LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-661403

The RAJA Portability Layer: Overview and Status

R. D. Hornung, J. A. Keasler

September 24, 2014

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

(U) The RAJA Portability Layer: Overview and Status

Richard D. Hornung, Jeffrey A. Keasler, et al.

Lawrence Livermore National Laboratory, Livermore, CA

Abstract

As computer architectures become increasingly complex and diverse, application developers face difficult challenges to achieve high performance while maintaining code portability. The problem is especially acute for large ASC multiphysics codes. Efficient parallel execution often requires tuning algorithms and data access to match processor and memory system constraints. Changing compiler directives and parallel programming model constructs on thousands of individual loops in a large code is disruptive and unwieldy. RAJA is a programming approach that we have been developing at Lawrence Livermore National Laboratory to encapsulate platform-specific concerns, related to both hardware and parallel programming models. The RAJA abstraction layer simplifies porting C/C++ codes to various programming models and architectures by reducing effort and developer disruption. In this report, we motivate and describe key aspects of RAJA. We also present a preliminary assessment of RAJA based on exploration in three ASC hydrodynamics codes at LLNL, which was one part of a three-part ASC Level 2 milestone, completed in September 2014.

Introduction

Over the past couple of decades, HPC application performance has improved dramatically with advances in computer architectures and CPU clock rates. During this period, hardware platforms have remained relatively homogeneous and consistent. Thus, application scientists have been able to focus on algorithm development and coarse-grained parallelism (mostly MPI) with little concern for fine-grained parallelism and a detailed understanding of hardware variations across platforms. The performance and portability challenges now facing ASC codes are rooted in recent, disruptive changes to HPC node architectures. Exploiting the full range of hardware capabilities forces developers to express ample fine-grained parallelism in varied forms, such as SMT (Simultaneous Multithreading), SIMT (Single Instruction, Multiple Threads), and SIMD (Single Instruction, Multiple Data). Also, careful management of data locality and memory access is becoming paramount with the emergence of deeper memory and cache hierarchies with different sizes and access timing rules.

To achieve high performance portably, ASC codes need to adopt algorithms and programming styles that can express various forms of parallelism, that do not overburden code maintainability, and which can be explored incrementally. RAJA is designed to integrate with legacy codes simply and to provide a model for development of new codes that are portable from inception. Basic insertion of RAJA in a code enables “zeroth-order” architecture portability. Once in place, a wide range of architecture-specific tuning optimizations can be pursued without substantial application code disruption.

RAJA is based on standard C++ language features, which works well with LLNL ASC codes most of which use C/C++ as their main programming language. The fundamental conceptual abstraction in RAJA is an inner loop, where the overwhelming majority of computational work in most physics codes occurs. RAJA is lightweight, customizable, and based on

concepts used heavily in LLNL codes. It can be added incrementally and used selectively, which facilitates exploration of implementation alternatives. Finally, RAJA can encapsulate different programming models so a code need not be bound to a particular technology.

The main goal of the RAJA portion of the ASC Level 2 milestone was to assess whether RAJA is a viable approach for architecture portability in LLNL ASC codes. To perform this assessment, we considered a variety of questions about RAJA related to several concerns:

- Performance
  - Does RAJA benefit or hinder performance? By how much?
  - When does it work well and when does it not?
- Portability
  - Does RAJA enable portability across current architectures?
  - Does it simplify access to various forms of parallelism?
- Programmability
  - How easy is it to adapt a code to the RAJA model?
  - Does it provide the features we need?
  - Does it enhance or hinder code flexibility and maintenance?
- Long-term viability
  - What are the benefits and limitations of adopting the RAJA approach, or a similar model?
  - What are the expectations to resolve issues that are not under our direct control?
  - What are the prospects for RAJA to adapt to future architectures and programming models?
  - Are there any showstoppers?

Background and Motivation

In this section, we provide background and motivation for RAJA. We begin by describing central, common, concepts employed in mesh-based numerical operations in LLNL ASC physics codes. Then, we discuss challenges related to exposing high performance fine-grained parallelism in physics algorithms.

Data and algorithm organization in physics codes

A typical ASC code has clearly-defined mesh and data abstractions. Generally, a problem is decomposed and distributed across the nodes on a partition of a parallel HPC platform. Each compute node is considered a distributed memory “locale” to which some number of MPI processes is assigned. A data structure, often called a “domain”, owns a description of part of the mesh and the field data for that mesh part. Exactly one MPI process owns each domain and each process may own multiple domains. The basic elements of a domain structure are illustrated in Figure 1. Data on a domain is disjoint from data on other domains; that is a domain is a “data locality context”, representing the finest level of data partitioning in a code typically. Each mesh field is associated with a fixed centering on the mesh, such as element or nodal, and the data for each field is held in a distinct array. Field arrays are 1-dimensional regardless of the problem dimension. Also, there usually exist other metadata on a domain, for example to map materials to elements, as well as non-mesh data, such as tables of physical data (e.g., material properties, equation of state, etc.) shared by domains on a node.

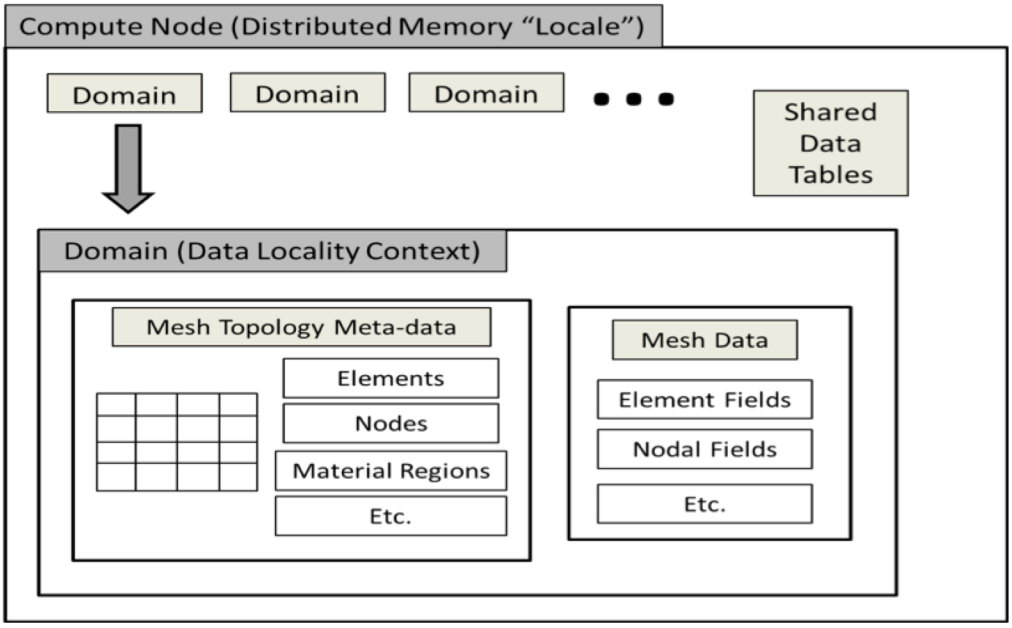

Figure 1. Basic organization of a typical domain structure in a mesh-based physics code.

The mesh topology defines the organization of elements and nodes on the mesh. Generally, there are two fundamental mesh configurations, structured and unstructured. A *structured* mesh uses an N-dimensional Cartesian index space, which defines uniform node connectivity and a single element geometry. Structured mesh algorithms often use nested loops to traverse logically rectangular regions on a mesh. Such operations rely on zero-overhead implicit relationships between mesh entities, and allow a high level of compile-time optimization due to stride-1 data access patterns. An *unstructured* mesh is composed of arbitrarily connected node points that define the elements they surround; thus, an unstructured mesh admits arbitrary element geometries. Due to the irregular connectivity, relationships between unstructured mesh entities are defined using lists of array indices. For example, eight nodes define each element on a three-dimensional hexahedral mesh, so eight nodal-array indices are stored to access nodal field data for each element. Use of indirection arrays to manage relationships among mesh entities requires additional memory traffic, involves a much higher ratio of integer to floating point operations, and precludes many compiler optimizations. Regardless of the underlying mesh topology, most ASC physics codes employ algorithms involving regular, stride-1 memory accesses as well as those requiring indirection arrays. So, efficient implementations of both types of operations are important to every code.

Mesh data is often organized into a *hierarchy of contexts*, typically, where a context represents a relationship between the mesh and data on the mesh. There will be multiple *topological* contexts, one for node-centered quantities, one for element-centered quantities, face-centered quantities, etc. An element context will have child contexts that enumerate the elements associated with each material region. Often, material region contexts are further partitioned into clean elements (single material) and "mixed" elements (containing multiple

materials). When contexts are nested, local indices are typically used within a child context to index into arrays associated with a parent context.

The context hierarchy in an ASC code is designed to map the conceptual organization of physics operations to the underlying data structures. Most physics operations are encoded in loops; a large code will have tens of thousands of loops, typically. However, within a given code, there are relative few *loop patterns*. Common loop patterns involve:

- Simple traversal within a context (e.g., loop over all elements, nodes, etc.)
- "Parent-child" interactions within a topological context (e.g., loop over all elements containing material "A" and update values for some field defined over all elements)
- Relations between fields in different topological contexts (e.g., difference stencils involving node- and element-centered quantities)

Other operations may involve more elaborate data dependencies, but are less common.

Fine-grained parallelism challenges

Presently, numerical kernels in most LLNL ASC codes are usually serial and operate on data associated with an entire domain. However, efficient parallelism is tied closely to memory-locality. One way to improve locality in a multithreaded environment is to use many small domains, allowing more threads to simultaneously share data caches without contention. Unfortunately, domain overhead measured in terms of additional memory needed for non-shared data, and domain management operations that are hard to amortize away, can lead to space or performance problems on current multicore systems. Thus, an alternative to traditional domain partitioning will be required to exploit massive on-processor parallelism.

A better option is to employ fine-grained data "chunking" within a domain where a chunk of data can be assigned to a work thread or passed to an algorithm kernel. Proper chunk size selection can balance both instruction and data cache usage so that neither cache becomes overly strained. For example, if an algorithm works on a single element at a time (typical for a complex material model), the amount of code executed may not fit into an instruction cache. So the *algorithm* is always streamed from main memory (as though there is no instruction cache), while the data may be perfectly cached, with room to spare. On the other hand, if the chunk size is larger than will fit into the highest level of processor data cache, then the *data* is always streamed from main memory (as though there is no data cache).

