Visibility Algorithms for Dynamic Dependence Analysis and Distributed Coherence

Michael Bauer
NVIDIA
mbauer@nvidia.com

Elliott Slaughter
SLAC National Accelerator Laboratory
eslaught@slac.stanford.edu

Sean Treichler
NVIDIA
sean@nvidia.com

Wonchan Lee
NVIDIA
wonchanl@nvidia.com

Michael Garland
NVIDIA
mgarland@nvidia.com

Alex Aiken
Stanford University
aaiken@stanford.edu

Abstract

Implicitly parallel programming systems must solve the joint problems of dependence analysis and coherence to ensure apparently-sequential semantics for applications run on distributed memory machines. Solving these problems in the presence of data-dependent control flow and arbitrary aliasing is a challenge that most existing systems eschew by compromising the expressivity of their programming models and/or the performance of their implementations. We demonstrate a general class of solutions to these problems via a reduction to the visibility problem from computer graphics.

Keywords: Visibility, Legion, Dynamic Analysis, Coherence

1 Introduction

Implicitly parallel and distributed programming systems are popular with users who require a supercomputer or the cloud for large-scale computing, but are unfamiliar with or unwilling to develop explicitly parallel and distributed applications [2, 3, 8, 14, 16, 19, 29]. These systems provide high-productivity programming models based on automatic discovery of parallelism from computations over implicitly-distributed collection data types, such as arrays and dataframes.

Any implementation of an implicitly parallel and distributed programming model has two primary responsibilities. First, it must discover parallelism by performing a dependence analysis on sequences of distinguished computations, often called *operators* or *tasks*, that access subsets of the collections, which we call *regions*. Tasks that access disjoint regions, or access regions in ways that cannot interfere with other tasks (e.g., only reading data) can be executed in parallel. Second, the system must ensure coherence of regions so that tasks observe the most recent updates to data, consistent with a sequential shared-memory semantics of the original program.

In this paper we present and evaluate algorithms for managing coherence and detecting data dependences in the general case when regions can name arbitrary subsets of collections (and so regions can overlap, or *alias*), when regions are

```
struct Node {
   up   : float
   down : float
}

task t1(p<Node>, g<Node>):
   read-write p.up, reduce::+ g.down;
task t2(p<Node>, g<Node>):
   read-write p.down, reduce::+ g.up;

task main(N<Node>):
    read-write N.up, N.down;
    P[1..3] = ... # create primary partition of N
    G[1..3] = ... # create ghost partition of N
    while (*)
        for i = 1..3 t1(P[i],G[i])
        for i = 1..3 t2(P[i],G[i])
```

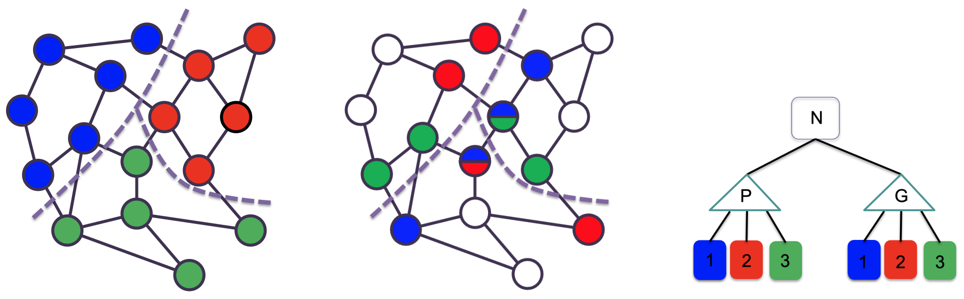
Figure 1. A stylized graph computation with tasks using two different partitions of the collection of nodes in the graph.

dynamically computed rather than statically specified, and when the sequence of tasks is determined by data-dependent control flow. Due to these requirements, we perform our implementation and evaluation using the Legion runtime [5] because it supports all three features, which are essential for many applications with data-dependent behavior.

The data-dependent nature of these problems mandates that solutions are dynamic dependence and coherence analyses. Most existing systems with such analyses impose restrictions to simplify their implementations, such as requiring that all regions be pairwise independent. We refer to systems with this model as using *name-based coherence*, because every data element must always have a unique name (i.e., can be in only one region); in contrast, in a *content-based coherence* system, data elements may be part of multiple regions simultaneously. Content-based coherence is more general than name-based coherence: content-based coherence can trivially support name-based coherence using a single region for each name. We further contrast content-based and named-based systems in Section 9.

Our main technical contribution is the insight that the problems of dependence analysis and content-based coherence in implicitly-parallel task-based distributed systems are closely related to the *visibility problem* in computer graphics (Section 3). After providing some background and an

PPoPP '23, February 25-March 1, Montreal, QC, Canada
©2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577515



(a) Primary partition (b) Ghost partition (c) Region tree

Figure 2. Primary and ghost partitions of a graph's nodes and the associated region tree.

example program that we use for the duration of the paper (Section 2), we demonstrate how to reduce these problems to the visibility problem. We then adapt three well-known visibility algorithms of increasing sophistication to address content-based coherence: the painter's algorithm, Warnock's algorithm, and ray casting (Sections 4–7). We evaluate implementations of all three algorithms on several different applications across machine scales (Section 8). The results show that ray casting has the best overall performance by a significant margin, and as a result the ray casting algorithm is the one currently in use by the Legion project. Finally, we discuss related work (Section 9) and conclude (Section 10).

2 Example and Background

In this section we give more background on the Legion programming model [5] and introduce a program that we will use as a running example throughout the paper. The program in Figure 1, which outlines a typical simulation on an undirected graph, illustrates the value of supporting content-based coherence. Each task accesses two different *subregions* of data: a set of nodes that form a piece of the graph and a set of ghost nodes adjacent to that piece. Subregions are simply subsets (not copies) of the elements of a region. Instead of computing these subsets at each task launch, Legion allows programs to name the subregions by creating *partitions* of regions [23, 25], in this case P (for primary) and G (for ghost) of the node region N of the graph. Partitions are arrays of subregions. An example primary partition P of a graph's nodes into three disjoint subregions is shown in Figure 2(a), where all the nodes of one color belong to the same subregion. The primary partition can be mapped across the machine (e.g., each subregion assigned to a different GPU) to enable parallel computation by tasks on the subregions. The ghost nodes for each subregion r of the primary partition are the nodes outside of r that are connected to r by at least one edge; these represent places where information must be exchanged between the pieces of the graph during the simulation. Figure 2(b) shows the ghost partition corresponding to the primary partition in Figure 2(a). For example, all nodes colored blue in 2(b) form the ghost subregion for the blue subregion of 2(a). We omit the partitioning of edges; the node computations are sufficient for illustration.

