# Structure Slicing: Extending Logical Regions with Fields

Michael Bauer
Stanford University
mebauer@cs.stanford.edu

Sean Treichler
Stanford University
sjt@cs.stanford.edu

Elliott Slaughter
Stanford University
slaughter@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

***Abstract*—Applications on modern supercomputers are increasingly limited by the cost of data movement, but mainstream programming systems have few abstractions for describing the structure of a program's data. Consequently, the burden of managing data movement, placement, and layout currently falls primarily upon the programmer.**

**To address this problem we previously proposed a data model based on *logical regions* and described Legion, a programming system incorporating logical regions. In this paper, we present *structure slicing*, which incorporates *fields* into the logical region data model. We show that structure slicing enables Legion to automatically infer task parallelism from field non-interference, decouple the specification of data usage from layout, and reduce the overall amount of data moved. We demonstrate that structure slicing enables both strong and weak scaling of three Legion applications including S3D, a production combustion simulation that uses logical regions with thousands of fields, with speedups of up to 3.68X over a vectorized CPU-only Fortran implementation and 1.88X over an independently hand-tuned OpenACC code.**

## I. Introduction

Modern supercomputers have evolved to incorporate deep memory hierarchies and heterogeneous processors to meet the demands of performance and power efficiency from the computational science community. Each new architecture has been accompanied by new software for extracting performance on the target hardware [1, 2, 3, 14, 18]. While these programming systems contain many ways for describing parallelism, they offer little support to programmers for managing data. Consequently, the responsibility for orchestrating all data movement, placement, and layout both within and across nodes falls primarily on the programmer. At the same time, data movement has become increasingly complex and expensive and now dominates the performance of most applications.

To aid programmers in managing program data, we previously introduced Legion, a data-centric programming model based on *logical regions* [6, 25, 26]. Logical regions are typed collections that can be recursively partitioned into logical subregions, thereby allowing applications to name the subsets of data used by different sub-computations. Logical regions further benefit applications by decoupling the specification of which data is accessed by computations from where that data is placed within the memory hierarchy. Using a *mapping interface*, Legion applications separately control where instances of logical regions are placed in the machine [6].

Despite these advantages, our experience writing Legion programs over the last two years has convinced us that a fundamental dimension was missing from the design: describing compound data types with multiple *fields*. Consider, for example, the combustion simulation S3D [11], which was one of the six applications used for acceptance testing of Titan [19], the current number two supercomputer [4]. S3D models both the physics of turbulent gas dynamics as well as the chemistry of combustion via direct numerical simulation. Physical space is discretized into a grid of cells and each cell maintains more than a thousand intermediate values or fields corresponding to different physical and chemical properties (e.g., temperature, pressure, etc.). Each cell can therefore be considered a structure with numerous field values. S3D contains many computational phases, each of which is primarily a physics or chemistry phase, as shown in Figure 1. Physics phases are mainly *stencils* that access neighboring cells, but require only a few fields from each cell. Chemistry phases require many fields, but always from a single cell.

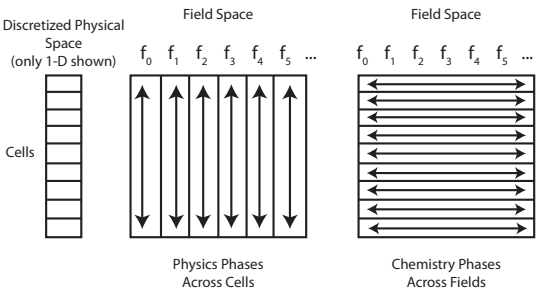



Fig. 1. Across-Cell and Across-Field Phases in S3D.

In practice, many scientific computing applications share S3D's structure: there are one or more large distributed data structures (regular grid, irregular mesh, graph) and each element within these data structures holds many tens, hundreds, or even thousands of field values. Furthermore, all computations on these data structures go through phases operating on different subsets of these fields. The crucial insight is that few, if any, computations in an application require access to all of the fields. In this paper, we present *structure slicing* as an extension to Legion which allows computations to explicitly slice logical regions to name the subsets of fields they will access. Armed with this information, a field-aware Legion implementation can automatically derive several important performance benefits.

- *Task Parallelism*: Legion can automatically infer implicit task parallelism between computations using disjoint sets of fields.
- *Decoupled Data Layout*: The specification of data

SC14, November 16-21, 2014, New Orleans, LA, USA
978-1-4799-5500-8/14/$31.00 

usage for computations can be decoupled from the actual data layout by making data layout a Legion mapping decision.

- *Reduced Data Movement*: By knowing precisely which fields are required for tasks, Legion can automatically determine the minimum subset of data that needs to be communicated when computations are moved to new processors on remote nodes or accelerators with their own address spaces.

The rest of this paper is organized as follows. In Section II we motivate our design by giving concrete examples of how S3D benefits from structure slicing. Each of the remaining sections presents one of our technical contributions:

- We describe the semantics of structure slicing, including the need for dynamic field allocation. We detail the incorporation of fields into Legion's definition of non-interference and show how Legion's mapping interface decouples data layout from data usage (Section III).
- We describe the implementation of structure slicing in Legion, including the necessary changes to Legion's dynamic dependence analysis as well as the implementation of data movement routines for managing field data (Section IV).
- We port several applications, including a full implementation of S3D, to use a structure slicing Legion implementation. We demonstrate that the performance benefits conferred by structure slicing enable both strong and weak scaling (Section V).

Section VI describes related work and Section VII concludes.

## II. Motivation

To motivate our design, we begin by presenting a small code example from S3D that illustrates the need for structure slicing. Listing 1 shows a short pseudo-code excerpt from the *right-hand side function* `rhsf` of S3D (lines 57-66). The `rhsf` function evaluates the values on the right-hand side of the Navier-Stokes partial differential equations and is parameterized to operate across a range of chemical mechanisms. The `rhsf` function is invoked multiple times per time step on each node by an explicit Runge-Kutta solver and routinely consumes in excess of 97% of the execution time of an S3D run. In this section we focus on the computation performed by the `rhsf` function on each node.

The `rhsf` function operates on an array of cells. The `Cell` type (declared on lines 1-13) shows the first 42 fields of the 139 field struct for H2, the smallest chemical mechanism. Interesting mechanisms used in real research, such as dimethyl ether (DME) and heptane [11], require 548 and 1046 fields per cell respectively. In the original Fortran, these fields are implicitly encoded as a fourth dimension on every array, with the ordering of dimensions fixing the layout of the data. In both C and Fortran versions of the code, modifying the data layout would require changes to a large fraction of the 200K lines of code in S3D.

