Regent: A High-Productivity Programming Language for HPC with Logical Regions

Elliott Slaughter
Stanford University
slaughter@cs.stanford.edu

Wonchan Lee
Stanford University
wonchan@cs.stanford.edu

Sean Treichler
Stanford University
sjt@cs.stanford.edu

Michael Bauer
NVIDIA Research
mbauer@nvidia.com

Alex Aiken
Stanford University
aiken@cs.stanford.edu

ABSTRACT

We present Regent, a high-productivity programming language for high performance computing with logical regions. Regent users compose programs with tasks (functions eligible for parallel execution) and logical regions (hierarchical collections of structured objects). Regent programs appear to execute sequentially, require no explicit synchronization, and are trivially deadlock-free. Regent's type system catches many common classes of mistakes and guarantees that a program with correct serial execution produces identical results on parallel and distributed machines.

We present an optimizing compiler for Regent that translates Regent programs into efficient implementations for Legion, an asynchronous task-based model. Regent employs several novel compiler optimizations to minimize the dynamic overhead of the runtime system and enable efficient operation. We evaluate Regent on three benchmark applications and demonstrate that Regent achieves performance comparable to hand-tuned Legion.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages*; D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*

Keywords

Regent; Legion; logical regions; task-based runtimes

1. INTRODUCTION

Modern supercomputers feature distributed memory architectures with deep memory hierarchies. Currently, the state of the art in programming this class of machines is the MPI+X hybrid programming model. While MPI+X codes achieve good performance, they do so at a cost to programmer productivity and performance portability. Users of MPI+X must explicitly manage data movement and synchronization within and between nodes. Furthermore, they must also explicitly overlap communication with computation for optimal performance, a task made difficult by the need to interface with two disparate programming models. In addition, the degree to which communication and computation must be overlapped to achieve good performance depends on machine-specific factors, resulting in poor performance portability in aggressively hand-tuned codes.

An alternative that is receiving considerable attention is writing programs for *task-based* runtimes. While there is significant variation among the current approaches [9, 4, 6, 19, 27], the common element is a graph of tasks to be executed, where the graph's edges capture ordering dependencies between tasks. The advantage of the task-based approach is that the computation is expressed at a higher level than MPI+X, which allows for both more aggressive optimization by the programming model's implementation and correspondingly less effort by programmers to express the same optimizations by hand (as well as better portability). A disadvantage of all the current task-based models is that they are runtime systems (i.e., libraries) embedded in a host language that does not understand the task-based model's higher-level semantics. Programmers must do additional work to maintain important invariants across calls to the runtime system, resulting in a programming interface that is more complex and verbose than a true programming language implementation could provide. Furthermore, important optimizations that require static analysis to be feasible are simply beyond the scope of dynamic runtime systems.

To address these challenges we present Regent, a high-productivity programming language for high performance computing. Regent features two key abstractions: *tasks* and *logical regions*. Regent programs look like ordinary sequential programs with calls to *tasks*, which are functions that the programmer has marked as eligible for parallel execution. Regent guarantees that any parallel execution is consistent with the sequential execution of a Regent program. Internally, dependencies between tasks are inferred automatically, freeing the user from the need to explicitly synchronize or manage data movement around the machine. Regent programs are also trivially deadlock-free and avoid a number of classes of mistakes possible in lower level distributed programming.

Logical regions [9, 33, 10], or simply regions, are collections of structured objects. Regions have no fixed location

SC '15, November 15 - 20, 2015, Austin, TX, USA

ISBN 978-1-4503-3723-6/15/11…$15.00

DOI: http://dx.doi.org/10.1145/2807591.2807629

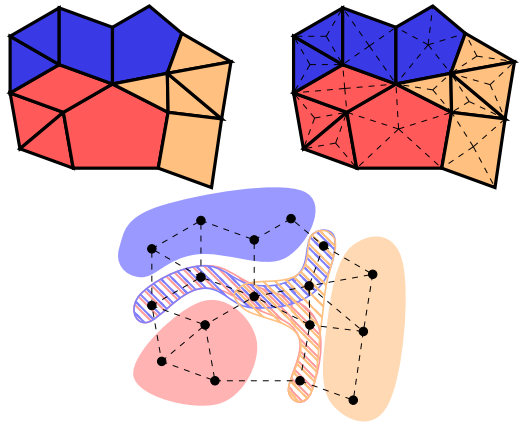


Figure 1: Partitioning in PENNANT: zones (top left), sides (top right), and points (bottom).

in the memory hierarchy—for example, because they may be striped across nodes—and no fixed layout in memory—for example, because different tasks or processors may prefer array-of-structs or struct-of-arrays layouts. Regions can be recursively partitioned to match the hierarchical structure of memory, and to facilitate parallel execution on subsets of data. Regions can also be partitioned multiple times to express sophisticated communications patterns involving multiple views of the data, including ones with aliasing between views.

We describe an optimizing compiler for Regent that translates Regent programs into efficient implementations for Legion, a dynamic, task-based asynchronous runtime system with native support for tasks and logical regions [9]. Regent simplifies the Legion programming model. Many details of programming to the Legion runtime system can be managed statically by the Regent compiler, resulting in Regent programs that are both written at a higher level and with fewer lines of code than the corresponding Legion programs. Several novel optimizations allow the Regent compiler to achieve performance equivalent to hand-tuned codes written directly to the Legion C++ API.

To motivate the Regent programming model, we present excerpts from a Regent implementation of PENNANT [23], a Lagrangian hydrodynamics code. Each of the following sections discusses one of our contributions:

- Section 3 presents the Regent programming model and provides a detailed comparison with Legion.
- Section 4 presents compiler optimizations that are important for achieving high performance in Regent programs.
- Section 5 discusses the implementation of the Regent compiler.
- Section 6 evaluates the performance of three applications written in Regent.

Section 7 discusses related work, and Section 8 concludes.

2. MOTIVATING EXAMPLE

To motivate our presentation of Regent, we begin by introducing the language through a series of excerpts from a

```
1 task adv_pos_full(points : region(point), dt : double) where
2   reads(points.{px0, pu0, pf, pmaswt}), writes(points.{px, pu})
3 do
4   var fuzz = 1e-99
5   var dth = 0.5 * dt
6   for p in points do
7     var pap = (1.0 / max(p.pmaswt, fuzz))*p.pf
8     var pu = p.pu0 + dt*pap
9     p.pu = pu
10    p.px = p.px0 + dth*(pu + p.pu0)
11  end
12 end
```

Listing 1: PENNANT kernel task in Regent.

```
1  task simulate(zones_all : region(zone),
2                zones_all_p : partition(disjoint, zones_all),
3                points_all : region(point),
4                points_all_private : region(point),
5                points_all_private_p : partition(disjoint, points_all_private),
6                conf : config)
7  where
8    reads(zones_all, points_all_private),
9    writes(zones_all, points_all_private)
10 do
11   var dt = conf.dtmax
12   var time = 0.0
13   var tstop = conf.tstop
14   while time < tstop do
15     dt = calc_global_dt(dt, dtmax, dthydro, time, tstop)
16     for i = 0, conf.npieces do
17       adv_pos_full(points_all_private_p[i], dt)
18     end
19
20     -- ...
21
22     for i = 0, conf.npieces do
23       dthydro min= calc_dt_hydro(zones_all_p[i], dt, dtmax)
24     end
25     time += dt
26   end
27 end
```

Listing 2: PENNANT main task excerpt in Regent.