Careful ordering of array accesses is also important to improve cache reuse, which is critical for good performance; e.g., ensuring that all entries in a cache line are used before the cache line must be reloaded. In a multi-material hydrodynamics code, a material model may likely be the primary work unit on a domain. Ordering elements so that data for elements with the same material are adjacent in memory can provide an optimal cache mapping. When materials move between mesh elements due to advection, it may be wise to periodically permute mesh data to retain memory adjacency. Optimal cache reuse will likely occur when data layout mirrors the needs of dominant numerical operations. However, which loops dominate runtime for a code is often highly problem-dependent. Flexibility to permute data could save an application from using poor memory access patterns for a given architecture. Reordering can also enable "lock-free" computations in a multithreaded environment. When using traditional programming language constructs, such as C-style for-loops, all execution and data access details are fixed in the application source code. Without some sort of abstraction layer, such as RAJA, altering implementation details is difficult and may require *writing and maintaining* multiple versions of individual loops.

Programming model concerns

There has been a clear trend in HPC toward node-level parallel programming models that extend programming languages, like C and C++, via compiler directives and library routines. OpenMP [1, 2], CilkPlus [3], CUDA [4], and other models can support multithreading and/or processor heterogeneity where CPUs and accelerators are combined. These models are standardized and supported well by compiler vendors making them viable for ASC production codes. However, *no existing programming model is a clear "best choice"* for all architecture considerations. Moreover, each model has unique programming characteristics and models are not easily interchangeable. Yet, interchangeability is necessary to manage performance portability. RAJA enables the use of different programming models in an application without exposing their idiosyncrasies to application scientists and without requiring multiple versions of computational kernels to be coded to different models.

RAJA Overview

The RAJA model addresses many concerns discussed in the previous section by providing a means to encapsulate loop implementation details, such as data access and execution patterns. RAJA is designed to keep the look and feel of serial code at the application level, which greatly simplifies maintenance and reasoning about algorithms and implementation choices. RAJA shares concepts with other C++ abstraction approaches, such as Thrust [5], Bolt [6], Kokkos [7], etc. However, RAJA supports constructs used heavily in LLNL ASC codes that are absent in other models.

RAJA has two main goals. The first is to insulate application developers from non-portable compiler and platform-specific directives, and parallel execution and programming model implementation details. The second is to simplify tuning of data layout and access patterns for diverse memory hierarchies. In this section, we describe the basic concepts in RAJA.

Fundamental Concept: Separate loop body from loop traversal

To encapsulate architecture-specific concerns and insulate them from application code, RAJA decouples a loop body from its traversal. Consider a "daxpy" operation implemented using a traditional C-style for-loop:

```
double* x; double* y;
double a;
// ...
for ( int i = begin; i < end; ++i ) {
   y[i] += a * x[i];
}
```

The corresponding RAJA form is:

```
Real_ptr x; Real_ptr y;
Real_type a;
// ...
forall< exec_policy >(IndexSet, [&] (Index_type i) {
   y[i] += a * x[i];
} );
```

There are several encapsulation aspects in the RAJA form shown here:

- **Data type encapsulation.** RAJA provides data and pointer types, seen here as "Real_type" and "Real_ptr", to hide non-portable compiler directives and data attributes (such as alignment). These additional compiler-specific data decorations often enhance a compiler's ability to optimize so that higher performance may be extracted from the underlying hardware. These types are not required to use RAJA, but are a good idea in general for HPC codes.
- **Traversal template and execution policy.** The "forall()" template method and the specified template parameter encapsulate the details of loop execution; for example, run the loop sequentially or in parallel, enable SIMD, etc.
- **Indexsets.** In the simple RAJA example above, "begin" and "end" integral loop bounds could have been passed to the iteration method. The RAJA Indexset abstraction is much more powerful and allows encapsulation of complex loop iteration patterns and memory access patterns based on data placement.

Note that the loop body is identical in both the C-style and RAJA loop forms above. A key RAJA design point is to have a minimal impact on application source code. Using the standard C++11 lambda function language feature, the loop body and necessary variables are *captured without modification* and used within the iteration method [8]. C++ compiler support for lambda functions is new and still maturing. We believe that robust lambda support is essential for adoption of RAJA in LLNL codes.

In the C-style loop all details of the loop execution (iteration sequence, data access pattern, etc.) are explicitly coded at the application level. Changing any aspect of execution requires direct modification of the loop source code. In the RAJA version, on the other hand, all loop execution details are hidden. This allows changing the execution of the loop by simply changing the execution policy type and/or iteration method. To make this more concrete, consider two potential RAJA OpenMP iteration templates (loop execution policy in red):

```
// (A) typical OpenMP multithreaded execution
template< typename LB >
void forall(omp_exec, int begin, int end, LB loop_body) {

#pragma omp parallel for
    for ( int i = begin; i < end; ++i ) loop_body( i );
}

// (B) OpenMP 4.0 accelerator execution
template< typename LB >
void forall(omp_acc_exec, int begin, int end, LB loop_body) {

#pragma omp target
#pragma omp parallel for
    for ( int i = begin; i < end; ++i ) loop_body( i );
}
```

Example (A) shows how the loop iterations could be launched using a standard OpenMP parallel for construct on a multi-core CPU, for example. Example (B) shows how the loop could be launched on an accelerator device, such as a GPU, using OpenMP 4.0 standard device target directives [2]. Here, data mapping between the host and device is implicit.

In CUDA, the notion of a loop is fundamentally absent. A "loop iteration" is expressed in a CUDA kernel function that is launched over a thread block on a CUDA-enabled GPU device. Each iteration executes on a different CUDA thread. The code snippets below illustrate potential RAJA "back end" code for CUDA. So that application code looks like that for other parallelization approaches, we pass a loop body to a C++ template method (D), which has the same arguments as other iteration methods. This template launches a GPU kernel (C) that executes a loop iteration on a separate GPU thread.

```
// (C) kernel function template
template< typename LB >
__global__ void loop_it(int begin, int N, LB loop_body) {
   int i = blockIdx.x * blockDim.x + threadIdx.x ;
   if ( i < N ) {
      loop_body( begin + i ) ;
   }
}

// (D) traversal template that launches GPU kernel
template< typename LB >
void forall(cuda_exec, int begin, int end, LB loop_body) {
   const int blocks = (end – begin + 63)/64 ;
   loop_it<<< blocks, 64 >>>(begin, end-begin), loop_body ) ;
}
```

It is important to note that support for C++ lambda functions in the NVIDIA nvcc compiler is under development. Early releases of CUDA 6.x support lambda functions in GPU kernels. However, it is still not possible to launch a lambda on a GPU when it is defined in code compiled for a host CPU. This functionality is needed for the RAJA model to encapsulate CUDA constructs and achieve full CPU-GPU portability. We have verified that the approach shown above works with current nvcc compilers when a loop body is expressed as a C++ functor object. We are actively involved in discussions with NVIDIA compiler developers on future expanded lambda support.

Fundamental Concept: Partition iteration space into work units (Segments)

For CPU-GPU portability, we need a single abstraction that makes it easy to manage both types of parallelism. Loop iterations map to threads differently on a multi-core CPU and a GPU. Figure 2 shows the key difference; on a CPU a contiguous block of loop iterations are mapped to each thread, while on a GPU adjacent iterations are mapped to adjacent threads within a thread warp. Each iteration of a loop is associated with a "footprint" of data array

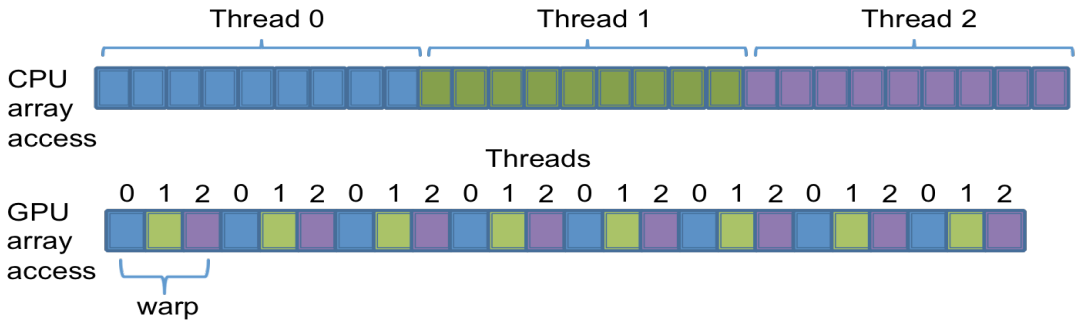

Figure 2. Loop iterations map to threads differently on CPUs and GPUs.

values in memory. In RAJA, we bundle loop iterations into *Segments*, which helps to manage data access patterns for different threading models.

Earlier, we noted that multiphysics codes employ operations involving stride-1 array data access as well as unstructured accesses involving indirection arrays. Often, these different access patterns are used on the same data array and may even be combined in the same physics operation. Figure 3 shows two different segment types, "range" and "list". A RAJA *range* segment defines a contiguous set of iteration indices. Constraints can be applied to the iteration bounds and also to their alignment with memory constructs. For example, range

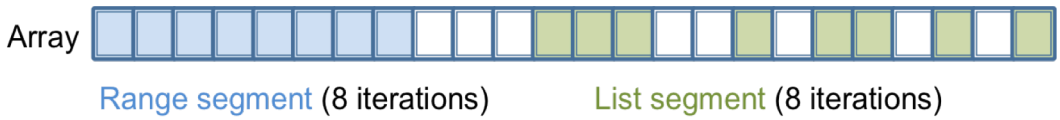

Figure 3. "Range" and "List" segments iterate over sets of array elements differently.

segments can be aligned multiples of a SIMD or a SIMT width, which helps compilers generate more efficient code. Iteration over a range segment usually involves a simple for-loop:

```
for ( int i = begin; i < end; ++i ) loop_body( i );
```

A *list* segment bundles iterations that do not meet range segment criteria. A typical iteration over a list segment involves a for-loop, with indirection applied:

```
for ( int i = 0; i < seglen; ++i ) loop_body( segment[i] );
```

Runtime segment construction can impose constraints that complement compile-time pragmas and optimizations, which can be hidden in RAJA iteration templates.

