Dependent Partitioning

Sean Treichler
Stanford University
sjt@cs.stanford.edu

Michael Bauer
NVIDIA Research
mbauer@nvidia.com

Rahul Sharma
Stanford University
sharmar@cs.stanford.edu

Elliott Slaughter
Stanford University
slaughter@cs.stanford.edu

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Abstract

A key problem in parallel programming is how data is *partitioned*: divided into subsets that can be operated on in parallel and, in distributed memory machines, spread across multiple address spaces.

We present a *dependent partitioning* framework that allows an application to concisely describe relationships between partitions. Applications first establish *independent partitions*, which may contain arbitrary subsets of application data, permitting the expression of arbitrary application-specific data distributions. *Dependent partitions* are then derived from these using the *dependent partitioning operations* provided by the framework. By directly capturing inter-partition relationships, our framework can soundly and precisely reason about programs to perform important program analyses crucial to ensuring correctness and achieving good performance. As an example of the reasoning made possible, we present a static analysis that discharges most consistency checks on partitioned data during compilation.

We describe an implementation of our framework within Regent, a language designed for the Legion programming model. The use of dependent partitioning constructs results in a 86-96% decrease in the lines of code required to describe the partitioning, eliminates many of the expensive dynamic checks required for soundness by the current Regent partitioning implementation, and speeds up the computation of partitions by 2.6-12.7X even on a single thread. Additionally, we show that a distributed implementation incorporated into the the Legion runtime system allows partitioning of data sets that are too large to fit on a single node and yields a further 29X speedup of partitioning operations on 64 nodes.

OOPSLA '16, October 30-November 04 2016, Amsterdam, Netherlands
Copyright © 2016 held by owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4444-9/16/10. . . $15.00
DOI: http://dx.doi.org/10.1145/2983990.2984016

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.4 [*Software Engineering*]: Program Verification; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Data partitioning; distributed memory; parallel execution; program verification; Legion; Regent

1. Introduction

The partitioning of the data used by a parallel application is critical for both performance and scalability. Independent subsets of data can be processed concurrently and the right placement of data within the memory hierarchy minimizes data movement. Every general-purpose parallel programming model must confront the tension between a programmer's desire for expressivity (i.e. the ability to describe exactly the best partitioning for a given application) and the compiler and runtime's need for tractability (i.e. the ability to perform analysis and optimizations). [1]

As an example, consider a distributed computation on a graph-based data structure, such as the simulation of an electric circuit. On today's supercomputers, such computations are generally limited by data movement over the network between the systems in the cluster. The resulting importance of the distribution of data (the nodes and edges of the circuit's graph) to different processes (*ranks*) is such that it is worth the expense of computing a near-optimal partitioning of the data with a graph partitioning tool such as PARMETIS[22] before running a long simulation. (The hashing or greedy clustering approaches used by graph processing frameworks such as Pregel[24] and PowerGraph[19] focus instead on computing "good enough" partitions quickly.)

In addition to an optimal assignment of graph nodes to "owner" ranks, data movement is further minimized by computing a partition of the circuit graph's edges, as well as

[1] Domain-specific programming models are often able to compute adequate partitions themselves (e.g. for computations on graphs[24, 28]), but acceptable solutions for arbitrary data structures and arbitrary computations over them are elusive.

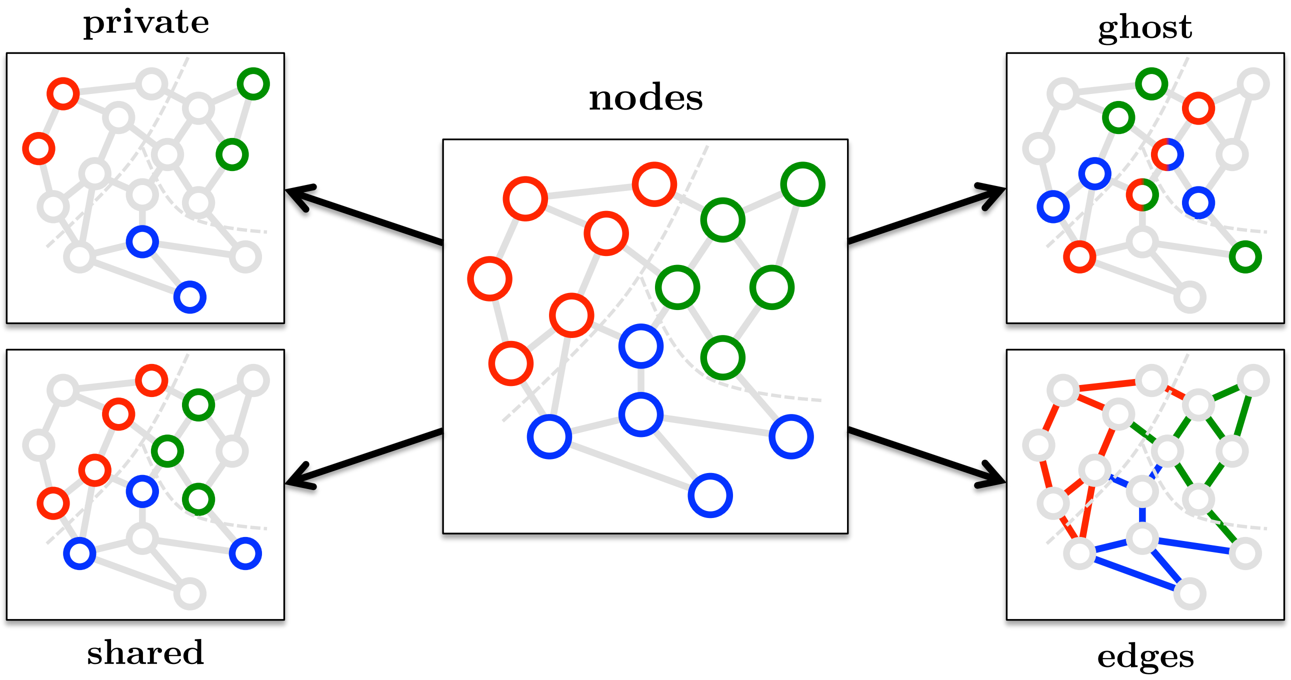


Figure 1: Partitioning of a graph-based computation. The *independent* partition is in the center, and four *dependent* partitions are derived from it.

computing whether nodes are "private" (i.e. only accessed by the owner rank) or "shared", and which nodes are "ghost" nodes for each rank (i.e. accessed by that rank but owned by another). Figure 1 summarizes the various partitions used for this computation. These additional partitions may not be chosen freely, because the application code will in general be written with assumptions about how the rest of the data is organized relative to the original partition of the nodes. For example, a step in the application that iterates over the edges owned by a rank might require that all the nodes on which these edges are incident be in the private, shared, or ghost partitions for that rank.

Existing programming models generally provide abstractions for partitioning that are at one of the two extremes on the tractability vs. expressivity spectrum, both of which have significant drawbacks. One programming model that favors tractability is Chapel[8], which provides a small set of primitive *domain maps* such as *blocked*, *cyclic*, or *replicated* distributions. These domain maps concisely describe common data distribution patterns for structured data (e.g. computations on dense arrays or grids). The primitives are easy for the programmer to specify, and the Chapel compiler and runtime are able to efficiently analyze how two or more of these domain maps interact. However, the expressivity of such a system is limited. For computations on irregular data structures such as the circuit graph above, the nodes and edges must be manually ordered into arrays (which often requires the duplication of ghost nodes) so that they may then be distributed using the available domain map primitives. The programmer is solely responsible for maintaining and testing this mapping — neither the Chapel compiler nor the runtime can help in guaranteeing correctness.

In contrast, the Legion[5] programming model allows a *logical region* to be partitioned into any computable subsets, thereby maximizing the programmer's ability to express partitioning schemes. This generality is achieved by allowing arbitrary code (including external libraries) to be used for partitioning, with only the results of the execution of the code being provided to the Legion runtime as a *coloring*. The Legion runtime is able to perform some consistency checks on the resulting partitions, but only at runtime — the use of arbitrary code to compute partitions rules out most attempts at static analysis. Further, the use of colorings for even simple partitions prevents the Legion runtime from optimizing the computation or distributing it across multiple ranks, limiting the size of problem that can be tackled.

