Legion: Expressing Locality and Independence with Logical Regions

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*—Modern parallel architectures have both heterogeneous processors and deep, complex memory hierarchies. We present Legion, a programming model and runtime system for achieving high performance on these machines. Legion is organized around *logical regions*, which express both locality and independence of program data, and *tasks*, functions that perform computations on regions. We describe a runtime system that dynamically extracts parallelism from Legion programs, using a distributed, parallel scheduling algorithm that identifies both independent tasks and nested parallelism. Legion also enables explicit, programmer controlled movement of data through the memory hierarchy and placement of tasks based on locality information via a novel mapping interface. We evaluate our Legion implementation on three applications: fluid-flow on a regular grid, a three-level AMR code solving a heat diffusion equation, and a circuit simulation.**

I. Introduction

Modern parallel machines are increasingly complex, with deep, distributed memory hierarchies and heterogeneous processing units. Because the costs of communication within these architectures vary by several orders of magnitude, the penalty for mistakes in the placement of data or computation is usually very poor performance. Thus, to achieve good performance the programmer and the programming system must reason about *locality* (data resident close to computation that uses it) and *independence* (computations operating on disjoint data, and therefore not requiring communication and able to be placed in possibly distant parts of the machine). Most contemporary programming systems have no facilities for the programmer to express locality and independence. The few languages that do focus primarily on array-based locality [1], [2], [3] and avoid irregular pointer data structures.

In this paper we describe Legion, a parallel programming system based on using *logical regions* to describe the organization of data and to make explicit relationships useful for reasoning about locality and independence. A logical region names a set of objects. Logical regions are first-class values in Legion and may be dynamically allocated, deleted and stored in data structures. Regions can also be passed as arguments to distinguished functions called *tasks* that access the data in those regions, providing locality information. Logical regions may be *partitioned* into disjoint or aliased (overlapping) *subregions*, providing information for determining independence of computations. Furthermore, computations access logical regions with particular *privileges* (*read-only*, *read-write*, and *reduce*) and *coherence* (e.g., *exclusive access* and *atomic access*, among others). Privileges express how a task may use its region arguments, providing data dependence information that is used to guide the extraction of parallelism. For example, if two tasks access the same region with read-only privileges the two tasks can potentially be run in parallel. Coherence properties express the required semantics of concurrent region accesses. For example, if the program executes $f_1(r); f_2(r)$ and tasks f_1 and f_2 both require *exclusive* access to region r, then Legion guarantees the result will be as if $f_1(r)$ completes before $f_2(r)$ begins. On the other hand, if the tasks access r with *atomic* coherence, then Legion guarantees only atomicity of the tasks with respect to r: either task $f_1(r)$ appears to run entirely before $f_2(r)$ or vice versa.

Logical regions do not commit to any particular layout of the data or placement in the machine. At runtime, each logical region has one or more *physical instances* assigned to specific memories. It is often useful to have multiple physical instances of a logical region (e.g., to replicate read-only data, or to allow independent reductions that are later combined).

To introduce the programming model, we present a circuit simulation in Section II, illustrating regions, tasks, permissions and coherence properties, the interactions between them, and how these building blocks are assembled into a Legion program. Subsequent sections each describe a contribution in the implementation and evaluation of Legion:

- We define a *software out-of-order processor*, or SOOP, for scheduling tasks with region arguments in a manner analogous to how out-of-order hardware schedulers process instructions with register arguments (Section III). In addition to pipelining the execution of tasks over several stages, our SOOP is distributed across the machine and is also hierarchical to naturally extract nested parallelism (because tasks may recursively spawn subtasks).
- Of central importance is how tasks are assigned (or *mapped*) to processors and how physical instances of logical regions are mapped to specific memory units

This work was supported by the DARPA UHPC project through a subcontract of NVIDIA (DARPA contract HR0011-10-9-0008), and by the Army High Performance Research Center, through a subcontract from Los Alamos National Laboratory for the DOE Office of Science, Advanced Scientific Computing Research (DE-AC52-06NA25396). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

SC12, November 10-16, 2012, Salt Lake City, Utah, USA
978-1-4673-0806-9/12/ $31.00 © 2012 IEEE

```
1  struct Node { float voltage, new_charge, capacitance; };
2  struct Wire⟨rn⟩ { Node@rn in_node, out_node; float current, ... ; };
3  struct Circuit { region   r_all_nodes; /* contains all nodes for the circuit */
4                   region   r_all_wires; /* contains all circuit wires */ };
5  struct CircuitPiece {
6    region  rn_pvt, rn_shr, rn_ghost; /* private, shared, ghost node regions */
7    region  rw_pvt;                   /* private wires region */ };
8
9  void simulate_circuit(Circuit c, float dt) : RWE(c.r_all_nodes, c.r_all_wires)
10 {
11   // The construction of the colorings is not shown.  The colorings wire_owner_map,
12   // node_owner_map, and node_neighbor_map have MAX_PIECES colors
13   // 0..MAX_PIECES − 1. The coloring node_sharing map has two colors 0 and 1.
14   //
15   // Partition of wires into MAX_PIECES pieces
16   partition⟨disjoint⟩ p_wires = c.r_all_wires.partition(wire_owner_map);
17   // Partition nodes into two parts for all−private vs. all−shared
18   partition⟨disjoint⟩ p_nodes_pvs = c.r_all_nodes.partition(node_sharing map);
19
20   // Partition all−private into MAX_PIECES disjoint circuit pieces
21   partition⟨disjoint⟩ p_pvt_nodes = p_nodes_pvs[0].partition(node_owner_map);
22   // Partition all−shared into MAX_PIECES disjoint circuit pieces
23   partition⟨disjoint⟩ p_shr_nodes = p_nodes_pvs[1].partition(node_owner_map);
24   // Partition all−shared into MAX_PIECES ghost regions, which may be aliased
25   partition⟨aliased⟩ p_ghost_nodes = p_nodes_pvs[1].partition(node_neighbor_map);
26
27   CircuitPiece pieces[MAX_PIECES];
28   for(i = 0; i < MAX_PIECES; i++)
29     pieces[i] = { rn_pvt: p_pvt_nodes[i], rn_shr: p_shr_nodes[i],
30                   rn_ghost: p_ghost_nodes[i], rw_pvt: p_wires[i] };
31   for (t = 0; t < TIME_STEPS; t++) {
32     spawn (i = 0; i < MAX_PIECES; i++) calc_new_currents(pieces[i]);
33     spawn (i = 0; i < MAX_PIECES; i++) distribute_charge(pieces[i], dt);
34     spawn (i = 0; i < MAX_PIECES; i++) update_voltages(pieces[i]);
35   }
36 }
37                         // ROE = Read−Only−Exclusive
38 void calc_new_currents(CircuitPiece piece):
39         RWE(piece.rw_pvt), ROE(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
40   foreach(w : piece.rw_pvt)
41     w→current = (w→in_node→voltage − w→out_node→voltage) / w→resistance;
42 }
43                         // RdA = Reduce−Atomic
44 void distribute_charge(CircuitPiece piece, float dt):
45         ROE(piece.rw_pvt), RdA(piece.rn_pvt, piece.rn_shr, piece.rn_ghost) {
46   foreach(w : piece.rw_pvt) {
47     w→in_node→new_charge += −dt ∗ w→current;
48     w→out_node→new_charge +=  dt ∗ w→current;
49   }
50 }
51
52 void update_voltages(CircuitPiece piece): RWE(piece.rn_pvt, piece.rn_shr) {
53   foreach(n : piece.rn_pvt, piece.rn_shr) {
54     n→voltage += n→new_charge / n→capacitance;
55     n→new_charge = 0;
56   }
57 }
```

Listing 1.   Circuit simulation.