Note that the ghost partition of nodes is not a mathematical partition: it is not *complete* (some nodes are not included in any subregion) and some nodes have multiple colors because they are included in more than one subregion. Thus the ghost subregions are also not disjoint and the partition G is said to be *aliased*. Figure 2(c) shows an arrangement of regions and partitions for this program into a hierarchical structure called a *region tree* that captures their relationships: the set of all nodes (the root region N) has two partitions (represented by the triangles labeled P and G) each with three subregions representing subsets of data in N.

After constructing the primary and ghost partitions of the nodes, the program in Figure 1 enters a loop where it repeatedly alternates between two phases t1 and t2. The signatures of these tasks are given, including their effects on the region's *fields* (members of the struct Node): t1 reads and writes the up field of each node in its primary subregion and performs reductions (summations) to the down field of each node in its ghost subregion. Task t2 reverses the roles of the up and down fields.

It is easy to check that all three task calls on line 16 can execute in parallel. First, in each call to t1, the accesses through the primary partition and the ghost partition are touching different data because they refer to different fields up and down. The t1 tasks read and write their primary subregion, which can proceed in parallel because the primary subregions are disjoint. The ghost region accesses in different invocations of t1 may touch the same node's down field (since some nodes are included in more than one ghost subregion), but the accesses are reductions and so can be performed separately and the three sets of results combined later (e.g., when the field is read by another task). The justification that the t2 tasks (line 17) can run in parallel is symmetric.

Now consider what happens between lines 16 and 17. Task t2 on line 17 is reading values of the down field in the primary partition that were written by reductions to the ghost partition on line 16. Similarly, the reductions through the ghost partition on line 17 must be applied to the most recent

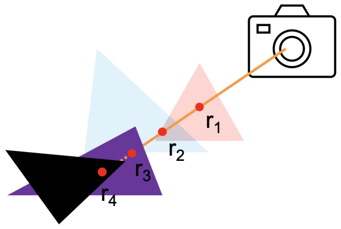


Figure 3. Rendering a 3D scene as a 2D image.

values of the up field written through the primary partition on line 16. Similar implicit communication occurs going from line 17 to line 16 of the next loop iteration. Complex communication patterns can arise when a task writes data using one partition and another task reads that data through a different partition; it is one of the strengths of the implicitly parallel model that the programmer only needs to identify the desired partitions of the data and not to explicitly manage the communication. It is the system's responsibility to guarantee *coherence*: that each read by a task t sees the correct, current version of the data produced by tasks preceding t in sequential program order regardless of where tasks are scheduled. Correctly relaxing this sequential order into a parallel schedule requires analyzing *dependences* between tasks. A dependence exists between two tasks that access the same data unless both tasks read that data or both perform reductions with the same associative operator.

As mentioned earlier, while Legion supports content-based coherence, most systems cannot support the program in Figure 1 as written because they are *name-based*, reasoning only about the names of regions and not their contents. Consequently all subregions must be pairwise disjoint to avoid aliasing (no data element may have more than one name), and therefore only a single disjoint partition of a region is supported [2, 8, 14, 16, 19, 29]. In our experience, name-based coherence will force at least one of two compromises when writing a program with multiple natural views (partitions) of the data. The program in Figure 1 could be written using only the primary partition. In this approach, communicating the ghost nodes requires communicating an entire piece of the graph, as the only units that can be communicated are named subregions; thus, one compromise is to name regions at the coarsest granularity required and accept that some tasks will communicate more data than necessary. A more efficient strategy creates a separate set of regions that hold only the ghost nodes; explicit copies manage the coherence of the ghost node regions with the primary partition, which forgoes the advantages of implicit communication.

3 The Visibility Problem

Visibility in computer graphics is the problem of determining which primitive objects (usually triangles) in a scene are visible to the camera and therefore should contribute to the rendered values of pixels [10, 13, 20]. Complications arise because one object may be in front of, or *occlude*, another from the perspective of the camera and some objects may be partially or wholly transparent. Figure 3 depicts the common case in graphics of rendering a 2D image from a 3D scene.

Rendering algorithms solve the visibility problem to compute a value for each pixel in an output image as a function of the primitives in the scene and the position of the camera. Only the nearest objects to the camera ultimately contribute to the value of a pixel, unless some objects are at least partially transparent as seen in the blue and red triangles in Figure 3. The degree of transparency for an object is its *alpha* value, with 0 fully transparent, 1 fully occluding, and values in between representing fractional transparency. The process of accumulating the effects of multiple semi-transparent primitives to a pixel is referred to as *alpha blending*.

The basic visibility problem is computing what is visible along a line. Assuming primitive objects are triangles, and ignoring degenerate cases, each triangle in the scene intersects with the line at either zero or one points. The triangles that share a point with the line can then be ordered by where those points appear along the line, as depicted in Figure 3. Here four triangles in the scene intersect with a line from the camera at points r_1 to r_4, with r_1 closest to the camera. The value of the pixel defined by the view along the line from the camera is $v = b(r_1, \alpha_1, b(r_2, \alpha_2, b(r_3, \alpha_3, b(r_4, \alpha_4, 0))))$, where b is the blending function, α_i is the alpha value of the i-th triangle, and 0 is the identity for b (i.e., $b(x, \alpha, 0) = \alpha x$). Since the point r_3 on the purple triangle is occluding ($\alpha_3 = 1$), the black triangle is not visible along the line; i.e., $v = b(r_1, \alpha_1, b(r_2, \alpha_2, b(r_3, 1, 0)))$.

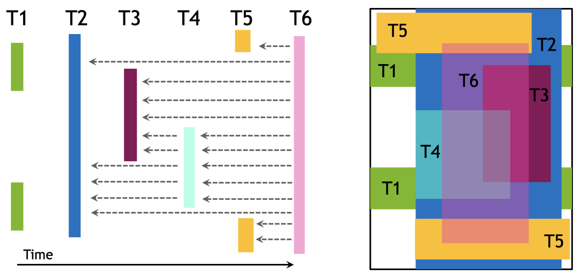
3.1 Coherence

We reduce the coherence problem (content-based or name-based) to the visibility problem in computer graphics. Consider a single element v of a region, such as a point in an N-dimensional array or dataframe. We assume there is a global clock that assigns a time t to each operation o performed by any task on v. Let $\langle o_1, t_1 \rangle, \ldots, \langle o_n, t_n \rangle$ be that sequence of operations ordered by increasing t_i. The three possible operations are a write w_x, which assigns x to v, a reduction f^i_x which applies the reduction operator f^i to the current value of v and x, and a read r, which reads the current value of v. We define a blending function B on this sequence:

$$\begin{aligned} B(\langle o_1, t_1 \rangle, \ldots, \langle o_n, t_n \rangle, v) &= B(\langle o_2, t_2 \rangle, \ldots \langle o_n, t_n \rangle, b(o_1, v)) \\ b(w_x, v) &= x \\ b(f^i_x, v) &= f^i(x, v) \\ b(r, v) &= v \end{aligned}$$

Consider a read $\langle r, t_i \rangle$. Then $B(\langle o_1, t_1 \rangle, \ldots, \langle o_{i-1}, t_{i-1} \rangle, 0)$ is the value of v at time t_i, which is visibility on the line extending backwards in time along v.