Our *dependent partitioning* framework is a new, middle ground between these two extremes. We split the partitions used by an application into two types. *Independent partitions* are those that are computed from "scratch" (e.g. the central node ownership map in Figure 1) and preserve the expressivity benefit of a Legion-like approach, using a *filtering* operation that is similar to Legion's colorings, but can work on distributed data sets, allowing the partitioning performance to scale with the available computing resources (yielding up to a 29X speedup in our experiments for partitioning operations on 64 nodes compared to a single node) and enabling the partitioning of data sets that do not fit on a single node.

```
pgrm := stmt+
stmt := idx id = idxexpr;
      | field id : idxtype→rngtype;
      | function id : basetype→basetype;
      | property propstmt;
      | val id : basetype;
      | for id in idxexpr { stmt+ }
      | immutable id [, id+] { stmt+ }
      | assert [ propstmt ⇒] asrt;
asrt := idxexpr ≤ idxexpr;
      | idxexpr ∗ idxexpr;

basetype := int32 | float32 | ptr | bool | . . .
          | ⟨basetype, basetype⟩
idxtype  := id
rngtype  := basetype | idxtype[+]
idxexpr  := ispace( basetype [, constexpr, constexpr])
          | idxexpr{ id | id relop constexpr }
          | idxexpr{ id | id→id relop constexpr }
          | idxexpr→id
          | idxexpr←id
          | idxexpr & idxexpr
          | idxexpr | idxexpr
          | idxexpr - idxexpr

propstmt := propexpr relop propexpr
          | propstmt ∧ propstmt
          | propstmt ∨ propstmt
propexpr := id | id(id) | constexpr
          | propexpr + propexpr
          | propexpr - propexpr
          | propexpr * constexpr
          | propexpr / constexpr
          | propexpr % constexpr
relop    := = | ≠
          | < | ≤
          | > | ≥
```

Figure 2: Dependent Partitioning Language grammar

Dependent partitions are the second type, and are those derived from independent partitions and/or other dependent partitions (e.g. the four smaller partitions in Figure 1). The computation of these is based on a small set of *dependent partitioning operations* that are similar in style to Chapel's domain maps. They are easy to use, resulting in dramatic (86-96%) reductions in the amount of application code needed to compute dependent partitions and can be optimized by the runtime, resulting in 2.6-12.7X speedups of these partitioning operations on a single thread. Dependent partitioning operations (described in more detail in Section 2) are based on set operations and reachability via pointers within the application data, and are carefully chosen to provide the tools needed by these applications while preserving the ability to perform static analysis, allowing most consistency checks between the various partitions to be discharged during compilation.

We have designed the dependent partitioning framework to be general enough to be implemented in any programming model that includes the concepts of stable "names" for objects (e.g. pointers), sets and subsets of those names, and fields defined for each object in a set. We evaluate our framework with an implementation in the Regent language[31] and the Legion runtime[5], and borrow their terminology in many cases, but a natural mapping exists as well to Chapel *domains* and *arrays* defined over those domains.

The rest of this paper is organized as follows. The majority of the paper focuses on developing concepts and algorithms for a data partitioning sublanguage. We begin by describing an abstract Dependent Partitioning Language (DPL) in Section 2 and give several examples to demonstrate its expressivity. Section 3 covers the static analysis required to discharge most consistency checks at compile time and discusses when and why completeness is sacrificed. The final parts of the paper describe the integration of DPL into a full programming language for parallel computation. We describe the changes made to the Regent compiler (Section 4) and the Legion runtime (Section 5) to implement the DPL framework. Section 6 provides our experimental results and Section 7 places our effort in the context of related work.

2. Dependent Partitioning Language

Figure 2 defines the syntax of an abstract Dependent Partitioning Language (DPL) that we use to describe our framework. DPL is a sublanguage that includes only the features needed for creating, manipulating and automatically reasoning about data partitions. Any full programming language incorporating DPL inherits DPL's capabilities but would also need additional standard constructs (e.g., while loops, conditionals, etc.) to carry out computations on the data itself.

DPL is an imperative language in which *index spaces* (i.e. sets of indices) are computed and tested against constraints. Program data in DPL is stored in *fields*, which associate a value of a specified *range type* with each index in the *domain* index space. The range type may be either a base type or another index space — in the latter case it may be *null-extended* to allow null pointers (which are not a part of any index space) to be stored. The value associated with an index in a field may change during a program's execution, but at any given point in time a field represents a mathematical function from its index space to the values associated with each index. Dually, we can view an index space as a collection of objects of a given type, fields as member variables of the type, and an index i as the identity of an object whose field $i{\rightarrow}f$ is stored in $f[i]$.

The actual contents of any field are unimportant in DPL, as we wish the consistency checks to be valid for all possible inputs. To simplify the presentation, we omit constructs for updating fields and instead focus on scopes in which one or more fields are *immutable*, in which case multiple computations based on those fields' contents are guaranteed to have used the same values. Outside any of its immutable scopes, a field is conservatively assumed to change arbitrarily between every reference.

An implementation of the circuit partitioning example from Section 1 is given in Figure 3. The code defines an assignment of circuit nodes to subcircuits and then uses the graph topology to define the necessary subsets of nodes or wires (i.e. edges) and verify their internal consistency. The initial `nodes` and `wires` index spaces are declared (lines 1-2) using the opaque `ptr` base type. Index space declarations

```
1  idx nodes = ispace(ptr);
2  idx wires = ispace(ptr);
3  field subcircuit_id : nodes → int32;
4  field in_node : wires → nodes+;
5  field out_node : wires → nodes+;
6
7  val N : int;
8  idx partitions = ispace(int, 0, N);
9
10 immutable subcircuit_id, in_node, out_node {
11   for p in partitions {
12     idx owned_nodes = nodes{ n | n→subcircuit_id = p };
13     idx owned_wires = owned_nodes ← in_node;
14     idx cross_wires = wires{ w | w→in_node→subcircuit_id !=
15                                  w→out_node→subcircuit_id };
16     idx all_shared = cross_wires → out_node;
17     idx my_private = owned_nodes − all_shared;
18     idx my_shared = owned_nodes & all_shared;
19     idx my_ghost = (owned_wires → out_node) − owned_nodes;
20
21     assert (owned_wires → in_node) ≤ (my_private | my_shared);
22     assert (owned_wires → out_node) ≤ (my_private | my_shared |
23                                        my_ghost);
24     assert my_private ∗ my_shared;
25   }
26 }
```

Figure 3: Circuit partitioning in DPL

are lexically scoped and once declared, an index space is immutable. Lines 3-5 declare the fields we need for this example: an integer `subcircuit_id` for each node and two node pointers (`in_node` and `out_node`) for each wire. Note that fields are given function types *indexspace* → *range* (the + qualifier permits null values).

DPL also includes scalar values, which capture runtime constants (line 7). A limited form of scalar variable is available for iteration over the indices in an index space (line 11), but a loop variable may not be modified in the loop body.

Line 10 declares the fields locally immutable so that we can compute index spaces from them (lines 12-19) and check their consistency (lines 21-24). Line 12 uses the (opaque) contents of the `subcircuit_id` field to *filter* the original `nodes` index space down to the subset of nodes whose subcircuit ID match the loop variable `i`. As mentioned above, `n → subcircuit_id` can be thought of as a field lookup of node `n`, or dually, as a function application `subcircuit_id(n)`. Filter operations use a syntax similar to set comprehensions, with the domain of the variable being limited to the index space to which the filter operation is being applied (i.e. `nodes` in this case). A filter operation may be classified as *simple* or *complex* based on the predicate used. A simple filter's predicate may either test the variable directly against a constant or use the variable to perform a field lookup and test that value against a constant.

Filtering is used to describe the independent partitions used in an application. The contents of the `subcircuit_id` field may be read from an input file or they may be the result of an online graph clustering algorithm. DPL places no constraints on the form or result of that computation — it simply captures how index spaces are derived from it.

Line 13 shows the use of arbitrary application data to derive a dependent partition. We want every wire to belong to the same subcircuit as its `in_node`, and the expression `owned_nodes←in_node` computes the set of edges e where `in_node`$(e) \in$ `owned_nodes`. Put another way, it is the *preimage* of `owned_node` under the function `in_node`. We use preimage (and image) in the standard mathematical sense: for any function $f : X \rightarrow Y$, we define the image $f^{\rightarrow} : 2^X \rightarrow 2^Y$ such that $f^{\rightarrow}(S) = \{f(x) \mid x \in S\}$ and the preimage $f^{\leftarrow} : 2^Y \rightarrow 2^X$ such that $f^{\leftarrow}(S) = \{x \mid f(x) \in S\}$. Although it remains uninterpreted, the knowledge that the current value of `in_node` is a mathematical function allows DPL to learn constraints on how `owned_nodes` and `owned_wires` may relate.