In Section III we demonstrate how Legion allows data layout to be decoupled from the specification of program data. In a heterogeneous environment, decoupling of data specification from layout using structure slicing facilitates

```
1  struct Cell {
2    double avmolwt, mixmw, temp, viscosity, lambda, pressure,
3      yspec_h2, yspec_o2, yspec_o, yspec_oh, yspec_h2,
4      yspec_h20, yspec_h, spec_h02, yspec_oh,
5      ds_mixarg_h2, ds_mixarg_o2, ds_mixarg_o,
6      ds_mixarg_oh, ds_mixarg_h20, ds_mixarg_h, ds_mixarg_h02,
7      ds_mixarg_h202, ds_mixarg_n2, grad_vel_x_x,
8      grad_vel_x_y, grad_vel_x_z, grad_vel_y_x, grad_vel_y_y,
9      grad_vel_y_z grad_vel_z_x, grad_vel_z_y, grad_vel_z_z,
10     grad_yy_h2, grad_yy_o2, grad_yy_o, grad_yy_oh, grad_yy_h20,
11     grad_yy_h, grad_yy_h02, grad_yy_h202, grad_yy_n2,
12     tau_x_x, tau_x_y, tau_x_z, tau_y_y, tau_y_z, tau_z_z, ...
13 };
14
15 /* Example across−cell computation: gradient molar species */
16 /* Only one direction of stencil shown */
17 #define STENCIL_1D(x, field, ae, be, ce, de) \
18   (ae * (cells[x+1].field − cells[x−1].field)) + \
19   (be * (cells[x+2].field − cells[x−2].field)) + \
20   (ce * (cells[x+3].field − cells[x−3].field)) + \
21   (de * (cells[x+4].field − cells[x−4].field))
22
23 void calc_grad_yy(Cell *cells, int num_cells, double dim_size) {
24   double ae = 4.0 / 5.0 * dim_size;
25   double be = −1.0 / 5.0 * dim_size;
26   double ce = 4.0 / 105.0 * dim_size;
27   double de = −1.0 / 280.0 * dim_size;
28   for (int i = 0; i < num_cells; i++) {
29     cells[i].grad_yy_h2 = STENCIL_1D(i, yspec_h2, ae, be, ce, de);
30     cells[i].grad_yy_o2 = STENCIL_1D(i, yspec_o2, ae, be, ce, de);
31     /* ... */
32     cells[i].grad_yy_n2 = STENCIL_1D(i, yspec_n2, ae, be, ce, de);
33   }
34 }
35
36 /* Example across−field computation: stress tensor */
37 void calc_tau(Cell *cells, int num_cells) {
38   for (int i = 0; i < num_cells; i++) {
39     double sum_term = cells[i].grad_vel_x_x +
40       cells[i].grad_vel_y_y + cells[i].grad_vel_z_z;
41     cells[i].tau_x_x = 2.0 * cells[i].viscosity *
42               (cells[i].grad_vel_x_x − sum_term);
43     cells[i].tau_y_y = 2.0 * cells[i].viscosity *
44               (cells[i].grad_vel_y_y − sum_term);
45     cells[i].tau_z_z = 2.0 * cells[i].viscosity *
46               (cells[i].grad_vel_z_z − sum_term);
47     cells[i].tau_x_y = cells[i].viscosity *
48               (cells[i].grad_vel_x_y + cells[i].grad_vel_y_x);
49     cells[i].tau_x_z = cells[i].viscosity *
50               (cells[i].grad_vel_x_z + cells[i].grad_vel_z_x);
51     cells[i].tau_y_z = cells[i].viscosity *
52               (cells[i].grad_vel_y_z + cells[i].grad_vel_z_y);
53   }
54 }
55
56 /* Right−Hand Side Function (RHSF) */
57 void rhsf(Cell *cells, int num_cells) {
58   calc_volume(cells, num_cells);
59   calc_temperature(cells, num_cells);
60   calc_thermal_coefficients(cells, num_cells);
61   calc_grad_yy(cells, num_cells);
62   calc_tau(cells, num_cells);
63   calc_diffusion_flux(cells, num_cells);
64   calc_reaction_rates(cells, num_cells);
65   /* ... */
66 }
```

Listing 1. S3D Right-Hand Side Function.

using different data layouts for tasks executed on different kinds of processors. For example, tasks run on GPU processors will likely perform best with a *struct-of-arrays* data layout for memory coalescing, while tasks run on CPU processors will often perform best with *array-of-structs* or hybrid [21] data layouts to leverage cache prefetch engines.

The `rhsf` function invokes two different kinds of functions on the array of cells in Figure 1. The `calc_grad_yy` function on lines 17-34 of Listing 1 (which computes the gradient of the molar fractions for each species using a stencil

computation) is an example of an across-cells function. In contrast, `calc_tau` on lines 37-54 (which computes the stress tensor for each cell using other fields within the same cell) is an example of an across-fields function. In S3D there is no function that requires access to all of the fields.

While the implementation of `rhsf` portrayed in this example is a sequential function, there exists significant field-level task parallelism among its subroutines. For example, it is safe to execute the adjacent functions `calc_grad_yy` and `calc_tau` in parallel because they access independent sets of fields and are therefore *non-interfering*, even though one function operates across fields and the other operates across cells. (We formally define non-interference in Section III.) In practice, it is common for there to exist in excess of 100-way task parallelism between the hundreds to thousands of fields in the `Cell` data type for larger mechanisms in S3D. We show in Section III how structure slicing enables Legion to implicitly infer field-level non-interference between tasks.

Structure slicing also enables two important data movement optimizations. First, on machines with hierarchical memory, structure slicing permits Legion to know exactly which fields must be moved for computations to run on an accelerator. For example, off-loading the data parallel `calc_tau` task onto a GPU requires only copying data from the `viscosity` and `grad_vel` fields into the GPU's framebuffer. Since these are a small subset of the total fields in a `Cell`, there is a significant improvement in performance by only moving the needed field data. We note that a version of S3D using OpenACC [19] can perform a similar operation, under the condition that data is laid out in system memory using a struct-of-arrays format so individual field data is dense, allowing OpenACC to copy individual dimensions of the array. Entangling layout with data movement optimizations in the OpenACC code results in code that is difficult to modify when exploring different mapping strategies and tuning for new architectures.

The second data movement optimization enabled by structure slicing is copy elimination. Consider two instances of the `grad_vel` task being performed for the same species in different dimensions. Both of these tasks require access to the `yspec` field for a given species. In the case where both tasks are mapped onto an accelerator, a structure slicing Legion implementation will dynamically detect the redundancy in data movement and ensure that the data is only moved once. This optimization can also be done in an OpenACC version of the code, provided the mapping decisions are known statically, but it is a tedious and error prone transformation to perform by hand in the presence of the thousands of fields in each S3D chemical mechanism. To compound matters, the optimization is implicit in the code, reducing code maintainability, and potentially introducing bugs under code refactoring. We cover the details of how Legion automatically performs both data movement optimizations in a safe way in Section IV.

## III. Structure Slicing

In this section we present extensions to the Legion programming model for structure slicing. We describe how fields are added to logical regions (Section III-A), how field information is used to infer non-interference of tasks (Section III-B), and extensions to the Legion mapping interface for controlling data layout (Section III-C).

### A. Logical Regions with Fields

In the original Legion design [6] (which for brevity we will call *original Legion*), logical regions are a cross product of an *index space* $I$ [9, 15] and a type $T$. An index space is an abstract set which defines the set of entries in a logical region (e.g., a set of opaque pointers, or a set of points in a Cartesian grid of arbitrary dimensions). For each point in $I$, there is a corresponding object of type $T$ in the logical region. The type $T$ is not restricted and can be either a base data type or a compound data type. Logical regions can be subdivided by dynamic *partitioning* operations, which subset the index space to define subregions.

To extend logical regions to support structure slicing, we introduce *field spaces*. Instead of using statically defined types, logical regions are created at runtime by taking the cross product of an index space $I$ and a field space $F$. Each field $f \in F$ has a type $T_f$. A pair $\langle i, f \rangle$ where $i \in I$ and $f \in F$ uniquely identifies an entry of type $T_f$ in the logical region.

Fields can be dynamically added to and removed from logical regions. To understand the need for dynamically allocated fields, consider a program that computes a temporary for every element in a collection and then performs a stencil computation over that temporary before summing the result with another field. The need for such temporary fields is common, and in many cases programmers will allocate *scratch* fields in static data types that are re-used throughout long computations like S3D's `rhsf` function.

There are two problems with how scratch fields are allocated and used in current programs. First, scratch fields consume memory at all times. Programmers often address this memory bloat by reusing a single scratch field for several (hopefully non-overlapping) temporary variables, at a significant cost to code maintainability. However, reuse of scratch fields can introduce false dependencies between two otherwise independent tasks that happen to reuse the same scratch field. Second, scratch fields add overhead to data transfers: The scratch fields of a structure in C or Fortran are copied along with all the other fields, whether they have data that will be used by the receiving computation or not. For applications such as S3D that use a large number of scratch fields and are constrained by system bandwidth, the costs of copying unused scratch fields can be large.

By allowing the applications to add fields to a field space when needed and remove them after the last use, false dependencies can be eliminated and memory footprint and transfer costs reduced.

