman, J. E. Guilkey, B. Kashiwa, J. Schmidt, and P. McMurtry. 2003. An Eulerian-Lagrangian Approach For Large Deformation Fluid Structure Interaction Problems, Part 2: Multi-physics Simulations Within A Modem Computational Framework. *WIT Transactions on The Built Environment* 71 (2003).

[9] 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.

[10] 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

[11] J. K. Holmen, D. Sahasrabudhe, and M. Berzins. 2021. A Heterogeneous MPI+PPL Task Scheduling Approach for Asynchronous Many-Task Runtime Systems. In *Proceedings of the Practice and Experience in Advanced Research Computing 2021 on Sustainability, Success and Impact (PEARC21)*. ACM.

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

[13] A. Humphrey. 2019. *Scalable Asynchronous Many-Task Runtime Solutions to Globally Coupled Problems.* Ph. D. Dissertation. School of Computing, University of Utah. http://www.sci.utah.edu/publications/Hum2019a/AlanHumphrey_phd_thesis_FINAL_2019.pdf

[14] A. Humphrey and M. Berzins. 2019. An Evaluation of An Asynchronous Task Based Dataflow Approach For Uintah. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 2. 652–657.

[15] 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.

[16] 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.

[17] 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.

[18] C. Lochbaum J. L. Kelly Jr. and V. A. Vyssotsky. 1961. A block diagram compiler. *Bell System Tech. J.* 40, 3 (1961), 669–678.

[19] A. Johnson. 2020. Area Exam: General-Purpose Performance Portable Programming Models for Productive Exascale Computing. (2020).

[20] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. 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.

[21] 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.

[22] B Kashiwa and E Gaffney. 2003. Design Basis for CFDLIB, Tech. Rep. LA-UR-03-1295. (2003).

[23] 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. 2018. Scalable Data Management of the Uintah Simulation Framework for Next-Generation Engineering Problems with Radiation. In *Supercomputing Frontiers*, Rio Yokota and Weigang Wu (Eds.). Springer International Publishing, 219–240. https://doi.org/10.1007/978-3-319-69953-0_13

[24] J.P. Luitjens. 2011. *The Scalability of Parallel Adaptive Mesh Refinement Within Uintah.* Ph. D. Dissertation. School of Computing, University of Utah. http://www.sci.utah.edu/publications/Lui2011b/JLuitjens_PhDThesis2011.pdf Advisor: Martin Berzins.

[25] D. S. Medina, A. St-Cyr, and T. Warburton. 2014. OCCA: A unified approach to multi-threading languages. *arXiv preprint arXiv:1403.0968* (2014).

[26] Q. Meng. 2014. *Large-Scale Distributed Runtime System for DAG-Based Computational Framework.* Ph. D. Dissertation. University of Utah, Salt Lake City, UT, USA.

[27] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. 2013. Investigating Applications Portability with the Uintah DAG-based Runtime System on PetaScale Supercomputers. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. 96:1–96:12.

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

[29] S.G. Parker. 2006. A Component-Based Architecture for Parallel Multi-physics PDE Simulation. *Future Generation Computer Systems (FGCS)* 22, 1-2 (2006), 204–216.

[30] B. Peterson. 2019. *Portable and Performant GPU/Heterogeneous Asynchronous Many-task Runtime System.* Ph. D. Dissertation. University of Utah, School of Computing. http://www.sci.utah.edu/publications/Pet2019a/bradpeterson-thesis.pdf

[31] B. Peterson, A. Humphrey, J. K. Holmen, T. Harman, M. Berzins, D. Sunderland, and H. C. Edwards. 2018. Demonstrating GPU Code Portability and Scalability for Radiative Heat Transfer Computations. *Journal of Computational Science* 27 (2018), 303–319.

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

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

[34] D. Sahasrabudhe. 2021. *ENHANCING ASYNCHRONOUS MANY-TASK RUNTIME SYSTEMS FOR NEXT-GENERATION ARCHITECTURES AND EXASCALE SUPERCOMPUTERS.* Ph. D. Dissertation. University of Utah, Salt Lake City, UT, USA, School of Computing.

[35] D. Sahasrabudhe and M. 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.

[36] D. Sahasrabudhe, R. Zambre, A. Chandramowlishwaran, and M. 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

[37] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland. 2013. Large Scale Parallel Solution of Incompressible Flow Problems using Uintah and hypre. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 458–465. https://doi.org/10.1109/CCGrid.2013.10

[38] 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.

[39] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer. 2021. November 2021 | TOP 500. https://top500.org/lists/top500/2021/11/.

[40] D. Sulsky, Z. Chen, and H. L. Schreyer. 1994. A particle method for history-dependent materials. *Computer methods in applied mechanics and engineering* 118, 1-2 (1994), 179–196.

[41] 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.

[42] W. R. Sutherlandr. 1966. *The On-line Graphical Specification of Computer Procedures.* Ph. D. Dissertation. MIT.

[43] R. L. Rivest T.H. Cormen, C. E. Leiserson and C. Stein. 2009. *Introduction to Algorithms, Third Edition.* MIT Press, Boston Mass.

[44] C. Trott. 2018. Apps Using Kokkos. https://github.com/kokkos/kokkos/issues/1950.

[45] Z. Yang, D. Sahasrabudhe, A. Humphrey, and M. Berzins. 2018. A Preliminary Port and Evaluation of the Uintah AMT Runtime on Sunway TaihuLight. In *9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2018)*. IEEE.

[46] R. Zambre, D. Sahasrabudhe, H. Zhou, M. Berzins, A. Chandramowlishwaran, and P. Balaji. 2021. Logically Parallel Communication for Fast MPI+Threads Communication. In *Proceedings of the Transactions on Parallel and Distributed Computing*. IEEE.