RAJA Segments can represent arbitrary loop iteration bundles that can be tuned and sized for specific architecture and memory configurations. Figure 4 shows two different element "tilings" on a domain that represent different data orderings (numbers) and iteration patterns

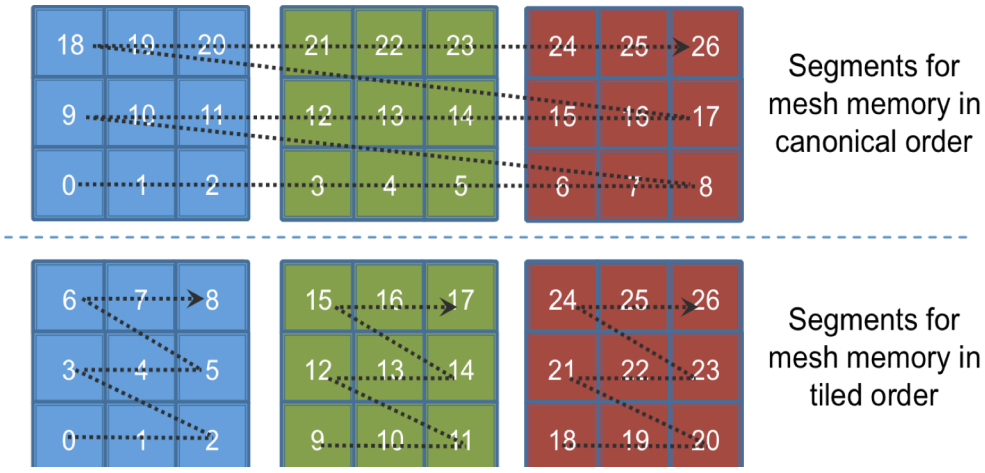

Figure 4. RAJA Segments can represent arbitrary "tilings" of filed data on a domain.

(dashed arrows). When loop bounds are abstracted in a segment, instead of hard-coded in an application, data arrays can be permuted for locality and cache reuse. For example, the canonical tiling in the upper part of Figure 4 can be transformed into the "compact" tiling in the lower part of the figure.

Segments can also work together with data allocation to further enhance optimization. A typical ASC code centralizes data allocation in macros or functions for consistent usage throughout a code. Data allocation can be based on segment configurations to apply other optimization-enhancing allocation techniques, such as "first-touch", which can result in improved NUMA behavior during multithreaded execution.

Fundamental Concept: Segment dispatch and execution (Indexsets)

A RAJA *Indexset* is an object that encapsulates a complete iteration space that is partitioned into a collection of segments, of the same or different types. To illustrate a simple use case, consider an array of indices to process; e.g., indices that enumerate elements on a domain containing a particular material:

```
int elems[] = {0, 1, 2, 3, 4, 5, 6, 7, 14, 27, 36,
               40, 41, 42, 43, 44, 45, 46, 47, 87, 117};
```

The indices may be assembled at runtime into an Indexset object by manually creating and adding segments to an Indexset object. A more powerful approach is to use a RAJA *Indexset builder* method that partitions the iteration space into a collection of "work segments" according to architecture-specific constraints. For example,

```
Indexset segments = createIndexset( elems, num_elems );
```

In this example, the resulting Indexset object may contain two range segments (`{0,…,7}`, `{40,…,47}`) and two list segments (`{14, 27, 36}`, `{87, 117}`). The Indexset object can be passed to an iteration template, as in the RAJA code examples shown earlier, that automatically dispatches the segments to execute a loop body (i.e., lambda function):

```
forall< exec_policy >( segments, loop_body );
```

A compile-time generated iteration template for each *segment type* executes portions of the loop associated with the segment type, possibly in parallel.

Indexset builder methods can be customized to tailor segments to hardware features and execution patterns to balance compile-time and runtime considerations. Presently, Indexsets enable a two level hierarchy of scheduling and execution. A *dispatch policy* is applied to the collection of segments. An *execution policy* is applied to the iterations within each segment. Examples include:

- Dispatch each segment to a CPU thread so segments run in parallel and execute range segments using SIMD vectorization.
- Dispatch segments sequentially and use OpenMP within each segment to execute iterations in parallel.
- Dispatch segments in parallel and launch each segment on either a CPU or GPU as appropriate.

It is important to note that RAJA allows all aspects of execution to be tailored and optimized by developers. They can modify segment dispatch and execution mechanisms in traversal methods, or build their own to explore alternative "work-around" implementations that may overcome problems when execution performance does not match expectations. Such complete control is not found in monolithic programming models, typically.

The RAJA indexset/traversal model also supports other more advanced features that we have recently begun to explore in the LULESH proxy-app. For example, indexset segments can be defined and arranged to encode dependence scheduling patterns to enable more efficient

parallelism. Key aspects of a “lock-free” segment scheduling mechanism in a RAJA iteration template are shown in the following code example:

```
#pragma omp parallel for schedule(static, 1)
for (int i = 0; i < num_seg; ++i) {
   while (seg_semaphore[i] != 0) {
      sched_yield();
   }

   seg_dispatch(seg_type[i], seg_info[i], loop_body);

   seg_semaphore[i] = seg_sem_reload_val[i];
   for (int j = 0; j < seg_num_dep_tasks[i]; ++j) {
      int dep = seg_dep_task[i][j];
      __sync_fetch_and_sub(&seg_semaphore[dep], 1);
   }
}
```

The dependence scheduling is controlled by a simple semaphore mechanism applied per segment. To manage dependencies in this fashion, three pieces of information are required. First, a “reload” value defines the number of external dependencies that must be satisfied before a segment can execute. As each dependency is satisfied the semaphore value is decremented; when it reaches zero, the segment can execute. Until then, the thread “yields” the CPU resource. After a segment is dispatched to execute, its semaphore value is reset to the reload value. Second, an “init” value is an override for the reload value. A subset of segments must be “primed” to execute; ideally, a number of segments at least as large as the maximum number of threads available segments should be able to execute immediately. The semaphore value for such segments is initialized to zero to indicate that they can execute from the start of the traversal. Semaphore values for all other segments are initialized to their reload values. Third, “forward dependencies” are the set of segments that must be notified when a segment execution completes. Here, notification means that the semaphore value in each forward segment is decremented by one. Later, we will show the performance benefit that can be achieved using this “lock-free” segment scheduling mechanism in LULESH. Such an approach could also be used to create “task graph” dependency scheduling for tile segments in wave-front algorithms, such as sweeps used in deterministic transport codes.

Traversals can also potentially support a portable, transparent, fine-grained transient fault recovery mechanism. The simplified code example below shows the basic idea:

```
#pragma omp parallel for schedule(static, 1)
for (int i = 0; i < num_seg; ++i) {
   bool done = false;
   while (!done) {
      try {
         done = true;
         seg_dispatch(seg_type[i], seg_info[i], loop_body);
      }
      catch (Transient_Fault) {
         cache_invalidate();
         done = false;
      }
   }
}
```

This fault recovery mechanism, embedded in a loop iteration template, can potentially catch any transient error condition and allow a code to recover. The C++ try/catch mechanism tests

whether a transient fault has occurred. If so, the data cache is invalidated and the loop is re-run with data values reloaded from memory. With such an approach, important aspects of transient fault recovery can be hidden and easily managed in a large code base. Also, the recovery cost for faults addressed by this method is commensurate with the scope of such faults. That is, a code can recover with minimal localized disruption and not need a full restart. Use of this approach would benefit from additional hardware and O/S support (i.e., the processor could emit specialized signals for fault conditions and the O/S could be more specialized to help process them), and language support (e.g., the C++ try/catch mechanism could be expanded to respond to O/S signals). In addition, each loop to which this mechanism is applied must be *idempotent*; i.e., it can be run an arbitrary number of times and produce the same result. This requires read-only and write-only arrays (no read-write arrays), which can increase memory usage and bandwidth requirements slightly. Later, we will show the software and performance impact of this RAJA-based fault recovery method in LULESH is acceptable.

Exploring RAJA in LLNL ASC hydrodynamics codes

To assess the viability of RAJA, we explored the approach in the Lagrange hydrodynamics portion of three ASC codes at LLNL: Ares, Kull, and ALE3D. Although preliminary and limited in scope, the explorations were adequate to determine whether RAJA is sufficiently flexible to serve as a loop-level abstraction layer across diverse code environments, to determine its impact on production source code, and evaluate its potential for performance portability. Performance evaluations were performed on TLCC2 and BG/Q platforms, the primary architectures used for production codes at LLNL. Also, on BG/Q, we compared the GNU 4.7.2 and the IBM xlc 12.1 compilers. Most LLNL codes use the xlc compiler on that platform. However, only the GNU compiler supports C++ lambda functions on that machine, which is needed for RAJA. This section summarizes our findings.

Ares

Ares is a multi-block, structured mesh code that uses little abstraction in its numerical kernels. Most physics algorithms in the code are written using traditional C-style for-loops with no encapsulation of data access or loop iteration patterns. Basic RAJA insertion enables encapsulation of these aspects of the code. We converted most of the loops executed during a single material Lagrange hydrodynamics computation to use RAJA. Performance experiments were done on LLNL BG/Q and TLCC2 platforms.

Ares-RAJA integration

To simplify RAJA integration into Ares, and minimize its impact on the look-and-feel of the source code, a lightweight C++ loop template API was constructed to encapsulate RAJA constructs. Loop templates were created to explicitly name each loop pattern in a way that is intuitive to Ares developers. An indexset collection was added to the Ares domain structure to hold the indexsets that were needed to run the loops. These were defined in existing setup routines where domain extents and indirection arrays are defined. Each named loop iteration method retrieves the appropriate index set from the collection when it is called. For example, loops over “real” zones on a domain in the original code, such as:

```
for ( int ii = 0 ; ii < domain->numRealZones; ++ii ) {
   int zone = domain->Zones[i];
   // loop body using “zone” as array index
}
```

and

```
for ( int j = domain->jmin; j < domain->jmax; ++j ) {
   for ( int i = domain->imin; i < domain->imax; ++i ) {
      int zone = i + j * domain->jp;
      // loop body using “zone” as array index
   }
}
```

were transformed to the following form:

```
forEachRealZone< exec_policy >( domain, [=] (int zone) {
   // loop body using “zone” as array index
}
```

In total, we converted 421 loops in Ares to similar implementations and employed three loop execution policies, which we called “DPWork”, “DPStream”, and “Seq”. We applied the DPWork and DPStream policies to data parallel loops containing roughly ten FLOPS or more per loop iteration and those that contained less than that, respectively. This distinction was based on some high-level profiling; we wanted to distinguish loops where threading was a clear win and when it was not on current platforms. We used the Seq policy for sequential execution on loops that were not easily parallelized. Overall, we found the basic integration of RAJA to be straightforward. The most difficult work was localized to setting up and manipulating RAJA index sets. Replacing C-style for-loop headers with calls to iteration template methods and identifying the appropriate execution policy for each loop was not hard, but somewhat tedious.

