SCI INSTITUTE
TECHNICAL REPORT

Portably Improving Uintah's Readiness for Exascale Systems Through the Use of Kokkos

John K. Holmen, Brad Peterson, Alan Humphrey, Daniel Sunderland, Oscar H. Díaz-Ibarra, Jeremy N. Thornock, Martin Berzins

UUSCI-2019-001

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

May 17, 2019

Abstract:

Uncertainty and diversity in future HPC systems, including those for exascale, makes portable codebases desirable. To ease future ports, the Uintah Computational Framework has adopted the Kokkos C++ Performance Portability Library. This paper describes infrastructure advancements and performance improvements using partitioning functionality recently added to Kokkos within Uintah's MPI+Kokkos hybrid parallelism approach. Results are presented for two challenging calculations that have been refactored to support Kokkos::OpenMP and Kokkos::Cuda back-ends. These results demonstrate performance improvements up to (i) 2.66x when refactoring for portability, (ii) 81.59x when adding loop-level parallelism via Kokkos back-ends, and (iii) 2.63x when more efficiently using a node. Good strong-scaling characteristics to 442,368 threads across 1728 Knights Landing processors are also shown. These improvements have been achieved with little added overhead (sub-millisecond, consuming up to 0.18% of per-timestep time). Kokkos adoption and refactoring lessons are also discussed.

THE UNIVERSITY OF UTAH

Portably Improving Uintah's Readiness for Exascale Systems Through the Use of Kokkos

John K. Holmen[a,], Brad Peterson[a], Alan Humphrey[a], Daniel Sunderland[b], Oscar H. Díaz-Ibarra[c], Jeremy N. Thornock[c], Martin Berzins[a]

[a]*Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT 84112*
[b]*Sandia National Laboratories, PO Box 5800 / MS 1418, Albuquerque, NM 87175*
[c]*Institute for Clean and Secure Energy, University of Utah, Salt Lake City, UT 84112*

Abstract

Uncertainty and diversity in future HPC systems, including those for exascale, makes portable codebases desirable. To ease future ports, the Uintah Computational Framework has adopted the Kokkos C++ Performance Portability Library. This paper describes infrastructure advancements and performance improvements using partitioning functionality recently added to Kokkos within Uintah's MPI+Kokkos hybrid parallelism approach. Results are presented for two challenging calculations that have been refactored to support Kokkos::OpenMP and Kokkos::Cuda back-ends. These results demonstrate performance improvements up to (i) 2.66x when refactoring for portability, (ii) 81.59x when adding loop-level parallelism via Kokkos back-ends, and (iii) 2.63x when more efficiently using a node. Good strong-scaling characteristics to 442,368 threads across 1728 Knights Landing processors are also shown. These improvements have been achieved with little added overhead (sub-millisecond, consuming up to 0.18% of per-timestep time). Kokkos adoption and refactoring lessons are also discussed.

Keywords: Char Oxidation Modeling; Hybrid Parallelism; Kokkos; Portability; Radiation Modeling; Uintah

1. Introduction

This study is motivated by ongoing boiler simulation efforts at the University of Utah's Carbon Capture Multidisciplinary Simulation Center (CCMSC). These efforts have used large-scale simulation to support the design and evaluation of an existing 1000 MWe ultra-supercritical clean coal boiler, which has been developed by Alstom (GE) Power. These simulations use a Large Eddy Simulation (LES) code within the Uintah Computational Framework [1]. Uintah is an open-source asynchronous many-task (AMT) runtime system that has been widely ported to a diverse set of leadership-class HPC systems.

Uintah has supported multicore and GPU-based systems via architecture-specific programming models (e.g., CUDA). The introduction of many-core systems, such as the Intel Xeon Phi-based DOE Cori system, pose new challenges for Uintah due to the increase in core/thread counts at the node-level. This increase makes Uintah's MPI+PThreads hybrid parallelism approach [2] undesirable at scale. This is due to the forced subdivision of a computational domain that this approach requires to support each additional thread used within an MPI process.

Rather than adopt OpenMP directly when extending Uintah to many-core systems, the Kokkos C++ Performance Portability Library [3] was adopted to extend the codebase in a portable manner to GPU-based, many-core, and multicore architectures. Kokkos provides developers with data structures and architecture-aware parallel algorithms that can be executed across multiple architectures. This is accomplished using Kokkos back-ends, which interface to programming models such as CUDA and OpenMP. This portability eases Uintah's future ports, including those for exascale systems.

Through the use of Kokkos, challenges pertaining to forced subdivision of a computational domain have been addressed within [4]. For many-core systems, [4] marks a departure from Uintah's parallel execution of existing serial tasks within an MPI process to serial execution of Kokkos-based data parallel tasks within an MPI process. However, challenges still remain with this pre-

liminary MPI+Kokkos hybrid parallelism approach. Specifically, the use of all threads within an MPI process for individual task execution poses thread scalability challenges when striving to maintain 1 MPI process per node (or NUMA region).

This challenge is addressed here by adopting partitioning functionality recently added to Kokkos via *partition_master* within Uintah's MPI+Kokkos hybrid parallelism approach. This functionality makes it possible to improve node utilization by allowing fewer threads to be used for individual task execution when tasks do not scale well across large thread counts. The resulting flexibility in run configuration has allowed us to achieve performance improvements up to 1.82x, 2.63x, and 1.71x when more efficiently using GPU-, many-core-, and multicore-based nodes, respectively, for Uintah's reverse Monte-Carlo ray tracing-based radiation model. At scale, the resulting MPI+Kokkos hybrid parallelism approach has allowed us to achieve good strong-scaling characteristics to 442,368 threads across 1728 Knights Landing processors for the same model. These improvements have been achieved with little additional overhead introduced (sub-millisecond, consuming up to 0.18% of per-timestep time).

To demonstrate the capabilities of Kokkos, a key combustion loop that models the char oxidation of a coal particle has been used as a case study and refactored to support both the Kokkos::OpenMP and Kokkos::Cuda back-ends. This refactor has allowed us to achieve serial performance improvements up to 2.66x over the original serial loop on a single core. When adding loop-level parallelism via Kokkos, the resulting Kokkos-based loop achieved performance improvements up to 81.59x, 48.05x, and 43.77x using full GPU-, many-core-, and multicore-based nodes, respectively, over the original serial loop on a single core. Perhaps more importantly, this refactor has helped establish the foundations for future Kokkos refactoring efforts within Uintah by helping identify a number of portability barriers encountered during the refactoring process.

The remainder of this paper is structured as follows. Section 2 provides an overview of the Uintah Computational Framework. Section 3 introduces Uintah's char oxidation model. Section 4 introduces Uintah's reverse Monte-Carlo

ray tracing-based radiation model, RMCRT. Section 5 presents Uintah-based interfaces to Kokkos. Section 6 describes Uintah's task scheduling approaches. Section 7 introduces Uintah's use of *partition_master* for task scheduling. Section 8 provides a collection of lessons learned when refactoring existing loops for the Kokkos::OpenMP and Kokkos::Cuda back-ends. Section 9 presents single-node results for the char oxidation model. Section 10 presents single-node and multi-node results for the RMCRT-based radiation model. Section 11 outlines related work and Section 12 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. Uintah is based upon novel techniques for understanding a broad set of fluid-structure interaction problems. [5]

Specializing in large-scale simulation, Uintah has been widely ported across a diverse set of leadership-class HPC systems. For multicore systems, good scaling characteristics have been demonstrated to 96K, 262K, and 512K cores on the NSF Stampede, DOE Titan, and DOE Mira systems, respectively [5, 6, 7]. For GPU-based systems, good strong-scaling characteristics have been demonstrated to 16K GPUs [8] 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 [4] using Uintah's preliminary MPI+Kokkos hybrid parallelism approach and 128 core groups on the NRCPC Sunway TaihuLight system [9]. The work presented here extends this by demonstrating good strong-scaling characteristics to 1728 Knights Landing processors on the NSF Stampede 2 system using the refined MPI+Kokkos hybrid parallelism approach presented here.