(Section IV). Often using application-specific information results in better mappings than a generic mapping strategy. We describe a *mapping interface* that allows programmers to give the SOOP a specification of how to map tasks and regions for a specific application, or even part of an application. This mapping API is designed so that any user-supplied mapping strategy can only affect the performance of applications, not their correctness.

- We present results of experiments on three applications: fluid-flow on a regular grid, a three-level AMR code solving a heat diffusion equation, and a circuit simulation. We compare each application with the best reference versions on three different clusters of multicore processors with GPUs, including the Keeneland supercomputer [4].

II. Example: Circuit Simulator

We begin by describing an example program written in the Legion programming model. Listing 1 shows code for an electrical circuit simulation, which takes a collection of wires and nodes where wires meet. At each time step the simulation calculates currents, distributes charges, and updates voltages.

The key decisions in a Legion program are how data is grouped into regions and how regions are *partitioned* into *subregions*. The goal is to pick an organization that makes explicit which computations are independent. A `Circuit` has two regions: a collection of nodes and a collection of wires (line 3 of Listing 1).[1] An efficient parallel implementation breaks this unstructured graph into pieces that can be processed (mostly) independently. An appropriate region organization makes explicit which nodes and wires are involved in intra-piece computation and, where wires connect different pieces, which are involved in inter-piece computation.

Figure 1(b) shows how the nodes in a small graph might be split into three pieces. Blue (lighter) nodes, attached by wires only to nodes in the same piece, are *private* to the piece. Red (darker) nodes, on the boundary of a piece, are *shared* with (connected to) other pieces. In the simulation, computations on the private nodes of different pieces are independent, while computations on the shared nodes require communication. To make this explicit in the program, we partition the nodes region into private and shared subregions (line 18). To partition a region, we provide a *coloring*, which is a relation between the elements of a region and a set of colors. For each color c in the coloring, the partition contains a subregion r of the region being partitioned, with r consisting of the elements colored c. Note that the partition into shared and private nodes is disjoint because each node has one color.

The private and shared nodes are partitioned again into private and shared nodes for each circuit piece (lines 21 and 23); both partitions are disjoint. There is another useful partition of the shared nodes: for a piece i, we will need the shared nodes that border i in other pieces of the circuit. This *ghost node* partition (line 25) has two interesting properties. First, it is a second partition of the shared nodes: we have two views on to the same collection of data. Second, the ghost node partition is *aliased*, meaning the subregions are not disjoint: a node may border several different circuit pieces and belong to more than one ghost node subregion (thus, `node_neighbor_map` on line 25 assigns more than one color to some nodes). The private, shared, and ghost node subregions for the upper-left piece of the example graph are shown in Figures 1(c), 1(d), and 1(e) respectively.

Figure 1(a) shows the final hierarchy of node partitions and subregions. The $*$ symbol indicates a partition is disjoint. This *region tree* data structure plays an important role in scheduling tasks for out-of-order execution (see Section III). The organization of the wires is much simpler: a single disjoint

[1]Note that all pointers declare the region to which they point. For example, the definition of `Wire` (line 2) is parametrized on the region `rn` to which the `Node` pointers in fields `in_nodes` and `out_nodes` point.

partition that assigns each wire to one piece (line 16).

Line 9 declares the main simulator function, which specifies the regions it accesses and the privileges and coherence it requires of those regions. The `RWE` annotation specifies that the regions `c.r_all_nodes` and `c.r_all_wires` are accessed with read-write privileges and *exclusive* coherence (i.e., no other task can access these two regions concurrently or be reordered around this task if they use either region). Privileges specify what the function can do with the regions; coherence specifies what other functions can do with the regions concurrently. Functions that declare their accessed regions, privileges, and coherence are called *tasks* and are the unit of parallel execution in Legion.

Lines 31-57 perform the actual simulation by making three passes over the circuit for each time step. Each pass loops over an array of pieces (constructed on lines 27-30 from the partitions), spawning a task for each piece. There are no explicit requests for parallel execution (`spawn` simply indicates a task call and does not mandate parallel execution) nor is there explicit synchronization between the passes. Which tasks can be run in parallel within a pass and the required inter-pass dependences are determined automatically by the Legion runtime based on the region access annotations on the task declarations. The tasks spawned on lines 32-34 are *subtasks* of the main `simulate_circuit` task. A subtask can only access regions (or subregions) that its parent task could access; furthermore, the subtask can only have permissions on a region compatible with the parent's permissions.

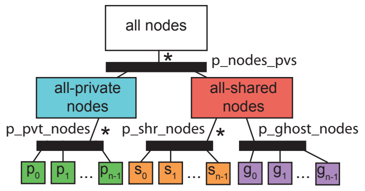
The `calc_new_currents` task reads and writes the wires subregion and reads the private, shared, and ghost node subregions for its piece. The `distribute_charge` task reads the piece's wires subregion and updates all nodes those wires touch. However, rather than using read/write privilege for the nodes (which would serialize these tasks for correctness), the task uses reorderable reduction operations and atomic rather than exclusive access. The final task `update_voltages` writes the shared and private nodes for a piece and reads the results of the previous task's reductions.

Listing 1 illustrates one way of constructing partitioned data structures in Legion: populate a region with data (the example makes use of whatever data has been allocated by the caller of `simulate_circuit`) and then partition it into subregions. One can also first partition an empty region into subregions and then allocate data in the subregions.

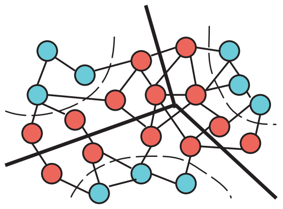
III. The Software Out-of-Order Processor

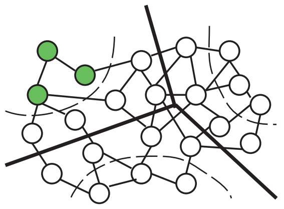
Legion uses a *software out-of-order processor*, or SOOP, to schedule tasks. A SOOP dynamically schedules a stream of tasks in a manner akin to an out-of-order processor scheduling a stream of instructions. Just as an instruction scheduler is constrained by register dependences, the SOOP is constrained by region dependences. The SOOP is pipelined, distributed, and extracts nested parallelism from subtasks. There are two major challenges in implementing an efficient task scheduler:

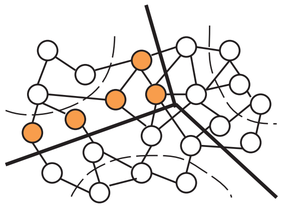
- For correctness, Legion must preserve data dependences between tasks, which is non-trivial because the same data may be in multiple different regions or subregions.
- For performance, Legion must hide the extremely long latencies associated with machines that have both distributed memory and many levels of memory hierarchy.



(a) Node region tree.


(b) p_nodes_pvs


(c) p_i


(d) s_i

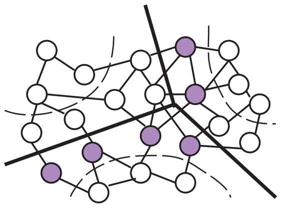

(e) g_i

Fig. 1. Partitions of r_all_nodes.