Visual transparency has a direct analog in coherence: reads are fully transparent, reductions are partially transparent



(a) Task time-line (b) Task updates

Figure 4. Visualization of content-based coherence.

(the "degree" of transparency depends on the reduction operator), and writes are fully opaque. The execution of a program of tasks using these operations on arbitrary subsets of a region induces a space-time volume upon which coherence must be analyzed. Our reduction shows the direct correspondence between visibility in graphics and coherence in task-based systems: one computes visibility in spatial dimensions, the other computes visibility in the temporal dimension.

Figure 4(a) shows a timeline of six tasks and Figure 4(b) shows the corresponding portion of an array that each task touches with updates stacked in order of time, most recent updates on top. All updates are writes (depicted as solid boxes) except for tasks T4 and T6, which perform reductions (partially transparent boxes).

When task T6 executes, T6's input data is assembled from the most recent tasks to update it. Figure 4(b) shows that T6 needs some data written by T5, T4, T3 and T2, but not T1, whose updates are occluded by T2. In Figure 4(a), the dotted arrows show this "looking back in time" to find the tasks that wrote each piece of the data needed by T6.

3.2 Dependence Analysis

The reduction presented above uses a global notion of time, which corresponds to the sequential execution order of tasks. Dependence analysis is needed to relax this sequential order to a partial (parallel) order on the execution of tasks such that the coherence of reads is still guaranteed.

Recall that we distinguish between *content-based coherence* and *name-based coherence*. In a name-based scheme, regions must be disjoint and there can only be a dependence between two tasks $s(X)$ and $t(Y)$ if $X = Y$. In content-based coherence, regions can overlap and there can be a dependence only if $X \cap Y \neq \emptyset$. Content-based coherence also easily supports irregular and sparse regions; the set of elements in a region need not be contiguous subranges of elements. The main cost of content-based coherence is discovering which pairs of regions have non-empty intersections.

Just because two tasks $s(X)$ and $t(Y)$ share data does not mean there is a dependence between them; there exists a *dependence* only if reversing the execution order of the tasks could lead to incorrect results. Each task declares whether it reads, reads and writes, or performs reductions with a specific reduction operator (such as summations) on its region arguments. The Legion runtime conservatively estimates whether there is a dependence between two tasks using the tasks' privileges [5]; we omit the details.

```
1       t1(P[1],G[1])   # t0
2       t1(P[2],G[2])   # t1
3       t1(P[3],G[3])   # t2
4       t2(P[1],G[1])   # t3
5       t2(P[2],G[2])   # t4
6       t2(P[3],G[3])   # t5
7       t1(P[1],G[1])   # t6
8       t1(P[2],G[2])   # t7
9       t1(P[3],G[3])   # t8
```

Figure 5. The initial sequence of tasks from `main` in Figure 1.

```
# S is the state of the runtime system
run_task(T(P_1 R_1,...,P_n R_n),S)
   foreach P_i R_i
      R_i,S := materialize(P_i,R_i,S)
   R_1,...,R_n := T(R_1,...,R_n)
   foreach P_i R_i
      S := commit(P_i,R_i,S)
   return S
```

Figure 6. Task execution.

Figure 5 shows the first nine tasks executed by the `main` task in Figure 1. Because our analyses are dynamic, the runtime system observes a sequence of task launches that it analyzes for dependences and coherence. Consider task t_6 on line 7 of Figure 5, the first task called the second time the loop on line 16 of Figure 1 is entered. To calculate what tasks t_6 depends on and to compute coherent versions of t_6's arguments, the runtime "looks" backwards in time to observe the effects of previous tasks on the regions of data t_6 references. In this case t_6 has a dependence on tasks t_3, t_4, and t_5 because t_6 reads a subregion from the primary partition that may overlap with reductions by these previous tasks to the ghost partition. In turn t_3 has dependences on t_0, t_1, and t_2 because t_3 adds reductions to values written by those tasks. Tasks t_4 and t_5 also have dependences on t_0, t_1 and t_2 for the same reason.

From this example we can see that dependence analysis is a subset of the coherence problem. Dependence analysis only requires the system to prove which tasks have touched the data another task may access, but it does not need to know the actual values of the elements, just that there could be elements in common. For example, in the task stream in Figure 5, the system will discover that there are no dependences between tasks t_{0-2}, t_{3-5}, and t_{6-8}, allowing those groups of tasks to execute in parallel. With coherence we must go one step further and compute the current values of the elements in common. Thus, our algorithms for coherence will also provide solutions for dependence analysis.

4 Preliminaries

Before presenting our visibility algorithms, we first define a common framework for their presentation. Figure 6 defines a function `run_task` that takes a task call $\mathsf{T}(\mathsf{P}_1\ \mathsf{R}_1, \ldots, \mathsf{P}_n\ \mathsf{R}_n)$ and the current state S of the runtime system. Here $\mathsf{P}_\mathtt{i}$ is the *privilege* that T has on region $\mathsf{R}_\mathtt{i}$ (see below). Each visibility algorithm must provide two functions `materialize` and `commit` together with an implementation of S. To explain `materialize` and `commit` we need additional details of tasks:

- A *region* R is a set of pairs $\{\langle i, v\rangle\}$ where i is an n-dimensional point and v is the value of the region at that point. Note that this definition limits regions to a single field. Each first component i can appear at most once in the set.
- A program uses a single region A. (Actual systems allow any number of regions as well as multiple fields per region, but this simpler setting is sufficient to illustrate the important ideas.) The domain $\mathsf{dom}(\mathsf{A}) = \{\mathtt{i} \mid \langle \mathtt{i}, \mathtt{v}\rangle \in \mathsf{A}\}$ is the set of A's first components.
- Reduction operators f must have an identity $0_\mathtt{f}$ to support partial accumulation.[1] For example, `+=` is a reduction operator with the identity 0.
- Each privilege $\mathsf{P}_\mathtt{i}$ in a task call is one of `read`, `read-write`, or $\mathtt{reduce}_\mathtt{f}$, where f is the reduction operator (e.g., a summation has the privilege $\mathtt{reduce}_+$). Two privileges *interfere* if two tasks with those privileges could have a dependence. The only non-interfering combinations of privileges are `read`/`read` and $\mathtt{reduce}_\mathtt{f}/\mathtt{reduce}_\mathtt{f}$, that is, two reductions with the same operator.
- Any region arguments R and R′ to a task must have disjoint domains $\mathsf{dom}(\mathsf{R}) \cap \mathsf{dom}(\mathsf{R}') = \emptyset$ unless the task only reads both regions or only reduces to both regions with the same reduction operator[2].

