Exploring the Use of Novel Programming Models in Land Surface Models

Ethan T. Coon*, Wael R. Elwasif†, Himanshu Pillai†, Peter E. Thornton*, Scott L. Painter*
* Environmental Sciences Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
† Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
Email: coonet@ornl.gov, elwasifwr@ornl.gov, pillaihk@ornl.gov, thorntonpe@ornl.gov, paintersl@ornl.gov

***Abstract*—A wide range of programming models are currently under rapid development to meet the needs of application developers looking to work on more complex machines. These models fill a variety of roles. Some look to abstract supercomputer architecture, including both processors and memory, to present a strategy for portable performance across a wide range of machines. Others look to expose concurrency by explicitly constructing task-driven dependency graphs that allow a scheduler to find parallelism. Here we explore the implications for application codes of adopting two such programming models, Kokkos and Legion, one from each class of models. We specifically focus on the software design implications on refactoring existing applications, rather than the performance and performance tuning of these models. We identify a strategy for refactoring the Energy Exascale Earth System Model's Land Surface Model, an extremely complex code for climate applications, and prototype a series of mini-apps that explore the adoption of Kokkos and Legion. In doing this, we identify commonalities across the models, leading to a series of conclusions about application software design and refactoring for the adoption of novel programming models. Specifically, we find that refactoring efforts to abstract physics algorithms from data structures enable the use of a variety of programming models. With this refactoring done, we find that, at least in the case of Kokkos and Legion, these types of programming models are sufficiently mature for active use by even small application software development teams.**

***Index Terms*—Software testing,Parallel processing,Distributed processing,Runtime,Coprocessors,Multitasking**

I. Introduction

Porting a complex scientific application currently based on an MPI-only programming model to today's complex supercomputing architectures can be a daunting task. As porting is often motiviated by and done in conjunction with enabling new scientific advances, the first step is identifying target scientific and performance metrics. With these metrics, the application is profiled and analyzed to determine an appropriate strategy for meeting these goals. If the code base is small, and the goals not too extreme, an application development team can potentially port the code in one or more low-level, device-specific languages (MPI+X). However, for complex applications, the performance-critical path may include hundreds of thousands of lines of code that must all be ported to a device. For significant advances in metrics, existing data-level parallelism may not identify sufficient concurrency to allow the scalability needed to reach the goal. In these cases, maintaining multiple implementations is not feasible, and alternative programming models, whether for device abstraction and/or for MPI alternatives, start to show their advantages.

One such complex application space is that of Earth System Models (ESMs) in general and Land Surface Models, the land component of an ESM. Understanding how components of the Earth system interact to determine climate is critical for local, national, and international interests. The Energy Exascale Earth System Model (E3SM) is the US Department of Energy's (DOE) flagship climate model, and is used to meet the scientific needs of the nation. The E3SM Land Model (ELM) is a key component which describes the structure, function, and evolution of physical, biological, and ecological characteristics of Earth's land areas. ELM maintains prognostic state variables for energy, water, carbon, nitrogen, and phosphorus, updating these on approximately half-hourly time steps through a comprehensive collection of process prediction modules. Two important challenges for computation in ELM are the wide range of global land environments represented, and the prevalence of heterogeneous landscape structure even at very fine horizontal resolution.

Climate modeling is an application with opportunities to leverage exascale computing power for great scientific progress [1]. Increased resolution is necessary to move toward cloud-resolving simulations in the atmosphere and eddy-resolving simulations in the ocean. Furthermore, understanding climatological impact on and feedbacks from the land model are of critical importance to water, food, energy, infrastructure, and other human-centric impacts [2]. While currently the land model is a small fraction of the computational time of a coupled E3SM run, anticipated scientific goals and therefore code versions of E3SM and ELM will require significantly improved computing power and scaling. As new architectures become available and the need for improved process representations, increased resolution, and more ensembles beg for more resources, all components of E3SM are working to improve their simulation throughput (measured in simulated years per wallclock hour) and scalability.

Toward this aim, E3SM has been an early explorer of

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

both MPI+X and MPI+X alternatives. One new effort within E3SM focuses on development of a Simple Cloud-Resolving E3SM Atmosphere Model (SCREAM) which will target a 3km horizontal resolution, allowing explicit representation of deep convection events. To allow portability to both GPU and CPU based architectures, this model uses the Kokkos performance portability library [17], following from the successful implementation of an E3SM hydrostatic computational core [4]. E3SM has been actively exploring task-based parallelism approaches in a variety of projects, including the LEAP project, which prototyped an implementation of the ocean model in the Legion task-based runtime [5].

To explore appropriate strategies for meeting the performance goals of DOE's land modeling community, we here consider approaches for porting a subset of processes from ELM onto novel architectures at exascale. We investigate the existing data model of ELM, and identify an approach to refactor data model independent kernels from ELM. We then develop the first in a series of mini-apps to test the suitability of a variety of programming models, including both layers for abstracting computer architecture and task parallel programming models to identify and leverage existing concurrency. We offer our experience in trying two of the more mature programming models, Kokkos and Legion, in the hopes that it will both help programming model developers identify opportunities for improvement and convince application developers that these alternatives to MPI+X are ready for adoption now. Note that this work will not focus on performance, but instead on approaches for designing and refactoring software for the adoption of these models. Despite this, our prototypes adopt the norms established by documentation and tutorials for these programming models, suggesting that performance should be comparable to that of other applications that adopt these models.

II. The Energy Exascale Earth System Land Model

The E3SM Land Model (ELM) uses physically motivated equations to solve for the exchange of water, energy, and biogeochemical species with other Earth system components. In doing this, it also solves for internal state, including extensive representations of the movement and storage of water, the life cycle of vegetation, and coupled biogeochemical cycles including carbon. These processes provide some of the key links between climate and society, including predicting climate impacts on society through water availability and food production capacity [2].