To solve the second problem Legion uses a *deferred execution model* that decouples the issuing of operations from when operations are performed. Using a low-level runtime event system (which we do not describe further in this paper), an issued operation waits for other operations on which it is dependent to complete before executing, but a waiting operation does not cause the SOOP to block. With deferred execution, it is worthwhile for the SOOP to run (potentially far) ahead of actual execution, allocating resources to and issuing tasks that may wait because they have data dependences on earlier tasks that have not finished executing. Pipelining and distributing the SOOP further improves throughput and hides latency.

To dynamically detect and enforce data dependences, the SOOP considers, for each task t, the regions t uses and ensures t waits on any earlier task whose region uses conflict with t's. There is a difficulty, however. The programmer's regions are logical; a region names a set of objects, but does not say where those objects are in memory. To ensure dependences are satisfied the SOOP must also *map* t: assign an appropriate physical instance to each of t's logical regions. Thus, dependence analysis requires knowing both which tasks t depends on and how those other tasks are mapped.

Our SOOP design solves this problem by dividing the analysis of dependences between two pipeline stages. The first SOOP stage computes task dependences at the granularity of logical regions (Section III-A), which does not give enough

	Exclusive	Atomic	Simultaneous	Relaxed
Exclusive	Dep	Dep	Dep	Dep
Atomic	Dep	Same	Cont	Cont
Simultaneous	Dep	Cont	Same	None
Relaxed	Dep	Cont	None	None

Fig. 2. Dependence table.

information to map t, but does identify all tasks that must map before t can map. The third SOOP stage maps t by carrying out a more refined analysis of the (already mapped) tasks on which t depends (Section III-C). Between these stages, the second stage distributes tasks to other processors (Section III-B). Once a task has been assigned a processor and physical instances it is issued for deferred execution (Section III-D). The final SOOP stage reclaims resources from completed tasks (Section III-E).

A. Stage 1: Mapping Dependences

Each processor in the system runs an instance of the SOOP. When a *parent* task spawns a *child* task, the child is *registered* with the SOOP on the parent's processor; registration records the subtask's logical regions, privileges and coherence properties. Children are registered in the sequential order the parent spawns them and enqueued for mapping dependence analysis. In the circuit simulation in Listing 1, the spawn statements on lines 32-34 register all three kinds of subtasks (in program order) on the processor where `simulate_circuit` executes.

Detecting mapping dependences between a newly registered task t and a previously registered task t' requires comparing the two sets of logical regions accessed. For each logical region used by t' that may *alias* (may share data with) a logical region used by t, the privileges and coherence modes are compared to determine whether a dependence exists. If both regions need only read privileges there is never a dependence, but if either task needs write or reduction privileges, the coherence modes are compared using the table in Figure 2. `Dep` indicates dependence while `None` indicates independence. `Same` is a dependence unless the two tasks use the same physical instance of the logical region. Since tasks have not mapped physical instances at this point in the pipeline, `Same` is always a mapping dependence. `Cont` indicates a dependence contingent on the privileges of the two tasks (e.g. an Atomic Read-Write task and Simultaneous Read-Only task will not have a dependence).

The table lists *simultaneous* and *relaxed* coherence modes that we have not yet discussed. Both modes allow other tasks using the region to execute at the same time and differ only in what updates must be observed. With simultaneous coherence, a task must see all updates to the logical region made by other tasks operating on the same region simultaneously (i.e., shared memory semantics). With relaxed coherence, a task may or may not observe concurrent updates.

A key property is that dependence analysis is not needed between arbitrary pairs of tasks. In fact, it suffices to check only *siblings* (children with the same parent) for dependences.

Observation 1. Let t_1 and t_2 be two sibling tasks with no dependence. Then no subtask of t_1 has a dependence with any subtask of t_2.

Recall from Section II that subtasks only access regions (or subregions) that their parent accesses. Thus if the regions used by t_1 and t_2 do not alias, the regions used by any subtasks of t_1 cannot alias the regions used by any subtask of t_2. If regions of t_1 and t_2 alias but there is no dependence because the regions have simultaneous or relaxed coherence, then by definition there is no dependence between the subtasks.

Consider the first two subtasks spawned on line 32 in Listing 1, `calc_new_currents(pieces[0])` and `calc_new_currents(pieces[1])`. The first of these tasks reads and writes the private wires subregion `pieces[0].rw_pvt` in exclusive mode, and reads in exclusive mode the node subregions `pieces[0].rn_pvt`, `.rn_shr`, and `.rn_ghost` (see line 39). The second subtask must be checked against the first for any dependences; the second subtask uses `pieces[1].rw_pvt` (read / write exclusive) and `pieces[1].rn_pvt`, `.rn_shr`, and `.rn_ghost` (read-only exclusive). It may be helpful to refer to the region tree for nodes in Figure 1(a) (recall that the wires region tree consists of a single disjoint partition). We give a few representative examples of reasoning about pairwise aliasing (not all pairs are covered):

1) `pieces[0].rw_pvt` and `pieces[1].rw_pvt` do not alias as they are different subregions of a disjoint partition.
2) `pieces[0].rn_ghost` and `pieces[1].rn_shr` alias as they are in different partitions of the same region.
3) Subregions of an aliased partition always alias (e.g., `pieces[0].rn_ghost` and `pieces[1].rn_ghost`).
4) `pieces[0].rw_pvt` and `pieces[1].rn_pvt` do not alias because they are in different region trees.

Cases 2 and 3 indicate a possible dependence, however the regions are accessed with only read privileges. All other cases for these two subtasks are similar to one of the examples above; thus, the two subtasks are independent.

A closer look at this example shows that whether r_1 aliases r_2 can be determined by examining their least common ancestor $r_1 \sqcup r_2$ in the region tree. If $r_1 \sqcup r_2$ either does not exist (the regions are in different region trees) or is a disjoint partition, then r_1 and r_2 are disjoint. If $r_1 \sqcup r_2$ is an aliased partition or a region, then r_1 and r_2 alias.

Consider tasks `distribute_charge(pieces[0],dt)` and `update_voltages[1]` in Listing 1. The former task uses region `pieces[0].rn_ghost` and the latter uses region `pieces[1].rn_shr`. The least common ancestor is the region of all shared nodes `p_nodes_pvs[1]`, so these two regions alias. Since both tasks modify their respective subregions, there is a dependence and the `update_voltages` task can only map after the `distribute_charges` task has mapped.

We can now describe the algorithm for mapping dependence analysis. The SOOP maintains a local region forest for each task t, the roots of which are t's region arguments and including all partitions and subregions used by t. Dependence analysis places the child tasks of t on the regions they use, maintaining the following invariant. Let r' be a region used by child task t', and let R be the set of regions r'' such that r'' is used by a task t'' registered after t' and t'' depends on t' because of aliasing between r'' and r'. Then t' is listed on the region that is the least common ancestor of the set $R \cup \{r'\}$. Placing t' on an ancestor of r' coarsens the dependences involving t', which can result in false dependences. However, the loss of information can be shown to be small and only affects when t can map; any information lost is recovered during mapping. Furthermore, the benefit is that it dramatically speeds up dependence analysis overall.