### B. Field-Based Non-Interference

A Legion application is decomposed into a hierarchy of *tasks*. Informally, a pair of tasks $t_1$ and $t_2$ is *non-interfering* (written $t_1 \# t_2$) if executing them in either order or concurrently cannot cause a difference in their behavior or the observed state of memory after both tasks have finished. The basic semantic guarantee of Legion is that if two tasks can interfere, then they will be executed in the original program order, thus giving Legion a default sequential semantics that is easy to understand and aids programmer reasoning.

To extract parallelism it is desirable for the Legion runtime to prove as many pairs of tasks are non-interfering as possible. To facilitate reasoning about non-interference, Legion requires that each task name the logical regions it will access. If two tasks access disjoint regions, for example, the runtime can prove the tasks are non-interfering and safely execute them in parallel. To enable finer grain reasoning when two tasks access the same region, Legion also requires tasks declare their *privileges* on region arguments (e.g., read-only, read-write, reduce). Thus, as another example, when two tasks access the same region $r$ but with read-only privileges, the two tasks are non-interfering on $r$.

A full treatment of non-interference in original Legion can be found in [25], which derives a sound approximation of the non-interference test that is efficient enough to be performed at run time. It also shows how the functional nature of Legion tasks with effects on logical regions enables a hierarchical scheduling algorithm and permits distributed non-interference tests to be performed on different nodes without communication, which is crucial for scaling Legion to thousands of nodes.

We now extend this framework to include non-interference on fields. Following the methodology of [25], we first define a precise non-interference test based on the actual execution of two tasks and then show a sound approximation that can be efficiently computed using region field privileges. In [25], a *memory operation* $\epsilon$ is a triple $(l, op, v)$ where $l$ is a memory location, $op$ is the operation performed on $l$ (*read*, *write*, or $reduce_o$ with a particular reduction operator $o$), and $v$ is the value (the value read, written or folded into $l$ using the named reduction operator). Non-interference of two memory operations $\epsilon_1$ and $\epsilon_2$ is then defined as follows:

$$\begin{aligned}\epsilon_1 \# \epsilon_2 \Leftrightarrow &(op_1 = read \wedge op_2 = read) \vee \\ &(op_1 = reduce_{id_1} \wedge op_2 = reduce_{id_2} \\ &\quad \wedge id_1 = id_2) \vee \\ &l_1 \neq l_2\end{aligned}$$

The first two conditions capture the non-interference of two read operations or two reduction operations (using the same reduction operator); these conditions are still sufficient for proving non-interference even under fields. The final condition looks for accesses to different memory locations $l_1$ and $l_2$. If we let $o_1$ and $o_2$ be the base address of the objects accessed by $\epsilon_1$ and $\epsilon_2$ and let $f_1$ and $f_2$ be the names of the fields being accessed, we can refine the notion of accessing the same location to the access of the same field within the same object

$$l_1 = l_2 \equiv (o_1 = o_2) \wedge (f_1 = f_2)$$

and then rewrite the non-interference test as

$$\begin{aligned}\epsilon_1 \# \epsilon_2 \Leftrightarrow &(op_1 = read \wedge op_2 = read) \vee \\ &(op_1 = reduce_{id_1} \wedge op_2 = reduce_{id_2} \\ &\quad \wedge id_1 = id_2) \vee \\ &\boldsymbol{o_1 \neq o_2} \vee \\ &\boldsymbol{f_1 \neq f_2}\end{aligned}$$

This reformulation splits the original different-location test into two tests, one that identifies data parallelism from accesses to different objects, and one that identifies task parallelism from accesses to different fields.

The non-interference test used in the original Legion runtime is an approximation that works at the granularity of privileges and regions rather than individual memory operations. This coarsening of the test is what makes it practical, as the cost of a single non-interference test is amortized over many accesses to the region's data. A further optimization is to perform the test based on logical regions rather than *physical instances*, the actual physical location(s) where data in that logical region is currently stored. (Multiple physical instances are permitted for a logical region when data has been replicated for improved access.) The analysis in [25] abstracts this translation of logical regions to physical instances in a mapping $M$ and shows that a valid mapping $M$ chosen by an application mapper preserves the soundness of the region-based non-interference test.

To incorporate structure slicing into the non-interference test, we augment the privilege declarations on tasks to also name the fields on each logical region that a task may access. Let task $t_i$ access the set of fields $\boldsymbol{fields}_i$ of region $r_i$ with the privilege $priv_i$, and let $M$ be the mapping of logical regions to physical instances. The test for non-interference of tasks $t_1$ and $t_2$, extended to include structure slicing, is given below. Here $r_1 * r_2$ is true if $r_1$ and $r_2$ are disjoint logical regions; $M(r_1) \cap M(r_2)$ is empty if the physical instances $M(r_1)$ and $M(r_2)$ share no memory locations.

$$\begin{aligned}&priv_1(r_1, \boldsymbol{fields_1}) \#_M priv_2(r_2, \boldsymbol{fields_2}) \Leftrightarrow \\ &\quad (priv_1 = \text{reads} \wedge priv_2 = \text{reads}) \vee \\ &\quad (priv_1 = \text{reduces}_{id_1} \wedge priv_2 = \text{reduces}_{id_2} \wedge id_1 = id_2) \vee \\ &\quad (r_1 * r_2) \vee \\ &\quad (M(r_1) \cap M(r_2) = \emptyset) \vee \\ &\quad \boldsymbol{(fields_1 \cap fields_2 = \emptyset)}\end{aligned}$$

This test is nearly identical to the one in [25]. The existing Legion runtime non-interference checks (which extract other forms of parallelism) are left unaffected, and a single new sufficient condition is added, which checks whether the sets of fields accessed by two tasks for a given privilege are disjoint. This additional check is performed dynamically (to support dynamic field spaces); we discuss an efficient implementation in Section IV. The similarity of the test allows a straightforward extension of the theorems proving the soundness of the approximate test and its suitability as the basis for scalable hierarchical scheduling.

### C. Field-Based Mapping

Legion features a dynamic mapping interface that decouples writing applications from tuning performance on specific target architectures [6]. Programmers work with logical regions with no implied layout or location in the memory hierarchy. During program execution the Legion runtime queries a *mapper object* about how the application should be mapped to the target architecture. Prior to the incorporation of structure slicing, mapping a task $t$ in Legion required a mapper to respond to the following queries from the Legion runtime:

- Select the target processor for the task $t$
- For each logical region $r$ that $t$ has privileges, select a target memory in which to create or reuse a physical instance of $r$

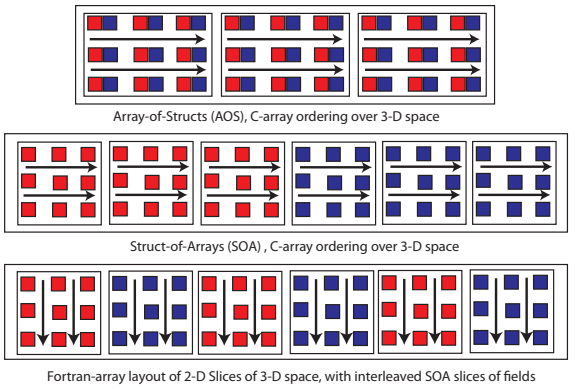



Fig. 2. Example physical region layouts for a 2-D stencil.

Initially applications use the Legion default mapper which answers these queries using basic heuristics. Legion applications are then tuned by customizing mappers based on application-specific and/or architecture-specific knowledge. Most importantly, mapping decisions made by mappers cannot impact the correctness of Legion applications, which is necessary for easy porting and tuning of Legion code [6].

To incorporate structure slicing into the Legion mapping interface we extend mapper objects to specify the layout of fields within physical instances. For each logical region requested for a task, Legion mappers are now queried to select a *layout schema* constraining how the region's index space and fields should be linearized in memory when creating a physical instance. For example, layout schemas can encode that fields should be laid out in struct-of-arrays (SOA) format, with data for each field compactly stored. Alternatively, data can be laid out in array-of-structs (AOS) format, or a hybrid format [21] that allows several values for each field to be stored compactly for use with vectorized SSE or AVX loads and stores. Layout specifications also require mappers to specify the ordering of points within the index space for a logical region. Ordering can be done by dimensions (e.g., C or Fortran array ordering) or alternatively with more flexible functions such as Morton space filling curves. Finally, layout schemas are flexible enough to interleave field data with different subsets of index spaces.