Line 14 uses another filter operation to compute the set of wires that cross between subcircuits (`cross_wires`), but uses a *complex filter*. The predicate for a complex filter may include multiple field lookups on either side of the (in)equality. Although these complex predicates are critical for expressing partitions in many cases, they can be problematic for static analysis, as we discuss in Section 3.

The counterpart to line 13's preimage operation is the *image* operation on line 16. We again treat a field (`out_node` this time) as a function and compute the image of the `cross_wires` index space to determine the set of nodes that are shared (i.e. accessed by another subcircuit).

The final way in which index spaces can be derived from others is by standard set union (`|`), intersection (`&`), and difference (`-`) operations. Lines 17 and 18 split the `owned_nodes` into the `my_private` and `my_shared` subsets based on whether any other subcircuit will access them. The ghost nodes for a given subcircuit can be determined by computing the image of just its `owned_wires` through the `out_node` field and subtracting out any nodes that are owned. There is no need to do the same for the `in_node` references, but we will test that assertion shortly.

The consistency requirements between different partitions are expressed using the `assert` keyword, which can be used to confirm the containment of one index space in another (`<=`) or the disjointness of two index spaces (`*`). Requirements are often expressed by computing additional dependent partitions and comparing them instead. Lines 21 and 22 use the image operation again to compute the nodes that can be directly reached from a wire in `owned_wires` and demand that they fall into that subcircuit's private, shared, or ghost nodes. (The omission of `my_ghost` on line 21 allows us to verify our claim above that the image through `in_node` was unnecessary.) This example's assertions are easily verified by hand, but an automated approach is obviously desirable and is discussed in Section 3.

2.1 Integer Arithmetic

The circuit example uses unstructured data, but it is equally important to capture structured (and semi-structured) cases in which partitions are defined in whole or in part by operations on the indices themselves. An example of this is red-black Gauss-Seidel iteration, in which the convergence of an iterative algorithm can be improved by separating elements into two sets (red and black) and alternately updating one based on the other. Figure 4 shows a DPL program that implements this partitioning.

The operations performed on indices are generally pure functions involving fairly simple arithmetic but commonly require the ability to force computed values into an interval, either by clamping the out-of-range values or by wrapping them around to the other end of the interval. Adding a whole sub-language for these index-based computations would complicate DPL considerably. Instead, DPL treats them very similarly to fields (i.e. as uninterpreted functions), but allows *properties* to be provided that can partially (or fully) specify the behavior of the function. The actual function is used at runtime to compute images and preimages, but DPL's static analysis is based entirely on the properties, and any verification results from that analysis will hold for any variant of the function that still satisfies the given properties.

The restrictions on what a property may describe are similar to those of filter predicates. They may refer to the inputs and outputs of a single function and any scalar values or loop variables in scope. They may perform comparisons of integer and floating-point values. They may also include linear integer arithmetic: addition and subtraction, as well as multiplication, division and remainder by integer constants. Examples can be seen on lines 4, 7, 11, and 14 of Figure 4.

Once declared, a function may be used in place of a field in any operation. A simple filter may contain a single function evaluation instead of a field lookup, but any combination of the two results in a complex filter operation.

Once these functions and their properties are defined, the set of elements that will be updated as part of a "red" iteration are computed on lines 17 and 22, and in turn the elements they will access are computed on lines 18 and 23. Line 26 asserts that these red blocks can be operated on in parallel and uses the ability to place conditions on when the assertion must hold.

```
1  val N : int;
2  idx elems = ispace(int32, 0, N);
3  function left : int32 → int32; −− subtract 1 and clamp to 0
4  property (left(x) ≥ 0) ∧ (left(x) ≥ (x−1)) ∧ (left(x) ≤ x);
5
6  function right : int32 → int32; −− add 1 and clamp to N−1
7  property (right(x) < N) ∧ (right(x) ≤ (x+1)) ∧ (right(x) ≥ x);
8
9  val B : int;
10 function blockid : elems → int;
11 property blockid(x) = x / (2∗B);
12
13 function isred : elems → int;
14 property isred(x) = (((x / B) % 2) = 0);
15
16 for i in ispace(int, 0, N / (2∗B)) {
17   idx red_update_i = elems{ x | x→blockid = i ∧ x→isred };
18   idx red_access_i = red_update_i |
19                         red_update_i→left | red_update_i→right;
20
21   for j in ispace(int, 0, N / (2∗B)) {
22     idx red_update_j = elems{ x | x→blockid = j ∧ x→isred };
23     idx red_access_j = red_update_j |
24                           red_update_j→left | red_update_j→right;
25
26     assert (i != j) ∧ (B > 0) ⇒ (red_access_i ∗ red_update_j);
27   }
28 }
```

Figure 4: Red-black Gauss-Seidel partitioning in DPL

2.2 Limits on Expressiveness

When designing a language with static analysis in mind, deciding what to exclude is important. We discuss a few conscious omissions: the immutability of index spaces, the limitation to linear integer arithmetic, and the question of transitive reachability.

Although DPL must necessarily include loops to describe interesting partitions and applications with iterative computations, the immutability and lexical scoping of index spaces ensure that the fragment of a DPL program that feeds into a given partitioning operation or consistency check is loop-free. Our analysis in the next section depends on this loop-freedom. A given iteration of a loop may not refer to index spaces computed in the previous iteration. Information may still flow from previous iterations (e.g. for incremental repartitioning), but it must flow through fields.

Second, the inability to express properties of functions that perform general integer multiplication or any kind of arithmetic on floating-point values is again necessary for tractable analysis. We also argue that such operations rarely appear in partitioning-related computations. In particular, the linear integer arithmetic contained in DPL is sufficient to precisely describe the standard domain maps in Chapel.

Finally, image and preimage operations compute reachability, but only allow a fixed number of steps that can be statically determined. Although transitive reachability (i.e. a fixed-point operator) is a very desirable property to describe, static analysis of it is intractable in most cases. Worse, the time complexity of a generic dynamic analysis used as a fallback is quadratic in the size of the index space. By excluding transitive reachability from DPL, the programmer is left with two choices:

1. Performing the fixed-point operation manually and supplying the result in a field, or
2. Finding an equivalent way to express the desired constraint using the available partitioning operations.

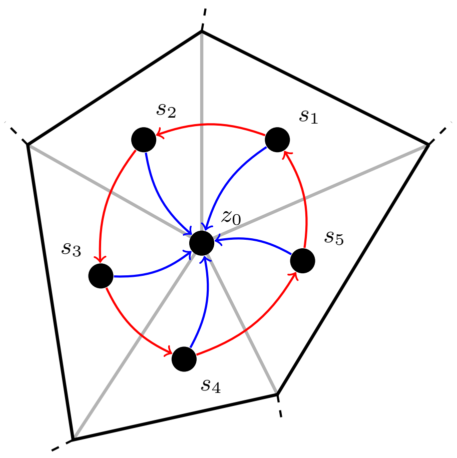


Figure 5: PENNANT mesh structure

```
1  idx zones = ispace(ptr);
2  idx sides = ispace(ptr);
3  field submesh_id : zones → int;
4  field mapsz : sides → zones+;
5  field mapss3 : sides → sides+;
6
7  idx bad_sides = sides{ s | s→mapsz != s→mapss3→mapsz };
8  idx bad_zones = bad_sides → mapsz;
9  idx any_bad_sides = bad_zones ← mapsz;
10
11 for i in ispace(0, N) {
12   idx my_zones = zones{ z | z→submesh_id = i };
13   idx my_sides = (my_zones ← mapsz) − any_bad_sides;
14
15   assert my_sides→mapsz ≤ my_zones;
16   assert my_sides→mapss3 ≤ my_sides;
17 }
```

Figure 6: PENNANT partitioning in DPL

An example of the latter situation can be found in the PENNANT[18] application, which simulates hydrodynamics in an irregular mesh and is used in our evaluation in Section 6. A *zone* in a PENNANT mesh is an arbitrary polygon which is divided into triangular *sides* (see Figure 5). Each side has a pointer to the zone of which it is a part and also to the next side. In a distributed computation, the zones are partitioned based on their `submesh_id` and then the corresponding partition of the sides is a preimage through the `mapsz` ("map sides to zones") field. This trivially satisfies the first consistency requirement on line 15 of Figure 6.

The application also depends on being able to walk the sides "around" a zone (line 16), which appears to necessitate a fixed-point operation. In this case (and in others like it), the key observation is that while walking all the sides reachable from a given side requires a fixed point, determining whether two adjacent sides are *not* in the same zone does not. The combination of a complex filter (line 7) to identify malformed sides, an image to compute the corresponding malformed zones (line 8), and then a preimage to share that information with the other sides of the same zones (line 9) provides the equivalent of the fixed-point operator in these cases, but is still statically analyzable and has only linear runtime cost.