The scale of representation of these processes is of critical importance to the accuracy and applicability of the land model's predictions [6]. The Earth's land surface is characterized by heterogeneity across scales, and capturing the effects of this heterogeneity is a key aspect of ELM's design. ELM places scale at the forefront of its design by modeling individual processes at their own native scale, then interpolating or restricting the effects of those processes across scales to couple with other processes. This led ELM to be built around an explicit scale hierarchy (see Figure 1), each with a set of native processes and the ability to move up and down the hierarchy through area-weighted averaging (restriction) or downscaling. Grid cells, typically arranged in a latitude-longitude mesh, tile the Earth and provide an organizing unit to couple to the atmosphere. Within each grid cell, topographic units potentially group fractions of the grid cell by their elevation, slope, and aspect/orientation. Land units split the topographic units by land cover (e.g. lakes, vegetated surfaces, urban surfaces, etc). The column represents the scale of physical processes such as water and biogeochemistry, and plant functional types (PFTs) split portions of the column by vegetation type, grouping plant species that function similarly. Note that only the grid cell is spatially explicit; all other scales are organized by aspatial characteristics. In this work, we will focus on the finest two scales, which represent much of the natural system physics while requiring restriction operations that are typical of each scale transition.

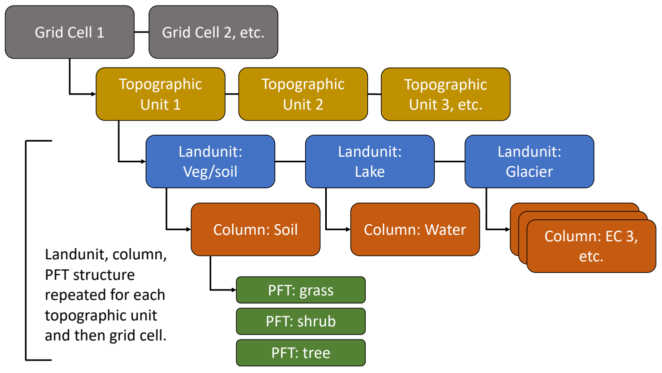

Fig. 1. ELM Scale Hierarchy

ELM's data model (see Figure 2) closely follows the scale hierarchy. At each level, the first dimension of each array stores all elements at a given scale. Additional arrays include, for each element, a parent index *up* the scale hierarchy, and indices to the beginning and end of the slice of children *down* the hierarchy. State and flux data at that scale are grouped into derived data types (structs) of arrays according to their physical functionality (i.e. water variables, vegetation variables, etc). Arrays are decomposed into subdomains at the grid cell level, and each grid cell is independent of all other grid cells.

ELM, like many ESM components, is written in MPI and Fortran 90 and is currently over 200,000 lines of code. This large code base is developed and maintained by a small team of order 10-20 active developers, the majority of whom focus on process development as opposed to software design and/or performance. Performance portability is therefore a priority; ELM currently runs on a variety of CPU-only clusters across DOE and academic institutions, and must continue to support multiple architectures, but does not have sufficient resources to optimize the code for each architecture independently. Since supporting multiple MPI+X paradigms is financially not

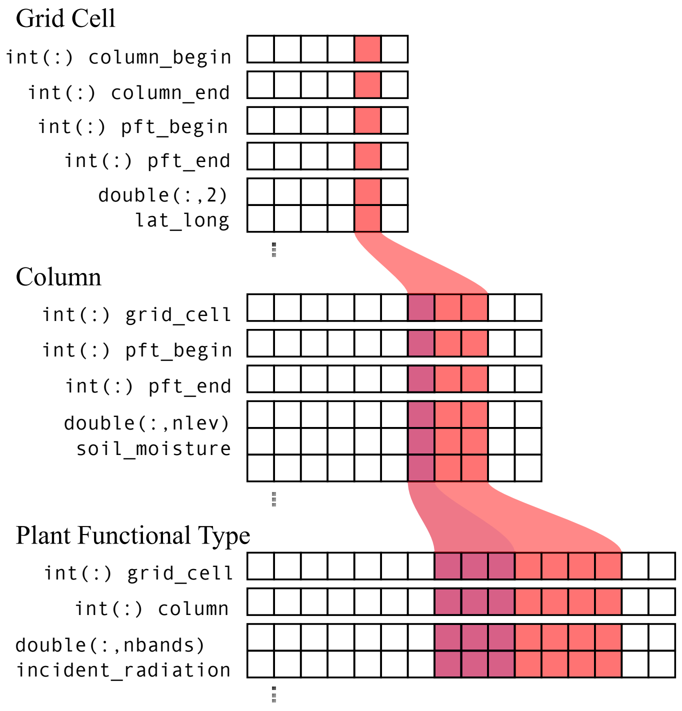

Fig. 2. ELM data model.

tractable, device abstraction layers such as Kokkos [17], RAJA [?], and HPX [8] are of interest.

While it is unclear whether data parallelism is sufficient to scale ELM to the needed applications, the rich set of processes simulated by ELM provides unique opportunities for task-based programming models. Subsystems such as hydrology, energy, vegetation, and biogeochemistry likely have process concurrency which is neither exposed nor leveraged in the current code. Many of the land processes represented (e.g. snow physics) are only relevant in certain parts of the world, while heterogeneity in a process's runtime due to effects such as biodiversity (e.g. many PFTs in tropical forests versus few in grasslands) suggest significant load imbalance across the Earth. Task parallel runtimes such as Legion [5], Charm++ [9], Uintah [10], HPX [8], ParSEC [11], and others, which explicitly construct a dependency graph, identify and exploit process concurrency, and automate load balancing, have great potential for ELM.

III. Refactoring for novel programming models

Prior to porting a CPU-only code to any alternative architecture, an extensive refactoring exercise can often make the process significantly easier. Critically, most novel programming models require some limitation on an application's interactions with its memory. As most applications convolve program data with physical memory, a significant portion of the refactoring process is simply separating these two concepts. Prior to adopting any programming model, we have spent significant effort to identify a refactoring strategy within our mini-app that would support refactoring a complex code such as ELM.

Many authors have advised "best practices" for major refactoring efforts, landing primarily upon the need for extensive unit and regression level tests, planning the refactor process, and splitting refactoring efforts into sub-steps that are conceptually orthogonal [12]–[14]. ELM development makes use of a sophisticated testing environment that covers all aspects of E3SM functionality. Because ELM has numerous features which can be selectively enabled or disabled, and because not all of those features are designed for co-operability, the testing environment establishes a canonical list of science configurations which must each pass a variety of system level tests. For example, recent development introduced interactions among carbon, nitrogen, and phosphorus [15]. That capability is tested for its ability to run from initial conditions, to restart seamlessly, and to maintain science results across a range of parallelization implementations.