A region argument $\mathsf{R}_\mathtt{i}$ to a task only names what data is needed: $\mathsf{dom}(\mathsf{R}_\mathtt{i})$ is the set of indices and it is up to the runtime system to fill in the correct values. The function `materialize` takes a region, its privilege, and the current runtime state and returns a region with the same domain and current values; `materialize` may also update the state (line 5 of Figure 6). Running a task calls the function on the region arguments, which may update those regions (line 6). The function `commit` records information needed to materialize correct region arguments for future task calls (line 8).

5 The Painter's Algorithm

In the next three sections we present three coherence algorithms based on visibility algorithms from computer graphics. Each algorithm refines the previous one, so the most sophisticated and best performing method, ray casting, reuses concepts developed for the other two.

In computer graphics, the painter's algorithm solves the visibility problem by rendering objects in a scene from back to front [17]. For a given pixel p, rendering a semi-transparent object (in front of all objects already rendered) accumulates onto p and rendering a fully occluding object overwrites p. The painter's algorithm for content-based coherence is in Figure 7. The code uses three auxiliary functions:

```
1     # S is a history: a list of (privilege, region) pairs
2     paint(R,S)
3     # The history is traversed from oldest to newest
4         for ⟨P′,R′⟩ in S
5             if P′ = read-write then
6                 R := (R ⊕ R′)/R
7             else if P′ = reduce_f then
8                 R := R ⊕ f(R/R′, R′/R).
9             # do nothing if P′ = read
10        return R,S
11
12    materialize(P,R,S)
13       # R[i] is initially undefined for all i in dom(R)
14       if P = reduce_f then
15           return {⟨i,0_f⟩| i ∈ dom(R′)}
16       else
17           return paint(R,S)
18
19    commit(P,R,S)
20       return S ++ ⟨P,R⟩
```

Figure 7. The painter's algorithm.

$$\begin{aligned} X/Y &= \{\langle i, v\rangle \in X \mid i \in \mathsf{dom}(Y)\} \\ X\backslash Y &= \{\langle i, v\rangle \in X \mid i \notin \mathsf{dom}(Y)\} \\ X \oplus Y &= X\backslash Y \cup Y \end{aligned}$$

In words, X/Y is the subset of X sharing points with Y, $X\backslash Y$ is the subset of X not sharing points with Y, and $X \oplus Y$ is the union of X and Y using Y's values for $\mathsf{dom}(X) \cap \mathsf{dom}(Y)$.

The state S in Figure 7 is a list of privilege-region pairs $\langle \mathsf{P}, \mathsf{R}\rangle$. The initial state is a list with one pair $[\langle \mathtt{read\text{-}write}, \mathsf{A}\rangle]$, which is the initial value of A. When a task finishes, `commit` appends the final state of its region arguments to S. Thus, the state is a *history* of the results of operations in order of increasing time—all the result pairs for the ith task call in program order are listed before the results of task $i + 1$ and after the results of task $i - 1$.[3]

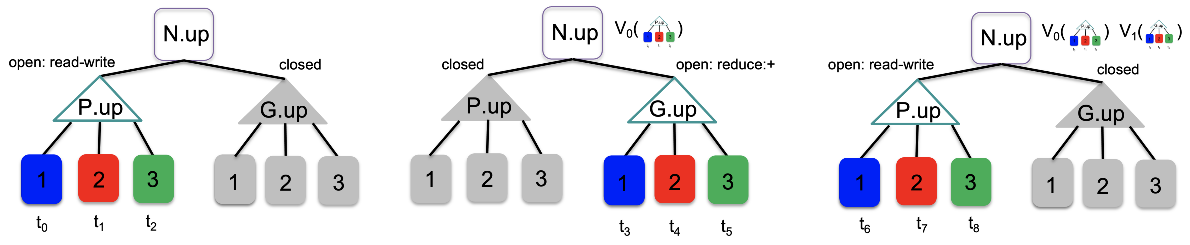
The `paint` function computes the most recent state of a region R by traversing the history and applying operations in order to R. For each privilege-region pair (line 3), if the privilege is `read-write`, then the overwritten portion of R is replaced while the portion that is not updated is retained (line 5). A $\mathtt{reduce}_\mathtt{f}$ is similar, except the updated portion is folded into R using the reduction operator (line 7). Note we lift reduction operators f pointwise to pairs of regions: $f(X, Y) = \{\langle i, v\rangle \mid \langle i, v_x\rangle \in X \wedge \langle i, v_y\rangle \in Y \wedge f(v_x, v_y) = v\}$. No modifications to R are needed for a `read` operation.

The `materialize` function behaves differently depending on the privilege of the operation to be performed on R. For a reduction the history is not examined at all and the region is initialized to the identity of the reduction operator—the reductions are accumulated locally and only applied in future

[1]There are interesting optimizations when reduction operators are commutative and/or associative, but they are beyond the scope of this work.

[2]Handling the case of $\mathsf{dom}(\mathsf{R}) \cap \mathsf{dom}(\mathsf{R}') \neq \emptyset$ with interfering privileges requires intra-task coherence that is beyond the scope of this work.

[3]Because of the restrictions on region argument aliasing it does not matter what the order is for the region arguments to a single task call.



(a) Tasks t_{0-2} (b) Tasks t_{3-5} (c) Tasks t_{6-8}

Figure 8. The region tree state after processing the task launches in Figure 5.

calls to `materialize`. This lazy application of reductions is important, because it minimizes data movement: eagerly performing reductions requires materializing the current value of the region, which copies data to the location of the task before it begins execution, while accumulating partial reductions only moves data when the result is needed [24]. In contrast, for a `read` or `read-write` operation the system must materialize the current values of the region.

5.1 Optimizations and Implementation

The algorithm in Figure 7 is simple but inefficient. When materializing a subregion R, the naive painter's algorithm requires testing every operation in the history for overlap with R. A more efficient approach is to use the region tree as an acceleration data structure: We store histories in the region tree such that the history relevant to a region R can be found along the path from the region tree's root to R. More specifically, each node R of the region tree maintains a subhistory R.h, and we guarantee that to materialize R, materializing the *path history* which is the concatenation of all the histories on the path from the root to R, in that order, yields the same result as the naive painter's algorithm. When a task t with region argument R and privilege p is launched:

1. $\langle t, p \rangle$ is appended to R.h.
2. Let R′ be an ancestor of R with child C. If $C \cap R \neq \emptyset$, then any tasks already recorded in C's subtree must precede $\langle t, p \rangle$ in the path history. We append a snapshot of C's subtree of task histories, called a *composite view*, to R′.h. We delete the task histories in C since they are now recorded at C's parent R′.

When traversing a path history, composite views are traversed (including any nested composite views) before continuing with the next element of the path.