Given these exclusions in the pursuit of static analyzability, the reader may question the inclusion of the complex filter operations. Complex filter operations are distinguished from the omitted features above in three critical ways:

1. Pragmatically, their exclusion would result in DPL being too weak to describe both the circuit and PENNANT examples, and likely many other common patterns.
2. Several heuristics (described in Section 3.2) are available to eliminate such expressions in many cases.
3. When the static analysis is unable to yield a result, the corresponding dynamic analysis has both time and space complexity that is linear in the index space sizes — no worse than any of the other partitioning operations. (This is due to the restriction that even complex filter operations must be quantifier-free.)

3. Static Analysis

This section describes the static analysis used to discharge most DPL assertions at compile time. We first describe a sound and complete version of the analysis that can be used in the absence of complex filter operations. We then cover the problems caused by the introduction of complex filter operations, and present a modified version of the algorithm that must necessarily be incomplete, but for which an efficient dynamic check can be defined to handle cases where the static analysis fails to yield a yes/no answer.

3.1 Simple Filter Operations

At a high level, the analysis in the absence of complex filter operations proceeds as follows:

1. Extract the loop-free subset of a DPL program that culminates in the assertion(s) to be checked.
2. Translate the DPL statements into a formula in a fragment of first-order logic that may contain quantifiers, linear arithmetic, and uninterpreted functions, but is restricted enough to remain decidable.
3. Translate the formula into an equisatisfiable one that is quantifier-free.
4. Use a decision procedure for the theory of Presburger arithmetic and uninterpreted functions to decide the satisfiability of this final formula.

The first two steps are straightforward. The immutability and lexical scoping of index spaces allows a syntax-directed translation of the DPL program to generate the necessary formulas in linear time. The translation maintains the following state:

DPL	**F (field mapping)**	**Logic**
idx A = ispace(int32);	$\{\}$	$P_A(x) \equiv \mathit{int32}(x)$
field f : A→A;	$\{\}$	
function g : int32→int32;	$\{g \to \alpha\}$	
property g(x) = x + 1;	$\{g \to \alpha\}$	$P_g(x,y) \equiv y = x + 1$
immutable f {	$\{g \to \alpha, f \to \beta\}$	
idx Y = A→f;	$\{g \to \alpha, f \to \beta\}$	$P_Y(x) \equiv \exists y, x = \beta(y) \wedge P_A(y)$
idx W = A - Y;	$\{g \to \alpha, f \to \beta\}$	$P_W(x) \equiv P_A(x) \wedge \neg P_Y(x)$
idx Z = W←g;	$\{g \to \alpha, f \to \beta\}$	$P_Z(x) \equiv \exists y, \alpha(x) = y \wedge P_g(x,y) \wedge P_W(y)$
assert Y $*$ Z;	$\{g \to \alpha, f \to \beta\}$	SAT[$P_Y(x) \wedge P_Z(x)$]
}	$\{g \to \alpha\}$	

Figure 7: Example of syntax-directed translation of DPL into logic

- The names of index spaces in scope, with unary predicates $P_I(x)$ that capture membership in each index space. The forms this predicate can take are described below.
- The for loop variables in scope, with any known bounds.
- The mapping **F** of DPL functions and currently immutable fields to function symbols. A field is assigned a fresh function symbol at the beginning of an `immutable` block. The assignment is removed from **F** at the end of that block. (When a field that does not appear in the immutable set is used in a DPL operation, it is given a fresh function symbol for each occurrence.)
- The properties associated with each DPL function. These take the form of binary predicates $P_f(x, f(x))$. For convenience, a field (or a function with no properties) has a predicate that is simply $\top$ (*true*).

When a line of the program contains index space expressions, they are parsed and a fresh predicate is created for each subexpression. A new index space created using the `ispace` keyword places constraints based on the base type and any supplied bounds:

$$C = \texttt{ispace}(T) \implies P_C(x) \equiv P_T(x)$$
$$C = \texttt{ispace}(T, v_1, v_2) \implies P_C(x) \equiv P_T(x) \wedge x \geq v_1 \wedge x \leq v_2$$

Set operations result in standard logical connectives:

$$C = A \,\&\, B; \implies P_C(x) \equiv P_A(x) \wedge P_B(x)$$
$$C = A \mid B; \implies P_C(x) \equiv P_A(x) \vee P_B(x)$$
$$C = A - B; \implies P_C(x) \equiv P_A(x) \wedge \neg P_B(x)$$

Image and preimage operations yield existential quantifiers:

$$C = A \to f \implies P_C(x) \equiv \exists y, x = \mathbf{F}[f](y) \wedge P_f(y,x) \wedge P_A(y)$$
$$C = A \leftarrow f \implies P_C(x) \equiv \exists y, \mathbf{F}[f](x) = y \wedge P_f(x,y) \wedge P_A(y)$$

Finally, the translation of a simple filter depends on whether the index is used directly or if a field lookup is performed (in which case it is very similar to a preimage):

$$C = A\{x \mid \phi(x)\} \implies P_C(x) \equiv P_A(x) \wedge \phi(x)$$
$$C = A\{x \mid \phi(x, f(x))\} \implies P_C(x) \equiv P_A(x) \wedge \exists y, (\mathbf{F}[f](x) = y \wedge P_f(x,y) \wedge \phi(x,y))$$

Whereas index space expressions generate unary predicates, DPL `assert` statements use those predicates to yield statements whose validity can be tested. As is standard, these statements are negated, yielding an existential statement suitable for a satisfiability query:

$$\texttt{assert}\ \ A \leq B \implies \exists x, P_A(x) \wedge \neg P_B(x)$$
$$\texttt{assert}\ \ A * B \implies \exists x, P_A(x) \wedge P_B(x)$$

Figure 7 gives an example DPL program and its translation into logic. The logic used is a carefully-chosen fragment of first order logic, and one whose decidability has, to our knowledge, not been previously studied. General first order logic allows arbitrary variables, quantifiers, function symbols, and predicates. A fragment disallowing function symbols and limiting the number of free variables to two (i.e. FO^2) is decidable [29], but too restrictive - DPL requires arbitrary function symbols to reason about images and preimages. Allowing a third free variable (i.e. FO^3) results in undecidability [6], as does nested use of function symbols [20]. Intuitively, function composition can be used to create new variables:

$$f(g(x)) = y \iff g(x) = z \wedge f(z) = y$$

DPL also requires the use of linear integer arithmetic (i.e. Presburger arithmetic), which is decidable with arbitrary quantifiers via Cooper's procedure [12], but undecidable with the addition of a single unary predicate [17]. The immutability of index spaces in DPL allows the elimination of predicates, avoiding this concern. Similarly, use of arbitrary function symbols and quantifiers with Presburger arithmetic results in undecidability, but these analyses were performed with no restrictions on the number of variables. We

```
function DPLSAT($\phi$)
  $\rho := \top$
  while $\neg$quantifier-free($\phi$)
    select top-level quantifier $\exists y, \psi(\bar{x}, y)$ in $\phi$
    if coinflip() $=$ true
      $\phi := \phi[\exists y, \psi(\bar{x}, y)/\top] \wedge \psi(\bar{x}, y')$
    else
      $\phi := \phi[\exists y, \psi(\bar{x}, y)/\bot]$
      $\rho := \rho \wedge \neg\psi(\bar{x}, q)$          -- q does not appear in $\psi$
  if $\rho \equiv \top$
    return PAUFSAT($\phi$)
  else
    for $v$ in free-vars($\phi$)
      $\phi := \phi \wedge \rho(\bar{x}, v)$
    return DPLSAT($\phi$)
```

Figure 8: DPLSAT Pseudocode

were unable to find any studies of Presburger arithmetic with limitations on the number of free variables and use of functions, perhaps due to doubts about the utility of that fragment. However, DPL inhabits this space (as a result of the limitations adopted in Section 2.2), and we show below that these restrictions do indeed result in a decidable fragment of first order logic.

The first step of the analysis of a DPL problem is to expand all of the predicates. The DPL assertion is valid if the resulting formula is unsatisfiable. For the example in Figure 7, expanding the assert on the last line yields:

$$\text{SAT}[\ (\exists y, x = \beta(y)) \wedge (\exists z, \alpha(x) = z \wedge z = x + 1 \wedge \neg(\exists w, z = \beta(w)))\ ]$$

The base type predicates *int32*($\cdot$) have been eliminated and all variables have been given unique names for clarity, but the predicate expansion has duplicated the subformula for P_Y. This is necessary for our decision procedure below, but we note that it can result in a formula that is exponential in the size of the original program in the worst case.