Therefore, we developed a three-step approach for developing mini-apps for testing novel programming models that we believe to be applicable to the entire codebase. To illustrate the strategy, we focus on the CanopyHydrology module of ELM, which includes rain and snow canopy interception and storage, the calculation of leaf wet and dry area fractions, and the initialization of new snow layers as new snow is provided to the surface.

First, the CanopyHydrology part of the existing Fortran codebase was *enucleated*; that is, the module was refactored to move all biogeophysical process representations to kernels that are agnostic to the data structure. Enucleation is not a new concept [?]; indeed it forms the core of libraries as wide-spread as the C++ standard library. Despite this, it historically has been rarely adopted by application software, where developers often see the data structure implementation as an integral part of the scientific model implementation. Introducing a formal data model, which allows application developers and computational scientists to co-design an interface to the data structure, is a key first step in the refactoring approach.

The choice of kernel granularity was driven by both data access patterns and the biogeophysics. This refactoring process required significant thought and communication between the application developer and the computational scientist to ensure that kernels were appropriately designed. Critically, kernel interfaces were designed to abstract away the scale hierarchy, which is fundamentally tied to the data structures and memory layout, from the mathematical equations that implement the physical processes. As few assumptions as possible were made about the underlying data layout; the goal is a library of process kernels that are usable in multiple applications with differing assumptions about the scale hierarchy. Listing 1 shows an example before the enucleation processes with the start locations of code that was migrated to kernels identified. Note that we have simplified all code snippets, including omitting details and changing some data layout (from the flattened model used by ELM, which requires more complex and verbose indexing, to a two-dimensional array model), to make the presentation simpler. In addition to those physics kernels, one mathematical kernel is identified representing a

Listing 1. Original Fortran Module

```
module CanopyHydrologyMod
  ! scale hierarchy indexing and data which is constant
  ! across all instances (i.e. the vertical mesh structure
  ! of columns) are module-level "singletons" which are
  ! read-only
  use ColumnType      , only : col_pp
  use VegetationType  , only : veg_pp
  ...
  implicit none
  !-------------------------------------------------------
  subroutine CanopyHydrology(bounds, num_nolakec, &
       filter_nolakec, num_nolakep, filter_nolakep, &
       atm2lnd_vars, canopyst_vars, temptrst_vars, &
       aerosol_vars, waterstate_vars, waterflux_vars)
    ! filters and bounds control indexing of loops
    type(bounds_type)    , intent(in)    :: bounds
    integer              , intent(in)    :: num_nolakec
    integer              , intent(in)    :: filter_nolakec(:)
    integer              , intent(in)    :: num_nolakep
    integer              , intent(in)    :: filter_nolakep(:)
    ! state variables, structs-of-arrays
    type(canopy_st_type) , intent(in)    :: canopyst_vars
    type(water_flx_type) , intent(inout) :: waterflux_vars
    ...
    ! associate variables in scale hierarchy and state
    associate( &
         pgridcell      => veg_pp%gridcell, &
         pcolumn        => veg_pp%column, &
         elai           => canopyst_vars%elai_patch, &
         qflx_snow_pft  => waterflux_vars%qflx_snow_pft, &
         qflx_snow_col  => waterflux_vars%qflx_snow_col, &
         h2osoi_ice_liq => waterstate_vars%h2osoi_ice_liq, &
         ...
      )
    ! Start PFT-level loop
    do f = 1, num_nolakep
       ! filters select based on underlying physical type
       p = filter_nolakep(f)
       ! indices across the scale tree
       g = pcoloumn(g)
       ! more filters based upon physics
       if (ltype(l)==istsoil) then
          ! physical processes begin here
          ! kernel: Interception
          ! =======================================
          h2ocanmx = dewmx(p) * (elai(p) + esai(p))
          qflx_snow_pft(c) = forc_snow(c) * ...
       end if
       ...
    end do ! (end pft loop)
    !
    ! integrate PFT-level variables to column-level
    ! kernel: reduction
    call p2c(bounds, num_nolakec, filter_nolakec, &
         qflx_snow_pft(bounds%begp:bounds%endp), &
         qflx_snow_col(bounds%begc:bounds%endc))
    ! Start column-level loop
    do f = 1, num_nolakec
       ! filters select based on underlying physical type
       c = filter_nolakec(f)
       ! indices across the scale tree
       g = cgridcell(c)
       ! kernel: SnowWater
       snow_level = ...
       h2osoi_liq(j,snow_level) = ...
       ...
    end do
  end associate
  end subroutine CanopyHydrology
end module CanopyHydrologyMod
```

reduction operation. The enucleation process for this example produced several physics kernels and the reduction kernel.

Second, module-level regression tests using the extracted kernels were designed and implemented. These tests consisted of a complete, stand-alone driver which loads input data previously saved from a full-system run and writes output data

Listing 2. Refactored Kernel Implementation

```
template<typename Array_t>
void CanopyHydrology_SnowWater(const double& dtime,
  const int& ltype,
  const int& ctype,
  int& snow_level,
  double& snow_depth,
  Array_t swe_old,
  ...)
{
  snow_level = ...
  swe_old[snow_level] = ...
}
```

for prognostic variables which can be tested for regression. The approach of extracting the needed data to drive a single module has previously been identified as a viable strategy to expand ELM's testing framework [16]. The granularity of the regression tests fits the layout of the original CanopyHydrology module, which comprises a few computational kernels executing a complete set of process physics for a single sub-system. A separate, stand-alone python test harness was written to compare output to the gold standard, enabling tests in multiple programming languages and models. In addition to the original extracted Fortran kernels, the kernels and test driver were also converted to C++.

Third, a series of mini-apps using different programming models was implemented. Template metaprogramming was used for the C++ versions of the kernels to facilitate the implementation in different programming models. Several kernels accept arrays of doubles and other primitives; by templating the kernel on array type (see Listing 2), the kernels are able to accept "array-like" objects including `Kokkos::View<>`, `Legion::FieldAccessor<>`, and a simple container based on `std::array<>`. The requirements on the container are simply that it provides an accessor `operator[]` and that it have view semantics, i.e. that copy construction provides a shallow view of the data, not a deep copy. The latter allows temporary values such as subviews to be passed to the kernel (as opposed to accepting the array by non-const reference, which would require first storing an lvalue). This allows a single implementation of the kernel to be used by a variety of programming models, while being identically used by simple, CPU-only implementations for testing. Much like we wish to abstract kernel implementation from kernel execution, we wish to abstract kernel testing from kernel execution testing; this design supports that goal. The developed mini-apps included multiple languages for both the kernel and mini-app driver, including Fortran-only, C++ only, Kokkos, and Legion (see Table III). The remainder of the paper describes our experience in adopting Kokkos and Legion, their suitability for this application, and their ease of adoption.