Regent implementation of the PENNANT mini-app. PENNANT [23] implements Lagrangian hydrodynamics for a subset of the functionality provided by FLAG [13], a production code at Los Alamos National Laboratory (LANL). PENNANT operates on 2D unstructured meshes, with data structures representing the fundamental elements in 0, 1, and 2 dimensions called points, edges and zones. To deal with zones with an arbitrary number of edges, PENNANT adds an intermediate data structure called a side, representing the triangular area between an edge and the center of the zone (see Figure 1 top right).

Simulation in PENNANT proceeds in alternating phases, reading values stored on zones and scattering to the points, and gathering values on points and writing to the zones. This structure is illustrated in an excerpt from the PENNANT main loop in Listing 2, where two phases are visible: the first phase advances points using forces previously computed (lines 16-18), while the second phase computes *dt* for the next timestep from the geometry of the updated mesh (lines 22-24). The rest of the PENNANT timestep loop follows in a straightforward way from these examples, calling each phase's kernels in turn.

To allow Regent programs to express that tasks execute on subsets of the data, regions can be *partitioned* into subregions. Partitioning a region is a primitive operation in Regent, and the resulting *partition*, which is the collection of subregions of the parent region, is a first-class value. In

```
1 void adv_pos_full(const Task *task,
2                   const std::vector<PhysicalRegion> &regions,
3                   Context ctx, HighLevelRuntime *runtime)
4 {
5   PhysicalRegion points0 = regions[0];
6   Accessor<double, SOA> points_px0_x(points0, PX0_X);
7   Accessor<double, SOA> points_px0_y(points0, PX0_Y);
8   Accessor<double, SOA> points_pu0_x(points0, PU0_X);
9   Accessor<double, SOA> points_pu0_y(points0, PU0_Y);
10  Accessor<double, SOA> points_pf_x(points0, PF_X);
11  Accessor<double, SOA> points_pf_y(points0, PF_Y);
12  Accessor<double, SOA> points_pmaswt(points0, PMASWT);
13  PhysicalRegion points1 = regions[1];
14  Accessor<double, SOA> points_px_x(points1, PX_X);
15  Accessor<double, SOA> points_px_y(points1, PX_Y);
16  Accessor<double, SOA> points_pu_x(points1, PU_X);
17  Accessor<double, SOA> points_pu_y(points1, PU_Y);
18  Future f0 = task->futures[0];
19  double dt = f0.get_result<double>();
20  double fuzz = 1e-99;
21  double dth = 0.5 * dt;
22  IndexIterator it(points0.get_logical_region().get_index_space());
23  while (it.has_next()) {
24    size_t count;
25    ptr_t start = it.next_span(count);
26    ptr_t end(start.value + count);
27    for (ptr_t p = start; p < end; p++) {
28      double frac = (1.0 / max(points_pmaswt.read(p), fuzz));
29      double pap_x = frac * points_pf_x.read(p);
30      double pap_y = frac * points_pf_y.read(p);
31      double pu_x = points_pu0_x.read(p) + dt * pap_x;
32      double pu_y = points_pu0_y.read(p) + dt * pap_y;
33      points_pu_x.write(p, pu_x);
34      points_pu_y.write(p, pu_y);
35      points_px_x.write(p, points_px0_x.read(p) +
36                        dth*(pu_x + points_pu0_x.read(p)));
37      points_px_y.write(p, points_px0_y.read(p) +
38                        dth*(pu_y + points_pu0_y.read(p)));
39    }
40  }
41 }
```

Listing 3: PENNANT kernel task in Legion C++ API.

```
1 runtime->unmap_region(ctx, pr_points_all_private);
2 Domain domain = Domain::from_rect<1>(
3   Rect<1>(Point<1>(0), Point<1>(conf.npieces - 1)));
4 IndexLauncher launcher(ADV_POS_FULL, domain,
5                        TaskArgument(), ArgumentMap());
6 launcher.add_region_requirement(
7   RegionRequirement(points_all_private_p, 0 /* projection */,
8                     READ_ONLY, EXCLUSIVE, points_all_private));
9 launcher.add_field(0, PX0_X);
10 launcher.add_field(0, PX0_Y);
11 launcher.add_field(0, PU0_X);
12 launcher.add_field(0, PU0_Y);
13 launcher.add_field(0, PF_X);
14 launcher.add_field(0, PF_Y);
15 launcher.add_field(0, PMASWT);
16 launcher.add_region_requirement(
17   RegionRequirement(points_all_private_p, 0 /* projection */,
18                     READ_WRITE, EXCLUSIVE, points_all_private));
19 launcher.add_field(1, PX_X);
20 launcher.add_field(1, PX_Y);
21 launcher.add_field(1, PU_X);
22 launcher.add_field(1, PU_Y);
23 launcher.add_future(dt);
24 runtime->execute_index_space(ctx, launcher);
```

Listing 4: PENNANT task launch in Legion C++ API.

PENNANT, partitions are created during program initialization (not shown) and passed as arguments to the main simulation task. Two such partitions are shown in Listing 2 on lines 2 and 5.

Partitioning a region assigns one or more *colors* (small integers) to the region's elements; there is one subregion per color containing all the elements with that color. Zones are an example of a *disjoint partition* where none of the subregions overlap. For performance, it is advantageous for the colored subregions to be compact, though the application will run correctly with any coloring. Figure 1 shows one possible coloring of zones.

Partitions of sides and points are computed from the partition of zones using the topology of the mesh. Sides simply take on the color of their corresponding zone. Points are partitioned in a more sophisticated way, to account for aliasing at the boundaries of the subregions of zones. Points are colored by every zone they are adjacent to, leaving each point with one or more colors. Points are first partitioned according to the number of colors assigned; points with only one color are placed in a subregion of private points while points with multiple colors are put into a subregion of ghost points. Each subregion is then further partitioned according to the colors of the points, as shown in Figure 1. This partitioning scheme allows Regent to limit data movement when reducing the forces applied by the zones on the points. Because the private points are known to be disjoint from all ghost points, and the private points are further partitioned into disjoint pieces for each submesh, those partitions of the mesh are isolated from any data movement in the system.

Regent tracks these relationships between regions, and ensures that tasks only access data for which they have declared the appropriate *privileges*: Regent tasks must say whether they plan to access a region with *read*, *read/write*, or *reduction* privileges (and, in the case of reduction privileges, the reduction operator must also be specified). Tasks may only call other tasks with subsets of their own privileges. The call to adv_pos_full in Listing 2 line 17 is safe because simulate holds a superset of the privileges required by adv_pos_full, but also because Regent understands that the partition access at points_all_private_p[i] is a subregion of points_all_private, for which simulate has privileges. Similarly, all pointer accesses (such as those in Listing 1 lines 7-10) are associated with a particular region, ensuring that the access stays safely within the bounds of the regions for which the task has privileges.