The final two steps of our strategy are combined in the form of a **DPLSAT** procedure that decides the satisfiability of our statements, using a known decision procedure for the theory of quantifier-free Presburger arithmetic and uninterpreted functions (**PAUFSAT**) as a subroutine[30]. Figure 8 gives pseudocode for a non-deterministic form of the **DPLSAT** algorithm, and we describe the intuition here.

The **DPLSAT** algorithm uses case analysis to eliminate quantifiers, exploring the subset of possible models in which a given quantified expression holds and those for which it does not hold. This is captured by the following transformation, which is generally applicable. A top-level existential quantifier (i.e. one not contained in another quantifier) is removed from the main formula and either added as a condition on a new free variable in the "assumed true" case or in its negated form (i.e. a universally quantified axiom) in the "assumed false" case:

$$\phi(\bar{x}, \exists y, \psi(\bar{x}, y)) \overset{SAT}{\iff} (\psi(\bar{x}, \top) \wedge \psi(y^+)) \vee (\forall y^-, \neg\psi(\bar{x}, y^-)) \wedge (\phi(\bar{x}, \bot))$$

Each case is further subdivided until ϕ becomes quantifier-free. If more than one quantifier was "assumed false", their bodies are conjoined, resulting in a statement of the form:

$$(\forall q, \rho(\bar{x}, \bar{y}, q)) \wedge \phi_{QF}(\bar{x}, \bar{y})$$

The universal quantifier can now be instantiated for all free variables in ϕ_{QF}. This preserves equisatisfiability because the statements generated for DPL assertions fit within a very restricted fragment of first-order logic, allowing the elimination of the universally quantified axiom from the statement once it has been instantiated for all free variables that exist at the time.

The body of every existential quantifier in any formula generated by DPL is in FO^2 (i.e. first-order logic with at most 2 free variables in any subformula), which means that the body of a nested quantifier can only refer to the most immediately enclosing quantified variable. Further, each body contains exactly one term that uses a function, and it is always of the form $f(x) = y$, i.e. a functional dependence between the quantified variable and the enclosing one. As a result, when these bodies are negated, the resulting ρ includes a disjunction of disequalities. In the instantiation of ρ for one of the free variables in ϕ_{QF}, these disequalities may be incompatible with functional dependences already present in ϕ_{QF}, making the rest of ρ relevant for satisfiability. However, any new quantifier created through this instantiation will create a new free variable that has a functional dependence with exactly one existing variable. The instantiation of the universal quantifier for this new variable always results in a disequality that can be trivially satisfied.

With this constraint, the **DPLSAT** algorithm is guaranteed to terminate and we have the following bounds on its complexity.

Theorem 1. **DPLSAT** *runs in NEXPTIME (and requires EXPSPACE).*

Briefly, each recursive call to **DPLSAT** may quadratically increase the number of quantifiers in ϕ in the worst case, but it reduces the maximum nesting depth of these quantifiers by at least one level. The loop in each call to **DPLSAT** is linear in the number of quantifiers occurring in ϕ. In practice, these worst-case bounds can be dramatically improved by performing simplifications on ϕ and ρ within the loop, pruning that part of the search if either becomes unsatisfiable.

Returning to our example, we have 3 quantifiers, and a maximum nesting level of 2. There are 6 ways in which the nondeterminstic choices in the quantifier-elimination loop can complete (if z is assumed unsatisfiable, w disappears from ϕ as well). Only one results in a new ϕ that isn't trivially unsatisfiable - the case in which y and z are hypothesized to be satisfiable, but w is not. This case yields:

$\phi \equiv f(y^+) = x \wedge g(z^+) = x \wedge z^+ = x + 1$
$\rho \equiv f(q) \neq z^+$

The simplification of ϕ is not able to eliminate any variables, so ρ is instantiated 3 times, with q being replaced by each of the free variables x, y^+ and z^+ in turn:

$$f(y^+) = x \wedge g(z^+) = x \wedge z^+ = x + 1 \wedge \\ f(x) \neq z^+ \wedge f(y^+) \neq z^+ \wedge f(z^+) \neq z^+$$

This formula is easily satisfied (e.g. $x = 0, y^+ = 0, z^+ = 1, f(\cdot) = 2$), yielding a model for the original formula. Thus we have shown the assertion in the DPL code in Figure 7 to be incorrect under some possible execution of the program. Had the code made g an identity function instead of a successor function, the instantiated formula would contain a contradiction (i.e. $f(y^+) = x \wedge z^+ = x \wedge f(y^+) \neq z^+$), rendering it unsatisfiable and the original assertion would be discharged.

This divide-and-conquer approach to the problem is similar to the techniques used in SMT solvers such as Z3[14] and CVC4[4]. The main difference is while SMT solvers rely on heuristics to guess which cases to try, we are able to precisely enumerate the cases that matter. Interestingly, our bounds on the complexity of **DPLSAT** suggest that a heuristic approach in which universal quantifiers are only instantiated on demand will also have a bounded execution time. Anecdotally, this appears to be the case - the Z3 SMT solver took under a second to prove the validity of each assertion in our test cases.

3.2 Complex Filter Operations

Our algorithm above relies on efficient queries for quantifier-free formulas, which are available only in the absence of complex filter operations. In complex filter operations, the composition of multiple functions, or a function with arithmetic, effectively creates additional free variables, extending the formula to FO^3 and beyond, fragments that are known to be undecidable[6].

As discussed in Section 2.2, complex filter operations are necessary to express many common application use cases, so we extend our analysis to attempt to discharge these assertions either entirely at compile time or with a dynamic test that is more efficient than an explicit test of the assertion condition. We use three heuristics.

For the first heuristic, we observe that any complex filter can be replaced by a simple filter using a fresh boolean function and the resulting formula is weaker than the original. Therefore, we can start by replacing all complex filter operations in this manner and test for satisfiability. If the modified formula proves to be unsatisfiable, the original formula must be as well and we have validated the assertion. If a satisfying assignment is found, we test it against the complex filter predicates that were removed. If it is compatible with these predicates, we have a true positive and the original assertion is invalid. However, if at least one of the complex predicates is not satisfied, the assignment represents a false positive and we continue with the next heuristic.

The second heuristic also attempts to avoid runtime checks entirely, and takes the pragmatic approach of adding back in the problematic predicate(s) and hoping that a modern SMT solver can handle it. As termination is no longer guaranteed, a timeout must be set on the query, but this works well in practice. Z3, with its inbuilt heuristics, found all of our complex filter test cases to be unsatisfiable in just a few seconds.

The final heuristic attempts to simplify the runtime check rather than eliminating it entirely, and targets the case where these complex filter operations are used to "sanitize" input data. An example of this was seen with PENNANT in Section 2.2. While this filtering is critical for the correctness of the program, it is expected that application mesh data structures will be well-formed in practice. This heuristic tries replacing a complex filter with trivial predicates that either include all or none of the elements of the input index space and retesting the assertion. If either is successful, an additional runtime check is added that tests this intermediate result for equality or emptiness (depending on which predicate succeeded). These runtime checks are trivial, and if they succeed, the more expensive check can be skipped.

4. Dependent Partitioning in Regent

Our dependent partitioning framework is intended to be applicable to a variety of programming models. In this section, we discuss an implementation in Regent[31], a language designed to target the Legion runtime. The Regent data model and type system make it easy to embed DPL partitioning operations, and the compiler can usually insert the necessary consistency checks automatically.

Data in a Regent program is stored in *regions*, each of which has one or more fields. Regent includes a first-class object called a *partition* which is equivalent to an array of subregions. The dependent partitioning operations in our framework must be extended to work on these arrays.

Partitions are created in Regent by supplying a region to the `partition` operation along with a *coloring* object. A coloring is an abstract data type that allows the application to describe any association of colors (generally small integers) to elements of a region. The operation creates an array of subregions where each subregion contains all the elements of the parent region of a particular color.