As an illustrative example, consider a simple 2-D stencil computation done over a 3-D grid with two different fields: A and B. Figure 2 shows three different layouts that could be selected for the physical instance with elements of field A shown in red and elements of field B shown in blue. The first layout shows the standard AOS layout with the grid serialized based on a C-ordering of the dimensions. This layout would be well suited to CPU kernels which perform both stencils simultaneously. Alternatively, the second layout depicts the standard SOA layout of the fields, with the array for each field serialized using C-ordering of the dimensions. The last layout in Figure 2 shows how fields can be interleaved with index space dimensions, with 2-D slices of different fields alternating in memory and each slice laid out with a Fortran-ordering on the grid dimensions. Such a layout would be extremely useful for describing locality if both fields are necessary for performing the stencil, but coalescing of memory accesses are necessary for use on a vectorized or GPU processor.

In order to maintain the invariant that mapping decisions are independent of the correctness of Legion applications, we need to update our Legion implementation to handle various layout specifications. In addition to handling the necessary data movement operations, a Legion implementation also needs to apply the necessary transformations when moving data between physical instances with different data layouts. As we show in Section IV, the cost of transforming data to support different layouts is often minimal compared to the cost of data movement itself and any overhead is quickly made up by gains in memory system performance when executing tasks.

## IV. Structure Slicing Implementation

We now describe an implementation of structure slicing in Legion. We first summarize necessary extensions to the Legion runtime interface (Section IV-A). We then present the three primary modifications to the runtime system: support for dynamic field non-interference tests (Section IV-B), modifications to the mapping interface (Section IV-C), and extensions to data movement routines (Section IV-D).

### A. Interface Extensions

To support structure slicing, we extend the original Legion interface to support the dynamic creation and deletion of field spaces as well as the dynamic creation and deletion of fields. As discussed in Section III, dynamic logical region creation with structure slicing uses field spaces to specify the available fields on a logical region. To support dynamic allocation and deallocation of fields, we exploit the natural level of indirection between Legion's logical regions and physical instances. The dynamic allocation of a field $f$ on a logical region $r$ guarantees that $f$ is available on future physical instances of $r$. Similarly, the destruction of a field $f$ on $r$ ensures that the field need not be allocated as a part of future physical instances.

We also extend the task launching mechanism to encode the necessary information for structure slicing. Previously, Legion tasks declared the logical regions they could access along with any privileges [6]. In our structure slicing version of Legion, instead of specifying privileges on entire regions, tasks request privileges on individual fields of logical regions.

### B. Dynamic Field Non-Interference Tests

The primary challenge in extending a Legion implementation to incorporate structure slicing information is performing efficient dynamic non-interference tests. The introduction of an additional dimension of non-interference adds dynamic analysis overheads; however, we show in Section V that the cost of this analysis can be minimized and often pays for itself by discovering additional task and data movement parallelism.

In Legion, subtasks launched within a parent task are issued in program order to the runtime system. For each subtask $t$ the runtime must perform non-interference tests between $t$ and any unfinished subtasks launched within the same parent task [6]. Recall from Section III-B that the hierarchical nature of the non-interference test obviates the need to test against any tasks launched by another parent [25]. Extracting parallelism from this stream of tasks is an inherently sequential process. A poor implementation could easily place this analysis on an application's critical path, so it is important that the non-interference tests be implemented efficiently.

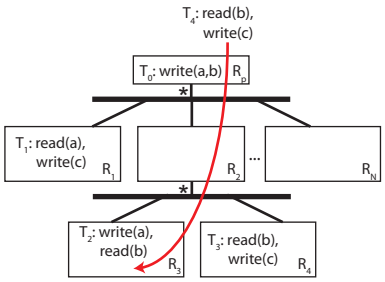



Fig. 3. Example Legion Non-Interference Test

In Section III-B we defined the three disjunctive conditions for demonstrating the non-interference of two tasks:

- access to independent sets of fields,
- access to independent sets of regions,
- or non-interfering privileges.

These three conditions may be tested in any order. If any one of the three tests evaluates to *true* then the tasks are non-interfering and any remaining tests can be skipped. Consequently the choice of test order can have a significant performance impact.

The order we use is region independence, field independence, and finally non-interference of privileges. This ordering stems from the observation that Legion programs commonly express data parallelism at several different granularities both across nodes and within nodes. It is therefore most likely that two tasks will be proven to be non-interfering using the independence of their logical regions. After this, field set independence is most likely. Finally, privilege non-interference is a more complicated test and is therefore placed last where it is least likely to be performed. While it is possible to write Legion applications that perform better with a different test ordering, it has been our experience that the performance of this ordering is sufficient for achieving high non-interference test throughput. We justify this choice of ordering by showing the breakdown of non-interference tests for several real applications in Section V-C.

We now build on the Legion non-interference test from [6, 25]. A *region tree* consists of two kinds of nodes, regions and partitions. Each region may have one or more child partitions, and each partition has a number of subregions. An example region tree is shown in Figure 3, where boxes are regions and horizontal lines are partitions (the $*$ notation on a partition indicates the subregions of the partition are disjoint).

At every point in time, the Legion runtime tracks which already-issued tasks are using which regions by annotating the logical regions in region trees with task and privilege information. To extend this algorithm to check field non-interference, we store the set of fields used in addition to the privilege information with each task in the region tree, as illustrated in Figure 3. Each task stored in the region tree has already performed its non-interference test and recorded itself on the regions it is using along with its field and privilege information. To perform the needed non-interference tests for a new task $T_4$, we begin by traversing the region tree from the root to where the task has requested privileges, in this case the logical region $R_3$. We only need to perform non-interference tests with tasks along this path because regions off of this path (such as $R_1$ in Figure 3), are already known to be disjoint with $R_3$. The tasks $T_1$ and $T_3$ can therefore be inferred to be non-interfering on logical region usage. Performing this traversal implements the first dimension of our non-interference tests, which examines region usage.

For all tasks that are encountered along the path, we apply the second and third dimensions of the non-interference test in order. We first check for non-interference on fields. Task $T_4$ is non-interfering with both $T_0$ and $T_2$ on fields $a$ and $c$, but interferes with both tasks on field $b$. For the interfering field $b$, we then perform privilege non-interference tests. In this case $T_4$ interferes with $T_0$ because $T_0$ is writing field $b$ and $T_4$ is reading field $b$, resulting in a true data dependence. However, because both $T_2$ and $T_4$ are reading field $b$, task $T_4$ is non-interfering with $T_2$ and can potentially run in parallel.

For applications such as S3D with large numbers of fields, it is challenging to implement non-interference tests on sets of fields efficiently. To make these tests fast we implement sets of fields as bit masks. For every region and privilege that a task requests, a bit mask is inserted into the region tree summarizing the fields used for the specific region and privilege. Using bit-wise operators, fast tests for both disjointness and intersection can be performed.

To further improve the performance of bit mask disjointness testing we place a compile-time upper bound on the number of fields that can be allocated in a bit mask, which allows fixed storage to be allocated for each bit mask. The runtime dynamically maps field allocations for a field space onto indices in the bit mask, and frees up indices when fields are deallocated. While this does place a restriction on the total number of fields in a field space, it does not limit the total number of fields in a program, as our implementation supports an unbounded number of field spaces. Additionally, the upper bound on fields can be programmer controlled, and increasing the upper bound simply increases the cost of the dynamic non-interference analysis.

By fixing an upper bound on the number of fields in a field space, we can optimize the representation of the bits in the bit mask. In the general case, bit masks are represented with unsigned 64-bit integers. However, when the hardware supports it, 128-bit SSE and 256-bit AVX vector data types are used along with the corresponding vectorized instruction for performing logical bit manipulation.