We can identify all mapping dependences between sibling tasks and place tasks in the correct place in the region tree in amortized time $\mathcal{O}(d)$ per region, where d is the depth of the region forest. For region r' used by task t', the analysis walks the unique path in the region forest from a root to r'; task t' depends on all tasks on this path when the walk completes. For each node m on the walk from the root to r' the following actions are taken:

- If m is a region with multiple child partitions and m' is the child on the path to r', then any task in a subtree of m other than m' on which t' depends is moved to m.
- If m is a non-disjoint partition and m' is the child on the path to r', then any task in a subtree of m other than m' that t' depends on is moved to m.
- If m is a disjoint partition and m' is the child on the path to r', then no tasks in a subtree of m other than m' can conflict with t'; we simply move on to m'.
- If $m = r'$ then any dependent tasks in r''s subtree are moved to r'.

There are two parts to this analysis: the walk from a region root to r' and the off-path work to move dependent tasks to the least common ancestor on the path. The walk to r' is bounded by the depth of the region tree. The off-path work can be made efficient by maintaining at each node a record of the *mode* of each subtree (whether the subtree has any tasks and with what privileges), enabling the off-path search to traverse directly and only to dependent tasks. The off-path work is proportional to how far a task is moved back up the region tree. Thus, for each region argument a task is initially placed at some depth in the forest and subsequently only moved up towards a root, so the overall work per region is (amortized) $\mathcal{O}(d)$.

We illustrate dependence analysis using the four tasks discussed above. Many more tasks are executed by the circuit simulation, but this subset illustrates the important points. Figure 3 shows the node region tree in three stages. In Figure 3(a), the two tasks `calc_new_currents(pieces[0])` (c_0) and `calc_new_currents(pieces[1])` (c_1) have been placed on their region arguments, which determines these two tasks are independent. Next, in Figure 3(b), the task `distribute_charge(pieces[0],dt)` (d_0) has dependences with both c_0 and c_1 caused by aliasing between the pieces of the shared node partition and the ghost node partition, so both c_0 and c_1 are moved to the `p_nodes_pvs[1]` region. Finally, the task `update_voltages[1]` causes d_0 to also be moved to `p_nodes_pvs[1]` for essentially the same reason; the final state of the node region tree is shown in Figure 3(c).

B. Stage 2: Distribution

The first step in distribution is that t waits for all tasks on which t depends to map; nothing happens to t during this time. Once t's mapping dependences are satisfied, t is ready to be mapped and is placed in the SOOP's mapping queue. At this point SOOPs on other processors may ask to steal t from its home SOOP. The home SOOP may decline the request, but if the request is granted, task t along with a copy of t's region forest is sent to the remote SOOP. If t has not been stolen when it reaches the head of the queue, the SOOP decides whether to execute t on the local processor or send it to another processor. Thus, task distribution to processors in Legion is both "pull" (stealing) and "push". The various policies (whether steal requests are granted, when to send a task to another processor, etc.) are part of the *mapping interface* and can be defined by the user (see Section IV).

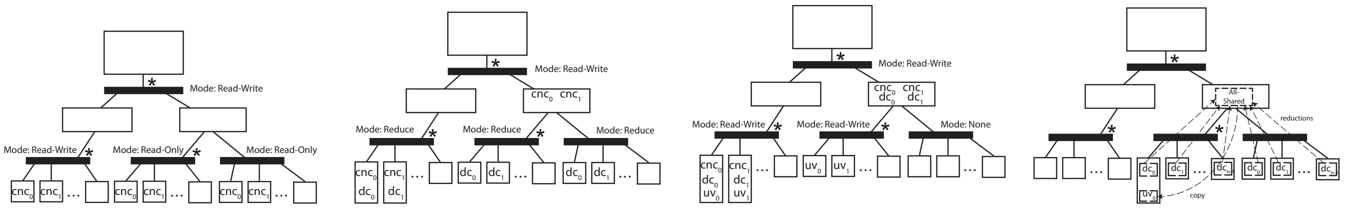
C. Stage 3: Mapping

There are two steps to mapping a region r used by task t. First, the mapper must ensure there is at least one *valid* physical instance of r for t to use. Not all instances have up-to-date data at all times. For example, if an instance of a subregion of r has been written by a previous task, it is necessary to copy that subregion back to an instance of r so that t sees the correct data. In the second step, the mapping interface selects one of the valid physical instances of r for t to use or creates a new one. We focus on the first step; the mapping interface is discussed further in Section IV.

Each logical region in the region tree maintains a list of its physical instances and which sibling tasks are using those instances and with what permissions. The important correctness property established by stage 1 is that all tasks on which t depends map before t maps and no task that depends on t maps before t finishes mapping. Thus, when t maps, the region tree will be in a state consistent with an execution in which all tasks mapped in program order.

Detecting or creating a valid instance of r is, at its core, dependence analysis on physical instances and thus similar to the mapping dependence analysis on logical regions in Section III-A. Consider mapping task t with task t' using physical instance s' of logical region r' in the region tree. There are four cases.

1) If t' has only read privileges for r' it cannot invalidate any instance of r.
2) If t' has read/write privileges for r', and r aliases r', then s' must be copied to an instance s'' of $r \sqcup r'$ and a fresh instance of r created from s''. Instance s' is removed from the region tree.
3) If t' has reduce privileges for r' and t has reduce privileges for r and both use the same reduction operator, then nothing is done. This is the only case in which



(a) Two calc_new_currents tasks. (b) distribute_charge depends on calc_new_currents. (c) update_voltages depends on distribute_charge. (d) An update_voltages mapping.

Fig. 3. Dependence analysis examples from `circuit_simulation`.

a writer does not require a valid instance—because reductions can be reordered, combining the instances of r and r' (if they alias) can be deferred to a later consumer of the data (see case 4).

4) If t' has reduce privilege for r' and t has read, write, or reduce privileges with a different operator than t' for r, and r aliases r', then s' must be reduced (using t''s reduction operator) to an instance s'' of $r \sqcup r'$ and then a fresh instance of r created from s''. Instance s' is removed from the region tree.

To map r, we walk from r's root ancestor in the region forest to r, along the way exploring off-path subtrees to find region instances satisfying cases 2 and 4. The details and analysis are similar to the implementation of dependence analysis in Section III-A; the amortized work per region mapped is proportional to the depth of the region forest and independent of the number of tasks or physical instances.

As part of the walk any required copy and reduction operations are issued to construct a valid instance of r. These operations are deferred, waiting to execute on the tasks that produce the data they copy or reduce. Similarly, the start of t's execution is made dependent on completion of the copies/reductions that construct the instance of r it will use.

Figure 3(d) shows part of the mapping of task `update_voltages[0]`; we focus only on the shared nodes. Because the immediately preceding `distribute_charges` tasks all performed the same reduction on the shared and ghost nodes they were allowed to run in parallel. But `update_voltages[0]` needs read/write privilege for `pieces[0].rn_shr`, which forces all of the reductions to be merged back to an instance of the `all-shared` nodes region, from which a new instance of `pieces[0].rn_shr` is copied and used by `update_voltages[0]`.

D. Stage 4: Execution

After a task t has been mapped it enters the execution stage of the SOOP. When all of the operations (other tasks and copies) on which t depends have completed, t is launched on the processor it was mapped to. When t spawns subtasks, they are registered by the SOOP on which t is executing, using t as the parent task and t's region arguments as the roots of the region forest. Each child of t traverses the SOOP pipeline on the same processor as t, possibly being mapped to a SOOP instance on a different processor to execute.

E. Stage 5: Clean-Up

Once task t is done executing its state is reclaimed. Dependence and mapping information is removed from the region tree. The most involved aspect of clean-up is collecting physical instances that are no longer in use, for which we use a distributed reference counting scheme.