Uintah release 2.1.0 in 2017 has four simulation components:

- *ARCHES:* Turbulent reacting flows with participating media radiation.
- *ICE:* Low-speed and high-speed compressible flows.
- *MPM:* Multi-material, particle-based structured mechanics.
- *MPM-ICE:* Fluid-structure interactions via MPM and ICE.

For ongoing boiler simulations, the CCMSC uses the ARCHES simulation component. ARCHES is a Large Eddy Simulation (LES) code described further in [10]. This code uses a low-Mach number (M <0.3), variable density formulation to model heat, mass, and momentum transport in turbulent reacting flows. Two typical and challenging models used for ARCHES-based boiler simulations are explored within this paper. The first is a key combustion loop that models the char oxidation of a coal particle. The second is a ray-tracing algorithm that models radiation, which is the dominant mode of heat transfer within the boiler.

3. Char Oxidation Modeling Within Uintah

The char oxidation of a coal particle involves a complex set of physics. This set of physics includes mass transport of oxidizers from the bulk gas phase to the surface of the particle, diffusion of oxidizers into the pores of the particle, reaction of solid fuel with local oxidizers, and mass transport of the gas products back to the gas phase. As implemented within ARCHES, the char rate computes the rate of chemical conversion of solid carbon to gas products, the rate of heat produced by the reactions, and the rate of reduction of particle size [11]. These rates are used in the Direct Quadrature Method of Moments (DQMOM) [12], which subsequently affect the size, temperature, and fuel content of the particle field. For each snapshot of time simulated, an assumption of steady-state is made that produces a non-linear set of coupled equations. This set of coupled equations is then solved pointwise at each cell within the computational domain using a Newton algorithm. The char model is the most expensive model evaluated during the time integration of physics within ARCHES.

Algorithm 1 provides an overview of the char oxidation model loop structure. The core loop refactored to use the Kokkos::OpenMP and Kokkos::Cuda

Algorithm 1 ARCHES Char Oxidation Model Loop Structure

```
1: for all mesh patches do
2:     for all Gaussian quadrature nodes do
3:         for all cells in a mesh patch do
4:             loop over reactions with an inner loop over reactions
5:             multiple loops over reactions
6:             loop over species
7:             loop over reactions with an inner loop over species
8:             for all Newton iterations do
9:                 multiple loops over reactions
10:                multiple loops over reactions with inner loops over reactions
11:            end for
12:            loop over reactions
13:        end for
14:    end for
15: end for
```

back-ends is the for loop beginning at Line 3 of Algorithm 1. This loop features approximately 350 lines of code with a number of interior loops and Newton iterations within. Outside of the core loop, there is a multiplier incurred by the number of Gaussian quadrature nodes, which results from the DQMOM approximation to the number density function. Inside of the core loop, there are a variety of multipliers incurred by the number of reactions and species computed. Additional complexity is introduced among these multipliers by the per-cell Newton iterations beginning at Line 8 of Algorithm 1. For example, the top bottleneck within the core loop has a multiplier of $GaussianQuadratureNodes * NewtonIterations * Reactions^2$ per cell. This algorithm has a theoretical arithmetic intensity of approximately 1.30 FLOPs per double precision number.

4. Radiation Modeling Within Uintah

Parallel reverse Monte-Carlo ray tracing (RMCRT) methods [6, 8] are one of several methods available within Uintah for solving the radiative transport equation. RMCRT models radiative heat transfer using random walks across rays cast throughout the computational domain. These rays are traced in reverse, towards their origin, to eliminate the need to trace rays that may never reach an origin. During ray traversal, the amount of incoming intensity absorbed by the origin is computed. This incoming intensity is then used to aid in solving the radiative transport equation.

RMCRT has good parallel scalability as rays can be traced simultaneously at any given cell and/or timestep [6]. As a result, a stand-alone RMCRT-based radiation model was developed in Uintah [13]. This model has since been (i) extended to support adaptive mesh refinement (AMR) [6], (ii) further adapted to run on GPUs at large-scale with this novel AMR approach [8], (iii) used to explore performance on the Intel Knights Corner coprocessor [14], (iv) extended to support the Kokkos::OpenMP back-end and used to explore performance of Uintah's initial MPI+Kokkos hybrid parallelism approach at scale on the Intel Knights Landing processor [4], and (v) partially extended to support Kokkos::Cuda for the non-AMR-based model [15].

The work presented here uses Uintah's 2-Level RMCRT-based radiation model. The core loop refactored to use the Kokkos::OpenMP and Kokkos::Cuda back-ends features approximately 500 lines of code with a number of interior loops. Inside of the core loop, there are a variety of multipliers incurred by the number of rays cast per cell and the number of cells that each ray is traced across. For example, the core loop has a multiplier of $100 RaysPerCell * CellsTracedAcross$ for the results presented with this paper. This algorithm has a theoretical arithmetic intensity of approximately 0.66 FLOPs per double precision number.

5. Uintah-Based Kokkos Interfaces for Application Developers

In a large existing codebase, such as Uintah, Kokkos must be adopted incrementally. For example, Arches features approximately 500 loops executing roughly 10,000 lines of code. These loops range from a single line of code to roughly 800 lines of code with an average of 20 lines of code per loop. Throughout our incremental adoption, Uintah's interfaces to Kokkos have been refined. This section presents the current state of these interfaces. These interfaces are a product of the refactoring efforts presented within this paper and Uintah's earlier proof-of-concepts, which can be found in [4], [15], and [16].

A key design philosophy maintained within Uintah is that application developers are isolated from infrastructure details via a task-based approach. This approach allows application developers to focus on writing loop-based tasks, leaving details such as task scheduling to infrastructure developers. Such a divide simplifies application development while allowing infrastructure changes to be made behind-the-scenes with minimal impact on application developers. Aligning with this philosophy, two new interfaces have been introduced within Uintah to ease refactoring efforts for application developers: (1) *Uintah::KokkosView3* and (2) *Uintah::parallel_<pattern>*.

Uintah::KokkosView3 is a custom data type used to store data warehouse variables as plain-old-data within unmanaged Kokkos views with 3-dimensional indexing. Note, more on unmanaged views can be found in 6.5.4 of `https://github.com/kokkos/kokkos/wiki/View`. As currently implemented, these views use the *LayoutStride* memory layout to accommodate arbitrarily strided mapping of indices to a memory location. Though Kokkos offers managed views, unmanaged views have been chosen as a result of Uintah offering means of managing memory allocations. Similarly, Uintah offers means of managing data access patterns through the *Uintah::parallel_<pattern>* interface to be discussed next.

A *Uintah::parallel_<pattern>* is a loop statement used to execute loop iterations either serially or in parallel using a Kokkos back-end. Currently supported

Uintah parallel patterns include *Uintah::parallel_for*, *Uintah::parallel_reduce_min*, and *Uintah::parallel_reduce_sum*. *Uintah::parallel_for* executes a traditional for loop. *Uintah::parallel_reduce_min* executes a reduction identifying the minimum of scalar data across loop iterations. *Uintah::parallel_reduce_sum* executes a reduction identifying the summation of scalar data across loop iterations. These interfaces have proved invaluable when refactoring Uintah.