IV. Performance portability through Kokkos

Demands of predictive understanding of land surface processes across the globe requires an immense amount of computational power. Environmental conditions on the land surface vary across scales, and gradients in things like water table depth and vegetative land cover vary over scales of meters to

TABLE I
IMPLEMENTED AND NOT-IMPLEMENTED COMBINATIONS OF KERNEL LANGUAGE AND MINI-APP PROGRAMMING MODEL

	Fortran kernel	C++ kernel
Fortran mini-app	✓	∅[1]
C++ mini-app	✓	✓
Kokkos OpenMP	✓	✓
Kokkos Cuda	∅	✓
Legion mini-app	∅[2]	✓

1. While this combination is clearly possible, it was not of interest. 2. This combination is possible through accessing Legion's raw pointers and ensuring that mapping from geometric region to physical region is done under a known, specific layout.

kilometers. Next-generation land surface models must leverage novel architectures in order to achieve sufficient performance in terms of "model years per wallclock hour," the preferred metric for climate code performance.

As architectures become more heterogeneous in terms of both processing units and memory, ensuring reasonable performance at large scale across a range of architectures becomes increasingly difficult. Critically, device and memory are interwoven; efficient performance requires data layout that is consistent with the expectations of the device. Kokkos is a programming model designed to abstract memory and device, allowing users to write code that can, through the use of template specializations for individual lower-level libraries and architectures, achieve some level of performance portability across architectures [3], [7], [17]. Kokkos is designed to target complex node architectures with N-level memory hierarchies and multiple types of execution resources. It currently can use OpenMP, Pthreads and CUDA as backend programming models.

A. *Wrapping ELM kernels within Kokkos*

The five ELM kernels identified were straightforwardly wrapped within two Kokkos kernel launches, one corresponding to the parallelism across PFTs and then restricted to columns via a parallel reduction, and the other a separate launch across columns (see Listing 3). The former demonstrates Kokkos' ability to represent hierarchical parallelism. First, thread teams are constructed, one for each water column. Each team does a parallel reduction, executing two kernels and performing the restriction operation.

Both Fortran and C++ implementations of the underlying kernels were successfully wrapped for use in Kokkos. To wrap the Fortran kernels, the `ISO_C_BINDING` intrinsic module was used to ensure primitive data types. Then, for kernels that must be passed arrays of memory, `Views` with specified layout (`LayoutRight`) data access are used to ensure that subviews of the 1D slice of 2D data are known to be contiguous in memory. This allows the kernel to pass a raw pointer to the Fortran kernel. We note that this eliminates Kokkos's ability to lay out arrays in memory to match the memory model of the device. This approach would hurt performance on a machine whose device preferred coalesced memory access patterns. However, as the kernels themselves were implemented in

Listing 3. Kokkos Driver
```
// view instantiation, 1D
typedef View<double*>  Vector_t;
Vector_t forc_snow("forc_snow", n_columns);
Vector_t qflx_snow_col("qflx_snow", n_columns);
// view instantiation, 2D
typedef View<double**>  ViewMatrixType;
ViewMatrixType elai("elai", n_columns, n_pfts);
ViewMatrixType qflx_snow_pft("qflx_snow_pft",
                             n_columns, n_pfts);
ViewMatrixType h2osoil_liq("h2osoi_liq",
                             n_columns, n_lev_soil);
...
// Data initialization, etc
...
// evaluate Interception and reduction kernels
typedef TeamPolicy<> team_policy;
typedef typename team_policy::member_type team_type;
parallel_for(team_policy(n_columns, AUTO()),
    KOKKOS_LAMBDA(const team_type& team)
{
  size_t c = team.league_rank() ;
  double sum = 0;
  parallel_reduce(TeamThreadRange (team, n_pfts),
      [=] (const size_t& p, double& lsum) {
          ELM::CanopyHydrology_Interception(
                  forc_snow(c), elai(c,p),
                  qflx_snow_pft(c,p), ...);
          lsum += qflx_snow_pft(team.league_rank(),p);
        }, sum);
  qflx_snow_col(team.league_rank()) = sum ;
}
// evaluate SnowWater kernel
parallel_for(n_columns,
    KOKKOS_LAMBDA(size_t c)
{
  ELM::CanopyHydrology_SnowWater(qflx_snow_col(c),
          subview(h2osoi_liq,c,ALL), ...);
}
```

Fortran, no effort was made to get this combination working on GPUs. Alternatively, the C++ kernels were implemented and tested on both cached and coalesced memory access patterns, on multi-threaded CPU-only and GPU architectures.

B. *Observations on the use of Kokkos*

Kokkos seems a sufficiently mature programming model for adoption by application developers with limited device porting experience. Bootcamps are increasingly available, and one training session was sufficient to understand appropriate design patterns. Examples, slides, and the Kokkos wiki provide support and documentation for simple usage. The development team of Kokkos is thriving, and clearly making efforts to support the adoption of the code by a wide range of application developers.

Kokkos enables performance portability by abstracting an application's data and data layout from the execution of functions on that data. This abstraction encourages the application developer to separate what is being executed from how and where it will be executed. We found that this data model supports developers in designing code that looks more like the underlying math and physics and is more reusable across applications. For instance, one goal of this work is the ability to treat ELM physics capability as a library of mathematical functions that are independent of the data model upon which those functions are applied. ELM's hierarchical data structure (Figure 1) assumes water columns cover scales much larger than the scale of vegetation variability. Therefore, the column contains multiple plant functional types, each assigned a fraction of the surface area of the water column. Alternatively, regional and local land surface models may resolve water at a much higher resolution, at which point only a single plant functional type per water column is appropriate. While the evaluation process and loops must be changed, the underlying physical kernels are still appropriate. By abstracting the data model from the kernel, kernels can be reused by different models, each with their own data model. While this is certainly possible without Kokkos, the data model of Kokkos encourages more flexible application design.