For comparison to the Regent code above, Listings 3 and 4 show implementations of the same tasks written with the Legion C++ API. Listing 3 corresponds to Listing 1 while Listing 4 corresponds to the three lines of Listing 2 lines 16-18. Clearly the C++ API is more verbose. However, beyond the length, these code samples also illustrate that Legion exposes a programming model with more moving parts than Regent. Regent is able to handle the additional aspects (which we discuss in Section 3) automatically within the compiler, and so hides them from the programmer and provides a higher-level interface while maintaining the performance of hand-tuned Legion.

3. PROGRAMMING MODEL

Before we discuss the Regent programming model in more detail, we explain more about Legion so that the reader can appreciate the differences between the two systems. Legion, the runtime which Regent targets, is implemented as a *software out-of-order processor* [9]. Tasks are issued to the Legion runtime in program order, but the underlying Legion scheduler may reorder tasks and execute them in parallel if it can prove that it is safe to do so. The Legion runtime analyzes each task's privileges for its region arguments to identify when tasks are using the same regions in ways that

either allow parallel execution or require that the tasks be serialized in the order they were issued.

Besides managing the tasks, the Legion runtime also manages the regions. Tasks are written using *logical regions*, which simply name collections of objects. During execution each logical region may correspond to any number of *physical instances*, which are actual allocated copies of the data. Separating the logical and physical levels allows important patterns, such as having multiple copies of read-only data, to be expressed directly.

The Legion abstractions allow programmers to write efficient task-based programs that run out-of-order, asynchronously, and in a distributed fashion. However, because Legion is embedded in C++, which does not understand the semantics of tasks and regions, the Legion API is forced to expose functionality beyond the logical layer of the programming model. Programmers must generally write Legion programs with some awareness of both the logical and physical levels.

Regent exposes only the logical level. The lower-level, physical details of the Legion execution model are hidden from the programmer. For a naive implementation, this would be disastrous for performance. However, with the support of static analysis and compiler optimizations, Regent is able to close the gap between logical and physical constructs and provide a seamless abstraction to the programmer.

In the rest of this section we present the Regent and Legion programming models in more detail, examine the differences between the two, and demonstrate how a Regent compiler is able to translate between them. In Section 4 we consider the optimizations performed by the compiler that enable this translation to be efficient.

3.1 Tasks

Tasks are the fundamental unit of control in both Regent and Legion. Tasks are issued in program order, exactly as they are written in the text, and every possible program execution is guaranteed to be indistinguishable from serial execution. As discussed in Section 2, tasks specify the regions they use and their permissions for those regions (whether the task performs reads, writes, or reductions to each region). In addition, tasks declare which *fields* of the objects in the region the task accesses. Together, the declaration that specific fields of a region are accessed with certain privileges is called a *region requirement*.

Whenever two tasks are *non-interfering*, accessing either disjoint regions, different fields of the same region, or the same fields with compatible permissions (e.g., both tasks only read the field or only perform the same reduction to the field), Regent allows those tasks to run in parallel. Wherever two tasks interfere, Regent inserts the appropriate synchronization and copy operations to ensure that the data dependence is handled properly. In addition to regions, tasks can also take partition arguments and specify partition requirements.

When writing (or compiling) to the Legion C++ interface, several additional aspects of the Legion runtime implementation are exposed, requiring additional user effort. Legion's dynamic dependence analysis imposes a cost with every task launched. To ensure that this overhead stays off the critical path, the Legion runtime is itself asynchronous and parallel [9]. The goal is for the runtime to run ahead of the application, issuing tasks and analyzing task interference in advance of when those tasks can actually run. Pipeline stalls, blocking operations, and excessive analysis costs can all cause the runtime to fall behind and hurt the performance of the application. Legion mitigates these issues by providing more sophisticated abstractions which can result in higher performance, but also have more complex semantics.

3.1.1 Avoiding Pipeline Stalls

Task execution in Legion is pipelined. In general, a task must complete a pipeline stage before it passes to the next stage. If a given stage stalls for any reason, that task and any task that depends on it also stalls. *Mapping*, described in greater detail in Section 3.3, is one pipeline stage. When a task is mapped, a processor is selected to execute the task and memories are chosen to hold the physical instances of each of its region arguments.

Because tasks can execute subtasks, Legion must wait for all subtasks to map before it can consider a parent task to have completed mapping. In general the only way to know that a parent task cannot issue more subtasks is that the parent task has terminated, which can result in unnecessary pipeline stalls when the task in question never intended to launch any subtasks.

Legion allows users to annotate tasks as *leaf* tasks if they launch no subtasks, a mechanism inherited from Sequoia [22]. In Legion, the runtime considers the mapping of a leaf task to be complete once the task itself is mapped, avoiding unnecessary pipeline stalls for dependent operations.

3.1.2 Avoiding Blocking Operations

Tasks can produce results in one of two ways: direct return values, or as a side-effect on a region argument. In Legion, operations can block whenever a parent task consumes a result produced by one of its child tasks. The Legion runtime provides ways of avoiding blocking on both kinds of task results.

When tasks produce direct return values, Legion wraps those values in *futures*. Users can block to obtain the value of a future, but Legion also supports passing futures as inputs to other tasks without blocking. In this way, the programmer can describe the flow of values between tasks without blocking, allowing the runtime to run further ahead and hide runtime analysis costs. Futures are visible in the C++ sample codes in Listing 3 lines 18-19 and Listing 4 line 23.

When a parent task needs to read the results of a region written by a child task, unless the parent task has explicitly indicated otherwise, the Legion runtime must conservatively assume that the parent task may attempt to access the regions used by the child task as soon as the child returns. To preserve sequential semantics, the runtime blocks the parent while the child task is in flight to ensure the child's results are available before the parent continues. To avoid blocking, parents must declare to the runtime that the region data is not required by *unmapping* (releasing) the physical instance of the region prior to calling the child.

3.1.3 Reducing Analysis Costs

Even when execution does not stall in the runtime or block in the application, the cost of dynamic analysis itself can cause the runtime to fall behind. One approach for reducing runtime overheads is to use *index space task launches*.

Conceptually, index launches simply represent a loop of task launches. Listing 2 lines 16-18 shows an example of a Regent loop that can be transformed into an index launch (with corresponding C++ code in Listing 4). However, the Legion runtime places several restrictions on index launches to ensure that they are well-behaved:

1. Arguments to all tasks in the index launch must be computed outside the launch, guaranteeing that arguments are available and that no arguments depend on side-effects from tasks within the launch.

2. Futures, if any, are added to the launch as a whole, not to individual tasks.

3. Requirements can be in one of two forms:

   - Individual region requirements add a single region to all tasks in the launch.
   - Partition requirements add a subregion of the partition for each task.

   Legion supports user-defined projection functions to allow the programmer to dynamically select subregions for each type of region requirement.

4. Because an index launch implies parallel execution, all the tasks must be non-interfering.

5. If tasks within the launch return a value, then the launch as a whole is allowed to either return a map with all the resulting futures, or to reduce the futures into a single value.

When executing an index launch, the runtime still performs dynamic checks to ensure that the tasks within the launch are non-interfering. However, Legion is able to amortize these checks across the entire index launch instead of performing them individually.