Listing 1 depicts an example of how loops were structured prior to adopting Kokkos. Listing 2 depicts an example of how loops are now structured to enable support for Kokkos parallel patterns. Comparing Listing 1 and 2, an application developer need only specify an iteration range (e.g., *range* in Listing 2), change the loop statement to use *Uintah::parallel_for*, and use *i,j,k*-based indexing for data warehouse variables to enable support for Kokkos parallel patterns. Note, these changes merely enable support for Kokkos parallel patterns. Additional loop-level changes may be necessary to ensure that a loop will build and/or execute properly for a given back-end. A collection of such changes and lessons from Uintah refactoring efforts can be found in Section 8.

```
for ( CellIterator iter = patch->getCellIterator(); !iter.done(); iter++ ) {
  IntVector origin  = *iter;
  char_rate[origin] = 0.0;
  // ...
}
```

Listing 1: Code listing illustrating how loops were structured prior to adopting Kokkos.

```
Uintah::BlockRange range( patch->getBeginCell(), patch->getEndCell() );

Uintah::parallel_for ( range, [&]( int i, int j, int k ) {
  char_rate(i,j,k) = 0.0;
  // ...
}
```

Listing 2: Code listing illustrating how loops are now structured to enable support for Kokkos parallel patterns.

As currently implemented, loop iterations within a *Uintah::parallel_<pattern>* can be executed serially, in parallel using the Kokkos::OpenMP back-end, or in parallel using the Kokkos::Cuda back-end. The manner in which a *Uintah::parallel_<pattern>* is executed is based upon whether or not Uintah has been built with Kokkos and, if so, which back-end(s) the Kokkos build supports. For example, a *Uintah::parallel_for* is executed using the Kokkos::OpenMP back-end when Uintah is built with a Kokkos build supporting the Kokkos::OpenMP back-end.

Listing 3 depicts an example of how loops are executed when Uintah is built without Kokkos. Listing 4 depicts an example of how loops are executed when Uintah is built with a Kokkos build supporting the Kokkos::OpenMP back-end. Listing 5 depicts an example of how loops are executed when Uintah is built with a Kokkos build supporting the Kokkos::Cuda back-end. In practice, a developer may wish to tune loop execution through aspects such as loop structure (e.g., singly-nested vs. triply-nested) or OpenMP loop scheduling parameters (e.g., chunk size). When using Kokkos parallel algorithms, care must also be taken to ensure that there are enough loop iterations to parallelize over. Parameters such as chunk size make it easy to inadvertently provide fewer iterations than there are OpenMP threads, leaving threads idle.

```
for ( int k = kBegin; k < kEnd; ++k ) {
for ( int j = jBegin; j < jEnd; ++j ) {
for ( int i = iBegin; i < iEnd; ++i ) {
  functor(i,j,k);
}}}
```

Listing 3: Code listing illustrating how loops may be executed when Uintah is built without Kokkos.

For application developers, these interfaces allow for easy adoption of Kokkos without requiring knowledge of intricacies such as Kokkos execution policies. For infrastructure developers, these interfaces offer an easy means of fine-tuning data access patterns and loop execution without requiring code changes across

```
Kokkos::parallel_for (
  Kokkos::RangePolicy<Kokkos::OpenMP, int>( 0, numItems ).set_chunk_size(1),
  [&, iSize, jSize, iBegin, jBegin, kBegin](const int& n)
{
  const int k = ( n / ( jSize * iSize ) ) + kBegin;
  const int j = ( n / iSize ) % jSize + jBegin;
  const int i = ( n ) % iSize + iBegin;
  functor(i,j,k);
});
```

Listing 4: Code listing illustrating how loops may be executed when Uintah is built with a Kokkos build supporting the Kokkos::OpenMP back-end.

```
Kokkos::TeamPolicy<Kokkos::Cuda> tPolicy( blocksPerLoop, threadsPerBlock );
typedef Kokkos::TeamPolicy<Kokkos::Cuda> policyType;

Kokkos::parallel_for (
  tPolicy,
  [=] __device__ ( typename policyType::member_type thread )
{
  const unsigned int nBegin = /* calculate per-block starting iteration */
  const unsigned int numItems = /* calculate total iterations */

  Kokkos::parallel_for (
    Kokkos::TeamThreadRange( thread, numItems ),
    [&, nBegin, iSize, jSize, iBegin, jBegin, kBegin](const int& n)
  {
    const int k = ( ( nBegin + n ) / ( jSize * iSize ) ) + kBegin;
    const int j = ( ( nBegin + n ) / iSize ) % jSize + jBegin;
    const int i = ( nBegin + n ) % iSize + iBegin;
    functor(i,j,k);
  });
});
```

Listing 5: Code listing illustrating how loops may be executed when Uintah is built with a Kokkos build supporting the Kokkos::Cuda back-end.

loops individually. Perhaps most importantly, these interfaces offer an easy means of extending support to additional Kokkos back-ends. The aim is for application developers to, hopefully, only need to refactor loops once to enable support for current and future Kokkos back-ends.

6. Task Scheduling within Uintah

Uintah's infrastructure offers a variety of task scheduling options. These options include an MPI-only task scheduler [17], an MPI+PThreads task scheduler [2], and a preliminary MPI+Kokkos task scheduler [4]. The work in [4] used MPI+Kokkos to overcome a scalability barrier on many-core systems that was imposed by the forced subdivision of a computational domain required by the MPI+PThreads task scheduler to support each additional thread used within an MPI process. In a similar manner, thread scalability barriers within task executors have necessitated the refinement of Uintah's preliminary MPI+Kokkos task scheduler.

A key notion in task scheduling is that of a task executor. A task executor resides within an MPI process and corresponds to the collection of compute resources used to execute an individual task. These resources may range from a single core/thread to an entire compute node. Task executors interact with per-MPI process task queues to select and execute ready tasks (e.g., tasks whose dependencies have been satisfied). Figure 1 presents an overview of the per-MPI process infrastructure supporting Uintah's MPI+PThreads task scheduler. In Figure 1, centrally-located ovals noted as "Running Data Parallel Task" and "Running Serial Task" correspond to task executors. Here, for example, CPU-based task executors are single-threaded with as many task executors as there are threads within an MPI process.

Across task schedulers, a variety of task executor configurations are supported within an MPI process. The MPI-only task scheduler uses a single task executor to support serial execution of serial tasks within an MPI process. The MPI+PThreads task scheduler uses multiple task executors to support parallel

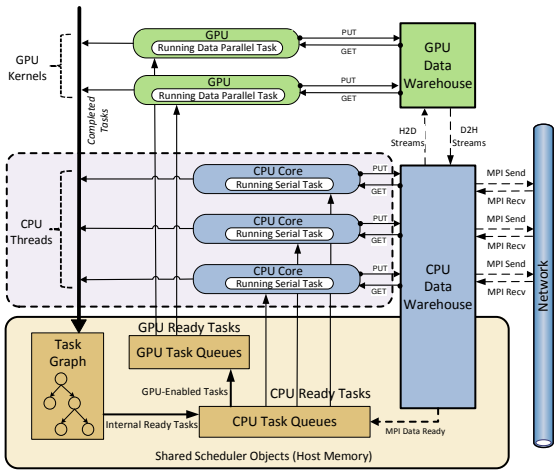
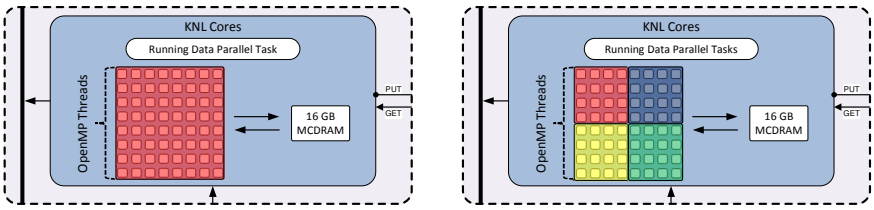


Figure 1: Uintah's multi-threaded MPI scheduler [4].

execution of serial tasks and CUDA-based data parallel tasks within an MPI process. The preliminary MPI+Kokkos task scheduler uses a single task executor to support serial execution of Kokkos-based data parallel tasks within an MPI process.