When the transformed code was presented to Ares developers, including code physicists, they identified several software engineering benefits related to improved code readability and maintenance. For example, named traversal methods clearly label iteration patterns in the code, and the named execution policies document how each loop will be run. Users also noted that encapsulating the loop logic would potentially eliminate coding errors. Finally, developers noted the potential to simplify porting to different architectures by parameterizing execution policies via *typedef* statements in header files.

To demonstrate additional RAJA benefits, we explored a few more advanced code transformations in Ares. One example involves a deep loop nest in the mixed-zone advection algorithm that uses integer arrays for both control logic and indirection. The following simplified code example shows key aspects of the loop structure:

```
for ( int iz = 0 ; iz < nmix_zones; ++iz ) {
   if ( domain->mixzone_advect[ iz ] ) {
      for ( int i = 0; i < numlocal_mat; ++i ) {
         if ( mzreg[ domain->mat[i] ].ndx[iz] >= 0 ) {
            var[ mzreg[ domain->mat[i] ].ndx[iz] ] = ...;
            // etc.
         }
      }
   }
}
```

This loop nest is further nested within a loop over mesh field variables that are processed by the advection algorithm. Notice, however, that none of conditional tests or indirect data accesses depend on the variable involved, only whether a "mixed" zone is advected or if the zone contains a given material. Such complicated loop organization induces unnecessary memory bandwidth needs and hinders many compiler optimizations.

To address these issues, we encoded conditional logic and indirection in RAJA index sets. We inverted the loop nest so that iteration over material regions was on the outside. The inner loop indirection and conditional logic is replaced with a call to an iteration method that iterates only over the advected zones with a given material. This allowed us to remove two levels of loop nesting. The resulting code is shown below:

```
for ( int i = 0; i < numlocal_mat; ++i ) {
      // ...
   forEachAdvectedMixedZoneInRegion< exec_policy >( domain, i,
      [=] (int ndx) {
         var[  ] = ...;
         // etc.
   } );
}
```

This code is simpler and easier to understand and runs faster than the original. On a test problem that stresses these operations, we observed a 1.6x serial speedup on TLCC2 for this loop and a 1.99x serial speedup on BG/Q. Starting from the original code compiled with xlc and moving to the RAJA version compiled with GNU results in a total serial speedup of 3.78x on BG/Q for the loop. Applying the transformation described here on just two loops yields ~8% speedup in the *overall runtime* on TLCC2 for the aforementioned test problem.

RAJA can also be used to reorder loop iterations to enable parallelism in loops that are not parallelizable as currently written. The loops could be restructured to expose the available parallelism. However, using the RAJA abstraction layer, different loop orderings can be explored easily without modifying the application code. For example, a common operation in a staggered-mesh code, like Ares, sums values to nodes from surrounding zones. This is illustrated in the left image in Figure 5. Using indexsets to define independent groups of computation and reorder the loop iterations enables different parallel implementation possibilities. The middle and right images in the figure show two options, (A) and (B). Different colors indicate independent groups of computation, which can be represented as segments in the indexsets. For option A, we iterate over groups sequentially (group 1 completes, then group 2, etc.) and operations within a group can be run in parallel. For option B, we process zones in each group (row) sequentially and dispatch rows of each color in

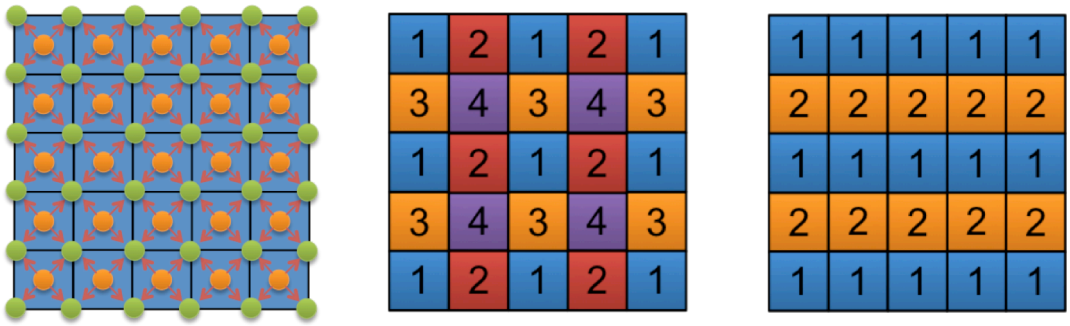

Figure 5. Zone-to-node sum operation (left), ordering option A (middle) and B (right).

parallel. For a 3D problem run on BG/Q, option A gives ~8x speedup with 16 threads over the original serial implementation. Option B provides ~17% speedup over option A at 16 threads. It is worth reiterating that no source code modifications are required to switch between these parallel iteration patterns with RAJA in place.

Ares-RAJA performance assessment

To evaluate the overall performance impact of RAJA on Ares, we ran multiple experiments with a single material 3D Sedov problem using various combinations of MPI tasks and OpenMP threads per task. Ares can partition a problem into any number of domains per MPI task and can run with or without OpenMP threading on loops over domains within an MPI rank. When OpenMP is used to parallelize loops over domains, we refer to this as coarse-grained threading in the following discussion. The simple Sedov problem represents an extreme performance case for fine-grained inner loop threading since most loops do very little work compared to more realistic, multi-material problems. In particular, for this problem, no combination of MPI tasks and OpenMP threads runs faster than using 64 MPI tasks on BG/Q (one task per hardware thread), assigning one domain to each MPI rank. Figure 6 summarizes the comparison between MPI-only and MPI plus OpenMP threads.

MPI Tasks	Threads per MPI task	Time (seconds)	Speedup	Parallel efficiency
1	1	1628	1.00	1.000
2	1	988	1.65	0.824
4	1	512	3.18	0.795
8	1	271	6.01	0.751
16	1	149	10.93	0.683
32	1	101	16.12	0.504
64	1	77	21.14	0.330
32	2	95	17.14	0.268
16	4	103	15.81	0.247
8	8	118	13.80	0.216
4	16	140	11.63	0.182
2	32	177	9.20	0.144
1	64	255	6.38	0.099

Figure 6. Summary of Ares Sedov problem performance run on a single BG/Q node for different combinations of MPI tasks and coarse-grained OpenMP threads per task. MPI-only always outperforms MPI plus threads using the same number of resources.

Fine-grained inner loop threading yields a performance benefit over running MPI-only in our study. Figure 7 shows strong-scaling speedup for Ares-RAJA compared to original Ares at each MPI task count. Ares is run with M MPI tasks and no threads. Ares-RAJA uses M MPI tasks and T OpenMP threads per task, where M x T = 64 in each case. Ares-RAJA shows a speedup over the original code at all MPI task counts, except M = 64. The performance benefit of fine-grained threading decreases as the number of MPI tasks increases, as expected based on the results shown in Figure 6. Nevertheless, the result leaves us optimistic about

future prospects for fine-grained threading in Ares since the Ares-RAJA version has 372 inner loop OpenMP parallel sections per timestep. It is important to note that, due to per-core memory constraints, a typical Ares run on BG/Q cannot use more than 16 MPI tasks per node. At that task count, we see a roughly 50% performance boost with inner-loop threading over MPI-only (red oval in the figure). The overhead of the RAJA layer in the code, ~10%, is seen at 64 MPI tasks. We elaborate on this issue in the Appendix.

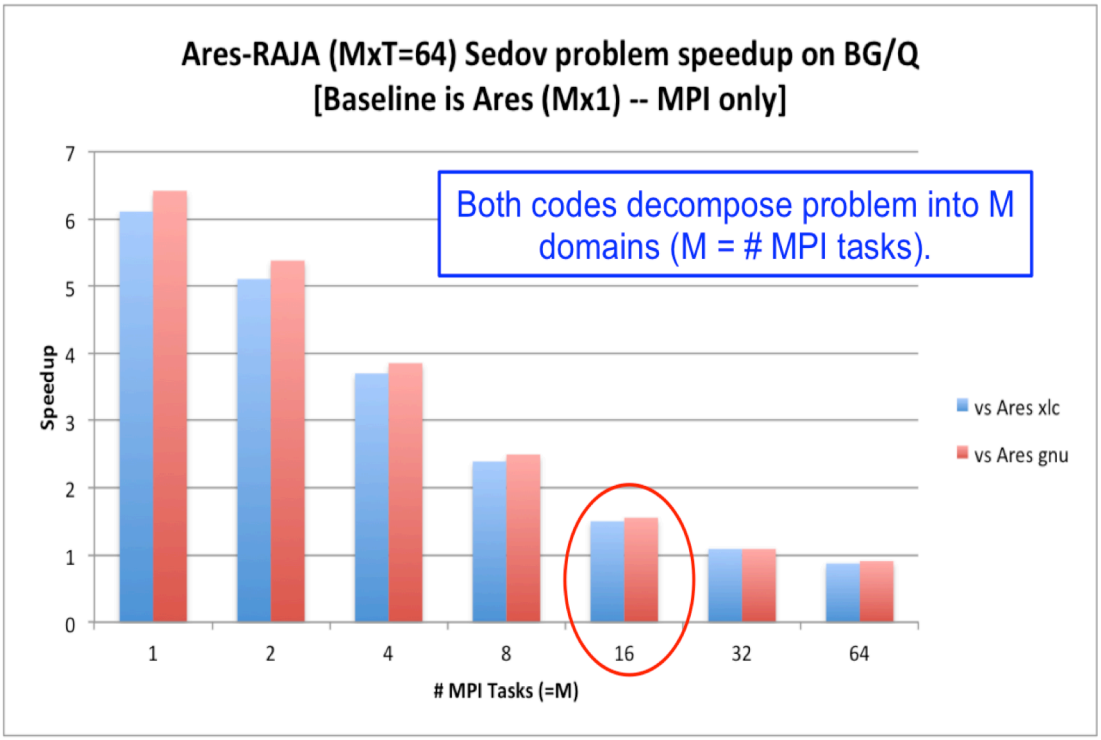

Figure 7. Strong scaling speeup of Ares-RAJA vs. Ares on BG/Q with xlc and gnu compilers. Ares is run with M MPI tasks. Ares-RAJA is run with M MPI tasks and T OpenMP threads per task (M x T = 64 in each case). Both codes decompose the problem into M equal-sized domains.