In this work, we found that the flexibility of generating different types of subviews to be useful. For instance, subviews which select a single row or column of two-dimensional arrays were used to pass one-dimensional arrays to kernels. Subviews can also be used to take a subset of a row or column; this will be useful in dealing with the flattened hierarchy described in Figure 2. Finally, `OffsetViews` offer the ability to, like Fortran, change the range of indices into an array. The ability to arbitrarily set the index of the "0th" entry in an array is used extensively in physics code in general and ELM in particular. For instance, water column data in ELM ranges from `-n_level_snow` to `n_level_soil`, with non-positive indices indicating "above ground" cells and positive indices indicating "below ground" cells. While this was not leveraged here in order to allow consistent kernels across Legion and Kokkos, `OffsetViews` offer great potential to ease refactoring Fortran code for C++ and Kokkos.

One missing capability that would be useful in this application is the ability to construct "filtered" or "random access" subviews. Often ELM code evaluates loops over a "filtered" set of columns, e.g. "for each column in a land unit that is of type `VEGETATED`." While it is unclear whether forming the subviews would enable architecture optimizations relative to the current approach of introducing an **`if`** statement at the beginning of each kernel, it would provide a more concise and flexible abstraction. Furthermore, while "ragged" two-dimensional array views seem to have been implemented via the Kokkos kernels package for use in compressed-row-storage spare matrices, they are not a part of Kokkos proper. Some documentation and better cross-referencing across multiple efforts in the Kokkos community would make the adoption of Kokkos easier for application developers.

V. Load balancing and task parallelism through Legion

One of the difficulties inherent in Earth System modeling in general and land surface models specifically is their sheer complexity. Processes include aspects of physics, biology, chemistry, economics, and social sciences. This rich set of processes are required to capture even the broad strokes of the evolution of the Earth's climate.

Task parallel programming models break up this complexity into a number of smaller tasks and their dependencies, then automate the formation and executiom of the associated dependency graph. The dependency graph, a directed, acyclic graph which describes both data dependencies and execution order, exists implicitly in all applications, but is typically mapped to computational resources explicitly by the developer. Task parallel programming models acknowledge that, in complex applications like land surface models, it is difficult if not impossible for a single developer to understand and reason about this graph. By providing tools for explicitly forming the dependency graph, the task scheduler can then identify opportunities for parallelism that may not have been recognized by the developer. As applications become more complex, the ability to recognize and exploit all available parallelism is crucial to ensure strong scaling. Thanks to this unique ability to expose parallelism in automated ways, task parallel runtimes are becoming more common in multiphysics applications [18]–[21].

The Legion Programming System is a task-based, data-centric programming model which provides abstractions to describe dependencies and locality of program data, along with a runtime, Realm, which asynchronously schedules and maps tasks and data onto devices and physical memory spaces. Legion provides a number of features which makes it appealing for current-day, complex applications. Most notably, it facilitates a data model, the `LogicalRegion`, from which persistent data can be mapped. The `LogicalRegion` combines a multidimensional indexed space (or listing of data entries) with a field space (or listing of variable names and types). It may be partitioned in a variety of ways, and each field and partition can then be the target of a data dependency. Tasks are added to the dependency graph with an arbitrary set of arguments and a collection of permissions (e.g. read-only, read-write, etc) on the `LogicalRegion`'s fields, and provide an arbitrary collection of return types as a `Future`, or handle for ensuring the completion of the task. Permissions can be extremely fine grained, depending upon a partition and a subset of the field space.

Tasks are added to the dependency graph in apparently sequential order which implies dependencies; the task runtime allocates resources to respect these dependencies and permissions [5], [22], [23]. The `LogicalRegion` is mapped onto (distributed) memory to form `PhysicalRegions` which are persistent across tasks, but can be moved across devices as necessary by the scheduler. The management of this mapping is the purview of the runtime, but can be controlled and guided by the application as needed. Mapping and therefore performance is independent of correctness, allowing the developer to focus on one at a time.

A. *Considerations of Legion for ELM*

Practically, Legion is a programming model that replaces MPI, but places stronger requirements on an application's data model. Specifically, as the scheduler must be able to instantiate the `PhysicalRegion` and move partitions of the `PhysicalRegion` on and off of various devices, applications may not control (persistent) data, instead describing the data within the confines of the `LogicalRegion`. ELM's existing data model (Figure 2), which uses integer indices to map between levels in the hierarchy, is well-suited for Legion's data model. Alternative approaches, such as pointers across levels of the hierarchy, would have been much more difficult to refactor for Legion. The key to ensuring limited communication and efficient code in Legion, then, is to ensure that partitioning is specified to match the access pattern of each task.

For ELM, this means that restrictions across hierarchy levels are most efficiently done if all needed data for a given restriction is in the same part of a partition. Furthermore, Legion's data parallelism and partitions are conceptually "coarse-grained." Together, these suggest our approach for this mini-app, which is that only the top level (grid cells) are partitioned, and each level of the hierarchy is partitioned consistently. Naive, independent partitioning of each level would result in a much denser data dependencies, as each restriction could depend on multiple chunks in the partition. Consistent partitioning ensures that reductions across the scale hierarchy can depend on only a single chunk, and careful mapping can ensure this is done locally on a single device without data movement, as would occur in a typical MPI implementation.

This partitioning is an over-decomposition; careful thought will be required at scale to ensure that the chunk size is large enough to hide Legion's overhead but small enough to allow efficient load balancing. Unlike a typical MPI implementation, over-decomposition of the grid cell level allows opportunities for load balancing to occur without user intervention.

B. Wrapping ELM kernels within Legion

To explore the use of Legion within ELM, a mini-app that passes regression tests was developed. The above set of C++ kernels was wrapped as Legion tasks and the mini-app was implemented within a "top level task" (see Listing 4). This task constructs `LogicalRegions` for the ELM data model, including partitioning consistent across levels of the hierarchy. It then loops in time, sets up permissions, and launches tasks for each kernel. An example of the task implementation of the restriction task is shown in Listing 5. The tasks create accessors of the correct type, then loop over each element (i.e. column or PFT) and call the kernel.

We note that the templated kernel can match "subviews" into the Legion accessor, allowing the same templated kernel to match Legion accessors and Kokkos views/subviews.

C. Observations on the usability of Legion

First, we note that Legion provides a rich set of tutorial material, documentation, mailing list, and community support. In our experience, the development team was responsive and friendly, and happy to explain and help new users. A huge amount of recorded video including Legion tutorials are available. This community support suggests a thriving community, which makes any code base both easier and more fun to develop.