Though use of a single task executor has eased development of the preliminary MPI+Kokkos task scheduler, this approach is undesirable for many-core systems. Specifically, the use of all threads within an MPI process for individual task execution poses thread scalability challenges when striving to maintain 1 MPI process per node (or NUMA region). To utilize fewer threads within a Kokkos-based data parallel task, Uintah's preliminary MPI+Kokkos task scheduler [4] has been extended to support the parallel execution of Kokkos-based data parallel tasks with an MPI process.

Figure 2 presents an example of how task execution within an MPI process may change as a result of this extension. Note, this figure assumes a run configuration using 1 MPI process and all available cores on a Knights Landing processor. Figure 2a shows the preliminary MPI+Kokkos task scheduler using a single task executor to execute Kokkos-based data parallel tasks using all cores. Figure 2b shows an example of how the refined MPI+Kokkos task scheduler can



(a) Serial execution of data parallel tasks. (b) Parallel execution of data parallel tasks.

Figure 2: Example of how a Knights Landing processor may be used when executing Kokkos-based data parallel tasks serially (a) and in parallel (b).

be used to execute 4 Kokkos-based data parallel tasks simultaneously using 4 task executors with each using a subset of available cores.

This refinement of the MPI+Kokkos task scheduler provides users with means of specifying both how many task executors to use within an MPI process and how many compute resources to use for individual task execution. This flexibility offers greater control over run configuration and task executor granularity when balancing communication and computation at scale. In doing so, this approach enables previously unsupported run configurations (e.g., multiple MPI processes per node in an MPI+X environment). Additionally, the resulting approach enables easy interoperability with OpenMP-based third party libraries as we are no longer mixing threading models (e.g., OpenMP and PThreads).

7. Task Executor Partitioning via Kokkos

The refinement of Uintah's MPI+Kokkos hybrid parallelism model has been made possible by partitioning functionality recently added to Kokkos via *partition_master*. This functionality allows a Kokkos execution space instance to be subdivided into multiple instances. In the context of Uintah, an execution space instance corresponds to a task executor. Similar to newly added CUDA support presented in [15], the introduction of *partition_master* marks another instance of Uintah's needs as an AMT runtime system helping drive Kokkos development.

Uintah's adoption of *partition_master* was relatively straightforward, requiring only a few lines of new code. Listing 6 depicts how *partition_master* has been used within Uintah. This code is called on a per-timestep basis and

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

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

Listing 6: Code listing illustrating Uintah-based code required to enable parallel execution of newly-written Kokkos-based data parallel tasks and existing serial tasks within an MPI process.

has replaced hundreds of lines of thread pool management code within Uintah's MPI+PThreads task scheduler [2]. At the start of a timestep, *partition_master* uses OpenMP to subdivide the original execution space instance into multiple instances. During a timestep, each instance calls *runTasks()* to select and execute all tasks for a given timestep. At the end of a timestep, *partition_master* restores the original execution space instance. Overheads associated with the subdivision and restoration of the original execution space instance will be presented in Section 9.

When using *partition_master* with multiple execution space instances, care must be taken to ensure that a node is fully utilized. Specifically, thread placement becomes critical as it is easy to inadvertently launch overlapping instances. Three OpenMP environment variables are important for using *partition_master*: *OMP_NESTED*, *OMP_PLACES*, and *OMP_PROC_BIND*. The *OMP_NESTED* environment variable enables nested parallelism and must be set to true to allow for multiple execution space instances within an MPI process. The *OMP_PLACES* and *OMP_PROC_BIND* environment variables manage thread placement. For best performance, Kokkos recommends use of *threads*

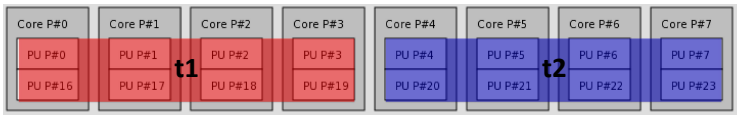


Figure 3: Disjointly placed task executors, fully utilizing a node with *OMP_PLACES=threads* and *OMP_PROC_BIND=spread*.

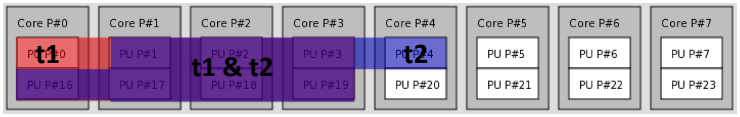


Figure 4: Oversubscribed task executors, under-utilizing a node with *OMP_PLACES=threads* and *OMP_PROC_BIND=close*.

and *spread* for *OMP_PLACES* and *OMP_PROC_BIND*, respectively. In particular, *spread* is critical for ensuring that task executors are placed disjointly across a node. Figure 3 depicts an example of properly placing task executors across a node using *OMP_PLACES=threads* and *OMP_PROC_BIND=spread*. Figure 4 depicts an example of improperly placing task executors across a node using *OMP_PLACES=threads* and *OMP_PROC_BIND=close*.

8. Lessons from Refactoring Loops for Kokkos Back-Ends

Uintah parallel patterns provide application developers with easy means of enabling parallel execution within existing serial loops. However, merely enabling support for Kokkos parallel patterns does not ensure that a loop will build and/or execute properly for a given back-end. Most notably, loops must now be written in a thread-safe manner. A collection of adoption recommendations maintained by the Kokkos team can be found in `https://github.com/kokkos/kokkos/wiki/Interoperability`. Additional insights gained from Uintah refactors are presented within this section.

Refactoring loops for the Kokkos::OpenMP back-end was straightforward. Issues encountered were limited to ensuring that thread-local variables were declared in a thread-safe manner (e.g., inside of parallel patterns). Refactoring loops for the Kokkos::Cuda back-end was more challenging as loops used C/C++

features that are not supported in CUDA. Below is a collection of lessons from efforts refactoring loops to support the Kokkos::Cuda back-end:

1. Eliminate use of C++ standard library classes and functions that do not have CUDA equivalents. Examples encountered include replacing use of *std::cout* with *printf*, replacing use of *std::string* with null-terminated arrays of characters, and hard-coding *std::accumulate*. More on C/C++ features supported in CUDA can be found in E and F of `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.
2. Replace use of *std::vector* with arrays of plain-old-data or *Kokkos::vector*. Functors and lambdas are passed into parallel patterns as const objects. As such, an *std::vector* is read only within a parallel pattern. More on *std::vector* and *Kokkos::vector* can be found in 12.3.3 of `https://github.com/kokkos/kokkos/wiki/Interoperability`.
3. Eliminate allocation of memory within parallel patterns. Memory must be allocated outside of parallel patterns to enable allocation across multiple Kokkos memory spaces.
4. Place arrays inside of a struct to pass them into a lambda. Passing an array itself into a lambda captures the host memory pointer address, which is not accessible from a GPU.
5. Copy class data members into local variables to pass them into a lambda. Passing members accessed via a C++ *this* pointer into a lambda captures the host memory *this* pointer address, which is not accessible from a GPU.

Several other insights were gained during these refactoring efforts, for example:

1. Use template metaprogramming to reduce code duplication. Examples encountered include templating data structures on the Kokkos memory space.
2. Favor using lambdas, instead of functors, for parallel patterns. Functors require duplication of parameter lists across multiple locations. This can be problematic for large parameter lists.