3.2 Regions

Logical regions are created as the cross product of an *index space* (set of indices) and a *field space* (set of fields). Logical regions can be compared to arrays of objects or structs, though this analogy falls short in several ways. In particular, as discussed previously, because a logical region may have multiple physical instances, there is no one-to-one mapping between a logical region and its representation in memory.

3.2.1 Physical Instances

In Legion, a logical region may, at any given point in time, map to zero or more physical instances. Access to each field of a physical instance is mediated through a field-specific *accessor*. In the Legion C++ interface, the programmer must manage distinct `LogicalRegion`, `PhysicalRegion`, and `Accessor` types.

In Regent, these differences disappear because the compiler manages the mapping from logical to physical regions (and physical regions to accessors) transparently for the programmer. Field spaces can be constructed concisely from struct types, and nested structs are automatically expanded into their component fields. Accessors are created automatically for whatever fields the programmer declared in the privileges for the task. These differences are illustrated in the difference between Listing 1 and Listing 3. In contrast to users of the Legion C++ API, Regent programmers can usually pretend that regions are simply arrays of structs or objects.

Physical instances in Legion may be stored in one of a number of layouts. Examples of common layouts include array-of-structs and struct-of-arrays, while more esoteric layouts may include arrays blocked for vectorized CPU instructions. Legion provides explicit accessor objects in order to constant-fold compile-time information about instance layouts for efficient access, as seen in Listing 3 lines 6-17. Regent manages accessors, along with instances, on behalf of the programmer.

Regent also manages the creation of region requirements for each task. For each task, Regent flattens the fields, groups them by privilege, and issues a region requirement for each privilege and set of fields; see the correspondence between Listing 2 line 17 and Listing 4 lines 6-22.

3.2.2 Partitions

Partitioning a region using the Legion C++ interface happens in two steps. First, the user creates an *index partition* of the index space to specify how the sets of indices are subdivided between the spaces. Second, the user applies the index partition to a logical region created using that same index space to obtain a corresponding *logical partition*. In Regent, these operations are combined, as the correspondence between logical regions and index spaces is managed for the programmer.

3.3 Mapping

As briefly described previously, mapping is the process of selecting a processor to run each task and a memory (and data layout) for each logical region. Mapping is under the control of the application, though Regent provides a *default mapper* with sensible settings to allow users to get up and running quickly.

Legion also provides a mapping interface, but because of the distinction between physical and logical constructs exposed in Legion, this mapping process is more involved.

In Legion, logical regions must be mapped to physical instances in a specific memory before they can be used. By default, at the start of a task Legion automatically maps each region used by the task, and when the task ends each of those regions is unmapped. Before launching a subtask a parent task must also unmap any region that the child task needs to use. By default, the Legion runtime unmaps all of the parent's regions before calling a child and remaps them when the child terminates. While this default behavior guarantees correct execution, if the parent and child have interfering privileges for a region (e.g., both can write the region) then the parent will most likely block until the child terminates, as the parent cannot remap the region until the child unmaps it (recall the discussion in Section 3.1.2).

For higher performance, Legion programmers can explicitly manage region mappings themselves through explicit *map* and *unmap* calls provided by the Legion interface. By unmapping a region, the programmer notifies the runtime that the data in that region is not required by the parent task until a corresponding map call is issued. In typical usage, programmers unmap all regions before entering a main loop, and remap all regions once the loop completes, which ensures that the runtime can avoid blocking when issuing tasks within that loop. An example of such an unmap call

can be seen in Listing 4 line 1.

4. OPTIMIZATIONS

As illustrated in Section 3, Regent simplifies the Legion programming model and provides a higher level of abstraction that is concerned only with logical, rather than physical, constructs. The Regent compiler is able to manage the correspondences between logical and physical constructs in a way that achieves significantly better performance than a naive implementation. This section describes a number of optimizations that together allow the Regent compiler to achieve performance comparable to hand-tuned code written to the Legion C++ API.

4.1 Mapping Elision

Regent frees programmers of the burden of managing physical instances of regions by statically computing correct and optimal placements of map and unmap calls. The Regent type system guarantees that the compiler has complete information about what regions can be accessed within any task. The compiler uses this information to perform a flow-sensitive analysis over the AST to determine the spans over which regions are used and inserts the map and unmap calls at the boundaries of spans when switching between usage in a parent and a child task. In the case where a region is not used at all within a task, the compiler issues a single unmap call at the top of the task and leaves the region unmapped for the entire duration of the task's execution. In contrast, the Legion runtime, in the absence of manually placed calls to map and unmap, is forced to continue to map and unmap the region throughout the task's execution.

4.2 Leaf Tasks

As discussed in Section 3.1.1, correctly identifying leaf tasks is an important optimization for Legion programs, as otherwise the Legion runtime must consider the mapping of a task still in progress until it can be certain all child tasks have mapped. Regent automatically infers at compile time which tasks are leaf tasks. The compiler knows all call targets and is therefore able to determine, using a flow-insensitive analysis, whether a given task calls any subtasks. These annotations are guaranteed to be correct and precise, in contrast to the user-provided leaf task annotations in Legion.

4.3 Index Launches

Whenever possible, the Regent compiler transforms loops of task launches into index space task launches. The analysis for this optimization proceeds in multiple phases:

1. The compiler begins with a structural analysis of the code to determine whether the loops in question are eligible for transformation into an index space launch. Currently all simple loops containing single task launches are considered eligible.

2. For each loop, the compiler determines whether the body of the loop (aside from the task call itself) is side-effect free. In particular, the loop body must not read or modify data that the task itself might read or modify. Doing so would introduces a loop-carried dependence and shows the loop is not fully parallelizable.

3. For each argument to the task launch, the compiler determines whether the argument in question is eligible to be transformed into an argument for an index task launch. Arguments must be one of:

   - a non-region value;
   - a region value that is provably loop invariant;
   - a region value that is provably an analyzable function of the loop index; i.e., it is an expression such as a partition access $p[i]$ indexed by the loop variable i.

4. The compiler then performs a static variant of Legion's dynamic non-interference analysis. For each region-typed argument, the compiler determines whether it is statically non-interfering with other region-typed arguments. As with the dynamic analysis, the compiler has several dimensions along which to prove non-interference:

   - disjointness, either because the region types are incompatible, or because the compiler can statically prove disjointness through the static tree of region partitions;
   - field disjointness, because the arguments use different fields; or
   - privileges, because both arguments use compatible privileges (e.g. both read-only, or both reductions with the same reduction operator).

If the analysis determines that a task launch is eligible for optimization, the compiler emits the code to perform the index task launch.

It is worth noting that while this optimization looks similar in principle to forall-style constructs in other languages and programming models, it behaves quite differently in many respects. In particular, when index launch optimization fails (because any of the properties above cannot be established), that does not imply the resulting code runs sequentially. The Legion runtime will perform its standard dynamic analysis, and will parallelize all tasks that are dynamically non-interfering, regardless of whether the compiler performs the optimization or not. This optimization simply allows the runtime to amortize the dynamic analysis costs in cases where the loops can be analyzed statically. Thus, Regent has a much more forgiving fallback for when static analysis is insufficient than language implementations that rely solely on static analysis.