Furthermore, we were pleased to find that Legion discovered potential concurrency that was not obvious, even in such a simple mini-app. The dependency graph constructed dynamically by the Legion runtime and shown in Figure 3 makes clear that the `CanopyHydrology_FracWet` and `SumOverPFTs` restriction tasks can be executed independently. We find it quite promising that, even within such a limited fraction of the overall code base (less than 10% of all anticipated tasks) and even within a single module which one would expect code to be more tightly co-dependent, concurrency exists. While it is likely that these concurrencies could be discovered by a careful analysis of the code, this potential is a powerful result of the adoption of Legion.

Listing 4. Legion Top Level Task

```cpp
void top_level_task(const Task *task,
      const std::vector<PhysicalRegion> &regions,
      Context ctx, Runtime *runtime)
{
  // canopy_state_vars
  auto phenology_fs_names =
      std::vector<std::string>{ "elai", ... };
  Data<2> phenology("phenology", ctx, runtime,
                    Point<2>(n_columns_g, n_pfts),
                    Point<2>(n_parts,1),
                    phenology_fs_names);
  // water_flx_type variables on columns
  auto flux_col_names = std::vector<std::string>{
    "forc_snow", "qflx_snow_col", ... };
  Data<1> water_flx_col("flux", ctx, runtime,
                Point<1>(n_columns_g),
                Point<2>(n_parts),
                flux_col_names);
  // water_flx_type variables on PFTs
  auto flux_pft_names = std::vector<std::string>{
    "qflx_snow_pft", ... };
  Data<2> water_flx_pft("flux", ctx, runtime,
                Point<2>(n_columns_g, n_pfts),
                Point<2>(n_parts,1),
                flux_pft_names);
  // soil variables on columns x soil cells
  auto soil_fs_names = std::vector<std::string>{
    "h2osoi_liq", ... };
  Data<2> soil("soil", ctx, runtime,
                Point<2>(n_columns_g, n_lev_soil),
                Point<2>(n_parts,1),
                soil_fs_names);
  ...
  // Data initialization, etc
  ...
  // create a color space for indexed launching.
  auto color_space = surface.color_domain;
  // launch Interception evaluation task
  CanopyHydrology_Interception().launch(ctx, runtime,
          color_space, phenology, water_flx_pft,
          water_flx_col);
  // launch reduction task
  SumOverPFTs().launch(ctx, runtime, color_space,
                      water_flx_pft, "qflx_snow_pft",
                      water_flx_col, "qflx_snow_col");
  // launch SnowWater evaluation task
  CanopyHydrology_SnowWater().launch(ctx, runtime,
          color_space, water_flx_col, soil);
}
```

Legion is, by intention, a low-level programming model. Significant amounts of boilerplate code is repeated in multiple places and in multiple forms to specify the same thing. For instance, the order of regions within the task must be specified in at least two places as a task's data is packed and unpacked: once at the launch of the task (i.e. adding the dependency to the graph), and once within the task itself (i.e. constructing an accessor of the correct type). Each of these is an opportunity for programmer error, and these "consistency bugs" are typically caught only at runtime.

Listing 5. Legion SumOverPFTs Task

```
FutureMap
SumOverPFTs::launch(Context ctx, Runtime *runtime,
        Rect<1>& color_space,
        Data<2>& pft, const std::string& summand,
        Data<1>& col, const std::string& sum)
{
  auto args = std::make_tuple(NULL);
  ArgumentMap arg_map;
  IndexLauncher launcher(taskid, color_space,
      TaskArgument(&args, sizeof(args)), arg_map);

  // -- permissions on input
  launcher.add_region_requirement(
      RegionRequirement(pft.logical_partition,
            pft.projection_id, READ_ONLY, EXCLUSIVE,
            pft.logical_region));
  launcher.add_field(0,flux.field_ids[summand]);

  // -- permissions on output
  launcher.add_region_requirement(
      RegionRequirement(col.logical_partition,
            col.projection_id, WRITE_DISCARD,
            EXCLUSIVE, col.logical_region));
  launcher.add_field(1,surface.field_ids[sum]);

  // -- launch the interception
  return runtime->execute_index_space(ctx, launcher);
}

void
SumOverPFTs::cpu_execute_task(const Task *task,
      const std::vector<PhysicalRegion> &regions,
      Context ctx, Runtime *rt)
{
 // get accessors
  using AffineAccessorRO2 = FieldAccessor<READ_ONLY,
        double,2, coord_t,
        Realm::AffineAccessor<double,2,coord_t> >;
  using AffineAccessorWO1 = FieldAccessor<WRITE_DISCARD,
        double,1, coord_t,
        Realm::AffineAccessor<double,1,coord_t> >;
  const AffineAccessorRO2 summand(regions[0],
        task->regions[0].instance_fields[0]);
  const AffineAccessorWO1 sum(regions[1],
        task->regions[1].instance_fields[0]);

  LogicalRegion lr = regions[0].get_logical_region();
  IndexSpaceT<2> is(lr.get_index_space());
  Rect<2> bounds = Domain(rt->get_index_space_domain(is));

  for (int g = bounds.lo[0]; g != bounds.hi[0]+1; ++g) {
    double sum_l = 0 ;
    for (int p = bounds.lo[1]; p != bounds.hi[1]+1; ++p) {
      sum_l += summand[g][p];
    }
    sum[g] = sum_l ;
  }
}
```

Furthermore, Legion data models require the composition of many conceptually independent classes. For example, constructing a partition of a `LogicalRegion` requires an `IndexSpace`, `FieldSpace`, `LogicalRegion`, `IndexPartition`, `FieldAllocator`, and `LogicalPartition`, along with names and IDs of each of these to track for debugging. This abstraction provides a richness of features that is very welcome, but conceptually these objects can easily be encapsulated under a common utility. Legion makes no attempt to hide the fact that it is a low-level model – higher level abstractions are realistically required to ensure efficient code development.

Instead, Legion is co-developed with the domain specific language Regent [24], which enforces consistency in the language design. However, developing application code in Regent comes with downsides. Specifically, interoperability of Regent with legacy tasks (written in C++) and embedding task scheduling within legacy control code (written in C++) is difficult if possible. Other approaches, such as Flexible Computational Science Infrastructure (FleCSI) [21], offer a complete abstraction layer for multiphysics applications, for which Legion is one backend. This is an intriguing possibility for future applications, but again is not well-suited for refactoring existing applications as it requires the complete adoption of the FleCSI framework, including mesh data abstractions.