A coloring may assign any number of colors (or none) to each element in a region and must allow random access, as colorings are often computed by following the topology of the application's data structures. Regent uses Legion's standard map-of-sets-of-indices representation. Ideally, other representations would be available that optimize for various common sparsity patterns, but this would com-

```
1  var p_nodes_eq = block_split(all_nodes, N)
2  var p_wires_eq = block_split(all_wires, W)
3
4  for i in 0, N do load_circuit(p_nodes_eq[i], p_wires_eq[i]) end
5
6  var p_nodes = partition(all_nodes, subckt, N)
7  var p_wires = preimage(all_wires, p_nodes, in_node)
8  var p_extern = difference(image(p_nodes, p_wires, out_node),
9                             p_nodes)
10 var all_shared = union(p_extern)
11 var all_private = difference(all_nodes, all_shared)
12 var p_pvt = intersection(p_nodes, all_private)
13 var p_shr = intersection(p_nodes, all_shared)
14 var p_ghost = intersection(p_extern, all_shared)
15
16 fspace Subcircuit(rn : region(Node),
17                   rw : region(Wire(rn, rn, rn))) {
18   rp : region(Node),
19   rs : region(Node),
20   rg : region(Node),
21   wp : region(Wire(rp, rs, rg)),
22   ...
23 } where rp * rs * rg, rp <= rn, rs <= rn, rg <= rn, wp <= rw
24
25 for i = 0, N do
26   subckt = Subcircuit(all_nodes, all_wires) {
27     rp = p_pvt[i],
28     rs = p_shr[i],
29     rg = p_ghost[i],
30     wp = p_wires[i],
31     ...
32   }
33 end
```

Figure 9: Regent circuit simulation with new partitioning operations

plicate the interface significantly. Our dependent partitioning framework lets us eliminate the coloring object entirely, instead using a field to represent the coloring. The result is the deletion of Regent's existing `partition` operation and coloring-related interfaces, and the following new dependent partitioning operations:

partition(*parent_region*, *field*, N)
image(*parent_region*, *source_partition*, *field*)
preimage(*parent_region*, *target_partition*, *field*)
union(*region_or_partition*, ...)
intersection(*region_or_partition*, ...)
difference(*region_or_partition*, *region_or_partition*)
block_split(*parent_region*, N)

The new `partition` operation takes a parent region, a field, and the number of subregions to create. For each i in $[0, N)$, a simple filter defines the subregion:

$$subregion_i = parent\{x \mid x \to field = i\}$$

The `image` and `preimage` operations accept an existing partition (i.e. an array of subregions) and map an image (or preimage) over the elements, returning a new array of image (or preimage) subregions:

$$image_i = parent \,\&\, (source_i \to field)$$

Regent's type system requires the inclusion of an additional intersection with the `parent_region` of the desired output partition. This can be efficiently fused with the image/preimage computation in the runtime.

The set operations accept either regions or partitions as arguments. If any of the arguments are partitions, the set operation is mapped over the elements of the arrays, yielding a new partition. The `union` and `intersection` operations may also be called with a single partition as an argument, in which case the elements of that partition are reduced to a single region that is the union (or intersection) of all the subregions of the original partition.

Finally, the `block_split` operation allows a region to be partitioned into N (roughly) equal subregions using arithmetic on the indices themselves. This operation can be performed before storage is allocated for the region's fields.

Figure 9 shows the new partitioning code for the circuit simulation in Regent. When a Regent program is modified to use these new dependent partitioning operations, there are four benefits that are realized:

1. The number of partitioning operations remains roughly the same, but change from a sequence of nondescript `partition` calls to a self-documenting description of the partitioning computation.
2. All the code related to creating and populating the coloring objects can be deleted.
3. Significant improvements in partitioning time, due to the runtime's ability to choose optimized data structures (discussed in the next section) and (in nearly all cases) the elimination of dynamic consistency checks.
4. The program's input file loading or generation code can be distributed as well, allowing the application to work on data structures that are too large to fit on a single node.

Consistency checks are generated by Regent when pointers or regions are "downcast" to a more restrictive type. An example of this is in lines 26-32 of Figure 9. To discharge these checks during compilation, the Regent compiler must perform the static analysis described in Section 3. Rather than implementing the analysis a second time, the Regent compiler translates the relevant parts of the program into DPL and performs the analysis on the DPL version.

Regent allows a programmer to assert that they expect iterations of a loop to be able to run in parallel. This requires static analysis to verify there are no loop-carried dependencies, which often demands that the regions accessed in each iteration are non-overlapping. The current Regent compiler can perform the needed analysis only in simple cases. With the use of the dependent partitioning framework, many cases that could not be checked before (such as the red-black Gauss-Seidel example in Section 2) are handled.

5. Dependent Partitioning in Legion

The output of the Regent compiler is C++ code that uses the Legion runtime API. Applications can also be written directly for the Legion runtime API (although without the benefits of the Regent compiler's type checking and static analysis), therefore it is important that the new dependent partitioning operations are supported by the Legion runtime. The Legion API was augmented to include new entry points for each of the operations added to Regent.

Although the effort involved in achieving an efficient and scalable implementation required solving many issues, we focus on two. First, we discuss the internal representation of an index space in Legion, and second, we cover a key optimization that improves the scalability of image and preimage operations.

5.1 Index Space Data Structure

Since index spaces are the key data type used in partitioning operations, it is important that they be efficient in both memory usage and query complexity. The construction cost for an index space is a secondary consideration. As our framework allows index spaces to be computed from arbitrary data, we must use a data structure that can describe any set of N indices, which will necessarily require $O(N)$ storage cost and $O(logN)$ query time. However, we also provide a mechanism for compressing commonly occurring patterns to improve memory usage and query time.

The index space data structure contains a `bounds` interval that gives an upper bound on elements of the index space, a `dense` flag that indicates if every index in the `bounds` interval is included, and a `cluster_list` stores the intervals containing elements when the `dense` flag is disabled. Each list entry includes an optional bitmask to capture non-dense clusters of indices. The intervals are sorted to maintain a logarithmic worst-case complexity for queries.

The Legion runtime operates in a distributed memory environment, and index spaces often need to be copied between nodes to perform partitioning operations. In some cases, imprecision in the index space membership is acceptable if it reduces the network traffic. Upon request, a precise index space is turned into an *approximate index space* that has a configurable maximum size by first omitting all of the bitmasks for non-dense clusters and then iteratively merging pairs of clusters into single intervals that cover both. This algorithm guarantees that an approximate index space is an upper bound for the original index space — i.e. every index present in the original index space is also present in the approximation. Approximate index spaces are a crucial component of the implementation of several dependent partitioning operations that we discuss below.

5.2 Scalability of Image and Preimage Operations

The Legion runtime allows data for a field to be distributed across multiple *region instances*, and the implementation of dependent partitioning operations that use field data. The internal interface for computing images is shown here:

```
Event compute_images(IndexSpace parent,
                     const vector<IndexSpace>& sources,
                     const vector<RegionInstance>& instances,
                     const vector<IndexSpace>& instance_domains,
                     vector<IndexSpace>& images,
                     Event wait_for);
```

The `parent` parameter is the index space for the Regent `parent_region`, while `sources` contains the index spaces for each region in the Regent `source_partition`. The index spaces that Legion will use to construct the output partition are filled into the `images` vector - this will have the same length as the `sources` input vector. The `wait_for` precondition and output events allow the Legion runtime to describe the necessary scheduling dependencies for this operation. Finally, the `instances` vector provides the list of instances containing the field data. Each instance can be thought of as a partial function f_j, with the corresponding `instance_domain` element giving the subdomain D_j over which each partial function is defined. Using these partial functions, we can write the image operation as a union of the image of each partial function (using P for the parent index space, S_i for a given source index space and I_i for the corresponding image output:

$$I_i = \bigcup_j P \cap f_j^{\rightarrow}(S_i)$$

Note that every S_i interacts with every f_j in this operation. Although the sizes of the `sources` and `instances` vectors need not match, they will often both be $O(N)$ in practice (where N is the number of ranks or cores in the machine). In the worst case (e.g. a completely random graph), this requires $O(N^2)$ work. However, in most common cases, there will be at least some structure to the data and $f_j^{\rightarrow}(S_i)$ will be a nonempty set for at most $M \ll N^2$ pairs. By using an *output-sensitive algorithm*, we can greatly improve the performance of these common cases. We first rewrite the above formula to make the known domain of each partial function explicit and then take advantage of the fact that the image of an empty set is an empty set:

$$I_i = \bigcup_{j,\ D_j \cap S_i \neq \emptyset} P \cap f_j^{\rightarrow}(D_j \cap S_i)$$

We have reduced the work to require only $O(M)$ image calculations, but we still have $O(N^2)$ intersection tests to perform. However, if we can perform these tests in a centralized location, we can use an *interval tree* to reduce the cost to $O(NlogN+M)$. Copying all the S_i and D_j to a single node can be prohibitive if the index spaces are sparse, so the final improvement is to construct the interval tree and perform the intersection tests using the approximate index spaces $\widetilde{S_i}$ and $\widetilde{D_j}$. This approximation is sound as the intersection of two upper bounds can be empty only if the actual sets are empty.