4.4 Futures

In Legion, tasks can return futures, which can be passed to other tasks without blocking, allowing applications to build chains of asynchronous operations ahead of the actual computation. The Regent compiler can automatically lift variables and simple operations to futures to take advantage of these benefits. This optimization has three phases:

- The compiler first performs a flow-insensitive analysis to determine which variables are assigned to futures at any point within each task. Any such variables are automatically promoted to hold futures.

- The compiler then issues calls to automatically wrap and unwrap futures when storing a concrete value into

a future-typed variable, or when reading a future-typed value because a concrete value is required. Tasks do not require arguments to be concrete, and can therefore be issued in advance of when the concrete arguments are ready.

- Finally, the compiler emits tasks to allow simple side-effect free operations (such as arithmetic) to be performed directly on futures.

4.5 Pointer Checks Elision

As noted in [33], static type checking of Legion programs allows certain classes of pointer checks to be elided. Regent preserves all the properties of the type system which make this possible. In particular, all pointer types in Regent explicitly contain one or more regions that they point into. Regent checks these annotations to ensure correctness at compile time, and elides the dynamic pointer checks, which are often prohibitively expensive at runtime.

4.6 Dynamic Branch Elision

In addition, Regent is able to elide certain classes of dynamic branches when accessing pointers in Legion. Pointers that can point into multiple different regions (e.g., private or ghost points in PENNANT) carry some dynamic tag bits encoding the region the pointer currently points to. In some cases, however, the memory for the two regions is actually co-located at runtime (e.g. because of a decision to map both regions to the same memory), allowing the dynamic branches on the tag bits to be elided. The compiler emits code that automatically detects such cases at runtime and selects the fast path when it is available.

4.7 Vectorization

Regent leaf tasks frequently feature loops over regions. In many cases, the Regent compiler is able to vectorize these loops automatically, often exceeding performance provided by traditional autovectorizers.

Regent performs runtime code generation to LLVM [28] via Terra [20]. While LLVM provides an autovectorizer, the low level of abstraction of the LLVM IR means that the vectorizer frequently misses vectorization opportunities or chooses the wrong optimization strategies for its vector code. Regent's native understanding of regions allows the vectorizer to make these decisions with improved precision. Regent uses Terra's built-in vector types to produce explicit vector instructions for LLVM, resulting in significant performance gains in many cases.

Regent derives this advantage in precision from two sources. First, Regent has improved information about aliasing through type, field, and region-based analysis. In particular:

- While accesses for composite types are ultimately expressed as array accesses to fundamental types (integers, double-precision floating point, etc.), Regent is able to compare the original types to determine if there is potential for aliasing.
- Furthermore, even for identical types, Regent knows which fields are accessed and may be able to use this information to prove independence.
- Finally, when two accesses are to different regions, Regent may be able to use its knowledge of region disjointness to prove that accesses are independent.

Beyond this, Regent has access to implicit information about the costs of potential vectorization opportunities through regions. Regions are hierarchical and distributed data structures intended to provide opportunities for parallelism. Therefore, when Regent sees an outer loop over a region, and an inner loop (over something other than a region), Regent can infer with high confidence that the outer loop is the better opportunity for vectorization. In some cases, largely because it lacks comparable information for its cost model, LLVM chooses to vectorize the inner rather than outer loop, resulting in degraded performance.

5. IMPLEMENTATION

We have implemented an optimizing Regent compiler using Terra [20], a low-level programming language with semantics comparable to C, but with extensive and sophisticated support for meta-programming via multi-stage programming [32]. Terra is embedded inside Lua [24], a high-level scripting language with first-class functions. Lua plays the same role for Terra that C++ templates play for C++, and provides many of the same benefits. However, Lua/Terra provides much better ease of use, because the meta-programming language is a full programming language rather than C++'s restricted template language.

Terra uses LLVM [28] to provide efficient JIT compilation of Terra functions to fast machine code. As noted in Section 4.7, Terra makes it possible to perform vectorization and specialization with full awareness of the vector instruction set supported by the machine. The use of LLVM as the JIT compiler also allows both Terra and Regent functions to call and link easily against native C libraries.

We have implemented Regent as a co-embedded language within Terra. The Terra API provides support for extending the parser with additional keywords, which when seen in the source program text cause Terra to invoke the embedded language compiler. Regent overloads a number of keywords—most notably, the `task` keyword—allowing the Regent language to interoperate seamlessly with both Lua and Terra. Regent tasks may call Terra functions and have access to all data types supported by Terra, including structs, arrays, and explicit vector types. The Regent compiler uses this information to provide automatic structure slicing [10] for struct types stored inside logical regions. Regent tasks may also be dynamically specialized, using Lua, to provide multiple implementations, which are JIT compiled prior to starting the Legion runtime.

6. EVALUATION

We evaluate the performance of the Regent implementation, and the effectiveness of the optimizations performed by the Regent compiler, on three applications: a circuit simulation; PENNANT, a Lagrangian hydrodynamics code; and MiniAero, an explicit solver for the compressible Navier-Stokes equations on an unstructured mesh. The experiments were done on the Certainty supercomputer [1]. Each node has two sockets with an Intel Xeon X5650 per socket for a total of 12 physical cores per node (24 threads with hyperthreading). Nodes are connected with Mellanox QDR Infiniband. The Legion runtime, along with all three C++ reference codes, have been compiled with GCC 4.9.2. Terra (and therefore Regent) uses LLVM 3.5.

We perform several experiments on the Regent and ref-

Key	Optimization
map	Mapping Elision
leaf	Leaf Task Optimization
idx	Index Launch Optimization
fut	Future Optimization
dbr	Dynamic Branch Elision
vec	Vectorization
all	All of the Optimizations Above

Figure 2: Legend key for knockout experiments.

Application	Regent	Reference
Circuit	825	1701
PENNANT	1789	2416
MiniAero	2836	3993

Figure 3: Lines of code (non-comment, non-blank) for Regent and reference implementations.

erence implementations of each application. First, we compare Regent absolute performance against the reference on the target machine.

Next, to demonstrate the impact of the compiler optimizations performed by Regent, we perform *knockout experiments* for each application, disabling each optimization presented in Section 4 in turn. In addition, we perform double knockout experiments, measuring performance with all possible pairs of two optimizations disabled, and call out a few interesting combinations. As several of the optimizations impact the achieved parallelism, we evaluate each configuration in a parallel configuration and compare against the best sequential performance achieved by Regent. The labels for the various optimizations are described in Figure 2. Pointer check elision has been previously demonstrated to have a significant impact on performance [33] and has been left out of the knockout to reduce clutter.

Finally, we evaluate the productivity of Regent by comparing the number of lines of codes in each Regent implementation against each reference. Figure 3 summarizes the results. Application-specific details are described along with each application below.

6.1 Circuit

Circuit, introduced in [9], is a distributed circuit simulation, operating over an arbitrary graph of nodes and wires. While in principle the topology of the graph can be arbitrary, Circuit is concerned primarily with topologies with interconnected dense subgraphs. Such graphs allow Circuit to achieve some level of scalability, though that scalability is ultimately limited by the global all-to-all communication pattern between the subgraphs.