3. Favor use of plain-old-data to avoid temporary object construction and deep object hierarchies.
4. Use the KOKKOS_INLINE_FUNCTION macro to annotate inlined functions.
5. Use the Kokkos pseudorandom number generator, based on [18], to avoid having to manage multiple generators.

Expanding on templates, one of the more difficult challenges faced during these refactoring efforts concerned loop compilation. With hundreds of existing loops in varying states of refactoring, means of selectively compiling loops for a target Kokkos back-end(s) is necessary to enable incremental refactoring. This challenge arises due to the decision to isolate application developers from Kokkos-specific calls via *Uintah::parallel_<pattern>*. While allowing an application developer to write a single loop statement that supports multiple Kokkos back-ends, this requires knowledge of the Kokkos back-end(s) that a loop supports to help ensure successful compilation and execution. For example, loops not yet refactored to support the Kokkos::Cuda back-end must be compiled for serial execution when Uintah is built with a Kokkos build supporting the Kokkos::Cuda back-end. To overcome this challenge, template metaprogramming and macros were used to tag tasks featuring Uintah parallel patterns with the currently supported Kokkos back-end(s).

9. Char Oxidation Modeling Results

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

9.1. Single-Node Studies

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

- *CharOx:CPU*: This is an existing implementation of the char oxidation model written to use serial tasks.

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

CPU-based results have been gathered on a node featuring two 2.7 GHz Intel Xeon E5-2680 Sandy Bridge processors with 8 cores (2 threads per core) per processor and 64 GB of RAM. GPU-based results have been gathered on a node featuring a Maxwell-based NVIDIA GeForce GTX Titan X GPU with 12 GB of RAM. KNL-based results have been gathered on a node featuring one 1.3 GHz Intel Xeon Phi 7210 Knights Landing processor configured for *Flat-Quadrant* mode with 64 cores (4 threads per core) and 96 GB of RAM.

Simulations were launched using 1 MPI process per node. CPU- and KNL-based problems used 1 patch per core with the exception of results in Table 2. GPU-based problems used 16 patches with the exception of results in Table 2. Note, a patch is the collection of cells assigned to a task executor. Run configurations were selected to use the extent of each node. Per-loop timings correspond to timings for the *Uintah::parallel_for* itself. Per-timestep timings correspond to timings for execution of a timestep as a whole. Results have been averaged over 7 consecutive timesteps and 80-320 loops per timestep depending upon patch count.

Table 1 depicts incremental performance improvements achieved when refactoring the char oxidation model. This table presents CPU-based results gathered using the CharOx:CPU implementation for three patch sizes (16^3, 32^3, and 64^3 cells) at various steps of the refactor. Tasks were executed using 16 task executors with 1 thread per task executor via 16 MPI processes. Step 0 corresponds to the original serial loop. Step 1 corresponds to refactoring the loop to use the *Uintah::parallel_for* interface described in Section 5. Step 2 corresponds to replacing use of *std::vector* inside of the loop with 1-dimensional arrays of doubles. Step 3 corresponds to replacing temporary object construction inside of the loop with 2-dimensional arrays of doubles. Step 4 corresponds to hard-coding short

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

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

Table 1: Dual-socket per-loop timings at various steps of the CharOx:CPU refactor on Intel Sandy Bridge. Note, refactor steps are cumulative.

virtual functions inside of the loop. Step 5 corresponds to refactoring data warehouse variables to use the *Uintah::KokkosView3* interface described in Section 5. Step 6 corresponds to replacing *std::string* comparisons inside of the loop with integer-based comparisons. Step 7 corresponds to restructuring the loop to improve data warehouse variable access patterns. These results demonstrate that performance is a by-product of refactoring for portability.

Table 2 depicts cross-architecture comparisons for char oxidation modeling. This table presents CPU-, GPU-, and KNL-based results gathered using the CharOx:Kokkos implementation with the Kokkos::OpenMP, Kokkos::Cuda, and Kokkos::OpenMP back-ends, respectively. For CPU-based results, tasks were executed using 16 task executors with 1 thread per task executor via 1 MPI process and 16 OpenMP threads. For GPU-based results, tasks were executed using 1 CUDA stream and 16 CUDA blocks per loop with 256 CUDA threads per block and 255 registers per thread. For KNL-based results, tasks were executed using 64 task executors with 4 threads per task executor via 1 MPI process and 256 OpenMP threads. Results are presented for nine patch counts (16, 32, 64, 128, 256, 512, 1024, 2048, and 4096 patches with 16^3 cells per patch). These results demonstrate portability of a single codebase across CPU-, GPU-, and KNL-based architectures. While the GPU outperforms both CPU and KNL,

PER-TIMESTEP LOOP THROUGHPUT - in milliseconds - CPU/GPU/KNL

16^3 Patches per Node	CPU - Kokkos::OpenMP	GPU - Kokkos::Cuda	KNL - Kokkos::OpenMP
16	34.60	9.76	X
32	69.29	20.71	X
64	138.49	41.79	117.41
128	277.08	76.69	230.96
256	554.89	150.93	461.85
512	1108.88	-	915.99
1024	2219.71	-	1878.02
2048	4444.84	-	3706.70
4096	-	-	7356.10

Table 2: Single-node per-timestep loop throughput timings comparing CharOx:Kokkos performance across Intel Sandy Bridge, NVIDIA GTX Titan X, and Intel Knights Landing. (X) indicates an impractical patch count for a run configuration using the full node. (-) indicates a problem size that does not fit on the node.

this is achieved at the expense of problem size restrictions.

Table 3 depicts *partition_master* overheads incurred at the start of a timestep. Table 4 depicts *partition_master* overheads incurred at the end of a timestep.

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

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

Table 3: Dual-socket start-of-timestep partition_master overheads across OpenMP wait policies for CharOx:Kokkos on Intel Sandy Bridge.

These tables present CPU-based results gathered using the CharOx:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells) and various combinations of OpenMP wait policies. Tasks were executed using 16 task executors with 2 threads per task executor via 1 MPI process and 32 OpenMP threads. The *OMP_WAIT_POLICY* environment variable decides whether threads spin (active) or yield (passive) while

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

Table 4: Dual-socket end-of-timestep partition_master overheads across OpenMP wait policies for CharOx:Kokkos on Intel Sandy Bridge.

they are waiting. *OMP_WAIT_POLICY* defaults to yielding (passive). The *KMP_BLOCKTIME* environment variable sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping. *KMP_BLOCKTIME* defaults to 200 milliseconds. These results demonstrate that the overheads incurred when using *partition_master* are negligible in the context of this problem. Further, the default OpenMP wait policies are sensible for Uintah's MPI+Kokkos hybrid parallelism approach. For the default OpenMP wait policies, start-of-timestep *partition_master* overheads account for 0.0202%, 0.0138%, and 0.0026% of the elapsed time per timestep for 16^3, 32^3, and 64^3 patches, respectively. For the default OpenMP wait policies, end-of-timestep *partition_master* overheads account for 0.1555%, 0.1016%, and 0.0007% of the elapsed time per timestep for 16^3, 32^3, and 64^3 patches, respectively. Together, these *partition_master* overheads account for 0.1757%, 0.1154%, and 0.0033% of the elapsed time per timestep for 16^3, 32^3, and 64^3 patches, respectively, for default OpenMP wait policies.