For a more extreme evaluation of OpenMP thread performance, we performed a similar experiment that compared coarse-grained domain loop threading in the original code and fine-grained inner-loop in the Ares-RAJA version. Both versions of the code are run with M MPI tasks and T OpenMP threads per task on a BG/Q node, where M x T = 64 across the full range of MPI/OpenMP combinations. Ares uses 64 domains in each case and employs coarse-grained threading on 27 domain loops per timestep. Ares-RAJA uses M domains and T fine-grained threads on 372 inner loops per timestep. Figure 8 shows Ares-RAJA speedup compared to Ares with three different compiler options. In all cases, it is clear that a few coarse-grained thread parallel sections outperform many fine-grained thread parallel sections. The Ares-RAJA version gets to within 5-10% at 16 and 32 MPI tasks, depending on the compiler used for the Ares version. The conclusion to be drawn from this experiment is that OpenMP overheads are too high for fine-grained threading on many of the loops in our codes *independent of RAJA*. Clearly, vendors must address this. In a large multi-physics application, thousands of threaded regions will be required per time step to expose sufficient parallelism and run efficiently on future architectures.

Lastly, we performed similar comparisons on the LLNL TLCC2 architecture (Intel ES-2670 16 core "Sandy Bridge" node). In this case, we used the Intel C++ compiler, version

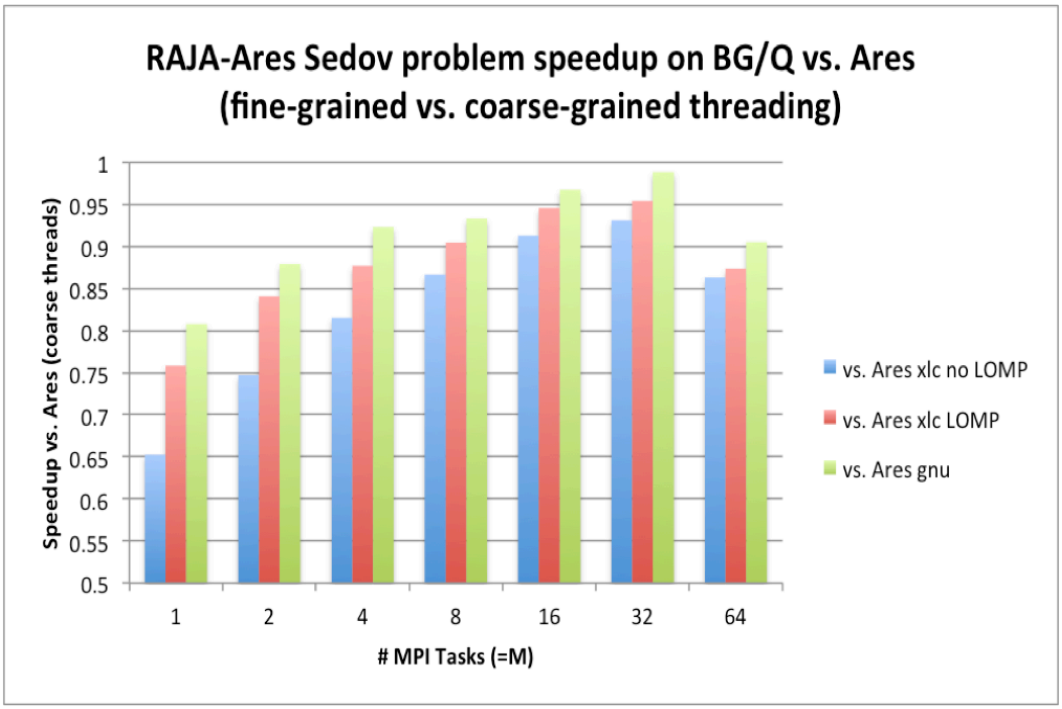

Figure 9. Strong scaling speedup of Ares-RAJA vs. Ares on BG/Q with xlc (LOMP runtime and no-LOMP) and gnu compilers. Both versions of the code are run with M MPI tasks and T OpenMP threads per task (M x T = 64). Ares uses 64 domains in each case, and uses coarse-grained threading on domain loops (27 per timestep). Ares-RAJA uses M domains and T fine-grained threads on inner loops (372 per timestep).

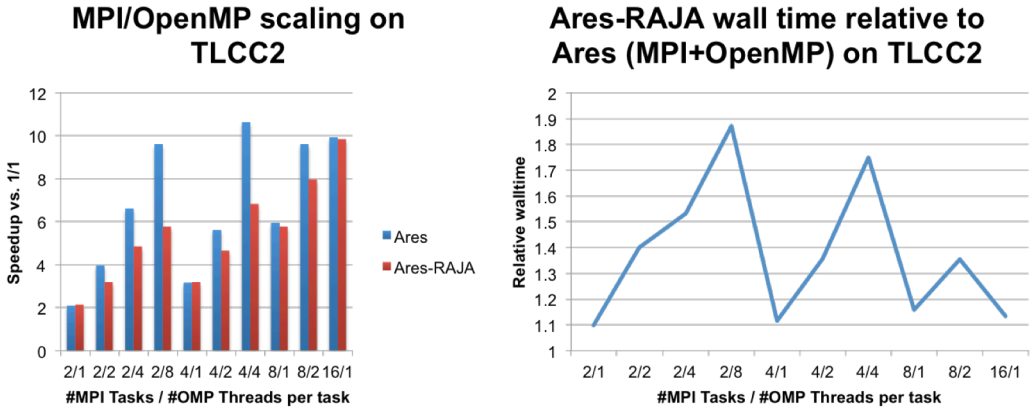

Figure 8. Fine-grained OpenMP inner loop threading (Ares-RAJA) shows considerable scaling and overhead issues when compared to coarse-grained domain loop threading (Ares) on TLCC2.

14.0.174. As in the BG/Q experiments, M is the number of MPI tasks and T is the number of OpenMP threads per task. Again, the Ares version uses M x T domains, assigning T domains to each MPI rank (one domain per thread). The Ares-RAJA version uses M domains, with T threads applied per inner loop. Fine-grained OpenMP threading performs considerably worse on TLCC2 than on BG/Q and this was clear in our results, which are shown in Figure 9. The left plot show the scaling of each version of the code compared to a serial version (no MPI,

no OpenMP). The right plot shows overall runtime of the Ares-RAJA version compared to the Ares version. Note that RAJA incurs serial overhead of 10-16% (points with 1 thread per MPI task). Relative performance degrades significantly as threads are added to each MPI task. After some analysis of compiler output, we have identified this as an issue involving the combination C++ template/lambda constructs and OpenMP directives, which we are working with the Intel compiler team to fix. Again, we elaborate more on this issue in the Appendix.

Kull

Kull is an unstructured mesh code that supports arbitrary polyhedral elements. It makes heavy use of C++ templates and all loops use custom iterators over entities on a mesh. Compared to Ares, a smaller number of loops in Kull were converted to use RAJA. However, enough loops in the Lagrange hydrodynamics portion of Kull were transformed to assess how well RAJA works with Kull iterators.

Kull-RAJA integration

Kull developers converted 129 loops to use RAJA-style traversals. Due to the custom nature of Kull iterator syntax, Kull-specific iteration templates were required for basic loop conversion. Also, no RAJA indexset concepts were used in the initial Kull exploration, as this would require non-trivial modification to Kull iterator and Field abstractions. Since C++ lambda functions are able to capture the bodies of Kull loops unchanged, RAJA insertion was straightforward, requiring 2-3 lines of code change per loop. To illustrate, a typical Kull loop over zones in a material region looks like the following:

```
Field< Region, Zone, Scalar >& matRho = ...;
Field< Region, Zone, Scalar >& oldMatRho = ...;
// ...
for ( typename Region::ZoneIterator zi = region.zoneBegin();
      zi != region.zoneEnd(); ++zi ) {
   matRho[*zi] = oldMatRho[*zi] * oldZoneVolume[*zi] /
                 newZoneVolume[*zi];
   // etc ...
}
```

The same loop converted to RAJA-style is:

```
Field< Region, Zone, Scalar >& matRho = ...;
Field< Region, Zone, Scalar >& oldMatRho = ...;
// ...
forall< exec_policy >( region.zoneBegin(), region.zoneEnd(),
   [&] (typename Region::ZoneIterator zi) {
   matRho[*zi] = oldMatRho[*zi] * oldZoneVolume[*zi] /
                 newZoneVolume[*zi];
   // etc ...
} );
```

As in Ares, only the loop header is changed. It is important to note that, in general, OpenMP will not parallelize a loop that contains the common C++ idiomatic use of an iterator operator "!=" in a loop conditional (as in the original Kull code example above). To work-around this, RAJA-style iteration templates for Kull use an appropriate OpenMP canonical loop by using "<" instead.

Kull also uses multiple iteration variables for certain loop iteration patterns. OpenMP will not parallelize a loop with more than one iteration variable. For example, Kull "part loops" typically use two iteration variables, an integer for the part ID and an iterator over a part list.