The fixed upper bound on the number of bits in a bit mask also permits another important optimization. The most common operations on bit masks are testing for disjointness, intersection, and testing emptiness (e.g., whether any bits are set. For cases where the upper bound on the number of bits is large (e.g., more than a 1K bits, which is common in S3D) all three of these operations can be accelerated by maintaining a single 64-bit word summary mask that represents the logical union of all 64-bit words in the bit mask. The test for disjointness, intersection, and emptiness can be accelerated by performing them on the summary masks first, and only performing the full test if needed. These *two-level bit masks* are extremely important for applications such as S3D where the upper bound on the number of fields in a field space is large, but most tasks only request privileges on a few fields.

### C. Field-Aware Mapping

In addition to modifying the Legion runtime interface, we also extend the original Legion mapping interface. The original Legion mapping interface requires mappers to specify on which processors tasks should be run and the memory in which to place each physical instance of a requested logical region. To support structure slicing, we also require mappers to specify the layout of data in each physical instance using the layout schemas described in Section III-C.

To support this feature, the Legion runtime stores additional meta-data for physical instances. In addition to tracking the memory location for a physical instance and whether it contains valid data, the Legion runtime also tracks which fields contain valid data as well as the layout schema for the physical instance. To aid mappers in making intelligent mapping decisions, this meta-data is also made available through the mapping interface to allow mappers to know where current instances reside and their data layout. Using this information, mappers can either choose to use an existing physical instance with a given data layout, or create a new instance and specify the desired layout.

Supporting dynamically chosen data layouts challenges an important principle of the Legion programming model. Legion guarantees that mapping decisions cannot impact the correctness of an application. To maintain this property, the runtime provides generic *accessor* objects which introduce the necessary level of indirection to mediate the reading and writing of physical instances with arbitrarily chosen layouts. Accessors come in two flavors: *generic* and *specialized*. Generic accessors work for all layouts, while specialized accessors only work for a subset of layouts. Applications can register multiple functionally equivalent task variants using combinations of generic and specialized accessors. The Legion runtime will automatically select a specialized task variant if a compatible one exists for the specified physical instances, or choose a variant with generic accessors for handling arbitrary data layouts if no specialized variant exists.

### D. Field-Aware Data Movement

As part of its analysis, the Legion runtime tracks the physical instances that contain valid data for each logical region in a Legion program. To support structure slicing, we extend our implementation of the Legion runtime to store data specifically regarding which fields contain valid data in each physical instance. At different times, different subsets of the fields within an instance may contain valid data while other fields are invalid. In the case when a mapper chooses to re-use an existing physical instance, the Legion runtime can automatically determine which fields contain valid data, and which fields require copies to acquire valid data. As mentioned in Section II, this knowledge permits Legion to automatically perform copy elimination and determine when data in individual fields within physical instances can safely be reused by multiple tasks.

While Legion can easily infer the necessary copies between physical instances, actually performing the copies is more challenging. In our previous version of Legion, data movement between physical instances could be performed by linear copies of segments of memory [26]. By allowing physical instances to contain multiple fields and maintain different data layouts for those fields, the problem of moving data between physical instances is greatly complicated. For movement between instances with the same layout, we can still use linear copies, however, for movement between different layouts additional logic is required for transforming data layout.

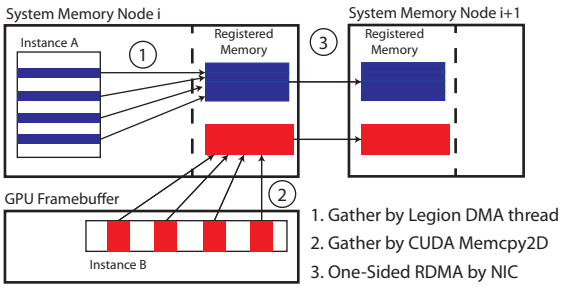



Fig. 4. Legion Data Movement Pathways

Fortunately, while the costs of these transformations are relatively large in shared memory machines, they are small compared to the cost of moving data in machines with hierarchical memory. For example, the cost of moving data over a PCI-E bus or between nodes will always be higher than for transposing data. Since most data movement in high performance Legion applications is between distinct memories, the additional cost of transforming data layouts is minimal.

In practice, our field-aware version of the Legion runtime fuses together both data movement and transformation within its data movement pathways. Figure 4 shows several examples of data movement pathways in Legion both within a node as well as between nodes. To support these data movement pathways, Legion maintains temporary buffers that are registered with various low-level programming APIs (e.g. uGNI, IBVerbs, CUDA). The memory for these buffers is pinned to facilitate direct access by the hardware DMA engines in both GPUs and network interface cards (NICs). Legion explicitly gathers and scatters data to and from these buffers to support bulk data movement operations for higher performance. This is a common, but tedious, optimization performed by hand in many applications, and one that happens automatically in Legion. Our Legion implementation further leverages the natural level of indirection afforded by these gather and scatter operations to automatically perform data transformation as part of data movement, rendering the cost of transforming data between physical instances with different layouts virtually free.

To perform the gather and scatter routines, we dedicate a CPU core per node to act as a DMA engine. The DMA core operates on a queue of requested data movement operations and knows how to perform fast data movement and transformation routines between all pairs of memories in our system for various data set sizes. This includes the use of fast, in-cache transpose routines and offloading data movement operations to hardware DMA engines when possible (e.g., using `cudaMemcpy2D` for gathering and scattering to and from GPU framebuffer memory). While the decision to dedicate an entire core on each node for DMA operations may seem excessive, it is reasonable in an environment where most applications, such as S3D, are dominated by the cost of data movement. If current scaling trends continue, more

applications will fall into this category, likely precipitating even more exotic hardware DMA engines for moving data. As these new features become available, Legion is well positioned to incorporate them in a way that is transparent to application developers and will require no changes to existing Legion application code.

## V. Performance Evaluation

To evaluate the benefits of structure slicing, we consider three Legion applications on two heterogeneous supercomputing systems. In addition to examining both strong and weak scaling scenarios, we use data made available by Legion's profiling tools to highlight several interesting points.

Two of our applications, Circuit and Fluid, are based on the original Legion versions described in [6]. The third is S3D, a production application [11]. There is no original Legion version of S3D; we compare with existing vectorized Fortran and OpenACC implementations.

We ran experiments on Keeneland [27] and Titan [4]. Both machines are high performance supercomputers with a mix of x86 CPUs and NVIDIA GPUs. However, there is a considerable difference between the computing power of various processors within these machines. The Keeneland Full Scale (KFS) system is an upgraded version of the machine used for the original Legion experiments [6]. The CPUs have been upgraded to 16-core Intel Sandy Bridge CPUs with support for AVX instructions, and the Infiniband interconnect has been upgraded from QDR to FDR. The three M2090 Fermi GPUs were not upgraded, but make use of a faster PCI-Express bus.

Titan is a larger system with over 18,000 nodes. Each node has an AMD Interlagos CPU with 16 first-generation Bulldozer cores, which perform poorly compared to the Sandy Bridge cores on Keeneland. Each Titan node does possess a single Kepler K20X GPU with nearly equivalent performance to all three M2090 GPUs on a Keeneland node. Titan is connected by a Cray Gemini interconnect with a 3D torus topology.

As evidenced by these two machines, heterogeneity is often present in more than just processor kinds. By having components from different technology generations, there are different ratios between compute throughput of different processors. On Keeneland, CPU and GPU throughput is evenly matched, while on Titan there is a severe imbalance. As we will see, these differences necessitate exploring different mapping strategies on each architecture.

### A. Circuit

The first Legion application we examine is Circuit, which simulates the behavior of an electrical circuit, defined as an arbitrary mesh in which edges are electronic components and nodes are the points at which they connect. We have two versions: a field-aware version and a baseline version that differs only in using a single field for entire structures, eliminating any benefit from structure slicing.