IV. Mapping Interface

As mentioned previously, a novel aspect of Legion is the *mapping interface*, which gives programmers control over where tasks run and where region instances are placed, making possible application- or machine-specific mapping decisions that would be difficult for a general-purpose programming system to infer. Furthermore, this interface is invoked at runtime which allows for dynamic mapping decisions based on program input data. We describe the interface (Section IV-A), our base implementation (Section IV-B), and the benefits of creating custom mappers (Section IV-C).

A. The Interface

The mapping interface consists of ten methods that SOOPs call for mapping decisions. A *mapper* implementing these methods has access to a simple interface for inspecting properties of the machine, including a list of processors and their type (e.g. CPU, GPU), a list of memories visible to each processor, and their latencies and bandwidths. For brevity we only discuss the three most important interface calls:

- `select_initial_processor` - For each task t in its mapping queue a SOOP will ask for a processor for t. The mapper can keep the task on the local processor or send it to any other processor in the system.
- `permit_task_steal` - When handling a steal request a SOOP asks which tasks may be stolen. Stealing can be disabled by always returning the empty set.
- `map_task_region` - For each logical region r used by a task, a SOOP asks for a prioritized list of memories where a physical instance of r should be placed. The SOOP provides a list of r's current valid physical instances; the mapper returns a priority list of memories in which the SOOP should attempt to either reuse or create a physical instance of r. Beginning with the first memory, the SOOP uses a current valid instance if one is present. Otherwise, the SOOP attempts to allocate a physical instance and issue copies to retrieve the valid data. If that also fails, the SOOP moves on to the next memory in the list.

The mapping interface has two desirable properties. First, program correctness is unaffected by mapper decisions, which can only impact performance. Regardless of where a mapper places a task or region, the SOOPs schedule tasks and copies in accordance with the privileges and coherence properties specified in the program. Therefore, when writing a Legion application, a programmer can begin by using the default mapper and later improve performance by creating and refining a custom mapper. Second, the mapping interface isolates machine-specific decisions to the mapper. As a result, Legion programs are highly portable. To port a Legion program to a new architecture, a programmer need only implement a new mapper with decisions specific to the new architecture.

B. Default Mapper

To make writing Legion applications easier, we provide a default mapper that can quickly get an application working with moderate performance. The default mapper employs a simple scheme for mapping tasks. When `select_initial_processor` is invoked, the mapper checks the type of processors for which task t has implementations (e.g., GPU). If the fastest implementation is for the local processor the mapper keeps t local, otherwise it sends t to the closest processor of the fastest kind that can run t.

The default mapper employs a Cilk-like algorithm for task stealing [5]. Tasks are kept local whenever possible and only moved when stolen. Unlike Cilk, the default mapper has the information necessary for locality-aware stealing. When `permit_task_steal` is called for a task, the default mapper inspects the logical regions for the task being stolen and marks that other tasks using the same logical regions should be stolen as well.

For calls to `map_task_region`, the default mapper constructs a stack of memories ordered from best-to-worst by bandwidth from the local processor. This stack is then returned as the location of memories to be used for mapping each region. This greedy algorithm works well in common cases, but can cause some regions to be pulled unnecessarily close to the processor, consuming precious fast memory.

C. Custom Mappers

To optimize a Legion program or library, programmers can create one or more custom mappers. Each custom mapper extends the default mapper. A programmer need only override the mapper functions he wishes to customize. Mappers are registered with the runtime and given unique handles. When a task is launched, the programmer specifies the handle for the mapper that should be invoked by the runtime for mapping that particular task.

Supporting custom mappers has two benefits. First, it allows for the composition of Legion applications and Legion libraries each with their own custom mappers. Second, custom mappers can be used to create totally static mappings, mappings that memoize their results, or even totally dynamic mappings for different subsets of tasks in Legion applications. We describe examples of custom mappers in Section V.

Cluster	Sapling	Viz	Keeneland
Nodes	4	10	32 (120)
CPUs/Node	2x Xeon 5680	2x Xeon 5680	2x Xeon 5660
HyperThreading	on	off	off
GPUs/Node	2x Tesla C2070	5x Quadro Q5000	3x Tesla M2090
DRAM/Node	48 GB	24 GB	24 GB
Infiniband	2x QDR	QDR	2x QDR

Fig. 4. System configurations used for the experiments.

V. Experiments

We evaluate the efficiency and scalability of Legion using three applications on three clusters (see Figure 4). All three clusters were Linux-based, and the Legion runtime was built using pthreads for managing CPU threads, CUDA[6] for GPUs, and GASNet[7] for inter-node communication. The RDMA features of GASNet were used to create a globally addressable, but relatively slow, *GASNet memory* that is accessible by all nodes. For each application, multiple problem sizes were used, and each size problem was run on subsets of each machine ranging from the smallest (a single CPU core or GPU) to the largest or near-largest (except Keeneland, where we limited runs to 32 nodes to get sufficient cluster time). By examining performance of the same size problem over progressively larger machines, we measure Legion's strong scaling. By increasing the problem size as well, we also measure weak scaling.

A. Circuit Simulation

The first experiment we investigate is the distributed circuit simulation described in Section II. The Legion SOOP runtime handles all of the resource allocation, scheduling, and data movement across the cluster of GPUs. In particular, Legion's ability to efficiently move the irregularly partitioned shared data around the system while keeping the private nodes and wires resident in each GPU's framebuffer memory is critical to achieving good scalability.

Circuits of two different sizes were simulated. The first had 480K wires, connecting 120K nodes. The second is twice as large, with nearly 1M wires connecting 250K nodes. In addition to running these tests on varying numbers of nodes, the number of GPUs used by the runtime was also varied. In no case did the changes to nodes or number of GPUs per node require changes to the application code.

The circuit simulation has a simple application-specific mapper. At initialization time, the mapper queries the list of GPUs in the machine and identifies each GPU's framebuffer memory and *zero-copy* memory (pinned memory that both the GPUs and CPUs on a node can access directly). Once the circuit is partitioned, the partitions are assigned a home GPU in round-robin fashion. Every task related to that partition is then sent to the home GPU, with no task stealing allowed. (In a well-partitioned circuit, load imbalance is low enough that the cost of moving the private data for a piece from one GPU to another outweighs any benefits.)

The regions for the tasks are mapped as shown in Figure 5. Wires and private node data are kept in each GPU's framebuffer at all times. An instance of the all-shared-nodes

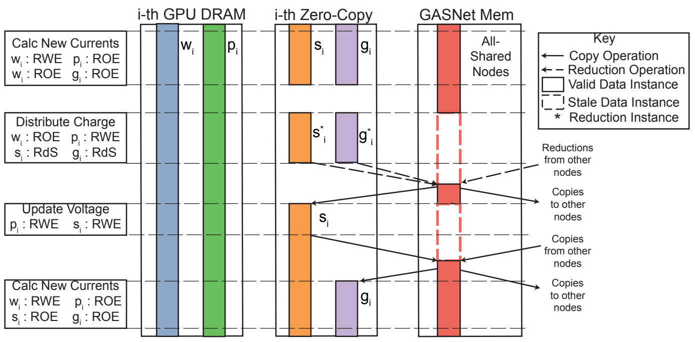


Fig. 5. Tasks and data for the circuit simulation on a cluster of GPUs.

region is placed in GASNet memory and instances for just the shared and ghost nodes needed by each GPU are placed into that GPU's zero-copy memory. This enables the application kernels to access the data as well as the necessary inter-node exchanges of data via the GASNet memory.