Application	Original LOC	Dependent Partitioning LOC	Reduction
PENNANT	163	6	96%
Circuit	159	8	95%
MiniAero	51	7	86%

Figure 10: Reduction in code required to compute partitions

The precise intersection is still needed within the image, but is only performed $O(M)$ times, and the complexity for the whole operation is $O(NlogN + M)$ as well.

A similar optimization is used for preimages, with the complication that we cannot directly intersect the instance domains D_j with the targets T_i in:

$$I_i = \bigcup_j P \cap f_j^{\leftarrow}(T_i)$$

Instead, we compute $R_j = f^{\rightarrow}(P)$ and use the identity:

$$A \cap f^{\leftarrow}(B) = A \cap f^{\leftarrow}(f^{\rightarrow}(A) \cap B)$$

to yield:

$$I_i = \bigcup_j P \cap f_j^{\leftarrow}(R_j \cap T_i)$$

and again perform approximate intersection tests. Noting that the intermediate R_j's will be discarded after this operation, we can save more time and memory by only computing an *approximate image* $\widetilde{R_j} = \widetilde{f_j^{\rightarrow}}(P)$ in which bitmasks are never generated and intervals are merged during the computation. Our final form again requires $O(NlogN + M)$ work to compute N preimages:

$$I_i = \bigcup_{j,\ \widetilde{R_j} \cap \widetilde{T_i} \neq \emptyset} P \cap f_j^{\leftarrow}(\widetilde{R_j} \cap T_i)$$

6. Evaluation

In addition to the qualitative benefits of catching many partitioning-related problems at compile time, our dependent partitioning framework provides quantitative improvements to both programmer productivity and application performance. To assess these benefits, we look at three applications that have been written and tuned for Regent. Two of these applications are the circuit simulation and PENNANT, which were discussed briefly in Sections 1 and 2. The third application is MiniAero, part of the Mantevo[2] project, which performs simulation of fluid dynamics in an unstructured 3-D mesh. Although all three applications perform computations on an unstructured graph or mesh, there are differences in how "unstructured" they are. Both PENNANT and MiniAero use a spatial decomposition to derive their independent partition, resulting in a nearest-neighbor communication pattern between the partitions. In contrast, the circuit example uses a randomized graph that has an all-pairs communication pattern between the partitions.

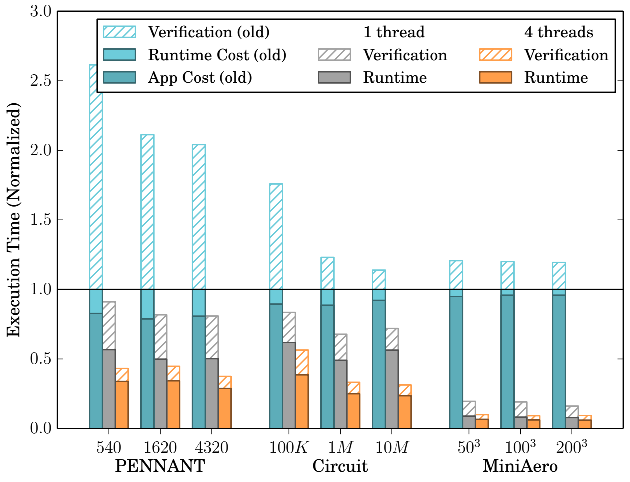
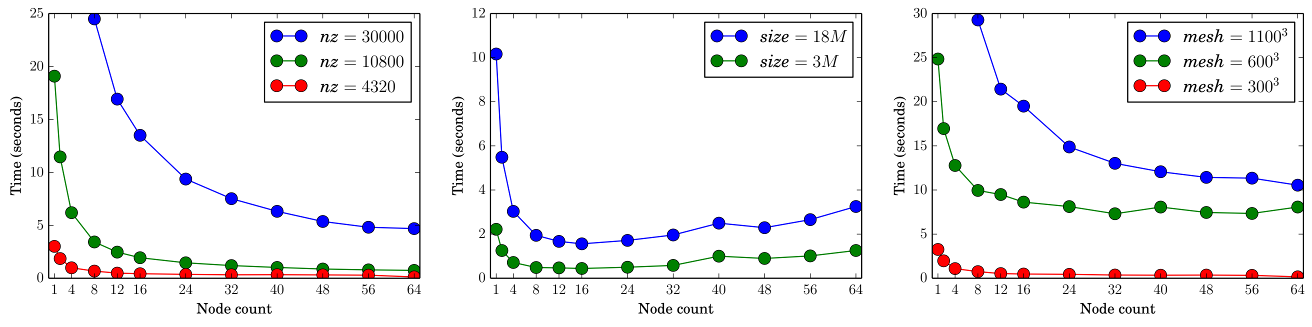


Figure 11: Partitioning time improvement on a single node

We rewrote the partitioning code of each application to use the new dependent partitioning operations in Regent that we described in Section 4. The effort took only minutes, and involved deleting nearly all of the code that was generating colorings for the old `partition` operation, and instead using images, preimages and set operations to achieve the same effect. This time also includes the time that was necessary to run the static analysis described in Section 3 and find two bugs in the newly written code, including the need to "filter" the PENNANT mesh as described in Section 2.2. Since there were no changes to the actual partitions being computed, no other code was changed in any of the applications. Figure 10 summarizes the dramatic improvements in the number of partitioning-related lines of code for each application. These results do not include the application-specific code that computes the assignments for the independent partition, as our framework allows that code to be used as is.

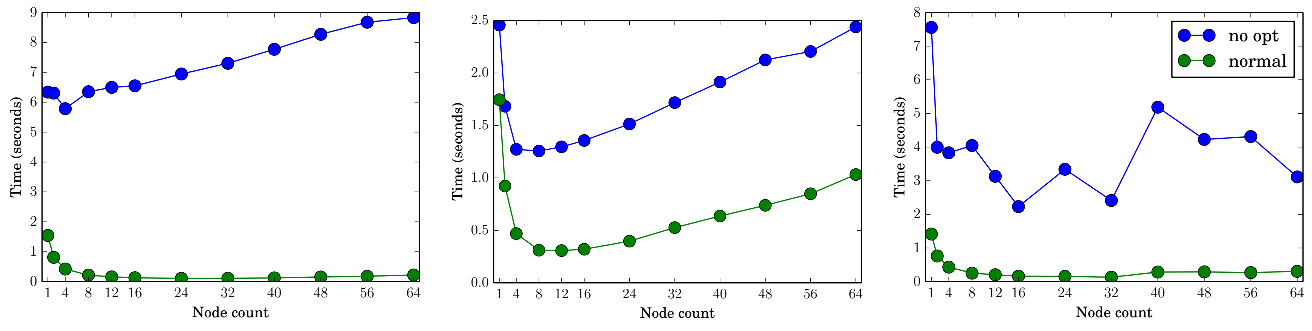
The next benefit that can be seen is in the performance improvements in the partitioning computation when run on a single compute node. Figure 11 shows these benefits for each of the applications. These three applications perform partitioning during initialization and then simulate for anything from seconds to hours, depending on the user's needs. To eliminate that variability, we report partitioning speedups considering only the time required for computing the partitioning and the subsequent verification in the original Regent code. Overall application speedup will vary between this upper bound and negligible, depending on the length of the simulation. (None of these perform any re-partitioning (e.g. for load balancing) during the computation, due in part to the cost of such repartitioning with existing programming models. It is our hope that the performance and productivity improvements of dependent partitioning make this a much more attractive option in the future.)

Each application was run with three input sizes, each roughly 10x larger than the next. For each case, the blue bar on the left shows the time taken by the old partitioning



(a) PENNANT (b) Circuit (c) MiniAero

Figure 12: Strong scaling of partitioning work up to 64 nodes.



(a) PENNANT (b) Circuit (c) MiniAero

Figure 13: Impact of intersection optimization.

method in Regent, and it is split into the time spent in the application to compute the coloring (the darker blue) versus the time spent in the runtime to convert the coloring into its internal representation (the lighter, but still solid, blue). Across all three applications, the majority of the partitioning-related effort falls to the application.

The middle grey bar in each case shows the time required to perform the equivalent dependent partitioning operations on a single thread. PENNANT partitioning times are improved by 76-100%, Circuit by 62-103%. MiniAero benefits the most from a data structure that efficiently stores clusters of indices, improving by 11.2-12.7X. The orange bar on the right in each case shows the further improvement in partitioning time when 4 runtime threads are used to perform the dependent partitioning operations. PENNANT sees an additional 45-74% improvement, Circuit gets 60-139% better. MiniAero improves by 30-35%, resulting in an overall 15.1-16.6X speedup compared to the original partitioning.