Table 5 depicts CPU-based OpenMP thread scalability within a task executor for char oxidation modeling. This table presents CPU-based results gathered using the CharOx:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, tasks were executed using 1, 2, 4, 8, and 16 task executor(s) with 16, 8, 4, 2, and 1 thread(s) per task executor, respectively, via 1 MPI process and 16 OpenMP threads. For 2 threads per core runs, tasks were executed using 1 and 16 task

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

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

Table 5: Dual-socket per-loop thread scalability within a task executor for CharOx:Kokkos on Intel Sandy Bridge. All speedups are referenced against 1 core per loop, 1 thread per loop timings. (*) indicates use of 2 threads per core for an individual loop. (**) indicates use of 2 sockets for an individual loop.

executor(s) with 32 and 2 threads per task executor, respectively, via 1 MPI process and 32 OpenMP threads. These results demonstrate that it is possible to achieve good loop-level scalability across dual-socket Sandy Bridge. When identifying optimal run configurations, this suggests that task execution times may vary little across variations of task executor counts and sizes. Comparing 1 core per loop, 1 thread per loop timings to Step 7 timings in Table 1 suggests that no performance has been lost when moving to CharOx:Kokkos. The use of additional OpenMP threads within a task executor has allowed for speedups up to 16.83x, 17.25x, and 17.60x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 16 cores with 2 threads per core over use of 1 core and 1 thread per loop. These results suggest that 2 threads per core can be used when enough per-core work is provided. Best per-loop timings achieve 43.59x, 43.77x, and 42.48x speedups for 16^3, 32^3, and 64^3 cells, respectively, over use of 1 Sandy Bridge core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 1).

Table 6 depicts KNL-based OpenMP thread scalability within a task executor for char oxidation modeling. This table presents KNL-based results gathered using the CharOx:Kokkos implementation with the Kokkos::OpenMP back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For 1 thread per core runs, tasks

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

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

Table 6: Single-socket per-loop thread scalability within a task executor for CharOx:Kokkos on Intel Knights Landing. All speedups are referenced against 1 core per loop, 1 thread per loop timings. (*) indicates use of 2 threads per core for an individual loop. (**) indicates use of 4 threads per core for an individual loop.

were executed using 1, 2, 4, 8, 16, 32, and 64 task executor(s) with 64, 32, 16, 8, 4, 2, and 1 thread(s) per task executor(s), respectively, via 1 MPI process and 64 OpenMP threads. For 2 threads per core runs, tasks were executed using 1 and 64 task executor(s) with 128 and 2 threads per task executor, respectively, via 1 MPI process and 128 OpenMP threads. For 4 threads per core runs, tasks were executed using 1 and 64 task executor(s) with 256 and 4 threads per task executor, respectively, via 1 MPI process and 256 OpenMP threads. These results demonstrate that it can be difficult to achieve good loop-level scalability across Knights Landing. When identifying optimal run configurations, this suggests that task execution times may vary across variations of task executor counts and sizes. As a result, the use of more, yet smaller, task executors have potential to improve node utilization. The use of additional OpenMP threads within a task executor has allowed for speedups up to 46.37x, 65.50x, and 76.90x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 64 cores with 2 threads per core over use of 1 core and 1 thread per loop. These results suggest that up to 4 threads per core can be used when enough per-core work is

provided. Best per-loop timings achieve 30.29x, 42.84x, and 48.05x speedups for 16^3, 32^3, and 64^3 cells, respectively, over use of 1 Sandy Bridge core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 1).

Table 7 depicts GPU-based performance when varying the number of CUDA blocks used per loop for char oxidation modeling. Table 8 depicts GPU-based

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

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

Table 7: Single-GPU performance for varying quantities of CUDA blocks per loop for CharOx:Kokkos on NVIDIA GTX Titan X using 256 CUDA threads per block. All speedups are referenced against 1 block per loop timings.

performance when varying the number of CUDA threads used per CUDA block for char oxidation modeling. These tables present GPU-based results gathered

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

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

Table 8: Single-GPU performance for varying quantities of CUDA threads per CUDA block for CharOx:Kokkos on NVIDIA GTX Titan X using 4 blocks per loop. All speedups are referenced against 128 threads per block timings.

using the CharOx:Kokkos implementation with the Kokkos::Cuda back-end for three patch sizes (16^3, 32^3, and 64^3 cells). For Table 7, tasks were executed using 1 CUDA stream and 1, 2, 4, 8, 16, and 24 CUDA block(s) per loop with 256 CUDA threads per block and 255 registers per thread. For Table 8, tasks were executed using 1 CUDA stream and 4 CUDA blocks per loop with 128,

192, and 256 CUDA threads per block and 255 registers per thread. The use of additional CUDA blocks per loop has allowed for speedups up to 10.00x, 9.67x, and 10.63x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using up to 24 blocks per loop over use of 1 block per loop. The use of additional CUDA threads per CUDA block has allowed for speedups up to 1.69x, 1.65x, and 1.63x to be achieved for 16^3, 32^3, and 64^3 cells, respectively, when using 256 threads per block over use of 128 threads per block. Best per-loop timings achieve 63.82x, 73.85x, and 81.59x speedups for 16^3, 32^3, and 64^3 cells, respectively, over use of 1 Sandy Bridge core and 1 thread per loop for the original serial loop without Kokkos (Step 0 in Table 1).

Revisiting performance, the measured L1 arithmetic intensity for this algorithm is 0.67 FLOPs per double precision number. This value is approximately 1.93 times lower than the algorithm's estimated value of 1.30 FLOPs per double precision number given in Section 3. This is attributed to poor utilization of the memory hierarchy. Specifically, this algorithm needs further tuning to improve cache utilization. In terms of GFLOPS, a 32^3 patch achieves approximately 40 GFLOPS, 104 GFLOPS, and 39 GFLOPS for CPU, GPU, and KNL nodes, respectively. These values represent approximately 18.22%, 1.64%, and 1.74% of peak performance for CPU, GPU, and KNL, respectively. These percentages are not unexpected given the low arithmetic intensity of the algorithm.

10. Radiation Modeling Results

This section presents results from experimental studies solving the Burns and Christon benchmark problem described in [19]. Past Uintah-based studies solving this problem on CPU-, GPU-, and KNL-based systems can be found in [4], [6], [8], and [15]. The studies presented here have been run as in past studies.

10.1. Single-Node Studies

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

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

Results have been gathered on the same nodes used for char oxidation modeling and described in Section 9.1. Simulations were launched using 1 MPI process per node. Run configurations were selected to use the extent of each node. Per-timestep timings correspond to timings for execution of a timestep as a whole. Results have been averaged over 7 consecutive timesteps.

Table 9 depicts cross-architecture comparisons for 2-level RMCRT. This ta-

PER-TIMESTEP TIMINGS - in seconds (x speedup) - CPU/GPU/KNL

Architecture	Implementation	512 - 16^3 Patches	64 - 32^3 Patches	8 - 64^3 Patches
CPU	2L-RMCRT:CPU	51.57* (-)	71.69 (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	36.30* (1.42x)	55.49 (1.29x)	X (-)
Best Configuration	2L-RMCRT:Kokkos	34.96* (1.48x)	42.03* (1.71x)	60.55* (-)
GPU	2L-RMCRT:GPU	32.08 (-)	46.58 (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	25.88 (1.24x)	36.66 (1.27x)	X (-)
Best Configuration	2L-RMCRT:Kokkos	19.96 (1.61x)	25.60 (1.82x)	43.63 (-)
KNL	2L-RMCRT:CPU	57.93** (-)	102.11 (-)	X (-)
Same Configuration	2L-RMCRT:Kokkos	43.82** (1.32x)	80.99 (1.26x)	X (-)
Best Configuration	2L-RMCRT:Kokkos	29.17** (1.99x)	38.78** (2.63x)	60.45** (-)

Table 9: Single-node per-timestep timings comparing 2-level RMCRT performance across Intel Sandy Bridge, NVIDIA GTX Titan X, and Intel Knights Landing. Same Configuration indicates use of the same run configuration as the existing non-Kokkos implementation. Best Configuration indicates use of the best run configuration enabled by additional flexibility introduced when adopting Kokkos. (X) indicates an impractical patch count for a run configuration using the full node. (*) indicates use of 2 threads per core. (**) indicates use of 4 threads per core.