For example,

```
int partID = 0;
// ...
for ( typename PartList<MeshType>::ConstIterator pi =
      partList.begin(); pi < partList.end(); ++pi, ++partID )
{
   Field< Region, Zone, Scalar >& eEDot = DEeDt[ partID ];
   // etc ...
}
```

RAJA can also work around this issue to enable parallelism in such a loop without major code restructuring. A possible RAJA-style iteration template for this is:

```
template< typename ITER_T, typename LOOP_BODY >
void forall_part( omp_parallel_for,
                  const ITER_T& begin, const ITER_T& end,
                  LOOP_BODY loop_body ) {
   #pragma omp parallel
   {
      int num_threads = omp_get_num_threads();
      int partID = omp_get_thread_num();

      #pragma omp for schedule(static, 1);
      for ( ITER_T i = begin; i < end; ++i ) {
         loop_body( i , partID );
         partID += num_threads;
      }
   }
}
```

While this does enable parallelism over a part loop in Kull, we found that very little was gained by doing so. This is due to either a small number of parts on any given domain, or a significant imbalance in the number of zones in different parts, at least in the problems that

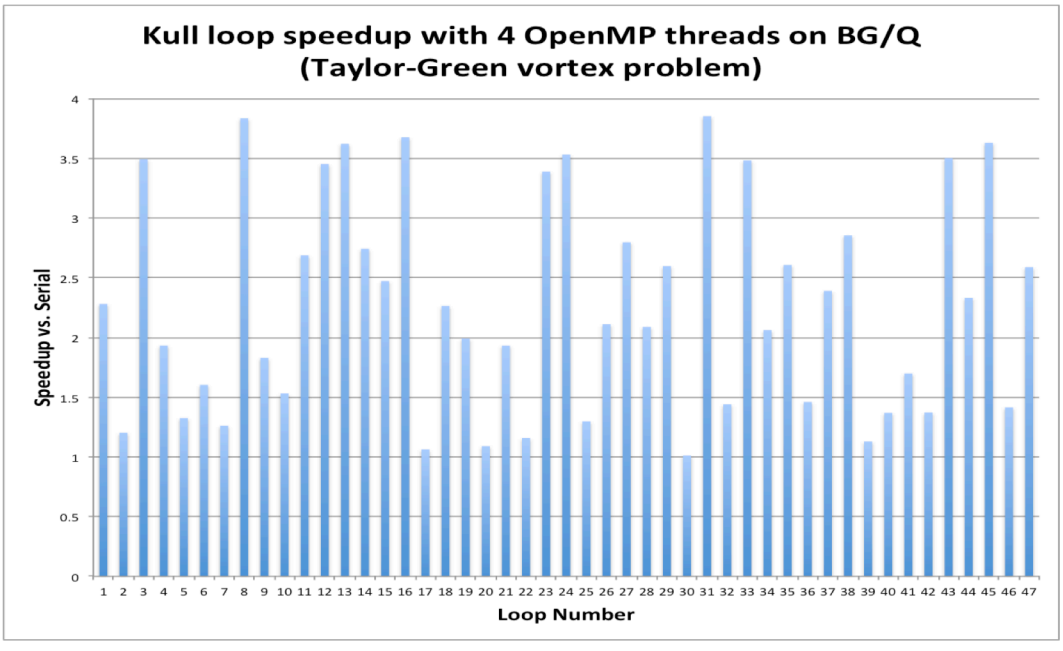

Figure 10. Speedup with 4 OpenMP threads vs. serial across a range of loops executed in Kull for a Taylor-Green vort4ex problem.

were explored. We found that parallelizing inner loops over zones within a part yielded the most performance benefit.

Kull-RAJA performance assessment

The study of thread scaling performance in Kull was very limited, but showed notable promise for the viability of fine-grained loop threading in that code. *Figure 10* shows speedups achieved with four OpenMP threads on BG/Q for individual loops across a range of loops executed during the run of a Taylor-Green vortex problem. Future effort profiling a variety of problems will help choose loops with the highest potential impact when parallelized. Also, execution policy alternatives should be explored, as well as in exercising other computer platforms, such as TLCC2, and higher OpenMP thread counts.

LULESH

LULESH is a proxy-app derived from the Lagrange hydrodynamics algorithm in ALE3D. ALE3D and LULESH are unstructured mesh codes. In contrast to Kull's general polyhedral element capability, ALE3D/LULESH primarily employs hexahedral elements with arbitrary connectivity. ALE3D/LULESH has a data access abstraction layer via simple use of C++ features. Thus, LULESH provides an important use case for RAJA exploration that is distinct from Ares and Kull. Due to its relatively small size and simplicity, LULESH also allowed us to explore more advanced RAJA features such as: data reordering and loop iteration tiling, lock-free segment scheduling, and fault tolerance.

LULESH-RAJA integration

LULESH v1.0 was the first application we explored with RAJA, beyond a preliminary investigation of RAJA concepts in the LCALS benchmark. Our goal was to explore a complete physics kernel, without tackling a full multiphysics code. We started with a highly optimized serial implementation of LULESH using only C language features (i.e., we removed the C++ data access layer). This was our "baseline" version, which provided a reference point to evaluate the effect (performance and code disruption) of inserting RAJA into a code.

Starting with the baseline LULESH, we performed several modifications to insert RAJA constructs. We replaced the single indirection array in the code for material indices with three RAJA indexset objects for element, node, and material indices, and replaced C-language for-statements with calls to RAJA iteration template methods. This required us to modify numerous function parameters, replacing indirection array pointers and length arguments with indexset object references. Finally, we removed five arrays used to gather domain fields for material operations and associated copy logic. RAJA indexsets allow such data reordering to be done "in-place". The overall conversion was straightforward and required modifications to roughly 4% of the total source code lines (LULESH is about 3000 lines total). Similar to Ares and Kull, the major change involved replacing for-loop headers with RAJA-style iteration method calls. Once RAJA was in place, we added about 150 lines of *initialization-only* code to explore different data tiling options. No numerical operations in the code were changed to enable this capability.

Error! Reference source not found. shows a serial runtime comparison for the variants of RAJA we studied. The results shown were generated on an Intel TLCC2 node and the code was built with the Intel compiler at a level of optimization comparable to what is used at

LULESH code version	Serial wall time (sec)	Page faults
Stock v1.0	126	16.9M
Baseline	121	16.9M
RAJA “canonical”	118	12.7M
RAJA “order-tiled”	118	12.7M
RAJA “index-tiled”	127	12.7M
Raw C – no indirection	114	12.7M

Table 1. Summary of serial performance of LULESH default problem configuration for different variants of the code. All runs were done on a TLCC2 node (Intel ES-2670) with the Intel compiler.

LLNL for production codes. The baseline variant (with the data access layer removed) runs 5 seconds faster than stock LULESH v1.0. The three RAJA variants use different indexsets. The “canonical” variant uses the same data layout as the baseline, represented as an index set with a single range segment. For the “order-tiled” variant, the mesh elements are permuted into stride-1 tile segments. This is encoded in an indexset as a collection of range segments. Finally, for the “index-tiled” variant, elements are left in canonical ordering with spatial tiles implemented in an index set as a collection of list segments. Run times for the first two RAJA variants are the same and slightly faster than the baseline. Recall that the RAJA versions have reduced memory movement resulting from the removal of “gather” operations mentioned above. The “index-tiled” variant is a bit slower since all data array accesses use indirection; this is comparable to the stock v1.0 version, in which indirection is used in nearly all loops. Lastly, we generated a “Raw C” version of the code, in which all indirection was removed. We believe this variant yields a lower bound on overall runtime. It is worth noting that the number of page faults drops significantly in the RAJA and “Raw C” variants, compared to the stock v1.0 and baseline variants. This is related to the removal of the “gather” arrays. The primary take-away message from this study is that the RAJA layer does not degrade serial performance. The transformation of LULESH to use RAJA also revealed several notable portability and maintainability benefits:

1. The kind of parallelism used for each loop is documented clearly by the template parameter associated with each loop iteration method call.
2. The named index set passed to each loop iteration method makes clear to which entity-set the loop is applied (e.g. node, element, or material subset). The original loop “begin” and “end” integer values were less clear.
3. Tiling support was isolated to index set creation, with no algorithm modifications.
4. There was *no change* to the core algorithms in inner-loops.

LULESH-RAJA performance assessment

We also compared the baseline version of LULESH v1.0 with OpenMP directives applied manually to each loop to a few different LULESH-RAJA variants to assess multithreading performance on BG/Q and TLCC2 platforms. To achieve loop-level multithreading in LULESH-RAJA, we simply altered one line of code to change the loop execution policy. The left plot in Figure 11 shows OpenMP thread speedup on BG/Q of LULESH-RAJA compiled with GNU (which supports C++ lambda functions) over LULESH compiled with the IBM xlc and GNU compilers. The right plot in the figure compares runtime relative to LULESH v1.0 compiled with xlc. The blue curve indicates that LULESH is only ~2% slower

when compiled with GNU than it is with xlc (until 32 threads, when the curve bumps up slightly). From this, we conclude that the GNU compiler is a reasonable alternative to xlc for the purposes of our study. The green and blue bars in the left plot show similar scaling with the two different compilers for LULESH. LULESH-RAJA scales much better (red bars), especially at higher thread counts. Runtime is also shorter at 16 or more threads (~70% of LULESH at 64 threads), as shown in the red curve in the right plot. However, LULESH-RAJA is noticeably slower at smaller thread counts; e.g., ~15% slower at one thread. This is a similar overhead issue we pointed out when describing Ares-RAJA performance earlier. We will discuss this in more detail in the Appendix.

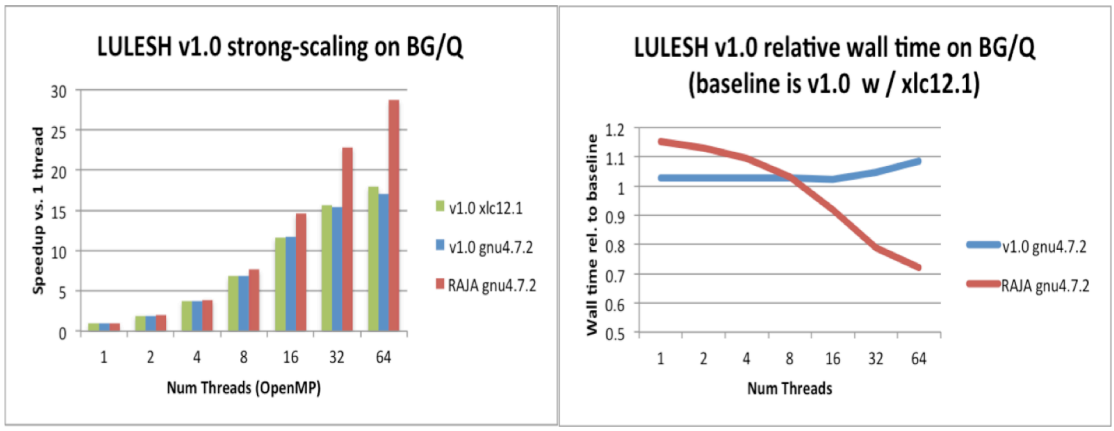

Figure 11. LULESH-RAJA shows good strong scaling vs. LULESH on BG/Q (left). However, runtime comparison (right) reveals compiler optimization issues, observed as overhead at low thread counts.