The hashed bars stacked on top of each case show the additional runtime cost of verifying the consistency of the computed partitions. This cost is comparable to the cost of the partitioning itself, and in current Regent programs, the user is tempted to disable the checks. With dependent partitioning, these consistency checks can be described in terms of the same set of dependent partitioning operations, and if they must be run, they see similar performance benefits. For all three of these applications, the static analysis described in Section 3 validated the assertions using Z3, and the runtime cost is only shown for comparison purposes.

At least as important as the speedups achieved by our dependent partitioning framework are the improvements to scalability, which significantly improve Regent's ability to partition large problem sizes. For these experiments, the number of pieces into which the data was partitioned was held constant to keep the amount of work stable as the node count is increased from a single node up to 64 nodes. Our experiments were performed on a large Infiniband cluster with each node containing two six-core Intel Westmere CPUs and 32GB of system memory. Figure 12 shows the results of our experiments. The PENNANT and MiniAero applications enjoy excellent strong scaling behavior, with PENNANT obtaining up to a 29x improvement on 64 nodes. In contrast, the circuit example receives some initial benefit from going up to 12 or 16 nodes, but beyond that, the communication required to compute a partitioning that involves all-pairs communication becomes the bottleneck.

Our final experiment explores the benefit of the intersection optimization discussed in Section 5. For each application, a single problem size was selected and scaling experiments were performed, first with the optimization enabled, and then again with the optimization disabled. For this case, the number of partitions into which the data is being divided is chosen to be 8 times the node count, anticipating the need to expose some parallelism within a node as well as between the nodes. The total cost of partitioning therefore increases with increasing node count. Despite this, the results (Figure 13) show the nearest-neighbor structure of the PENNANT and MiniAero meshes allows improvements in the time taken to compute a partitioning of their data, as long as the intersection optimization is enabled. Without the optimization, not only does the total work increase for all node counts, it now grows quadratically for all cases, causing erratic but generally increasing partitioning times for all applications as the node count increases.

7. Related Work

Most partitioned global address space (PGAS) programming models use data distribution techniques similar to those provided by Chapel[7, 8, 10]. A few also include the idea of "indirect" maps based on an application-supplied array of indices[1, 9], but do not perform analysis or include a dependent partitioning-like way to derive one map from another. A subset of the expressive capabilities of DPL's functions on indices exist in ZPL's direction vectors[15].

Many algorithms exist for computing efficient partitions of various data structures (e.g. graphs[11, 22] or meshes[13]). Our framework is designed to be complementary to these, accepting their results as independent partitions and then allowing further dependent partitions to be derived from those results. Similarly, some existing libraries provide optimized routines for computing ghost cells in structured meshes[3] or "cones" and "supports" in unstructured meshes[23]. These libraries do not provide any verification capabilities when used directly, but could be used by an implementation of DPL for the computation of images and preimages.

Domain-specific languages may be designed exclusively for distributed computations on a particular data structure such as a graph[24, 28] and internalize the problem of partitioning the application data. Although the application programmer is no longer involved, the performance and verification benefits of DPL may still be attractive to the implementers of such a DSL.

Various programming languages include *stencil analysis* that analyzes application code to extract memory access patterns[16, 21, 25, 26]. Many of these patterns map well to the dependent partitioning operations in our framework.

The need to compute reachability exists in non-distributed settings such as loop optimization. The Sparse Polyhedral Framework[32] is able to reason about the composition of functions (i.e. images) or of function inverses (i.e. preimages) that incorporate integer arithmetic. SPF does not require the ability to compose images with preimages, significantly simplifying the decision problem.

There are strong parallels between DPL and efforts in the database community. Index spaces have much in common with database *views* and the image and preimage dependent partitioning operations can be thought of as *semi-joins* in relational algebra. Optimization of database queries on distributed data is an area of active research[27, 33], but has primarily focused on the structure of the queries and assumes worst-case scenarios for the data. Any structure in the data is usually viewed as a cause of load imbalance and is randomized away by using hash functions or Bloom filters.

The decisions related to what to include or exclude from the framework were informed by the (un)decidability of various first-order logic fragments[6, 12, 17, 20, 29]. Our analysis relies on a decision procedure for quantifier free Presburger arithmetic with function symbols[30].

8. Conclusion

We have presented a language-independent framework for *dependent partitioning*. Our framework improves upon the partitioning capabilities of existing programming models by providing maximal expressivity for defining arbitrary *independent partitions*, but then provides a carefully chosen set of operations for the derivation of *dependent partitions*. These dependent partitioning operations are concise and straightforward to use, simplifying application code for partitioning and permitting static analysis that is able to discharge most consistency checks between an application's partitions at compile time.

We describe an implementation of our dependent partitioning framework in the Regent language and the Legion runtime, demonstrating significant improvements in partitioning performance (2.6X-12.7X on a single thread) and scalability (29X speedup when going from 1 node to 64) compared to the existing Regent partitioning mechanisms.

Acknowledgments

This work was supported by Los Alamos National Laboratories Subcontract No. 173315-1 through the U.S. Department of Energy under Contract No. DE-AC52-06NA25396 and by DoE Office of Science grant DE-SC0012426. The authors acknowledge the following award for providing computing resources that have contributed to the research results reported within this paper: MRI-R2: Acquisition of a Hybrid CPU/GPU and Visualization Cluster for Multidisciplinary Studies in Transport Physics with Uncertainty Quantification. This award is funded under the American Recovery and Reinvestment Act of 2009 (Public Law 111-5).

References

[1] High performance fortran language specification, version 2.0. `http://hpff.rice.edu/versions/hpf2/hpf-v20/index.html`, 1997.

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

[3] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Lang-

tangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[4] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Computer Aided Verification*, 2011.

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

[6] E. Börger, E. Grädel, and Y. Gurevich. *The classical decision problem*. Perspectives in mathematical logic. 1997.

[7] 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.

[8] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 2007.

[9] B. Chapman, P. Mehrotra, and H. Zima. Vienna Fortran-a Fortran language extension for distributed memory multiprocessors. Technical report, DTIC Document, 1991.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an Object-Oriented Approach to Non-uniform Cluster Computing. In *Proceeings SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 519–538, 2005.

[11] C. Chevalier and F. Pellegrini. Pt-scotch: A tool for efficient parallel graph ordering. *Parallel computing*, 34(6):318–331, 2008.

[12] D. C. Cooper. Theorem proving in arithmetic without multiplication. *Machine Intelligence 7*, pages 91–99, 1972.

[13] W. Dawes, S. Harvey, S. Fellows, N. Eccles, D. Jaeggi, and W. Kellar. A practical demonstration of scalable, parallel mesh generation. In *47th AIAA Aerospace Sciences Meeting & Exhibit*, pages 5–8, 2009.

[14] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. 2008.

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

[16] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, et al. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2011.

[17] P. J. Downey. Undecidability of presburger arithmetic with a single monadic predicate letter. Technical Report TR-18-72, Harvard University (Cambridge, MA US), 1972. URL http://opac.inria.fr/record=b1005540.

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

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[20] Y. Gurevich. The decision problem for standard classes. *J. Symb. Log.*, 41(2):460–464, 1976.

[21] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector SIMD architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 13–24. ACM, 2013.

[22] G. Karypis and V. Kumar. Parallel multilevel series k-way partitioning scheme for irregular graphs. *Siam Review*, 1999.

[23] M. Lange, M. G. Knepley, and G. J. Gorman. Flexible, scalable mesh and data management using petsc dmplex. In *Proceedings of the 3rd International Conference on Exascale Applications and Software*, pages 71–76. University of Edinburgh, 2015.

[24] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[25] P. McCormick, C. Sweeney, N. Moss, D. Prichard, S. K. Gutierrez, K. Davis, and J. Mohd-Yusof. Exploring the construction of a domain-aware toolchain for high-performance computing. In *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 1–10. IEEE Press, 2014.

[26] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. J. Kelly. Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12. IEEE, 2012.

[27] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms:[extended abstract]. In *Proceedings of the 31st symposium on Principles of Database Systems*, pages 37–48. ACM, 2012.

[28] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[29] D. Scott. A decision method for validity of sentences in two variables. *Journal of Symbolic Logic*, 27:377, 1962.

[30] R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *J. ACM*, 26(2):351–360, 1979.

[31] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken. Regent: A high-productivity programming language for HPC with logical regions. In *Supercomputing (SC)*, 2015.

[32] M. M. Strout, G. Georg, and C. Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 61–75. Springer, 2012.

[33] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems (TODS)*, 31(1):1–38, 2006.