We can reduce the number of composite views by recording at every node R whether it is *open* (there are non-empty histories in R's subtree) or *closed* (all histories in R's subtree are empty). An additional improvement is to record the privileges used in R's subtree at R. When performing step (2) above, we can then skip creating composite views for subtrees that are closed or only have histories with privileges that do not interfere with $\langle t, p \rangle$. Finally, often a composite view occludes an earlier composite view v', allowing v' to be deleted from the history; we omit the details of a conservative test to identify such occlusions for lack of space.

Figure 8 gives an example using the sequence of tasks in Figure 5. The indices in the figure indicate the step at which the task or composite view is added to the region tree. When t_0 is launched there are no previous tasks so the task is recorded in the history at P.up[1]. When t_1 is launched, we record t_1 at P.up[2]. No composite view is created because P is a disjoint partition, so $P.up[1] \cap P.up[2] = \emptyset$. The launch of t_2 is similar; Figure 8(a) shows the region tree state after t_{0-2} have been recorded. Task t_3 uses the G.up[1] subregion and so takes a different path through the region tree with different reduction privileges. Because P.up and G.up overlap and the `read-write` privileges used in `P.up` interfere with the reduction privilege used by t, a composite view V_0 of the subtree at `P.up` is appended to the root's history. Even though G.up is an aliased partition, t_4 and t_5 use the same reduction privilege as t_3 and so these tasks are simply added to the histories of G.up[2] and G.up[3], respectively. The region tree state after the launch of t_{0-5} is shown in Figure 8(b). When t_6 is launched, another composite view V_1 of G.up is appended at the root. Figure 8(c) shows the state of the region tree after tasks t_{0-8} have been analyzed.

Both region trees and composite views are distributed data structures with partial components living on different nodes for scalability reasons. The number of subregions in region trees and composite views is often directly proportional to the number of nodes in the machine and therefore storing any such data structure on a single node is likely to be prohibitively expensive. We construct composite views bottom up, with minimal communication between nodes to capture the state of the uppermost levels. Since composite views are immutable after construction, we can safely replicate nodes of composite views across the machine on demand.

6 Warnock's Algorithm

A disadvantage of the painter's algorithm is that even with optimizations, materializing a region R may result in testing R against many history entries (especially composite views) that are no longer visible due to subsequent updates. Warnock's algorithm, given in Figure 9, avoids considering updates irrelevant to materializing a region. It does this by performing a spatial decomposition to address the visibility

```
1     # S is a set of equivalence sets.
2     # An equivalence set is a (region,history) pair.
3     refine(R,S)
4        $S' := \emptyset$
5        for $\langle R', H\rangle$ in S
6           if $dom(R') \cap dom(R) = \emptyset$ then
7              $S' := S' \cup \{\langle R', H\rangle\}$
8           else if $dom(R) = dom(R')$ then
9              $S' := S' \cup \{\langle R', H\rangle\}$
10          else
11             $S' := S' \cup \{\langle R'/R, H\rangle, \langle R' \backslash R, H\rangle\}$
12       return $S'$
13
14    materialize(P,R,S)
15        $S' := refine(R, S)$
16        $Es := \{\langle X, H\rangle \in S' \mid dom(X) \subseteq dom(R)\}$
17        $R := \emptyset$
18        for $\langle R', H\rangle$ in Es
19           if P = reduce$_f$ then
20              $X := \{\langle i, 0_f\rangle \mid i \in dom(R')\}$
21           else
22              $X := paint(R', H)$
23           $R := R \cup X$
24        return $R, S'$
25
26    commit(P,R,S)
27        $S' := \emptyset$
28        for $\langle R', H\rangle$ in S
29           if $R'/R = R'$ then
30              if P = read-write then
31                 $S' := S' \cup \langle R', \langle P, R/R'\rangle\rangle$
32              else
33                 $S' := S' \cup \langle R', H \mathbin{+\!+} \langle P, R/R'\rangle\rangle$
34           else # refine guarantees $dom(R) \cap dom(R') = \emptyset$
35              $S' := S' \cup \langle R', H\rangle$
36        return $S'$
```

Figure 9. Warnock's algorithm.

problem [27]. The algorithm divides the scene into parts and then determines the number of primitives visible in each sub-scene. The divide-and-conquer approach continues recursively until the number of primitives visible in each sub-scene is trivial (one or a small number) or the sub-scene has been refined down to a single pixel. Once the scene has been segmented into simple sub-scenes, each sub-scene is easily rendered independently.

The state S is now a set of *equivalence sets* consisting of a pair of a region R and a history H with the property that every operation in H is *relevant* to every element of R: if $\langle P', R'\rangle \in H$, then $dom(R) \subseteq dom(R')$. Initially there is one equivalence set, the global collection A with the history $[\langle \texttt{read-write}, A\rangle]$. Any two equivalence sets are always disjoint, and the union of all equivalence sets always covers A.

When a task is launched, the function `refine` splits equivalence sets as necessary to maintain the relevant invariant that each equivalence set represents a set of points with the same history. If a task works on a region R that has a non-trivial overlap with an existing equivalence set R′, then R′ is split into two equivalence sets consisting of R′/R and R′\R (line 11). We then apply the painter's algorithm to each equivalence set (lines 18-24) to build up the materialization of R. The spatial separation of the histories is seen in the `commit` function, which appends a task's operation on region R only to the equivalence sets that make up R. In the case that the task is writing, it can clear the prior history of the

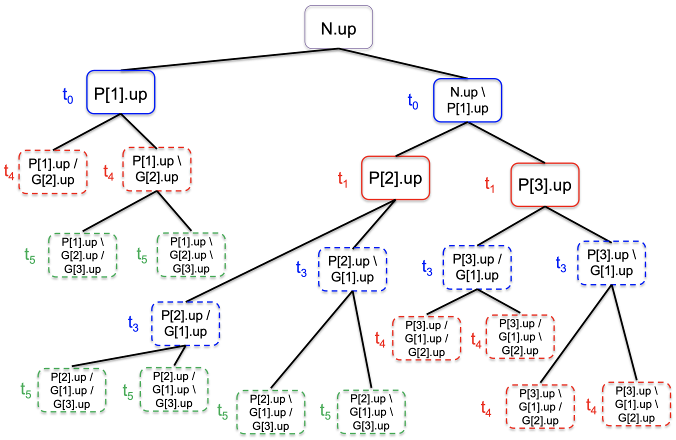

N.up
t_0
P[1].up
t_0
N.up \ P[1].up
t_4
P[1].up / G[2].up
t_4
P[1].up \ G[2].up
t_1
P[2].up
t_1
P[3].up
t_5
P[1].up \ G[2].up / G[3].up
t_5
P[1].up \ G[2].up \ G[3].up
t_3
P[2].up \ G[1].up
t_3
P[3].up / G[1].up
t_3
P[3].up \ G[1].up
t_3
P[2].up / G[1].up
t_4
P[3].up / G[1].up / G[2].up
t_4
P[3].up / G[1].up \ G[2].up
t_5
P[2].up / G[1].up / G[3].up
t_5
P[2].up / G[1].up \ G[3].up
t_5
P[2].up \ G[1].up / G[3].up
t_5
P[2].up \ G[1].up \ G[3].up
t_4
P[3].up \ G[1].up / G[2].up
t_4
P[3].up \ G[1].up \ G[2].up

Figure 10. The equivalence sets created by Warnock's algorithm for the task launches in Figure 5

equivalence set and then store itself as the only entry in a new history, thereby ensuring that histories are precise and only contain the most recently "visible" tasks (lines 30-31).