We ran similar experiments on TLCC2 using different threading models in LULESH-RAJA (OpenMP, CilkPlus, and lock-free OpenMP). Figure 12 shows the results. Thread speedup (left plot) shows that each LULESH-RAJA variant strong scales better than LULESH with OpenMP. Comparing runtimes (right plot) reveals an overhead issue with LULESH-RAJA and OpenMP similar to BG/Q. As on BG/Q, the slowdown is ~15% for one thread, and again, runtime is ~70% of LULESH at the maximum node thread count (16 threads in this case). CilkPlus scaling is not as good as OpenMP, but CilkPlus does not exhibit the overhead issue at small thread counts. Finally, lock-free thread scheduling version of LULESH-RAJA is considerably faster at all thread counts than the other parallel variants presented here. At 16 threads, it runs nearly twice as fast as LULESH with OpenMP directives applied directly on loops. Also, note that LULESH-RAJA with lock-free scheduling does not require extra data allocation and copying that is needed in the other variants to enable thread parallelism on all loops in the code. This efficiency overcomes the overhead at small thread counts.

Lastly, we have also run preliminary experiments simulating transient fault injection to evaluate performance of the RAJA-based fault recovery mechanism described earlier in this report. We found the fault detection and recovery mechanism to have very low overhead; for example, a sample serial run with 47 randomly injected faults executes within ~0.5% of a run with no injected faults. The code runs in just over two minutes, so this is an extreme fault frequency. In the current implementation, an entire loop must complete before a fault is identified and the loop is re-run. Additional operating system and language support could enable the recovery process to begin immediately after the fault occurs. We did observe faults occurring outside of loop executions; such faults require another mechanism to be caught and handled properly. As we noted earlier, the RAJA-based fault recovery mechanism

requires each loop to be idempotent. This can increase code runtime, due to increased memory allocation and data copy operations, which depends on how many loops must be changed to make them idempotent. Changing LULESH to make all loops idempotent increase total runtime 2-5% on TLCC2 depending on the number of OpenMP threads we used. We view these results as positive and demonstrate the RAJA can enable some degree of portable, transient fault tolerance to a code in a manner that is transparent to the

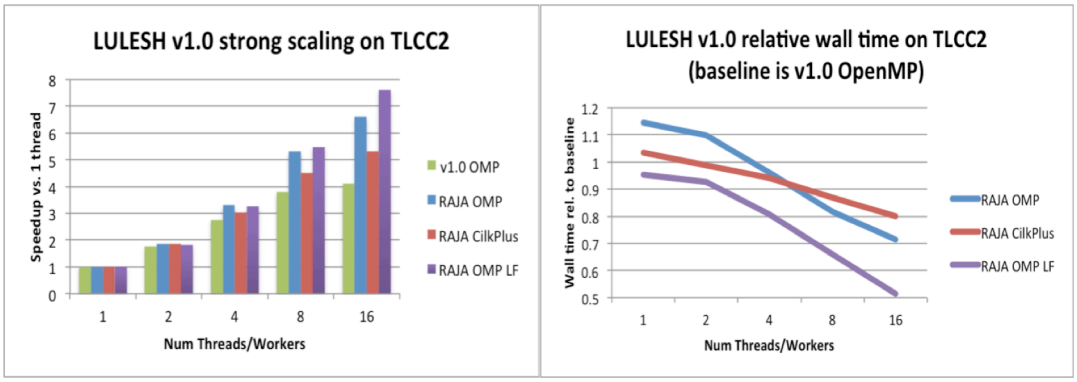

Figure 12. LULESH-RAJA with OpenMP, CilkPlus, and OpenMP lock-free threading models scales better than LULESH with OpenMP. Runtime comparison (right) shows a similar overhead issue with LULESH-RAJA and OpenMP at small thread counts to BG/Q (Figure 11). Lock-free scheduling overcomes the issue to run faster than LULESH at all thread counts.

application and with acceptable overhead.

Conclusions and future work

In this report, we have described some key challenges facing multiphysics codes as they are transitioned from an MPI-centric programming model to a model that adds massive fine-grained multithreading. We introduced the main concepts in the RAJA portability model, and discussed initial explorations involving portions of three ASC multiphysics codes at LLNL. We discussed both qualitative and quantitative advantages of RAJA to give the reader a broad view of how RAJA might impact their own multiphysics application.

Given the diversity and complexity of emerging architectures, RAJA can potentially enable a high level of optimization and performance, while encapsulating architecture-specific constructs in a large application. Loop traversal optimizations are instantiated at compile-time, while indexsets defining the iteration space are generated at runtime. Indexsets can help reduce data motion, place arrays in memory to reduce cache conflicts, and align arrays for better hardware efficiency. Much of the memory movement in vector-related gather/scatter operations can also be avoided with judicious use of RAJA, while still achieving many of the compiler optimizations that gather/scatter provides.

Multithreading programming models such as OpenMP and Cilk are currently supported. We expect that switching between CPU and GPU implementations can be done with RAJA without disrupting application code. However, concrete verification of this claim requires additional compiler support for OpenMP 4 and/or CUDA that is in development. RAJA should be able to encapsulate the essential features of the parallel programming models being considered for ASC code development at LLNL in the foreseeable future. Application scientists should be able to write source code cleanly and achieve a high level of portability without being required to possess detailed knowledge of non-portable code annotations and

constructs. Execution policy types passed to traversal methods are a self-documenting way to describe parallelism in each loop, while allowing the behavior of an equivalence class of similarly structured loops to be controlled from a single code site (e.g., a header file). At present, many of the drawbacks we have seen with RAJA are due to missing features in either compiler or run-time layers – drawbacks that exist with or without RAJA.

Recall the questions we set out to answer about RAJA that we stated in the introduction of this report. With respect to the previously stated concerns, we can conclude the following at this point:

- **Programmability**
  We have learned that it is easy to insert RAJA into legacy codes at LLNL. We have shown that RAJA supports distinct loop patterns in major ASC codes at LLNL by integrating it directly in a subset of Ares, Kull, and ALE3D (via LULESH). Basic insertion of the model is straightforward and non-disruptive, requiring code changes that are small and simple. Code transformations can be tedious, and we are exploring ways to guide and automate the process with the ROSE compiler team at LLNL. Application developers, who did the bulk of RAJA integration described in this report, found that the approach provides a variety of software engineering benefits that potentially make code easier to read, write, and maintain.

- **Portability**
  Our exploration of RAJA portability was limited to the BG/Q and TLCC2 platforms and multi-core threading models as these are the primary targets for inner loop threading in our codes, currently. We have shown that it is straightforward to control a variety of aspects of fine-grained parallelism once RAJA is in place. RAJA allows loop-level execution control to be centralized in a large code base, which enables *manageable* platform portability.

  Indexsets and iteration templates allow abstraction of many important concepts as we move toward future advanced architectures. These include, but are not limited to:
    - SIMD instruction-level parallelism
    - Loop-level multithreading and GPU device dispatch
    - Exploration of data permutations to increase locality/cache reuse and/or reduce synchronization dependencies
    - Asynchronous task-level parallelism and lock-free loop ordering
    - Fine-grained loop execution rollback for transient fault recovery

  A code can be parameterized to employ different execution mechanisms at compile-time to run efficiently on different machines by placing loop iteration templates and execution policies in header files. RAJA also provides a bridge between code-specific and programming model constraints, and simplifies the development of "work-arounds" to improve compiler/runtime-system performance.

- **Performance**
  Introduction of fine-grained loop threading via RAJA has been shown to yield performance benefits over the current MPI-only parallelism model employed in LLNL production codes. For example, introducing 4-way OpenMP threading on inner loops in Ares led to a 50% overall speedup over the typical 1 MPI task/core run strategy on BG/Q. We studied the OpenMP threading performance impact in Kull on a loop-by-loop basis and observed that many loops showed speedup; for example, 2x to more than 3x speedup using 4 cores on BG/Q. While similar performance gains

could be obtained from applying OpenMP directives directly in these codes, by using a model like RAJA, we hope to ensure that we have a platform for further improvements and/or programming model options without unworkable application code disruption.

On TLCC2, on the other hand, the introduction of inner loop threading showed no real advantage or even performance degradation, except for LULESH, which is much more highly-tuned for fine-grained parallelization than production application codes. This is not unexpected, because fine-grained threading on standard multi-core CPUs, with out-of-order instructions, does not show significant benefit in most cases. More importantly, current OpenMP runtime overheads are too high to make fine-grained inner loop threading viable. As the overheads are reduced in the future, we hope that such fine-grained loop threading will become advantageous.

- **Long-term viability**
  We believe that RAJA has the potential to enable new and legacy codes to be portable across the range of hardware node architectures we anticipate in the foreseeable future. However, the variety of alternatives vendors are exploring leaves us with some uncertainty and concerns about RAJA. Support for the C++11 standard (lambda functions, in particular) is not ubiquitous or robust at this point. We are working with compiler vendors to resolve key issues (see Appendix), and we believe that optimization and performance will improve as compiler support matures. Other efforts such as Kokkos, Thrust, etc. will also benefit from improved lambda support. Also, it is unclear at this point how well RAJA will work with other programming models that we have not tried yet, and those, like the OpenMP 4 device model, that are being developed. Thus far, we have explored only a few of the more common options applicable to the current main platforms at LLNL. Our goal has been to "do no harm" to code maintainability and performance while achieving a modest level of portability. A compelling demonstration of performance benefits in production codes has not been achieved yet. Independent of RAJA, OpenMP overheads are still too high for fine-grained multithreading in production codes without significant code and algorithm restructuring. Furthermore, we have observed that when C++ templates and lambda functions are combined with OpenMP directives, compilers tend to shut down many optimizations that are accessed when OpenMP pragmas are placed directly on traditional for-loops. We describe our findings in the Appendix and are working with compiler vendors to address the problems.

In the future, we will explore RAJA in new programming model (OpenMP 4, CUDA, etc.) and hardware (GPUs, Intel MIC) environments. We will also broaden our explorations of LLNL codes to include additional physics algorithms and problems of interest. Lessons learned from applying RAJA to production codes so far is driving future development and code transformations that will be beneficial for future architectures, independent of RAJA.

We are continuing to explore features of RAJA that could enable a physics application coding style that looks increasingly like serial code, while exposing more parallelism through encapsulation of loop execution details. We are exploring ways to expand memory-layout and placement choices that are not possible without a model like RAJA. Our overarching goal is to allow multiphysics code developers to easily tailor implementation choices to different architectures, compilers, and programming models with minimal code disruption as these concerns are encapsulated completely within the RAJA layer.