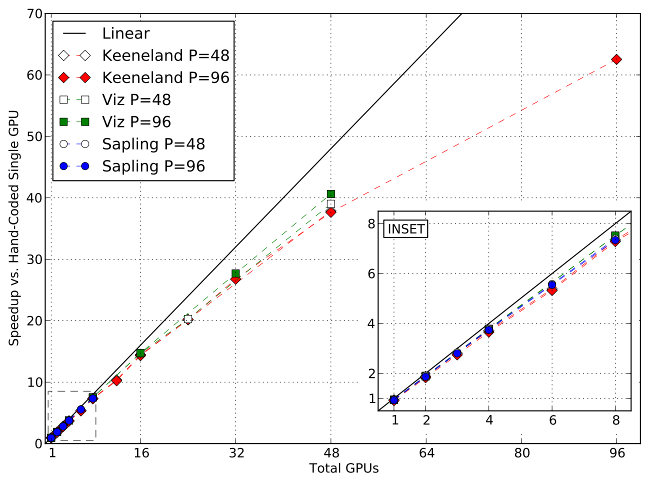
Figure 6(a) shows the performance of the Legion circuit simulation relative to a hand-coded single-GPU implementation written in CUDA. The hand-coded implementation is able to keep the entire simulation state in fast framebuffer memory. Each line shows the scaling of a particular problem size as the number of compute nodes is varied. Our results demonstrate excellent strong scaling, with speedups of 39.0X for the small problem on 48 GPUs and 62.5X for the larger problem size on 96 GPUs. The inset in the graph shows the relative performance for small numbers of GPUs. On a single GPU, our Legion implementation is within 5% of the performance of the hand-coded simulation.

Figure 6(b) shows the fraction of the overall simulation time (summed over all nodes) spent in the application kernels compared to the various pieces of the Legion SOOP runtime. As the node count increases, the non-communication overhead stays relatively constant. As expected, the communication overhead grows quadratically with the number of nodes.

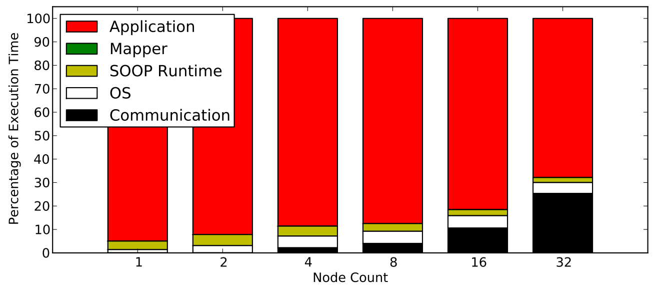
B. Particle Simulation

Our second experiment is a port of the *fluidanimate* benchmark from the PARSEC benchmark suite[8], which does a particle-based simulation of an incompressible fluid. Each particle interacts only with nearby particles. The benchmark divides the space in which the fluid can move into a three-dimensional array of cells such that the range of interaction is limited to just the cells adjacent (including diagonals) to the one a particle resides in. The application divides the array into *grids* and assigns each grid to a thread. Per-particle locking safely accesses particles in cells that lie on the edge of a grid. This fine-grained locking scheme along with the assumption of a shared address space gives good scaling in a multicore processor, but prohibits running beyond a single node.

To extend the scaling further, our port uses region partitioning to divide the array into grids in a similar way, but avoids relying on shared memory to handle interactions between grids. Instead, the Legion version creates explicit ghost copies of cells on grid boundaries and uses those ghost cells to exchange information between the grids.



(a) Circuit simulation speed relative to single-GPU implementation.



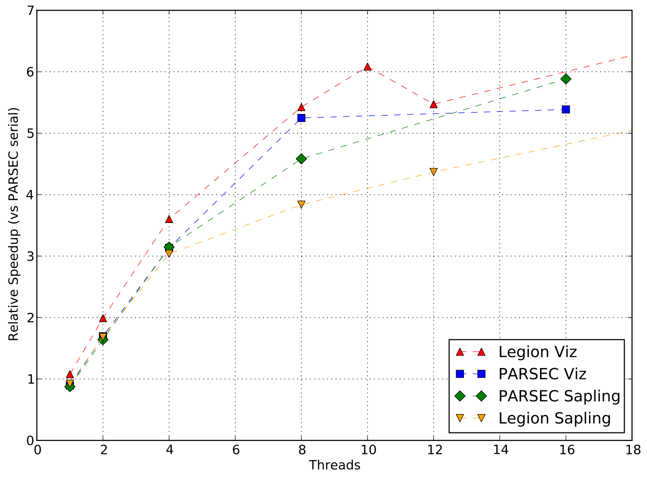
(b) Overhead of circuit simulation on Keeneland with 3 GPUs/node.

Fig. 6. Circuit simulation results.

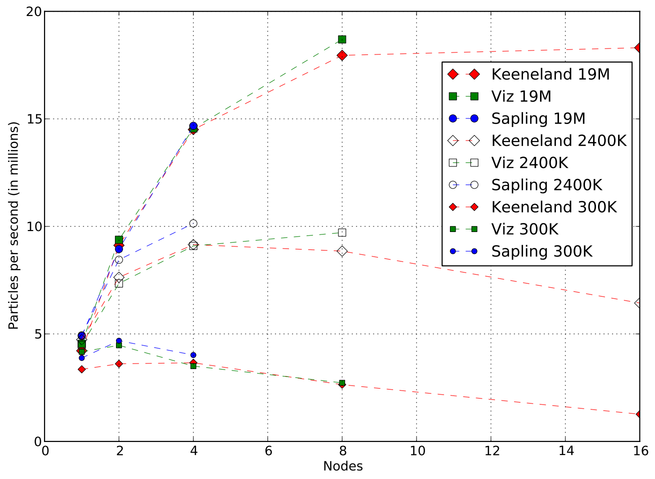
The particle simulation's mapper is very simple: it maps one grid's tasks onto each processor and maps all that grid's regions (both internal and ghost regions) into that processor's system memory. The exchange of ghost cell data between processors is handled by the Legion runtime as a ghost cell region is alternately mapped to two different memories.

Figure 7(a) compares the performance of the Legion implementation against the PARSEC version, using a relatively small problem (300K particles on a 15x21x15 array of cells). Speedups for both the Legion and threaded PARSEC implementations are measured relative to PARSEC's serial version, which eliminates all locking operations. Between 1 and 4 threads, the PARSEC and Legion results are nearly indistinguishable, indicating neither the Legion runtime nor the restructuring of the implementation to allow multi-node scaling impose any significant overhead. At 8 threads and above, performance begins to vary. Both the Legion and PARSEC versions on Viz flatten out as they over-subscribe the 12 physical cores. On Sapling, which has HyperThreading enabled, deviations from linear begin sooner as the operating system's thread placement choices begin to matter.

To measure scaling beyond a single node, three different problem sizes were run on each of the three systems. The results are presented in Figure 7(b). For the smallest problem (300K particles), we observe a 20% speedup from 1 to 2 nodes (16 threads total), but slow down beyond that due to communication overhead; at 4 nodes there are twice as many



(a) Single-node particle simulation speed.



(b) Multi-node scaling.

Fig. 7. Fluid simulation results.

ghost cells as interior grid cells. The larger problem sizes (2.4M and 19M particles) perform much better, with scaling of up to 5.4x when going from 1 node to 16 because of a lower communication-to-computation ratio.