Figure 10 gives an example of the equivalence set tree created by Warnock's algorithm for the sequence of task launches shown in Figure 5. We show just the sequence of equivalence set refinements for the up field. Each equivalence set q is labeled with the subset of `N.up` that q represents as well as the task t that caused q to be added to the refinement tree. A solid outline of a node indicates that a task's region argument is from the P partition, while a dashed line indicates a region argument from the G partition (recall Figure 2).

Initially `N.up` is the only equivalence set. Task t_0 has privileges on `P[1].up` which causes the root to be refined into N.up/P[1].up = P[1].up and N.up\P[1].up. Task t_1 has privileges on `P[2].up`, which is disjoint from `P[1].up`, so only node N.up \ P[1].up is refined into N.up \ P[1].up / P[2].up = P[2].up and N.up \ P[1].up \ P[2].up = P[3].up. After tasks t_1 and t_2, Warnock's algorithm has discovered the subregions of the P partition, so task t_2, with privileges on `P[3].up`, does not add additional equivalence set nodes.

The tasks t_{3-5} work on the ghost partition subregions G[1 − 3]. Task t_3 takes G[1].up as an argument. Recall that G[1].up is the set of ghost nodes for P[1].up, which means that G[1].up may share nodes with P[2].up and P[3].up (again, see Figure 2). Thus t_3 causes P[2].up and P[3].up to be split into those elements that are and are not ghost cells in G[1].up. Task t_4 takes G[2].up as an argument, which causes the then-leaf nodes in the subtrees under P[1].up and P[3].up to be further refined. Finally task t_5 takes G[3].up as an argument, which causes the final refinements in the subtrees under P[1].up and P[2].up. Note that not all equivalence sets computed in the subtree under P[1].up are shown because P[1].up/G[2]/G[3] and P[1].up/G[2] \ G[3] are empty. Task t_5 is the last task of the first iteration of the loop in Figure 1. Each subsequent iteration uses the same regions, so no further refinements are needed to the tree in Figure 10.

6.1 Optimizations and Implementation

As in the previous section, we can greatly improve the performance of Warnock's algorithm with some optimizations. The most important challenge is speeding up determining which equivalence sets represent a given region (line 16 of Figure 9). Since each equivalence set is always refined into new equivalence sets that cover the original equivalence set, the history of refinements defines a search tree that is a bounding volume hierarchy (BVH) [9]. For any new subregion R to discover its constituent equivalence sets, it starts at the root node of the equivalence set BVH, tests for intersections with any child nodes, and recursively traverses any overlapping children. The set of leaf nodes reached are exactly the most recent equivalence sets needed for performing the dependence and coherence analyses for R at that point in the execution of the program. After performing this initial traversal, we can memoize the equivalence sets that compose R. Since Warnock's algorithm only refines equivalence sets, the next task that uses R later in the program can begin the search for R's current constituent equivalence sets at each of the memoized equivalence sets.

The equivalence set BVH is a distributed data structure. The only mutable state in the BVH is the histories stored at the leaves, so we can safely replicate intermediate nodes in the BVH throughout the machine since the names of any child nodes are immutable after an equivalence set is refined. Replication is crucial to avoiding sequential bottlenecks at scale when many nodes are attempting to execute tasks that all need to traverse the upper levels of the BVH to identify constituent equivalence sets for their subregions.

7 Ray Casting

Historically, the painter's algorithm was the first coherence algorithm we implemented, but it had scalability issues due to the inability to precisely prune occluded elements of histories, and so subsequently our adaptation of Warnock's algorithm was developed. Warnock's algorithm can also lead to different scalability issues when too many equivalence sets are created (see Section 8). Our realization that these algorithms are related to visibility in computer graphics, which was not apparent earlier, motivated investigating whether ray casting, the preferred approach in modern rendering [28], could work better than the other approaches.

In ray casting, a ray is cast from each pixel of the image into the scene, recording the objects that it passes through (if semi-transparent) and the object that eventually blocks it. The effects of those intersections are then accumulated to compute the final value of the pixel[4] (recall Figure 3).

The difference between content-based coherence using ray casting and Warnock's algorithm involves the handling of writes. With Warnock's algorithm, equivalence sets are only refined into smaller equivalence sets. While this makes it easy to discover which equivalence sets represent a subregion, it often leads to many equivalence sets with identical histories after a write is performed. In the ray casting approach, every task t performing a write for a region R creates a new equivalence set representing the points in R with a history initially containing just ⟨read-write, R⟩. After creating a new equivalence set R for a write, we prune all equivalence sets that are occluded by R. Figure 11 uses the `materialize` and `commit` routines from Warnock's algorithm. The new function `dominating-write` creates a new equivalence set for R and prunes the equivalence sets occluded by R (line 2).

```
dominating_write(R,S)
    S′ := {⟨R, [⟨read-write, R⟩]⟩} ∪ {⟨R′,H⟩ ∈ S| dom(R) ∩ dom(R′) = ∅}
    return S′

materialize(P,R,S)
    R′, S′ := warnock::materialize(P,R,S)
    if P = read-write then
        S′ := dominating_write(R′,S′)
    return R′,S′

commit(P,R,S)
    return warnock::commit(P,R,S)
```

Figure 11. Ray casting algorithm.

The new handling of writes means that equivalence sets may be combined as well as refined. Ray casting serves as the primary mechanism for addressing two problems caused by these now non-monotonic changes in the collection of equivalence sets compared to Warnock's algorithm. To prune equivalence sets in the `dominating-write` function, we use ray casting to discover the equivalence sets that have been occluded by the region R (line 2 of Figure 11). Additionally, we use ray casting to discover the constituent equivalence sets of region R in each invocation of the `materialize` function (line 16 of Figure 9 via line 6 of Figure 11) because the collection of equivalence sets could change after the invocation of the `materialize` function for each task.

Ray casting produces the same equivalence sets shown in Figure 10 for the tasks $t_1 - t_5$ of Figure 5, but note that these equivalence sets are stored at the leaves of the P partition, which is the only disjoint and complete partition in the example in Figure 1. The first task of each loop has `read-write` privileges on P[1].up. The write privilege causes any refinements and their histories of P[1] to be discarded, reducing the number of equivalence sets and simplifying the history until the tasks that operate on ghost regions are called again. The tasks with read-write privileges on P[2].up and P[3].up similarly coalesce equivalence sets and simplify the history.