The performance of the Circuit application was measured on both Keeneland and Titan and is shown in Figure 5. The size of the circuit being simulated is kept constant as the number of nodes is increased to demonstrate strong scaling. Several of the leaf computation tasks in the application have been optimized

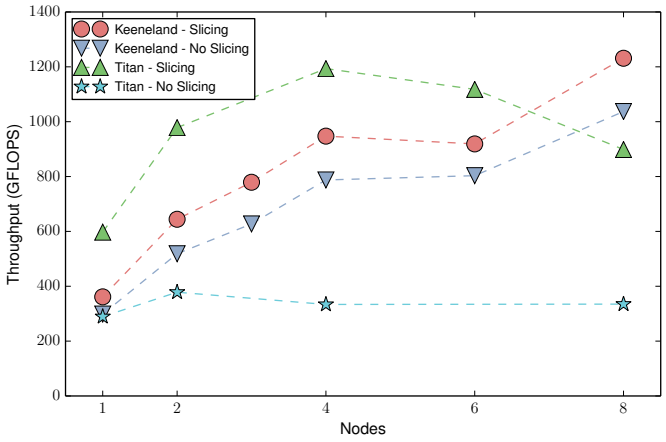



Fig. 5. Circuit Performance

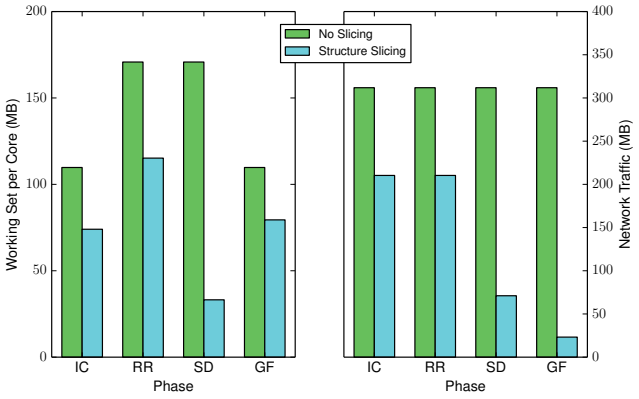



Fig. 6. Fluid: Working Set Sizes and Network Traffic

compared to the version that was tested in [6]. While this greatly improved single-node performance, the reduction in computation time increases the relative cost of Legion runtime and communication overhead, which is visible in the relatively poor strong scaling of the baseline, especially on Titan. There is no task parallelism to make use of in Circuit, but structure slicing is able to reduce the network traffic by only sending fields that are needed or have been updated by another task, and allows the use of the GPU-preferred SOA layout of data. This results in performance improvements of up to 19% on Keeneland and 257% on Titan, which is much more sensitive to data layout.

### B. Fluid

Fluid is a Legion port of the *fluidanimate* simulation from the PARSEC [5] suite. Although the original PARSEC version of Fluid was designed for a single node, the original Legion version achieves modest performance improvements and scaling on multiple nodes by replacing the fine-grain mutexes used in the PARSEC implementation with coarser-grain scheduling using logical regions [6].

A single simulation time step in Fluid consists of four phases. For both the original Legion version of Fluid and the new structure slicing version, Figure 6 shows the working set and the amount of network traffic between each of the

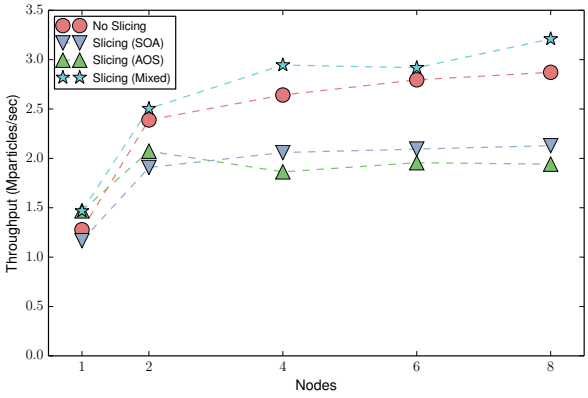



Fig. 7. Fluid Performance - Small Data Set

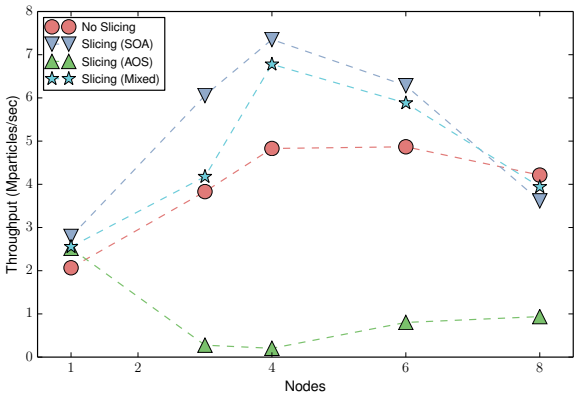



Fig. 8. Fluid Performance - Large Data Set

four phases: initialize cells (IC), rebuild reduce (RR), scatter densities (SD), and gather forces (GF). Structure slicing allows the Legion runtime to reduce working sets by 27-80% and the total network traffic per time step by 59%.

The relatively small problem sizes used in this benchmark present an interesting challenge. For multi-node runs, communication costs quickly dominate execution. The reduction in bytes of network traffic required due to structure slicing helps, but only if the data layout is one that can be efficiently transferred by the NIC. A structure-of-arrays (SOA) layout addresses this problem, while an array-of-structures (AOS) layout results in many sparse transfers for individual fields. However, an AOS layout can yield higher performance for CPU tasks if the working set fits in the CPU's cache.

There is therefore a tension between the SOA layout preferred by the NIC and the AOS layout preferred by the CPU. In existing programming models, the programmer would be forced to hard code a single decision into the source. The Legion mapping interface solves this problem by permitting experimentation with multiple different mapping strategies in order to select the best one. We try three different strategies for laying out physical instances: all AOS, all SOA, and a mixed strategy that uses AOS layout for instances local to a node while using SOA for instances shared between nodes. Figures 7 and 8 show performance results for each of these three strategies on Titan on two different problem sizes. For the small problem size, the mixed mapping strategy works best by matching AOS performance on a single node (where no

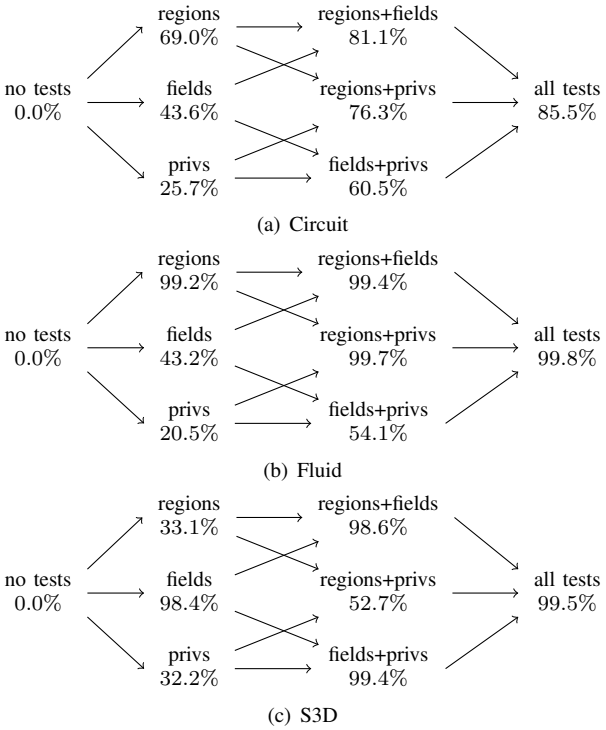


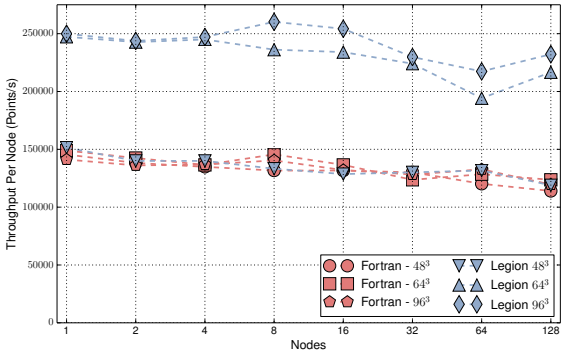

Fig. 9. Non-Interference Test Success Rates by Application

data is shared) and exceeding the SOA-based performance for larger node counts. On the larger problem size, data no longer fits in cache and therefore the full SOA mapping strategy performs best. In contrast, the volume of sparse inter-node transfers that result from the AOS mapping strategy can no longer be handled by the runtime's data layout transformation buffers, and the performance of AOS drops precipitously. The Legion mapping interface makes it possible to explore the tradeoffs inherent in mapping decisions seamlessly with no code modifications to find the optimal performance points.