ble presents CPU-, GPU-, and KNL-based results for three 2-level RMCRT implementations using a problem featuring 128^3 cells on the fine mesh and 32^3 cells on the coarse mesh. Results are presented for three fine mesh configurations (512, 64, and 8 patches with 16^3, 32^3, and 64^3 cells per patch, respectively). These results demonstrate portability of a single codebase across CPU-, GPU-, and KNL-based architectures. Further, these results suggest that no performance has been lost when moving to 2-Level RMCRT:Kokkos. For CPU-based results, optimal run configurations with 2L-RMCRT:Kokkos have allowed for speedup up to 1.48x and 1.71x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported run configurations using the existing non-Kokkos implementation. For GPU-based results, optimal run configurations with 2L-RMCRT:Kokkos have allowed for speedup up to 1.61x and 1.82x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported run configurations using the existing non-Kokkos implementation. For KNL-based results, optimal run configurations with 2L-RMCRT:Kokkos have allowed for speedup up to 1.99x and 2.63x to be achieved for 16^3 and 32^3 patches, respectively, over previously supported run configurations using the existing non-Kokkos implementation. For CPU and KNL, these results suggest that it is advantageous to use all threads within a core.

Revisiting performance, the measured L1 arithmetic intensity for this algorithm is 0.21 FLOPs per double precision number. This value is approximately 3.19 times lower than the algorithm's estimated value of 0.66 FLOPs per double precision number given in Section 4. This is attributed in part to the random nature of this algorithm. This randomization requires irregular data access patterns that result in poor cache utilization. These data access patterns led to a greater difference between measured arithmetic intensity and theoretical for 2-level RMCRT than for the char oxidation model, which features more regular data access patterns.

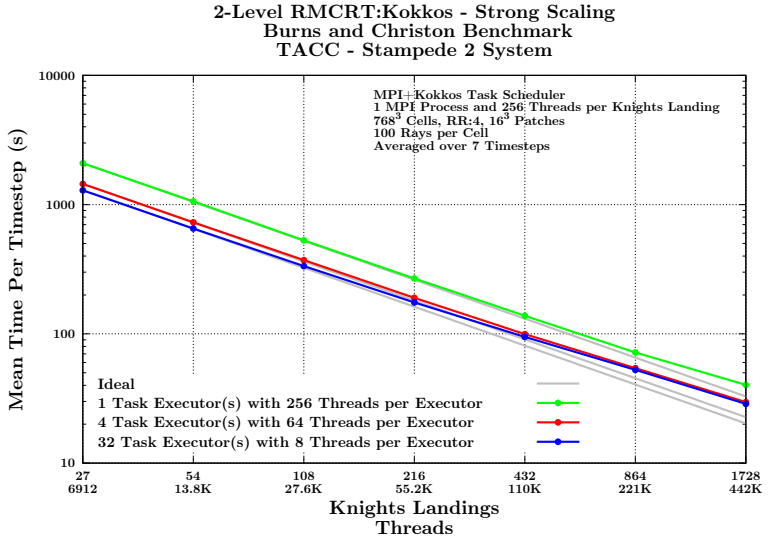


Figure 5: Strong-scaling results to 1728 nodes for 2-level RMCRT:Kokkos on Stampede 2's Knights Landing processors.

10.2. Strong-Scaling Studies

While single-node studies help understand how to efficiently use a node, it is also important to conduct multi-node studies to ensure that results apply at scale. To demonstrate scalability of the resulting MPI+Kokkos hybrid parallelism approach, such studies were conducted on the Knights Landing portion of the NSF Stampede 2 system. This portion of Stampede 2 features the Intel Xeon Phi 7250 Knights Landing processor and offers a variety of memory and cluster mode configurations. These studies explored various task executor counts and sizes across nodes configured for *Cache-Quadrant* mode with a problem that fit within the 16 GB memory footprint of MCDRAM.

Figure 5 depicts strong-scaling of 2-level RMCRT:Kokkos. This figure presents KNL-based results gathered using the 2-Level RMCRT:Kokkos implementation with the Kokkos::OpenMP back-end for a problem featuring 768^3 cells on the fine mesh and 192^3 cells on the coarse mesh. For this problem, the fine mesh was decomposed into 110,592 patches with 16^3 cells per patch. Results are presented for three run configurations (1, 4, and 32 task executor(s) with 256, 64, and 8

threads per task executor, respectively, via 1 MPI process and 256 OpenMP threads).

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

Further, these results demonstrate that it is possible to achieve good strong-scaling characteristics to 442,368 threads across 1728 Knights Landing processors using this MPI+Kokkos hybrid parallelism approach. This is encouraging as it suggests a potential for reducing the number of per-node MPI processes by a factor of up to the number of cores/threads per node in comparison to an MPI-only approach. This is advantageous for many-core systems where the number of MPI processes required to utilize increasingly larger per-node core/thread counts becomes intractable.

11. Related Work

MPI+X hybrid parallelism approaches are commonly used by codebases emphasizing large-scale simulation. A variety of programming models are available for use as the X within MPI+X. For many-core and multicore systems, OpenMP and PThreads are often used. For GPU-based systems, combinations of OpenMP/PThreads and CUDA/OpenMP are often used. An evaluation of these and other programming models can be found in [3] and [20].

Diversity among current and emerging HPC systems has increased the desirability of portable programming models over architecture-specific program-

ming models. Uintah has adopted MPI+Kokkos to extend the codebase in a portable manner to GPU-based, many-core, and multicore systems. Kokkos is one of several programming models that enable interoperability among programming models such as CUDA and OpenMP. At Sandia National Labs, Kokkos has been integrated within Trilinos [21] and used in codes such as Albany [22] and LAMMPS [23]. Examples of similar portable programming models include HEMI [24], OCCA [25], and RAJA [26].

Uintah is one of many asynchronous many-task (AMT) runtime systems and block-structured adaptive mesh refinement (SAMR) frameworks. Examples of similar AMT runtime systems include Charm++ [27], HPX [28], Legion [29], PaRSEC [30], and StarPU [31]. Examples of similar SAMR frameworks include BoxLib [32] (superseded by AMReX [33]) and Cactus [34]. An analysis of performance portability for representative AMT runtime systems, including Uintah, can be found in [35]. A survey of representative SAMR frameworks, including Uintah, can be found in [36].

12. Conclusions and Future Work

This work has helped improve Uintah's portability to future architectures and readiness for exascale systems. Specifically, it has shown performance improvements and portability of a single codebase across GPU-based, many-core, and multicore architectures for two key models within Uintah, char oxidation and RMCRT-based radiation. When refactoring for portability, serial performance improvements up to 2.66x have been achieved over the original serial loop on a single core. When adding loop-level parallelism via Kokkos, performance improvements up to 81.59x have been achieved using a full node over the original serial loop on a single core. Perhaps more importantly, this work has also helped establish the foundations for future Kokkos refactoring efforts within Uintah.

Infrastructure advancements have been made possible by the adoption of partitioning functionality recently added to Kokkos within Uintah's MPI+Kokkos

hybrid parallelism approach. This functionality makes it possible to improve node utilization by allowing fewer threads to be used for individual task execution when tasks do not scale well across large thread counts. When more efficiently using a node, performance improvements up to 2.63x have been achieved using newly supported run configurations over previously supported run configurations on a full node. The resulting flexibility in run configuration also offers greater control over the balance between communication and computation at scale. These improvements have been achieved with little additional overhead introduced (sub-millisecond, consuming up to 0.18% of per-timestep time).