7.1 Optimizations and Implementation

To accelerate ray casting, practical implementations in computer graphics rely on a BVH decomposition of the scene to efficiently detect the objects a ray intersects [9]. Because equivalence sets are coalesced by writes, there is no longer

[4]In graphics, when shading surfaces recursive "shadow" rays are also cast to sample the illumination at a surface point. There is no analog of shadow rays in our case; we only need to determine the visible tasks.

a stable BVH based on equivalence sets. Instead we use a heuristic based on which partitions tasks are using to select a subtree of the root region with only disjoint and complete partitions, which naturally defines a BVH. If the application switches to using a different subtree with disjoint-complete partitions, the runtime shifts the equivalence sets to the new subtree. In rare cases when no subtree with disjoint-complete partitions exists, the runtime creates a K-d tree [6]. While the construction of the BVH data structure based on a region subtree is different, the remainder of the implementation of ray casting is quite similar to Warnock's algorithm.

8 Performance Evaluation

Over a decade we have implemented and tuned all three algorithms in the Legion runtime. Each implementation was tested on many of the top supercomputers in the world at the time of development. To conduct experiments comparing the performance across approaches, we backported a recent version of Regent [21], a programming language that targets Legion, to all three versions of the Legion runtime. Similarly, we ran all experiments with a backported, recent version of Realm [24], the low-level portability layer that sits beneath Legion, in order to ensure as uniform behavior as possible.

No other current task-based runtime system fully supports content-based coherence, so it is not possible for us to compare with other runtime systems (see Section 9). On the other hand, having all three approaches implemented and tuned on a common platform also eliminates potentially significant but orthogonal differences in performance, making our evaluation of the algorithms directly comparable.

We consider three benchmark codes. The first is a 2-D stencil computation that computes a 9-point[5] stencil on a structured, regular grid of cells intermixed with data-parallel computations [26]. The second benchmark is a graph-based circuit simulation that iteratively models the behavior of circuit elements as edges between nodes with different voltages [22]. The structure of the graph is irregular and induces different communication patterns between each subset of the graph. The graph computation also makes use of reductions to describe parallel updates to voltages from different circuit elements contributing to the same voltage node. The skeleton program in Figure 1 is derived from this benchmark. The final benchmark is the Pennant mini-application [12]. Pennant is a 2-D Lagrangian hydrodynamics code for unstructured meshes. Pennant also performs reductions to handle updates in parallel, and has several distinct reduction operators that are used in different parts of the code. Each benchmark has been tuned and tested for scalability in prior work.

The experiments were run on the Piz Daint supercomputer [1] with one Legion process per node. All tasks are mapped to the single GPU on each node. We did not use Legion's tracing [15], which memoizes the dependence and coherence analyses. Therefore, these experiments do not measure Legion's peak performance, but rather measure the performance of the different coherence algorithms. For the ray casting and Warnock's algorithm implementations, we also perform experiments using Legion's novel *dynamic control replication* (DCR) [4]. (Unfortunately the painter's algorithm implementation predates a stable implementation of DCR.) The purpose of DCR is to shard the work of a single task across multiple nodes, which is important for tasks that launch a large number of subtasks. Consider the `while` loop in Figure 1, which repeatedly launches a number of subtasks proportional to the size of the machine. Since tasks are internally sequential (parallelism is between tasks, not within tasks), such a task becomes a sequential bottleneck at scale; applying DCR to the task essentially transforms it into an SPMD-style execution, with each shard of the task handling a subset of the task launches. DCR stresses our implementations by distributing the source of coherence and dependence analyses across the nodes.

For each benchmark, we measured the performance of two phases of execution. The first phase is the *initialization* time, which is all the time from application start to the end of the first iteration of the application's top-level loop. The performance of the initialization phase measures Legion's ability to discover and analyze the coherence requirements of the application, either in the form of making initial composite views for the painter's algorithm or constructing BVH and equivalence set data structures for Warnock's algorithm and ray casting. The more complex the partitions and communication patterns, the more initialization time may be required. The second phase is the steady-state performance of the remainder of the computation; since all of the benchmarks execute a repetitive loop with no changes to the structure of partitions or task dependences, once the initial analysis is done the performance stabilizes. This phase measures the weak-scaling capabilities of our implementations.

8.1 Initialization Performance

Figures 12, 13, and 14 show the wall clock initialization time for the Stencil, Circuit, and Pennant benchmarks, respectively. The most striking feature of these plots is the universally poor performance of equivalence sets based on Warnock's algorithm from Section 6, which is unable to scale beyond 128 to 256 nodes. Because Warnock's algorithm always refines overlapping equivalence sets, in the worst case it can generate a number of equivalence sets equal to the powerset of the number of regions used by a program. None of our benchmarks exhibits such exponential behavior, but the superlinear nature of the approach still explodes the number of equivalence sets that must be constructed, ultimately dooming scalability. Running with DCR slightly mitigates this behavior by better distributing the work of refining equivalence sets across all the nodes in the machine, but the equivalence set explosion still dominates in the limit.

[5]Two cells in each direction from the center, excluding "corner" cells.

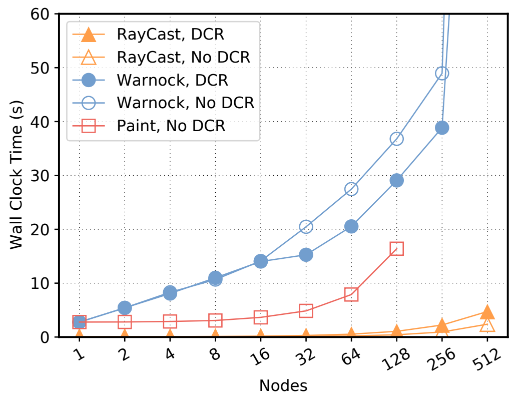


Figure 12. Stencil initialization time.

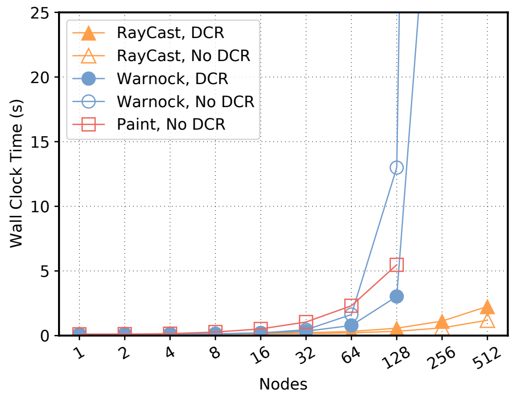


Figure 13. Circuit initialization time.