We compare the performance of Regent against a hand-tuned and manually vectorized CPU implementation written to the C++ Legion API. We evaluate both implementations on a graph with 800K wires connecting 200K nodes. Figure 4a shows the performance of Regent against the baseline C++ Legion implementation running on up to 8 nodes on Certainty. Notably, the fully-optimized Regent implementation—which is written in a straightforward way with no use of explicit vectors or vector intrinsics, and is less than half the total number of lines of code—achieves performance comparable to the manually vectorized C++ code, exceeding the performance that can be achieved by using the LLVM 3.5 vectorizer alone.

Figure 5a demonstrates the impact of disabling individual and pairs of optimizations on the performance of the Regent implementation of Circuit. Certain optimizations impact the parallelism available in the application; index launch optimization and mapping elision are two such optimizations. When both are disabled simultaneously, the code runs sequentially. As described in Section 4.1, the Legion runtime, in the absence of the map and unmap calls placed by the compiler, must copy back the results of each task execution before returning control to caller. This creates an effective barrier between consecutive tasks, but the effect is not noticeable as long as index launch optimization is able to parallelize the task launches. Disabling both optimizations serializes the code. But if either optimization is disabled by itself, the application continues to run in parallel at somewhat reduced throughput.

Dynamic branch elision does not have a significant impact on the performance of Circuit and has been omitted to reduce clutter.

6.2 PENNANT

PENNANT [23] is a mini-app for Lagrangian hydrodynamics representing a subset of the functionality of FLAG [13], a LANL production code. Figure 4b evaluates Regent against an OpenMP implementation of PENNANT on a problem containing approximately 2.6M zones. Implementation details of the Regent version are discussed in Section 2.

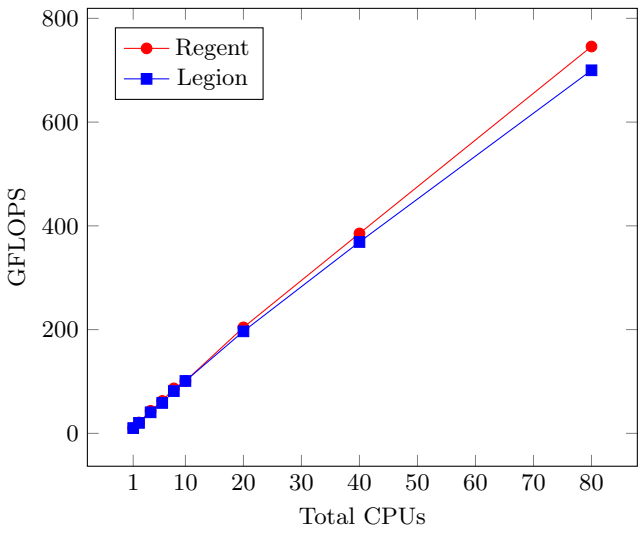
Regent performs better than OpenMP for all core counts up to 10, surpassing OpenMP by 8% at 10 cores. Starting at 12 cores, Regent performance degrades because the additional compute threads interfere with threads Legion uses for dynamic dependence analysis and data movement. The Legion runtime is also unable to exclusively allocate physical cores for each thread and abandons pinning altogether, leading to increased interference between application threads.

PENNANT is largely memory-bound, and is thus significantly impacted by the NUMA architecture of the machine. OpenMP performance was substantially impacted by CPU affinity, and a manual assignment of threads to cores was needed for optimal performance. Regent automatically binds threads to cores when possible and round robins threads between NUMA domains, thus performing well with minimal manual tuning.

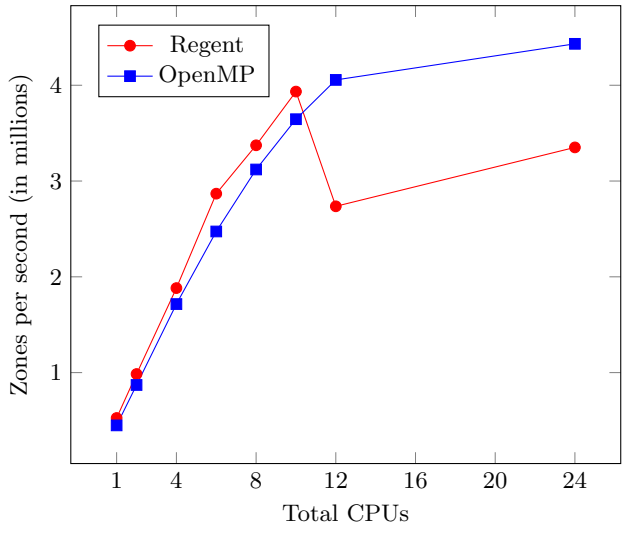
Regent achieves good performance despite an overall decrease in lines of code. The numbers listed in Figure 3 exclude a number of routines, shared between both implementations, for generating the input mesh and exporting the output of the simulation to files.

Figure 5c shows that the Regent implementation of PENNANT exhibits varied behavior with certain combinations of optimizations disabled. As with Circuit, the combination of index launches and mapping elision causes the application to execute sequentially. However, PENNANT offers a different response to the combination of index and leaf optimizations. PENNANT's pattern of task launches is such that when leaf optimization is disabled, the Legion runtime must stall for mapping to complete in order to ensure that all the dependencies are correctly captured. Circuit is structured differently from PENNANT and is therefore not impacted significantly by the leaf optimization (in combination with index launches or otherwise).

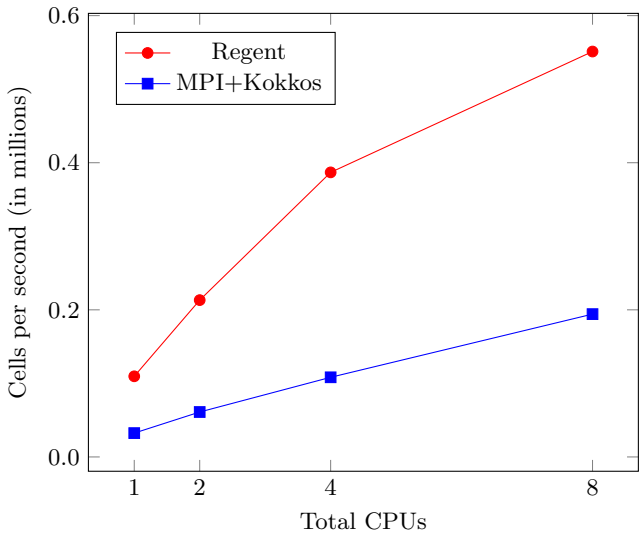
PENNANT also shows the most benefit from eliminating dynamic branches. In contrast to Circuit and MiniAero



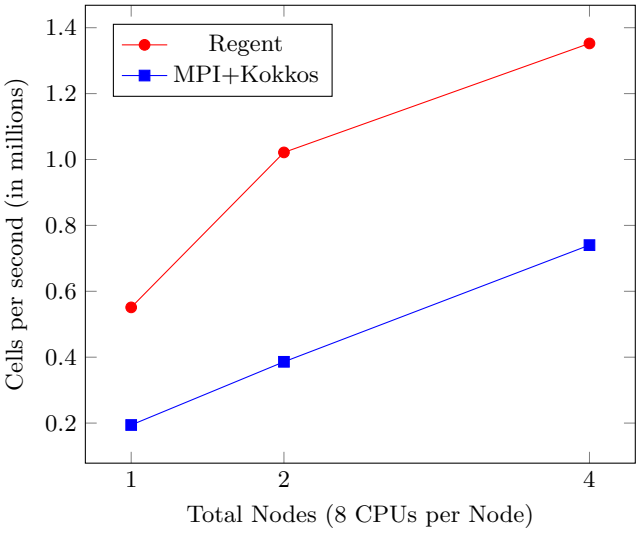
(a) Circuit performance vs Legion.