Acknowledgements

We thank Esteban Pauli for doing most of the RAJA integration and performance experiments in Ares, and Koushik Ghosh, Aaron Black, et al. for initial explorations of RAJA integration in Kull.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-TR-661403.

Appendix

From the start of our RAJA development, we observed numerous unexpected performance issues. After considerable investigation, we identified specific compiler deficiencies as root causes and discovered "work-arounds" that resolved some issues. Then, we engaged several vendor compiler teams and made concrete recommendations to fix the problems and provided evidence of their feasibility. The issues are described in a LLNL technical report [10]. We also created the LCALS: Livermore Compiler Analysis Loop Suite benchmark to study and monitor the issues [11]. This benchmark is a suite of loops ("Livermore Loops" modernized and expanded, essentially) implemented in C++ that uses various software constructs, including RAJA. The suite has formed the basis for much of our dialogue with compiler vendors. It is used to:

- Generate simple test cases for vendors showing specific optimization and language support issues
- Try solution suggested and provided by vendors and report findings
- Introduce and motivate RAJA-like encapsulation concepts not previously on vendors' "RADAR"
- Track version-to-version compiler performance

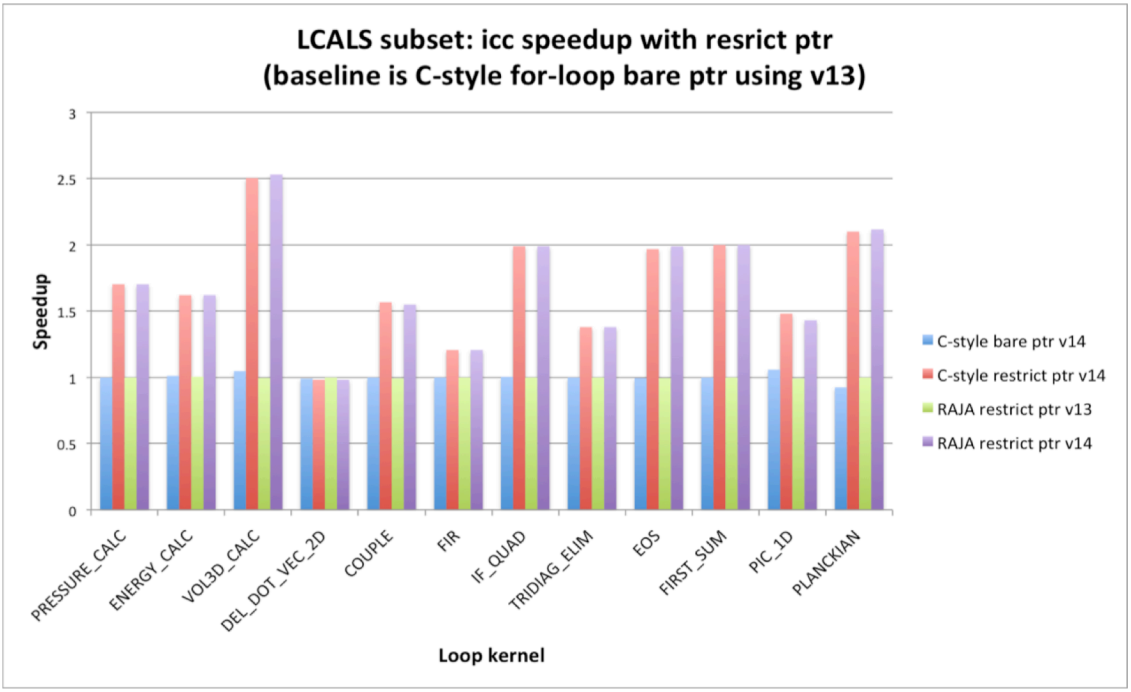

Figure 13. Working with the Intel compiler team, we have resolved SIMD optimization issues involving C++ lambda variable-capture and the Intel compiler. This speedup comparison plot shows that the RAJA-style loops perform as well as C-style loops with restrict applied to data pointers via a typedef meachanism with the v14 Intel compiler (red and purple bars, respectively). With the v13 version of the compiler, the restrict semantics were not enforced and so RAJA-style performance was the same as C-style loops without restrict applied (green and blue bars, respectively).

The performance issues we have observed are due to deficiencies in compilers, runtimes, and programming language/model standards. For example, the C++11 standard is relatively new, especially lambda functions that we view as essential for RAJA adoption in production codes. Ideally, RAJA-style traversal templates should be inlined and optimized at each loop

site where they are used. A key issue we identified was that compiler hints we provided on data pointers were not being propagated through the lambda variable-capture mechanism in many cases. Effectively this prevented most SIMD vectorization optimizations. We worked this issue with the Intel compiler team and it was resolved over the course of a couple of recent major compiler releases. Figure 13 illustrates this for a subset of LCALS. We believe that other issues like this can be resolved if adequate attention is given to them. We are optimistic this will happen as lambda use expands and matures.

Another key deficiency we have observed is that compilers do not optimize aggressively when OpenMP directives are used. For example, we have shown that SIMD is often disabled. Also, we see little evidence that compilers optimize across adjacent parallel regions. This has potentially grave implications for fine-grained use of "`#pragma omp parallel for`" in HPC codes. The most troubling issue we have encountered is that when OpenMP is combined with C++ language features, such as templates, many optimizations are not applied. That is, optimizations that are applied when OpenMP directives are used on C-style for-loops are absent when a RAJA-style abstraction layer is used. Moreover, the observed performance differences can be huge and vary by loop and compiler. Figure 14 compares runtimes on BG/Q for RAJA-style loops using C++ lambda functions to represent the loop bodies and the GNU compiler (left), and RAJA-style loops using C++ functors to represent the loop bodies and the IBM xlc compiler (right). The baselines for comparison in either case are the same loop kernels with OpenMP directives placed on C-style ("Raw") for-loops for the given compiler. Note that relative performance difference in each case is roughly the same across the range of threads shown; i.e., each curve is effectively flat. In the left plot (GNU compiler) the runtime difference among the loops only spans 0 –20 %. However, in the right plot, runtime differences are as large as 15 – 20 times for some loops. It is interesting to note that order of loop kernels, with respect to which differs least/most, is completely different between the two plots. It is also interesting to note that if one looks at thread scaling for this study, it appears that all loops scale well achieving speedup factors of 12 – 16 times at 16 threads. Similar behavior is observed on

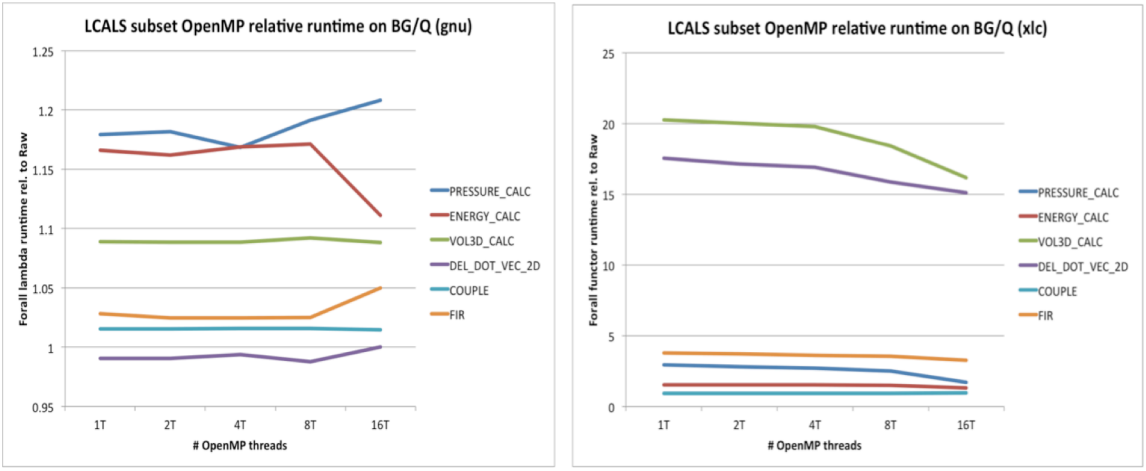

Figure 14. Loop runtimes can be vastly different when OpenMP directives are used on "Raw" C-style for-loops vs. applied within RAJA-style traversal templates. These plots compare relative runtime for a subset of LCALS, RAJA-style vs. C-style, from 1 to 16 threads on BG/Q. In the left plot, RAJA loops use C++ lambda functions and the code is built with the GNU compiler. Loop runtime differences vary ~ 0 – 20%. On the right, RAJA loops use C++ functors and the code is built with the IBM xlc compiler. Loop runtime differences are 15 – 20 X in some cases.

TLCC2 platforms with the Intel compiler. We are currently working with the Intel team to resolve the issue.

Small additions and clarifications in the C++ and OpenMP standards could have a large, positive performance impact on HPC codes by making it possible to develop techniques, like RAJA, for portability, by way of complete encapsulation of platform-specific concerns, without adversely affecting performance. Other efforts we have mentioned in this report are exploring C++ abstractions to encapsulate hardware-specific concerns. Resolving the issues would benefit a broad community.

References

1. Chapman, B., G. Jost, and R. Van Der Pas, "Using OpenMP: Portable Shared Memory Parallel Programming", MIT Press, Scientific and Engineering Computation Series, Cambridge, MA (2008).
2. OpenMP Application Program Interface, version 4.0, OpenMP Architecture Review Board (2013).
3. Intel Cilk Plus website, www.cilkplus.org.
4. CUDA C Programming Guide, http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (2014).
5. Thrust website, https://developer.nvidia.com/thrust.
6. Bolt website, http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-libraries/bolt-c-template-library/.
7. Edwards, H. C., 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 (PMAM '12),* ACM, New York, NY, pp. 1—10 (2012).
8. C++ language reference website, http://en.cppreference.com/w/cpp/language/lambda.
9. ROSE Compiler Project website: www.rosecompiler.org.
10. Hornung, Richard D., and Jeffrey A. Keasler, "A Case for Improved C++ Support to Enable Performance Portability in Large Physics Simulation Codes", LLNL-TR-653681 (2013). Available online: https://codesign.llnl.gov/codesign-papers-presentations.php.
11. Hornung, Richard D., LCALS: Livermore Compiler Analysis Loop Suite. Available at https://codesign.llnl.gov/LCALS.php.