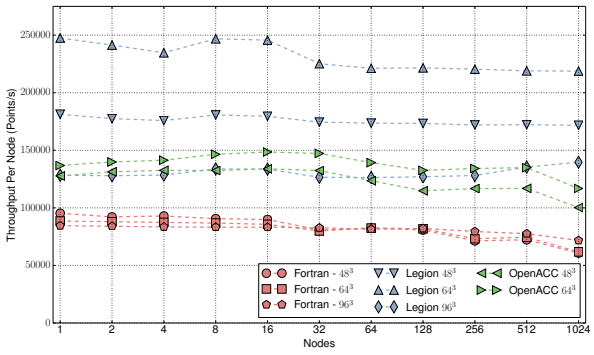
### C. Non-Interference Tests

As described in Section IV, there are three dimensions to non-interference tests in Legion. We now provide empirical evidence for our chosen ordering of these tests. Figure 9 shows decision diagrams for potential orderings of non-interference dimensions tests for each application. At each node, we show the percentage of all non-interference tests that would succeed with that subset of the tests. The percentage at the end shows the overall percentage of non-interference tests that succeed for a given application.

The goal is to minimize the overall cost of the tests, which favors the early use of cheaper and/or more effective tests. Although there is considerable variability between applications, region non-interference is the most effective test overall. As discussed in Section IV-B, the Legion runtime's region tree data structure makes this test inexpensive, and it is the clear choice for the first test. The combination of fast bit-mask tests with the benefit of finding significant parallelism in applications with many fields such as S3D justifies performing field non-interference second. Finally, the more expensive privilege non-interference test is placed last to minimize the number of invocations.

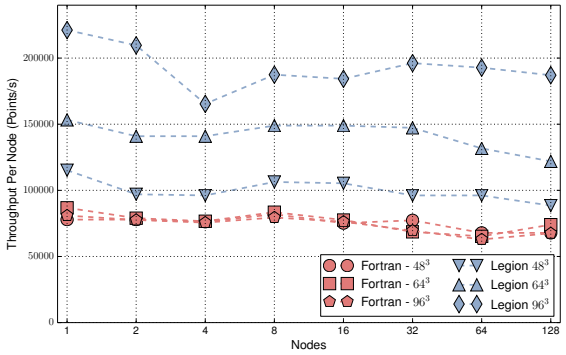



(a) Keeneland

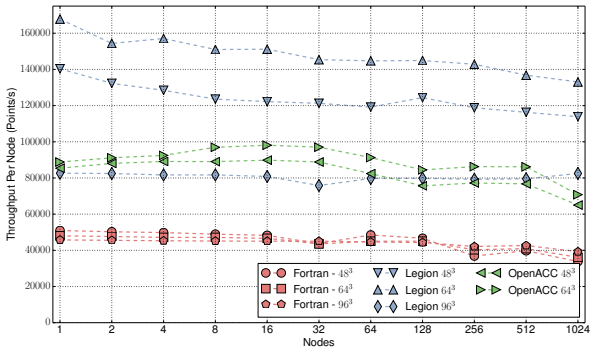



(b) Titan

Fig. 10. S3D Performance - DME




(a) Keeneland




(b) Titan

Fig. 11. S3D Performance - Heptane

### *D. S3D*

The final Legion application we use in our evaluation is a Legion port of S3D [11], a production combustion simulation code initially written in more than 200K lines of Fortran. We compare against two versions of S3D: a CPU-only version and an improved hybrid version from [19] that uses OpenACC.

In practice, the S3D algorithm was designed for weak scaling to support higher fidelity simulations when running on larger machines. In these experiments, we demonstrate weak scaling by holding the problem size per node constant (at either $48^3$, $64^3$, or $96^3$ grid points per node) as we scale out to larger node counts. Our graphs show the throughput per node averaged over 100 time steps, with a flat line representing perfect weak scaling. We show results for two different chemical mechanisms: DME and heptane. Results for the DME mechanism are in Figure 10(a) for Keeneland and Figure 10(b) for Titan, while the heptane mechanism results are in Figures 11(a) for Keeneland and 11(b) for Titan.

S3D consists of hundreds of Legion tasks that are launched every time step. The most obvious challenge is discovering the best way to map all of these tasks onto the available processors for both Titan and Keeneland. To address this problem, we developed several Legion mappers that support various mapping strategies designed to both balance work and minimize data movement. The two most successful mappers are a *mixed* mapping strategy that balances work between CPUs and GPUs, and a *GPU-centric* mapping strategy that keeps most work on GPU processors to minimize data movement between the system and framebuffer memories over the PCI-Express bus.

On Keeneland, the Legion version of S3D ranges from marginally faster than the AVX-vectorized Fortran version for the smallest problem size on the smaller DME mechanism to up to 3.06X faster for the largest problem size on the larger heptane mechanism. The variability is due to the bottleneck of the PCI-Express connections between the GPUs and the CPUs and the corresponding changes in the mapping strategy used in the Legion implementation. For the smallest problem, the optimal mapping is to place all work on a single GPU and avoid the latency cost of moving data across the PCI-Express bus. A Sandy Bridge CPU and a Fermi M2090 are roughly equal in their achievable double-precision performance, so parity in S3D performance is expected. For larger problems, the latency of PCI-Express transfers can be better hidden, and mapping strategies that spread work across multiple GPUs and the CPU cores results in significant improvement compared to the CPU-only Fortran implementation.

The results are much different on Titan, due to the extreme disparity in floating-point capability between the K20X GPU and the Bulldozer CPU cores. For the $48^3$ and $64^3$ problem sizes, where the GPU-centric mapping strategy is possible, the Legion implementation outperforms the CPU-only Fortran implementation by factors up to 3.54X for the DME mechanism

| | $48^3$ | $64^3$ | $96^3$ |
|---|---|---|---|
| Fortran | 63% | 70% | 84% |
| OpenACC | 78% | 85% | N/A |
| Legion | 94% | 88% | 98% |

(a) DME

| | $48^3$ | $64^3$ | $96^3$ |
|---|---|---|---|
| Fortran | 66% | 75% | 86% |
| OpenACC | 76% | 79% | N/A |
| Legion | 81% | 79% | 99% |

(b) Heptane

Fig. 12. Parallel Efficiency at 1024 Nodes.

and 3.68X for the heptane mechanism. For the $96^3$ problem sizes, the GPU's limited memory capacity forces a mixed mapping strategy that uses both CPU cores and the GPU. The structure slicing implementation of Legion automatically infers that only a subset of fields need to be moved to the GPU allowing the working set to fit in the constrained framebuffer memory. The overall performance is greatly reduced due to the significantly slower Bulldozer cores, but still exceeds the CPU-only Fortran implementation by up to 1.94X for the DME mechanism and 2.10X for the heptane mechanism.

The speedup of the Legion implementation compared to the OpenACC implementation on Titan is between 31-88% for the $48^3$ and $64^3$ problem sizes and can be attributed to two factors. First, the Legion implementation is able to use the computational power of both the Bulldozer cores and the K20X GPU, whereas the OpenACC code in many cases leaves the CPU cores idle while performing work solely on the GPU. Second, the latency of transferring data between the GPU's memory and the main system memory can be significant. The extra task and data movement parallelism discovered by structure slicing enables Legion to run tasks on the GPU for some fields while data transfers for other fields are in progress. No comparison is possible for the $96^3$ problem size. The OpenACC version cannot fit the necessary data in the GPU's framebuffer memory, and modifying it to employ an alternate mapping strategy that uses the CPU as well would involve significant code refactoring.