C. Adaptive Mesh Refinement

Our final application is based on the third heat equation example from the Berkeley Labs BoxLib project [9]. This application is a three-level adaptive-mesh-refinement (AMR) code that computes a first order stencil on a 2D mesh of cells. Updating the simulation for one time step consists of three phases. In the first phase, the boundary cells around a box at a refined level linearly interpolate their values from the nearby cells at the next coarser level. The second phase performs the stencil computation on each cell in every level. In the third phase, cells at a coarser level that have been refined are restricted to the average of the cells that they physically contain at the next finest level of refinement.

Achieving high-performance on this application is particularly challenging for several reasons. First, the application has a very high communication-to-computation ratio which, for a fixed problem size, begins as being memory bound and, with increasing node count, becomes network bound as the perimeter-to-area ratio of cell grids increases. Second, when choosing how to partition cells into grids, the programmer must consider the locality between cells within a level as well as across levels. For cross-level cell dependences, optimal mapping decisions can only be made at runtime as the location of refinements are dynamically determined. Finally, this application has parallelism both between tasks running at the same level and tasks running across levels, leading to complicated input-dependent data dependences.

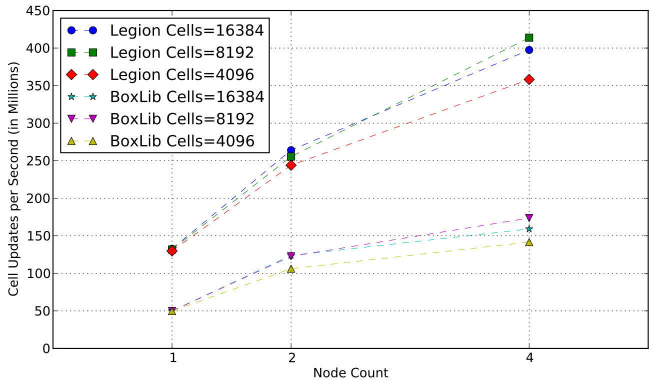
BoxLib's implementation partitions cells within a level into a number of grids based on the number of nodes in the machine and distributes one grid from each level to each node. This optimizes for memory bandwidth and load balance, but does not exploit cross-level locality between grids from different levels of refinement. Furthermore, BoxLib does not block grids into sub-grids to take advantage of intra-grid locality.

Our Legion implementation performs two optimizations that allow us to outperform BoxLib. First, for each level of refinement we recursively partition the logical region of cells based on the number of nodes in the machine and the sizes of the L2 and L3 caches. Our second optimization takes advantage of the cross-level locality. We wrote an application-specific mapper that dynamically discovers relationships between grids at different levels of refinement. The mapper dynamically performs intersection tests between logical regions containing grids of different refinement levels. If the mapper discovers overlaps between grids from different levels, the mapper places them on the same node in the machine. The mapper memoizes the intersection tests to amortize their cost. The mapper also dynamically load balances by distributing unconstrained grids from the coarsest level onto under-loaded nodes.

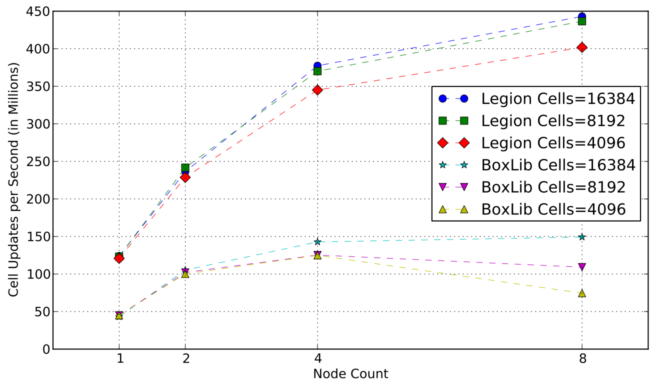
We compared our Legion implementation against BoxLib on three different problem sizes with a fixed number of cells per level of refinement, but with randomly chosen refinement locations. BoxLib also supports OpenMP and we took their best performance from using 1, 2, 4, or 8 threads per node. Our Legion implementation always uses one thread per node to illustrate that in this application locality is significantly more important than fine-grained data-parallelism.

Figure 8 gives the results. On just one node, blocking for caches using Legion achieves up to 2.6X speedup over BoxLib. As node count increases, the mapper's ability to exploit cross-level locality further increases the performance advantage to 5.4X by reducing the total communication costs.

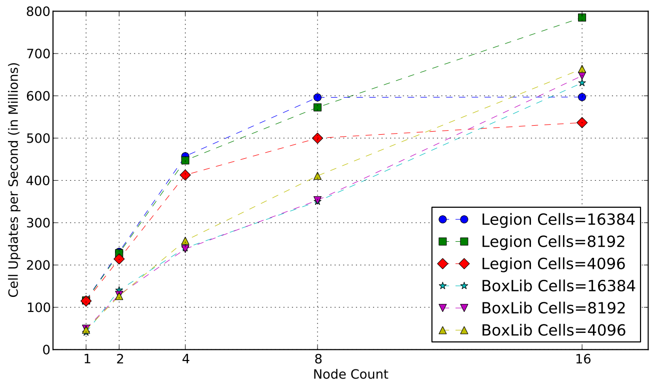
As the node count increases the AMR code becomes highly dependent on interconnect performance. BoxLib performs much better on Keeneland than on Viz due to the better interconnect. At higher node counts BoxLib begins to catch up (see Figure 8(c)) because our application's intra-level ghost-cell exchange algorithm uses GASNet memory to communicate ghost cells, requiring a linear increase in network traffic with the number of nodes. BoxLib uses direct node-to-node exchanges of ghost cells, similar to our fluid application. A future implementation of our AMR code will employ a similar ghost cell exchange algorithm to improve scalability.



(a) Sapling results.



(b) Viz results.



(c) Keeneland results.

Fig. 8. Throughput of adaptive mesh refinement code.

VI. Related Work

Legion began as an outgrowth of Sequoia, a locality-aware programming language capable of expressing computations in deep memory hierarchies [1]. Sequoia is a special case of the Legion programming model in which only arrays can be recursively partitioned, all access is exclusive, there is a static mapping of tasks and data (though extensions to Sequoia make this mapping more dynamic [10]) and, most fundamentally, the decomposition of tasks and the decomposition of data is one-to-one. Legion generalizes the Sequoia model by allowing for dynamic partitioning of pointer data structures through regions, enabling dynamic mappings through the mapper interface, and allowing different coherence properties. Legion's decoupling of the task tree from the region tree leads directly to the scheduling problem solved by our software out-of-order processor for tasks with region arguments.

The SSMP programming model is the most similar work to Legion [11]. Like Legion, SSMP supports dynamic detection of dependences between tasks based on data requirements. However, SSMP only supports a single disjoint rectilinear partition of an array, unlike Legion which supports multiple arbitrary partitions of regions. Furthermore, the SSMP runtime must perform dependence checks between every pair of tasks created in the system. Legion's programming model only requires dependence checks between tasks with the same parent task which enables scalable nested parallelism in distributed machines. SSMP only operates on shared memory machines.

Chapel has several concepts to support the expression of locality [12]. Domains are similar to logical regions in that they describe maps from indexes to objects. Domains can create sub-domains by slicing the index sets from a parent domain. Domains are a higher level concept than regions; the domain index sets support dimensionality and iterators, whereas logical regions can only be accessed by pointers. Also, the act of creating subdomains in Chapel does not track disjointness information, making it more challenging for the Chapel compiler or runtime to infer task independence.