(b) PENNANT performance vs OpenMP.



(c) MiniAero single-node performance vs MPI+Kokkos.



(d) MiniAero multi-node performance vs MPI+Kokkos.

Figure 4: Performance evaluations.

which are generally compute or memory bound, certain performance critical kernels in PENNANT contain long chains of dependent math instructions, which in turn depend on conditional memory accesses (when dynamic branch elision is not enabled). At 10 cores, throughput improves by 15% if dynamic branches can be eliminated.

6.3 MiniAero

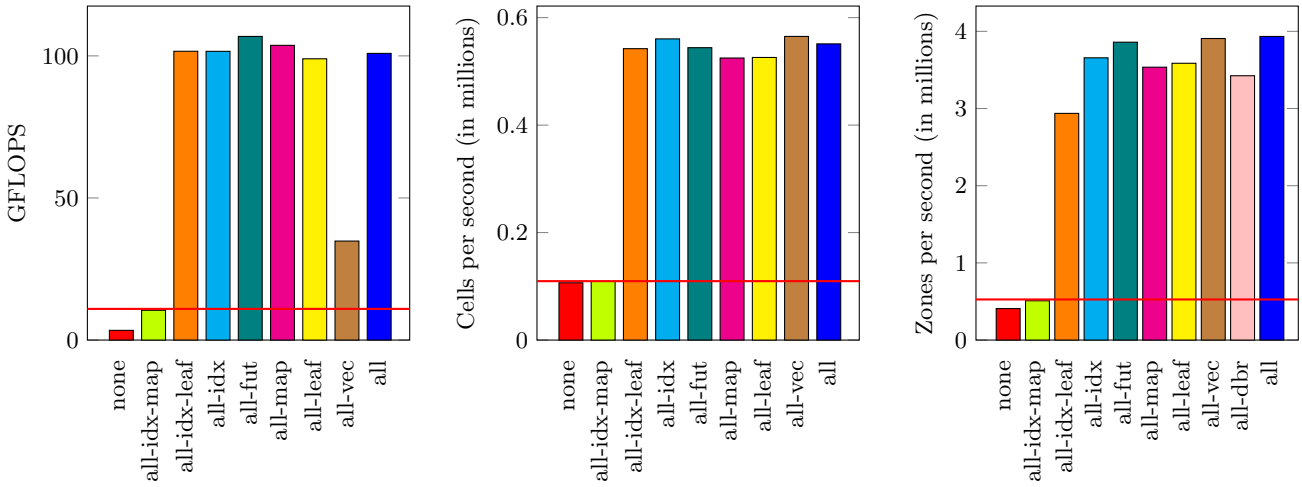
MiniAero [3] is a computational fluid dynamics mini-app that uses a Runge-Kutta fourth-order time marching scheme to solve the compressible Navier-Stokes equations on a 3D unstructured mesh. The baseline version of MiniAero is implemented as a hybrid code, using MPI for inter-node communication and Trilinos Kokkos [21] for intra-node parallelism. Figures 4c and 4d compare performance between a Regent implementation and the baseline MPI+Kokkos version on a problem size with 4M cells and 13M faces running on up to 4 nodes on Certainty. The Regent implementation for MiniAero is approximately 30% shorter by lines of code than the reference.

Regent outperforms MPI+Kokkos on 8 cores by a factor of 2.8X through the use of a hybrid SOA-AOS data layout, an approach similar to that taken in [10]. The improved data layout substantially boosts cache reuse and improves utilization of memory bandwidth.

Figure 5b demonstrates that MiniAero is sensitive to the combination of index launches and mapping elision. When both optimizations are disabled, the code runs serially. MiniAero is otherwise relatively resilient to the choice of optimizations performed. MiniAero is not significantly impacted by dynamic branch elision (not shown).

6.4 Limitations

While the Regent compiler significantly simplifies aspects



(a) Circuit (10 CPUs) (b) MiniAero (8 CPUs) (c) PENNANT (10 CPUs)

Figure 5: Knockout experiments. The red line in each graph shows the best sequential Regent performance.

of writing to the Legion programming model, our current implementation does have limitations. The most important one is that Legion's support for *simultaneous* access to regions by tasks is not currently implemented in Regent [9]. Briefly, simultaneous access allows Legion programs to implement SPMD style patterns, such as are typically used in UPC [14, 2], Titanium [35] and MPI [31] programs, where multiple concurrently executing tasks share access to the same region, mediated by explicit synchronization operations. Simultaneous access exists in Legion to address a problem that any dynamic task-based model with subtasks (i.e., not just Legion) faces. The performance of a task that launches substasks depends on the number of subtasks launched and the duration of those subtasks; the runtime overhead incurred is proportional to the number of subtasks launched, and so the more subtasks that are launched by a task, the longer those subtasks must run to keep the ratio of useful work to runtime overhead low. When launching very large numbers of tasks, say across a petascale supercomputer, the runtime overheads will dominate the useful computation of the tasks unless the tasks run for a long time. For applications in which the bulk of the available parallelism comes from repeated and regular data-parallel task launches, the SPMD structure allows a very large number of long-running tasks to be launched at application start, and then be further subdivided using the functional-style task calls we have shown in our examples in this paper (where each task has *exclusive* access to its region arguments [9]).

In our experience, such applications should be written in the SPMD syle at the outermost level of control if they are to scale past somewhere between 10 and 100 nodes, depending on the application domain. Because of that, and because the optimizations described in this paper are effective even on a single node, we have focused on experiments on one or a small number of nodes. There is no technical barrier to adding support for simultaneous access to Regent as it already exists in Legion. This approach is known to provide scalability to more than 10K nodes in Legion [7], and it may be the solution we eventually adopt, but simultaneous access does add another dimension of complexity to the programming model that we would like to avoid in Regent if possible. As future work we are interested in whether it is possible to provide scalability to large numbers of nodes in Regent for applications with regular (and repetitive) data parallel tasks by compiler transformations that target Legion's simultaneous coherence without exposing that feature to the Regent programmer.

7. RELATED WORK

Legion [9], though implemented as a runtime system, is designed to be a programming model for which many properties could in principle be statically checked. In particular, there is a static type system for key Legion abstractions [33]. Regent extends that work into a complete language and compiler, with a focus on productivity and performance. Among other things, partitions in Regent are first-class and can be passed to subtasks, facilitating certain interesting design patterns impossible in the initial type system.

Legion is itself the spiritual successor of Sequoia [22]. Most notably, Sequoia, in contrast to Legion, was entirely static: the application, machine specification, and mapping from application to machine were all given as inputs to the compiler, which produced an optimized (but entirely static) executable. Legion grew out of the difficulties encountered in adapting Sequoia to irregular, and thus more dynamic, applications [8]. Regent itself differs from the Sequoia compiler [29] in that the compiler plays a different role with respect to the runtime system, facilitating the efficient operation of the runtime, rather than the other way around. As a result, the optimization needs of each system differ.