The weak scaling of the Legion implementation is also better than both the Fortran and OpenACC versions. Figures 12(a) and 12(b) show the parallel efficiency of the DME and heptane mechanisms respectively at 1024 nodes. As expected, in all cases parallel efficiency increases with both larger problem sizes and larger mechanisms (recall heptane simulates 52 chemical species while DME simulates only 30 species). Both larger problem sizes and larger mechanisms provide additional work which can be used to better hide communication latencies. Not surprisingly, Legion confers the largest performance gains at scale relative to Fortran and OpenACC on the smallest problem size ($48^3$ DME) because structure slicing enables Legion to discover additional work and better hide communication latencies with computation.

## VI. Related Work

We briefly survey work related to structure slicing. First, there is a long history of two-dimensional views of data in relational databases [13]. The concurrent processing of transactions over two-dimensional relations relies on significant parallelism across rows [16], which corresponds to data parallelism. More recently, there has been considerable interest in column-oriented databases [23]. We are unaware of any concurrent transactional systems that leverage both field- and column-level parallelism for performance.

Structure slicing is loosely related to a common technique for achieving high performance in shared memory applications. Under heavy lock contention, programmers routinely turn to fine-grained locking techniques that assign locks to individual fields in data structures. The burden of the implementation of such a locking scheme falls entirely on the programmer with no help from the programming system and minimal tool support [22]. Structure slicing makes extracting parallelism on individual fields explicit and furthermore operates over distributed memory architectures.

Dynamic Parallel Java (DPJ) uses static region annotations on data structures to help identify parallelism in Java applications [7]. Although these annotations can be applied to fields in Java classes, they pertain to the objects pointed to by the field contents, rather than the fields themselves. In Legion structure slicing, privileges apply directly to the fields being accessed, supporting task parallelism between tasks accessing different fields of the same object.

The analysis performed for loop fission optimizations in many Fortran compilers [20] has similarities to structure slicing. This analysis relies both on the prohibition of aliasing of normal Fortran arrays and on the Fortran idiom of expressing arrays of complex data structures as separate arrays for each field. This can result in issues with memory locality for tasks that operate on many fields at a time. Structure slicing extracts task parallelism between fields without destroying locality and without restrictions on the data collections.

Several recent efforts [8, 10, 12, 21, 24] have taken advantage of the substructure of objects in arrays to change the layout of the data in memory to better match the different types of processors in a heterogeneous system, such as converting from an array-of-structs to a struct-of-arrays or transposing the contents of arrays. However, all require some changes to the application code and commit to a layout decision at compile time. By decoupling of the layout decision from the application code and allowing the decision itself to be made at run time Legion enables the layout to be optimized even when the right answer is data- or system-dependent or when it is impractical to generate code for all possible cases at compile time.

## VII. Discussion and Conclusion

The incorporation of structure slicing into the Legion programming model generalizes the logical region abstraction and allows it to encode a wider range of potential data structures. In many ways logical regions with structure slicing share the same properties as relations from the database literature, albeit without supporting the full cast of relational algebra operators. Structure slicing of logical regions on fields is similar to the projection ($\pi$) operation while partitioning logical regions into sub-regions is analogous to the selection ($\rho$) operation. Relations have been shown to be a useful abstraction for describing a wide array of data structures [17] and emulating them will

allow logical regions to handle the growing complexity of data structures and data decomposition patterns required by modern supercomputing applications.

The addition of structure slicing to logical regions has also opened up data layout as a new design dimension to be considered when writing Legion applications. Most importantly, the Legion programming model encapsulates data layout decisions within the mapper interface, ensuring that Legion applications remain portable. As hardware becomes increasingly heterogeneous and data movement costs continue to escalate, we anticipate seeing additional hardware support for data layout including programmable DMA engines and processors specialized for specific layouts. By abstracting data layout with logical regions and structure slicing, Legion codes are well positioned to be easily ported to and optimized for new hardware via different mapping strategies.

We have presented structure slicing, a general technique for extracting field-level task parallelism from applications that use structures with many fields. The task parallelism discovered by structure slicing composes well with the existing abstractions in Legion and better enables programmers to leverage the heterogeneous processors and hierarchical memories on existing supercomputers. To illustrate the benefits of structure slicing, we have ported several applications and shown significant performance improvements over optimized Fortran and OpenACC codes on heterogeneous architectures.

## ACKNOWLEDGEMENT

This work was supported by Los Alamos National Laboratories Subcontract No. 173315-1 through the U.S. Department of Energy under Contract No. DE-AC52-06NA2539, NSF Grant CCF-1160904, and the Army High Performance Computing Research Center. The authors would like to thank Ramanan Sankaran for his assistance with providing code and performance data for the OpenACC version of S3D. Special thanks go to Jackie Chen and Hemanth Kolla for their detailed explanations and patience answering questions about S3D. This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology, which is supported by the National Science Foundation under Contract OCI-0910735.

## REFERENCES

[1] OpenACC standard. http://www.openacc-standard.org.

[2] CUDA programming guide 5.5. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, Sept. 2013.

[3] Intel manycore platform software stack. http://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss, 2013.

[4] Top 500 supercomputers. http://www.top500.org, 2013.

[5] Christian B. and Kai L. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[6] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing locality and independence with logical regions. In *Supercomputing (SC)*, 2012.

[7] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. POPL, 2011.

[8] B. Catanzaro, A. Keller, and M. Garland. A decomposition for in-place matrix transposition. PPoPP, pages 193–206, 2014.

[9] B.L. Chamberlain et al. Parallel programmability and the chapel language. *Int'l Journal of HPC Apps.*, 2007.

[10] S. Che, J. W. Sheaffer, and K. Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Supercomputing (SC)*, pages 1–11, 2011.

[11] J H Chen, A Choudhary, B de Supinski, M DeVries, E R Hawkes, S Klasky, W K Liao, K L Ma, J Mellor-Crummey, N Podhorszki, R Sankaran, S Shende, and C S Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science and Discovery*, page 015001, 2009.

[12] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. PLDI, 1999.

[13] EF Codd. A relational model of data for large relational database. *Communications of the ACM*, 1970.

[14] Leonardo D. and Ramesh M. OpenMP: An industry-standard API for shared-memory programming. *IEEE Comput. Sci. Eng.*, 1998.

[15] S. J. Deitz, B. L. Chamberlain, and L. Snyder. Abstractions for dynamic data distribution. In *Int'l Workshop on High-Level Parallel Programming Models*, 2004.

[16] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[17] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Data representation synthesis. PLDI, 2011.

[18] Khronos. The OpenCL Specification, Version 1.0. The Khronos OpenCL Working Group, December 2008.

[19] J. Levesque, R. Sankaran, and R. Grout. Hybridizing S3D into an exascale application using OpenACC: An approach for moving to multi-petaflops and beyond. SC, pages 15:1–15:11, 2012.

[20] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[21] M. Pharr and W. Mark. ISPC: A SPMD compiler for high-performance CPU Programming. In *InPar*, 2012.

[22] S. Savage et al. Eraser: A dynamic data race detector for multithreaded programs. *Trans. Computer Sys.*, 1997.

[23] Mike Stonebraker et al. C-store: A column-oriented DBMS. In *Proc. of the 31st International Conference on Very Large Data Bases*, pages 553–564, 2005.

[24] R. Strzodka. Data layout optimization for multi-valued containers in OpenCL. *Journal of Parallel and Distributed Computing*, 72(9):1073–1082, 2012.

[25] S. Treichler, M. Bauer, and A. Aiken. Language support for dynamic, hierarchical data partitioning. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013.

[26] S. Treichler, M. Bauer, and Aiken A. Realm: An event-based low-level runtime for distributed memory architectures. In *Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[27] J.S. Vetter et al. Keeneland: Bringing heterogeneous GPU computing to the computational science community. *Computing in Science Engineering*, pages 90 –95, 2011.