In addition to domains, Chapel also supports the notion of domain maps and locales to enable the programmer to efficiently map domains onto hardware [13]. Locales are a flat array of abstract locations. Programmers can use locales by writing domain maps that specify how domains are subdivided and assigned to locales. Domain maps provide the same functionality as partitions and mappers in Legion, but require the user to correctly implement domain maps for the program to be correct. Legion explicitly isolates correctness from performance by defining the Mapper interface. In addition, Chapel's flat array of locales makes it challenging to fully utilize deep memory hierarchies. Chapel currently supports clusters and GPUs in isolation [14], but we are not aware of any results that make use of both.

X10 is another parallel programming language designed to operate on distributed memory machines [15]. X10's *places* enable programmers to talk about where to place both data and tasks. However, once data and tasks have been placed they are fixed, which mandates that data movement be explicitly managed by user level code or implicitly by the compiler [16]. Recently X10 has introduced regions into the compiler's intermediate representation [17]. Unlike Legion, regions in X10 are not visible to the programmer but are inferred from high level arrays through static analysis. X10 provides support for clusters of GPUs [18], but requires the programmer to write all code managing data movement through both the cluster and GPU memory hierarchies.

Deterministic Parallel Java (DPJ) is a parallel extension of Java that, like Legion, uses regions to express locality, but does static dependence analysis on region arguments to functions to find dependences[19]. The primary goal of DPJ is to provide a programming model that guarantees determinism while also supporting parallelism. As a result, the DPJ programming model is more restrictive than Legion. DPJ only supports a static form of Legion's exclusive and atomic coherence modes which mandates the same coherence for the lifetime of a region. DPJ can express the populate-then-partition style of using regions, but not the partition-then-populate. Overall, DPJ is more static, and thus provides more guarantees and less flexibility than Legion. DPJ efforts have so far focused on JVM implementations on shared memory machines.

There have also been several other efforts that use partitions to either avoid or detect conflicts in shared memory programs dynamically. Object assemblies are a mechanism for partitioning a shared memory heap to enable parallel execution [20]. Object assemblies only support a disjoint partitioning of the heap unlike Legion which allows for multiple partitions of any data structure. Legion further uses the information from partitions to operate on distributed architectures.

Galois is a programming system based on optimistic parallelism that dynamically detects memory conflicts between concurrent threads at the very fine granularity of object accesses. To achieve reasonable performance Galois supports a partitioning interface for breaking up data structures [21]. Galois relies on shared memory and only leverages partitioning for coarser locking and conflict detection. Several other recent efforts have proposed task schedulers based on dynamic detection of memory conflicts, but these also have assumed underlying shared memory hardware [22], [23], [24].

SPMD languages such as Titanium [25] and UPC [3] have mechanisms for describing array partitions in distributed memories. However, the partition operations supported only operate on two-level memory hierarchies consisting of local and global memory. Part of Legion's low-level runtime system is constructed using UPC's GASNet runtime system [7].

In previous programming systems regions have primarily been used as a construct for describing memory management schemes [26], [27] or for enforcing safety policies [28]. We follow [27] in Legion's decision to make regions first class. In these works, however, regions have memory layout implications. Logical regions in Legion enable the programmer to describe locality independent of memory layout.

VII. Conclusion

We have presented Legion, a programming model for expressing the abstract locality and independence properties of program data by using logical regions. The Legion programming model enables a distributed Software-Out-of-Order Processor (SOOP) scheduling algorithm. The SOOP algorithm leverages the locality and independence properties captured by logical regions to efficiently execute programs. We also described a novel mapping interface that guides the SOOP execution and allows tuning of program performance independent of correctness. We demonstrated that for three benchmarks our SOOP runtime enabled Legion implementations that were faster than existing codes and that the SOOP was able to scale on distributed, heterogeneous machines.

Acknowledgments

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. Michael Bauer was supported by an NVIDIA Graduate Research Fellowship.

References

[1] K. Fatahalian *et al.*, "Sequoia: Programming the Memory Hierarchy," in *Supercomputing*, November 2006.

[2] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The Cascade high productivity language," in *High-Level Parallel Programming Models and Supportive Environments*, 2004, pp. 52–60.

[3] 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.

[4] J. Vetter *et al.*, "Keeneland: Bringing heterogeneous gpu computing to the computational science community," *Comp. in Science Eng.*, 2011.

[5] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Symposium on Principles and Practice of Parallel Programming*, 1995.

[6] "Cuda programming guide 4.1," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, January 2012.

[7] K. Yelick *et al.*, "Productivity and performance using partitioned global address space languages," in *PASCO*, 2007, pp. 24–32.

[8] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[9] A. N. M. Lijewski and J. Bell, "Boxlib," https://ccse.lbl.gov/BoxLib/index.html, 2011.

[10] M. Bauer, J. Clark, E. Schkufza, and A. Aiken, "Programming the memory hierarchy revisited: Supporting irregular parallelism in Sequoia," in *PPoPP*, 2011, pp. 13–24.

[11] J. M. Perez, R. M. Badia, and J. Labarta, "Handling task dependencies under strided and aliased references," in *ICS*, 2010, pp. 263–274.

[12] B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int'l Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, August 2007.

[13] B. Chamberlain, S. Choi, S. Deitz, D. Iten, and V. Litvinov, "Authoring User-Defined Domain Maps in Chapel," 2011.

[14] A. Sidelnik *et al.*, "Using the High Productivity Language Chapel to Target GPGPU Architectures," 2011.

[15] P. Charles *et al.*, "X10: An object-oriented approach to non-uniform cluster computing," in *OOPSLA*, 2005, pp. 519–538.

[16] S. Chandra *et al.*, "Type inference for locality analysis of distributed data structures," in *PPoPP*, 2008, pp. 11–22.

[17] M. Joyner, Z. Budimlic, and V. Sarkar, "Subregion analysis and bounds check elimination for high level arrays," in *Compiler Construction*, 2011.

[18] "X10 2.1 cuda support," x10.codehaus.org/X10+2.1+CUDA, 2011.

[19] R. Bocchino, V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian, "A type and effect system for deterministic parallel Java," in *OOPSLA*, 2009, pp. 97–116.

[20] R. Lublinerman, S. Chaudhuri, and P. Cerny, "Parallel programming with object assemblies," in *OOPSLA*, 2009, pp. 61–80.

[21] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Optimistic parallelism benefits from data partitioning," in *ASPLOS*, 2008, pp. 233–243.

[22] H. Vandierendonch, G. Tzenakis, and D. Nikolopoulos, "A unified scheduler for recursive and task dataflow parallelism," in *PACT*, 2011.

[23] G. Tzenakis *et al.*, "BDDT: Block-level dynamic dependence analysis for deterministic task-based parallelism," in *PPoPP*, 2012, pp. 301–302.

[24] Y. Eom, S. Yang, J. Jenista, and B. Demsky, "DOJ: Dynamically parallelizing object-oriented programs," in *PPoPP*, 2012.

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

[26] E. D. Berger, B. G. Zorn, and K. S. McKinley, "Reconsidering custom memory allocation," in *OOPSLA*, 2002, pp. 1–12.

[27] D. Gay and A. Aiken, "Language support for regions," in *PLDI*, 2001, pp. 70–80.

[28] D. Grossman *et al.*, "Formal type soundness for Cyclones region system," Tech. Rep., 2001.