The performance improvements and portability demonstrated here offer encouragement as we prepare to extend more of Uintah to GPU-based HPC systems via the Kokkos::Cuda back-end. In particular, performance improvements suggest a potential to improve boiler simulations through the use of finer mesh resolutions and/or more simulated time. Next steps include furthering our understanding of loop-level tuning parameters. For Kokkos::OpenMP, this includes OpenMP loop scheduling parameters such as scheduling kind (e.g., dynamic, static, etc) and chunk size. For Kokkos::Cuda, this includes architecture-specific parameters such as the number of registers per CUDA thread.

13. Acknowledgements

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. An award of computing time was provided by the NSF Extreme Science and Engineering Discovery Environment (XSEDE) program. This research used resources of the Texas Advanced Computing Center, under Award Number(s) MCA08X004 - "Resilience and Scalability of the Uintah Software". This research also used resources donated to the University of Utah Intel Parallel Computing Center (IPCC) at the SCI Institute. Support for J. K. Holmen comes from the Intel Parallel Computing Centers Program. Additionally, we would like to thank all of those involved with the CCMSC and Uintah past and

present.

References

[1] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. Wight, J. Peterson, Uintah: A scalable framework for hazard analysis, in: Proceedings of the 2010 TeraGrid Conference, ACM, 2010, p. 3.

[2] Q. Meng, M. Berzins, J. Schmidt, Using hybrid parallelism to improve memory use in uintah, in: Proceedings of the TeraGrid 2011 Conference, ACM, 2011.

[3] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling many-core performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (12) (2014) 3202 – 3216.

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

[5] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, , C. Wight, 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.

[6] A. Humphrey, T. Harman, M. Berzins, P. Smith, A scalable algorithm for radiative heat transfer using reverse monte carlo ray tracing, in: J. M. Kunkel, T. Ludwig (Eds.), High Performance Computing, Vol. 9137 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 212–230.

[7] Q. Meng, A. Humphrey, J. Schmidt, M. Berzins, Investigating applications portability with the uintah DAG-based runtime system on PetaScale

supercomputers, in: Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis, 2013, pp. 96:1–96:12.

[8] A. Humphrey, D. Sunderland, T. Harman, M. Berzins, 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), 2016, pp. 1222–1231.

[9] Z. Yang, D. Sahasrabudhe, A. Humphrey, M. Berzins, 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, 2018.

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

[11] W. P. Adamczyk, B. Isaac, J. Parra-Alvarez, S. T. Smith, D. Harris, J. N. T. amd Minmin Zhoub, P. J. Smith, R. Zmuda, Application of les-cfd for predicting pulverized-coal working conditions after installation of nox control system, Energy 693–709.

[12] J. Pedel, J. N. Thornock, S. T. Smith, P. J. Smith, Large eddy simulation of polydisperse particles in turbulent coaxial jets using the direct quadrature method of moments, International Journal of Multiphase Flow 63 (2014) 23–38.

[13] A. Humphrey, Q. Meng, M. Berzins, T. Harman, 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, 2012.

[14] J. K. Holmen, A. Humphrey, M. Berzins, Chapter 13 - exploring use of the reserved core, in: J. Reinders, J. Jeffers (Eds.), High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches, Vol. 2, Morgan Kaufmann, Boston, MA, USA, 2015, pp. 229 – 242.

[15] B. Peterson, A. Humphrey, J. K. Holmen, T. Harman, M. Berzins, D. Sunderland, H. C. Edwards, Demonstrating GPU code portability and scalability for radiative heat transfer computations, Journal of Computational Science.

[16] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, M. Berzins, 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, ESPM2, IEEE Press, Piscataway, NJ, USA, 2016, pp. 44–47.

[17] Q. Meng, J. Luitjens, M. Berzins, Dynamic task scheduling for the uintah framework, in: Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10), 2010, pp. 1–10.

[18] S. Vigna, An experimental exploration of marsaglia's xorshift generators, scrambled, ACM Trans. Math. Softw. 42 (4) (2016) 30:1–30:23.

[19] S. Burns, M. Christon, Spatial domain-based parallelism in large-scale, participating-media, radiative transport applications, Numerical Heat Transfer 31 (4) (1997) 401–421.

[20] M. Martineau, S. McIntosh-Smith, M. Boulton, W. Gaudin, An evaluation of emerging many-core parallel programming models, in: Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'16, ACM, New York, NY, USA, 2016, pp. 1–10.

[21] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, K. S. Stanley, An overview of the trilinos project, ACM Trans. Math. Softw. 31 (3) (2005) 397–423.

[22] A. G. Salinger, R. A. Bartett, Q. Chen, X. Gao, G. Hansen, I. Kalashnikova, A. Mota, R. P. Muller, E. Nielsen, J. Ostien, et al., Albany: A component-based partial differential equation code built on trilinos., ACM Transaction on Mathematical Software.

[23] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, Journal of Computational Physics 117 (1) (1995) 1 – 19.

[24] M. Harris, Hemi: Simpler, More Portable CUDA C++, http://harrism.github.io/hemi/ (2017).

[25] D. S. Medina, A. St-Cyr, T. Warburton, Occa: A unified approach to multi-threading languages, arXiv preprint arXiv:1403.0968.

[26] R. D. Hornung, J. A. Keasler, The raja portability layer: overview and status, Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore, CA (2014).

[27] L. V. Kale, S. Krishnan, 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, OOPSLA '93, ACM, New York, NY, USA, 1993, pp. 91–108.

[28] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, D. Fey, 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, PGAS '14, ACM, New York, NY, USA, 2014, pp. 6:1–6:11.

[29] M. Bauer, S. Treichler, E. Slaughter, A. Aiken, 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, 2012, p. 66.

[30] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, J. J. Dongarra, Parsec: Exploiting heterogeneity to enhance scalability, Computing in Science Engineering 15 (6) (2013) 36–45.

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

[32] W. Zhang, A. S. Almgren, M. Day, T. Nguyen, J. Shalf, D. Unat, Boxlib with tiling: An AMR software framework, CoRR abs/1604.03570.

[33] M. Zingale, A. S. Almgren, M. G. B. Sazo, V. E. Beckner, J. B. Bell, B. Friesen, A. M. Jacobs, M. P. Katz, C. M. Malone, A. J. Nonaka, D. E. Willcox, W. Zhang, Meeting the challenges of modeling astrophysical thermonuclear explosions: Castro, maestro, and the AMReX astrophysics suite, Journal of Physics: Conference Series 1031 (2018) 012024.

[34] T. Goodale, G. Allen, G. Lanfermann, J. Massó, T. Radke, E. Seidel, J. Shalf, The Cactus Framework and Toolkit: Design and Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 197–227.

[35] J. Bennett, R. Clay, G. Baker, M. Gamell, D. Hollman, S. Knight, H. Kolla, G. Sjaardema, N. Slattengren, K. Teranishi, J. Wilke, M. Bettencourt, S. Bova, K. Franko, P. Lin, R. Grant, S. Hammond, S. Olivier, L. Kale, N. Jain, E. Mikida, A. Aiken, M. Bauer, W. Lee, E. Slaughter, S. Treichler, M. Berzins, T. Harman, A. Humphrey, J. Schmidt, D. Sunderland, P. McCormick, S. Gutierrez, M. Schulz, A. Bhatele, D. Boehme, P. Bremer,

T. Gamblin, ASC ATDM level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms, Tech. rep., Sandia National Laboratories (2015).

[36] A. Dubey, A. Almgren, J. Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Löffler, B. OShea, E. Schnetter, B. V. Straalen, K. Weide, A survey of high level frameworks in block-structured adaptive mesh refinement packages, Journal of Parallel and Distributed Computing.