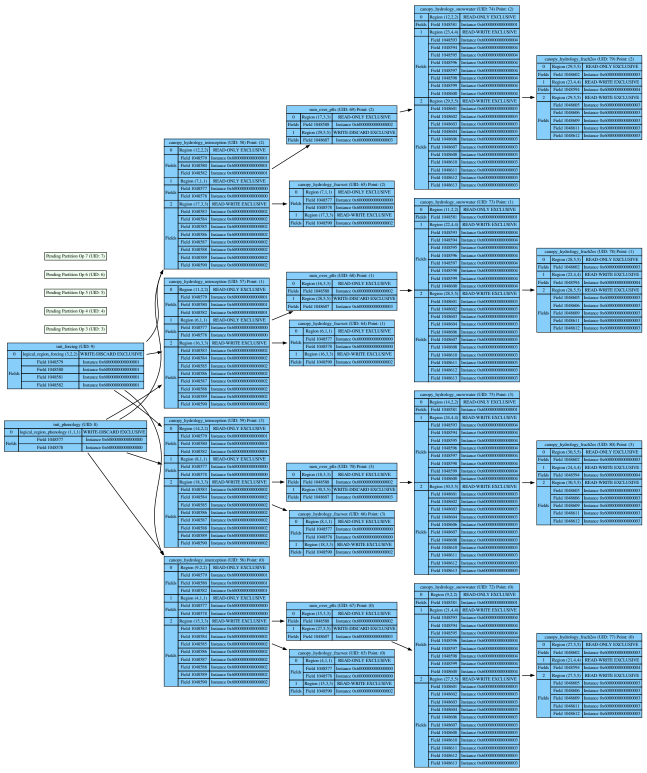
Fig. 3. Legion event graph of the four kernels and one reduction task, each split via indexed launch into four chunks. Common tasks on different chunks are colored the same. Potential concurrency was exposed in the third column, where the dependency graph shows how the `CanopyHydrology_FracWet` and `SumOverPFTs` tasks are independent. Note that, to simplify the presentation, this graph has been stripped of all setup and I/O and only one time step is shown.

Instead, it seems likely that a significant amount of the boilerplate required by Legion could be hidden in a middleware layer, enabled by C++ template metaprogramming, which would be useful to a wide range of applications. Indeed, even for this simple exercise, a data class to encapsulate `LogicalRegions` and their `LogicalPartitions` was implemented straightforwardly. While it is unlikely that such an approach will hide all of Legion's complexity, additional helper classes in C++ to encapsulate permissions, accessor construction, and other common low-level boilerplate would be a welcome addition to the Legion community landscape.

VI. Conclusions and Discussion

In this work we looked to develop an approach for refactoring a complex MPI-only code for hybrid architectures and extreme parallelism. We described an enucleation process by which kernels from the Energy Exascale Earth System Model's Land Model were isolated from their calling code, and packaged as a separate suite of physics process functions for use in a variety of applications. We showed how this approach led to a robust, testable capability that could be used with a variety of programming models. We then implemented a series of mini-apps, exploring the use of Kokkos to abstract architecture and Legion to identify concurrency and schedule tasks across resources. While not a new concept, this enucleation approach is not typical of application software, but provides multiple advantages for both adoption new programming models and for application software design. Libraries that encourage application software development in this form are emerging [**?**], [19], [21], and may become more common.

In this work, we found that Kokkos and Legion are two good examples of the maturation of programming models. Both benefit from a culture of developer-user interaction and support, and adoption is supported by a wide range of tutorials and introductions to enable application developers to understand the underlying concepts. While both would benefit from more complete documentation and a richer set of examples, both are ready for adoption by application developers. Both encourage the development of clear interfaces for physical processes which are data-structure agnostic, and the explicit identification of a data model and data access patterns. By exploring multiple programming models simultaneously, we feel comfortable that our approach has some robustness to ensure that investments in refactoring will not be premature.

While it is unclear what specific programming models for scientific applications will stand the test of time, it seems clear that some features of those models have emerged. Application developers would benefit from refactoring efforts to adopt some of these features. Specifically, future application codes must look more functional and less object-oriented than most codes today, with explicit data models which are abstracted from the mathematical processes evaluated on that data. Dependencies must be more explicit; by being explicit about limiting permissions and interfaces to only data that is actually used (as opposed to passing coarse-grained data structures), programming models will be able to infer and optimize the execution of these processes. By pushing unit and system level tests to finer granularity, application developers will gain invaluable knowledge about the appropriate granularity of processes while building the needed infrastructure to support refactoring efforts. Finally, we find that these changes will encourage more flexible, robust code which better enables scientific exploration of process uncertainty and better serves the science community.

While programming models for pushing "beyond MPI+X" are an active and dynamic research area, we find them to be rapidly maturing to the point that application development teams without programming model experts should feel empowered to explore and embrace them.

Acknowledgment

This work was supported by the US Department of Energy's Office of Science's Scientific Discovery through Advanced Computing (SciDAC) project, Coupling Approaches for Next-Generation Architectures (CANGA). This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

References

[1] H. T. Hewitt, M. J. Bell, E. P. Chassignet, A. Czaja, D. Ferreira, S. M. Griffies, P. Hyder, J. L. McClean, A. L. New, and M. J. Roberts, "Will high-resolution global ocean models benefit coupled predictions on short-range to climate timescales?" vol. 120, pp. 120–136. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1463500317301774

[2] P. E. Thornton, K. Calvin, A. D. Jones, A. V. Di Vittorio, B. Bond-Lamberty, L. Chini, X. Shi, J. Mao, W. D. Collins, J. Edmonds, A. Thomson, J. Truesdale, A. Craig, M. L. Branstetter, and G. Hurtt, "Biospheric feedback effects in a synchronously coupled model of human and earth systems," vol. 7, no. 7, pp. 496–500. [Online]. Available: http://www.nature.com/articles/nclimate3310

[3] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *2013 Extreme Scaling Workshop (xsw 2013)*, pp. 18–24.