Deterministic Parallel Java (DPJ) [11, 12] is a parallel extension to the Java programming language adding support for regions. Regions in DPJ also express locality, as they do in Regent. However, DPJ is a fully static system, while Regent is a hybrid system with a static type system and compiler optimizations but also an aggressively optimizing dynamic runtime. As a result, while Regent's semantic safety properties must be enforced at compile time, Regent is free to discover parallelism at runtime, giving it the ability to exploit dynamic information about the application when

insufficient static information is available.

An older but related language is Jade [30]. Jade provides an apparently-serial programming model that implicitly parallelizes through the use of a dynamic dataflow graph. Jade differs substantially in design and implementation from Legion because of the characteristics of the hardware at the time, when processors were much slower relative to networks and it was plausible to track dynamic dataflow on a per-object basis. Regent addresses these challenges by aggregating data with logical regions and providing language constructs for hierarchical decomposition of those regions. Regent's design also allows it to employ a hierarchical, distributed scheduling algorithm [33].

Swift/T [34, 5] (not to be confused with Apple Swift) is a more recent effort focusing on high-productivity scripted parallel workflows. Both Swift/T and Regent employ an implicitly parallel programming model, and both achieve parallelism through dataflow. Swift/T differs from Regent in that it is purely functional, while Regent allows tasks to have side-effects on regions, but serializes execution when tasks have the potential to interfere. Unlike regions, Swift/T's aggregate data types do not have a first-class concept of partitioning, as in Regent. Beyond this, while both Swift/T and Regent optimize for parallelism, the Swift/T makes no attempt to generate efficient sequential code while Regent is able to match the performance of hand-tuned and manually vectorized kernels.

Other runtime systems with support for task-based programming include Charm++ [27], Uintah [19], StarPU [6], and OCR [4]. While these systems differ substantially in the details, they all provide a common abstraction of a task as a fundamental unit of execution, with a graph of dependencies between tasks providing communication and synchronization. These systems differ from Regent in that, as dynamic runtime systems, they impose a small but potentially significant runtime overhead which must be overcome to achieve performance. Beyond this, these systems do not provide the ability to partition data multiple ways and to migrate data dynamically between these views as the application moves between different phases of computation. This makes several design patterns more straightforward to implement in Regent compared to the above systems.

Another related class of work is represented by PGAS and related models, such as UPC [14, 2], Titanium [35], Chapel [16, 15], X10 [18, 17, 25] and HPX [26]. Regent is similar to PGAS programming models in so far as pointers created in Regent are valid everywhere in the machine. However, Regent pointers can only be dereferenced when the task in question has privileges to the region that it points into. This property allows Regent to preserve a number of other important properties important for performance and correctness, such as guaranteeing non-interference of tasks executing in parallel.

8. CONCLUSION

We have presented Regent, a high-productivity programming language for high performance computing with logical regions. Regent provides a sequential programming model that parallelizes implicitly by performing dynamic dataflow on regions. We have presented an optimizing compiler for Regent which produces efficient Legion implementations, and evaluated the implementation on three benchmark applications.

Acknowledgments

The authors would like to thank Charles R. Ferenbaugh for his advice and assistance with the PENNANT code. The authors wish to express their appreciation to Janine C. Bennett, Greg Sjaardema, Kenneth Franko, Hemanth Kolla, Jeremiah Wilke, David Hollman, and the Mantevo project [3] for support with the MiniAero code.

This work was supported by the Department of Energy National Nuclear Security Administration under Award Number DE-NA0002373-1, and by Los Alamos National Laboratories Subcontract No. 173315-1 through the U.S. Department of Energy under Contract No. DE-AC52-06NA25396. The work used computing resources at the High Performance Computing Center at Stanford University [1], supported by the National Science Foundation under Award Number 0960306.

9. REFERENCES

[1] High Performance Computing Center at Stanford University. http://hpcc.stanford.edu/.

[2] UPC language specification v1.2. upc.lbl.gov/docs/user/upc_spec_1.2.pdf, 2011.

[3] Mantevo project. https://mantevo.org/, Nov. 2014.

[4] The Open Community Runtime interface. https://xstackwiki.modelado.org/images/1/13/Ocr-v0.9-spec.pdf, 2014.

[5] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *Supercomputing (SC)*, 2014.

[6] C. Augonnet et al. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, Feb. 2011.

[7] M. Bauer. *Legion: Programming Distributed Heterogeneous Architectures with Logical Regions*. PhD thesis, Stanford University, 2014.

[8] M. Bauer, J. Clark, E. Schkufza, and A. Aiken. Programming the memory hierarchy revisited: Supporting irregular parallelism in Sequoia. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, 2011.

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

[10] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Structure slicing: Extending logical regions with fields. In *Supercomputing (SC)*, 2014.

[11] R. Bocchino et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, 2009.

[12] 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.

[13] D. E. Burton. Consistent finite-volume discretization of hydrodynamics conservation laws for unstructured grids. Technical Report UCRL-JC-118788, Lawrence Livermore National Laboratory, Livermore, CA, 1994.

[14] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and language specification. UC Berkeley Technical Report:

CCS-TR-99-157, 1999.

[15] B. Chamberlain, S. Choi, S. Deitz, D. Iten, and V. Litvinov. Authoring user-defined domain maps in Chapel. 2011.

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

[17] S. Chandra et al. Type inference for locality analysis of distributed data structures. In *PPoPP*, pages 11–22, 2008.

[18] P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[19] J. Davison de St.Germain, J. McCorquodale, S. Parker, and C. Johnson. Uintah: a massively parallel problem solving environment. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 33–41, 2000.

[20] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. PLDI, 2013.

[21] H. Edwards and C. Trott. Kokkos: Enabling performance portability across manycore architectures. In *Extreme Scaling Workshop (XSW), 2013*, pages 18–24, Aug 2013.

[22] K. Fatahalian et al. Sequoia: Programming the memory hierarchy. In *Supercomputing*, November 2006.

[23] C. R. Ferenbaugh. PENNANT: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience*, 2014.

[24] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho. Lua - an extensible extension language. *Softw., Pract. Exper.*, 1996.

[25] M. Joyner, Z. Budimlic, and V. Sarkar. Subregion analysis and bounds check elimination for high level arrays. In *Compiler Construction*, pages 246–265, 2011.

[26] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A task based programming model in a global address space. In *Partitioned Global Address Space Programming Models*, 2014.

[27] L. Kalé and S. Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of OOPSLA'93*, pages 91–108, 1993.

[28] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[29] M. Ren, J. Y. Park, M. Houston, A. Aiken, and W. Dally. A tuning framework for software-managed memory hierarchies. In *Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, 2008.

[30] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 1998.

[31] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference*. MIT Press, 1998.

[32] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 2000.

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

[34] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Large-scale application composition via distributed-memory dataflow processing. In *Cluster, Cloud and Grid Computing (CCGrid)*, 2013.

[35] K. Yelick et al. Titanium: A high-performance Java dialect. In *Workshop on Java for High-Performance Network Computing*, 1998.