[4] L. Bertagna, M. Deakin, O. Guba, D. Sunderland, A. M. Bradley, I. K. Tezaur, M. A. Taylor, and A. G. Salinger, "HOMMEXX 1.0: a performance-portable atmospheric dynamical core for the energy exascale earth system model," vol. 12, no. 4, pp. 1423–1441. [Online]. Available: https://www.geosci-model-dev.net/12/1423/2019/

[5] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11.

[6] M. F. P. Bierkens, V. A. Bell, P. Burek, N. Chaney, L. E. Condon, C. H. David, A. d. Roo, P. Dll, N. Drost, J. S. Famiglietti, M. Flrke, D. J. Gochis, P. Houser, R. Hut, J. Keune, S. Kollet, R. M. Maxwell, J. T. Reager, L. Samaniego, E. Sudicky, E. H. Sutanudjaja, N. v. d. Giesen, H. Winsemius, and E. F. Wood, "Hyper-resolution global hydrological modelling: what is next?" vol. 29, no. 2, pp. 310–320. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/hyp.10391

[7] H. C. Edwards and D. Sunderland, "Kokkos array performance-portable manycore programming model," in *Proceedings of the 2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '12. ACM, pp. 1–10, event-place: New Orleans, Louisiana. [Online]. Available: http://doi.acm.org/10.1145/2141702.2141703

[8] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and 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 Press, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2676870.2676883

[9] L. V. Kale and S. Krishnan, "CHARM++: A portable concurrent object oriented system based on c++," in *In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pp. 91–108.

[10] J. D. d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: a massively parallel problem solving environment," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, pp. 33–41.

[11] G. Aupy, M. Faverge, Y. Robert, J. Kurzak, P. Luszczek, and J. Dongarra, "Implementing a systolic algorithm for QR factorization on multicore clusters with PaRSEC," in *Euro-Par 2013: Parallel Processing Workshops*, ser. Lecture Notes in Computer Science, D. an Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, Eds. Springer Berlin Heidelberg, pp. 657–667.

[12] M. Feathers, *Working Effectively with Legacy Code*. Prentice Hall Professional, google-Books-ID: fB6s_Z6g0gIC.

[13] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, google-Books-ID: 2H1_DwAAQBAJ.

[14] U. Yang, R. Bartlett, G. Hammond, X. Li, B. Smitth, and J. Willenbring. How to add and improve testing in your CSE software project. [Online]. Available: https://ideas-productivity.org/wordpress/wp-content/uploads/2016/04/IDEAS-TestingHowtoAddImproveTestinginyourCSESoftwareProject-V0.2.pdf

[15] X. Yang, P. E. Thornton, D. M. Ricciuto, and F. M. Hoffman, "Phosphorus feedbacks constraining tropical ecosystem responses to changes in atmospheric CO2 and climate," vol. 43, no. 13, pp. 7205–7214. [Online]. Available: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1002/2016GL069241

[16] D. Wang, Y. Xu, P. Thornton, A. King, C. Steed, L. Gu, and J. Schuchart, "A functional test platform for the community land model," vol. 55, pp. 25–31. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1364815214000255

[17] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," vol. 74, no. 12, pp. 3202–3216. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0743731514001257

[18] L. Kal, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, "NAMD2: Greater scalability for parallel molecular dynamics," vol. 151, no. 1, pp. 283–312. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0021999199962010

[19] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. A. Wight, and J. R. Peterson, "Uintah: A scalable framework for hazard analysis," in *Proceedings of the 2010 TeraGrid Conference*, ser. TG '10. ACM, pp. 3:1–3:8, event-place: Pittsburgh, Pennsylvania. [Online]. Available: http://doi.acm.org/10.1145/1838574.1838577

[20] S. Treichler, M. Bauer, A. Bhagatwala, G. Borghesi, R. Sankaran, H. Kolla, P. S. McCormick, E. Slaughter, W. Lee, A. Aiken, J. Chen, M. Bauer, A. Bhagatwala, G. Borghesi, R. Sankaran, H. Kolla, P. S. McCormick, E. Slaughter, W. Lee, A. Aiken, and J. Chen. S3d-legion : An exascale software for direct numerical simulation of turbulent combustion with complex multicomponent chemistry. [Online]. Available: https://www.taylorfrancis.com/

[21] J. Loiseau, H. Lim, B. K. Bergen, N. D. Moss, and F. Alin, "FleCSPH: a parallel and distributed smoothed particle hydrodynamics framework based on FleCSI," in *2018 International Conference on High Performance Computing Simulation (HPCS)*, pp. 484–491.

[22] S. Treichler, M. Bauer, and A. Aiken, "Language support for dynamic, hierarchical data partitioning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. ACM, pp. 495–514, event-place: Indianapolis, Indiana, USA. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509545

[23] A. Aiken, M. Bauer, and S. Treichler, "Realm: An event-based low-level runtime for distributed memory architectures," in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 263–275.

[24] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A high-productivity programming language for HPC with logical regions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. ACM, pp. 81:1–81:12, event-place: Austin, Texas. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807629

Appendix

Artifact Description/Artifact Evaluation

Summary of the experiments reported

A series of mini-apps which leverage a common library of application code and a variety of programming models were described. Design and testing of these mini-apps were reported.

Artifact Availability

Software Artifact Availability:: All author-created software artifacts are maintained in a public repository under an OSI- approved license (BSD-3 clause).

Hardware Artifact Availability:: There are no author-created hardware artifacts.

Data Artifact Availability: : There are no author-created data artifacts.

Proprietary Artifacts:: None of the associated artifacts, author-created or otherwise, are proprietary.

List of URLs and/or DOIs where artifacts are available:: https://code.ornl.gov/uec/elm_kernels/

Consideration for SCC:

No.

Baseline experimental setup, and modifications made for the paper

Operating systems and versions:: Tested on Ubuntu, OSX

Compilers and versions:: Tested on Gnu compiler collection (v7+), Apple Clang compilers (clang-1000.10.44.4)

Applications and versions:: As presented in hash `1b8a59c66de13200e79807eaf93875db0faf6baa`.

Libraries and versions::

- Kokkos, as built from Trilinos, https://github.com/trilinos/Trilinos, hash `a5f7683012ec5567679a5d065f91dcbe553c24a8`.
- Legion, https://github.com/StanfordLegion/legion, as built from hash `253ee9e3a44f3c4843b4e83c6b8e37da1651cfad`.

Artifact Evaluation

Verification and validation studies:: Regression tests for correctness of all mini-apps relative to original refactored application, as described in the manuscript.