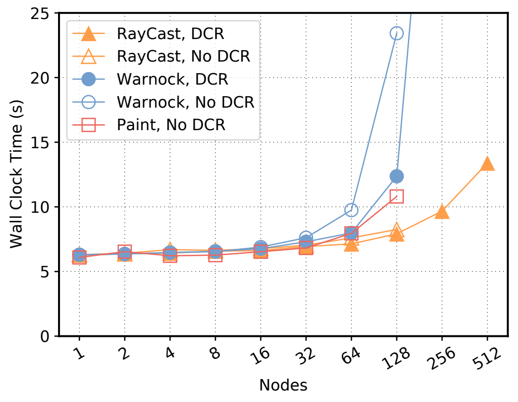


Figure 14. Pennant initialization time.

The performance of composite views based on the painter's algorithm from Section 5 scales roughly linearly with the number of nodes (note these are log-linear plots), which results in the initialization phase not scaling past 128 nodes. This outcome is the result of two interacting factors. First, while composite views are distributed data structures, they still have one root. Without DCR, the construction and traversals of the root is done on a single machine. At small node counts the effect is negligible, but at scale having one machine handling communication from every other node is a sequential bottleneck, which causes the analysis time to grow linearly with the size of the machine. This effect is observed in the initialization phase for all three benchmarks.

The ray casting approach from Section 7 easily performs the best. Ray casting is already very well distributed even without DCR, even demonstrating lower constant-time overheads compared to DCR in the case of Stencil and Circuit benchmarks. There is still linear growth in the initialization phase of the computation for ray casting with or without DCR, but rate of growth is much smaller than Warnock's algorithm (with or without DCR) and significantly smaller than the painter's algorithm (without DCR).

8.2 Weak Scaling Performance

Figures 15, 16, and 17 show weak scaling performance of the Stencil, Circuit, and Pennant benchmarks. To measure weak scaling performance, we increase the size of the problem being solved proportionately to the number of machine nodes used. The worst-performing approach was the painter's algorithm. In all cases, the painter's algorithm starts to tail off between 16 and 32 nodes because the number of children to examine for interference in each composite view grows with the size of the machine, which quickly comes to dominate the overall performance of the iterative phases of execution. While all three algorithms see large drops in performance at some point without DCR, the painter's algorithm experiences the loss of performance on smaller machines.

Warnock's algorithm performs better, scaling out to 32 to 128 nodes before noticeable performance degradation begins to occur without DCR, and scales out to at least 256 nodes for all three benchmarks with DCR. Since all three benchmarks do not change their partitioning schemes after the initialization phase, no equivalence sets undergo refinement during the iterative phase of execution. The scaling for Warnock's algorithm is then purely determined by the cost of performing the analyses on the existing equivalence sets. Without DCR, all analyses originate on a single node and must be communicated to equivalence sets on remote nodes, ultimately leading to a sequential bottleneck and decreased scaling. With DCR, Warnock's algorithm achieves good weak scaling since the coherence and dependence analyses are distributed across all the nodes and can traverse the already-computed equivalence sets in parallel.

Ray casting again performs the best of the three approaches. In all cases, its performance is slightly better than Warnock's algorithm, as its maintains fewer total equivalence sets in its lists by coalescing writes. Without DCR, ray casting suffers from the same sequential bottleneck that Warnock's algorithm does: all analyses originate on a single node and must be communicated to other nodes. With DCR, the analyses again are distributed across all the nodes equally, enabling parallel traversals of the list of equivalence sets and the equivalence sets themselves, yielding better scalability.

The results of these experiments show why ray casting has been adopted as the current approach in the Legion runtime:

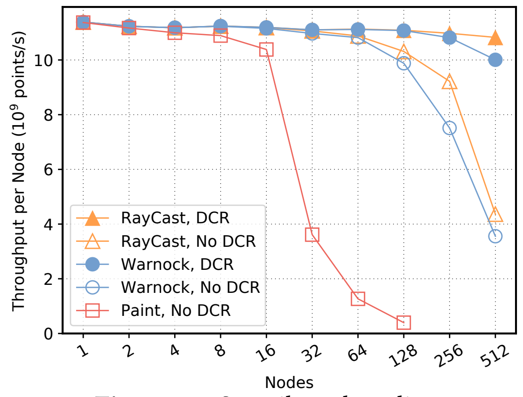


Figure 15. Stencil weak scaling.

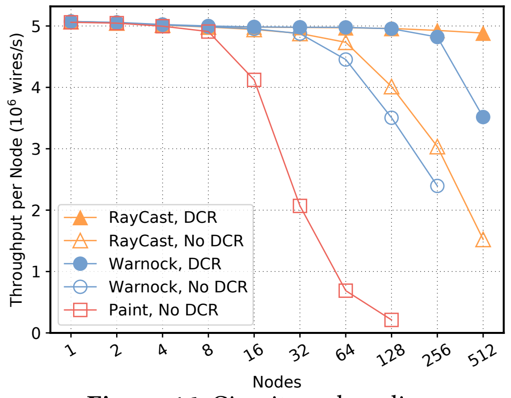


Figure 16. Circuit weak scaling.

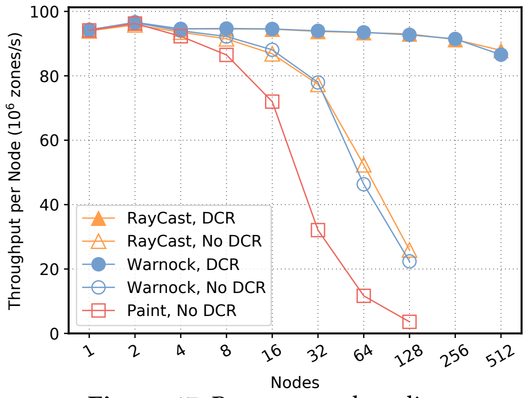


Figure 17. Pennant weak scaling.

it delivers the best performance for both initialization and steady-state execution.

9 Related Work

The closest related work on content-based coherence is in static-analysis based programming models. Sequoia [11], Deterministic Parallel Java [7], and Halide [18] are examples of compilers that analyze dependences by reasoning about non-trivial relationships between sets of data. These programming models are restricted to ensure the analyses can be done precisely at compile time, and they do not handle data-dependent control flow, dynamic creation and destruction of data, unbounded-size outputs, or dynamic load balancing.

StarPU is a task-based runtime system that supports dynamic discovery of parallelism [2]. Coherence in StarPU is name-based. Only one partition of the data can exist at a time; to use a different partition the first partition must be deleted (which causes all data in the partition's subregions to be copied back to the root region of the partition) and a new partition created. PaRSEC is a task-based system that supports a DSL for statically-analyzable parallelism [8] as well as a runtime interface for dynamically discovered parallelism [14]. Data in PaRSEC is stored in distributed data collections where each element is owned by a single rank, which is equivalent to a single, disjoint partitioning of the data. PaRSEC provides data views which may alias but cannot be used to write the contents of the underlying data collections. This permits the implementation of certain communication patterns such as halos for stencils, but forces an owner-computes style of computation that makes more dynamic or irregular algorithms challenging to implement.

Dask [19] and Ray [16] are distributed task-based runtimes embedded in Python with similar programming models. Both systems allow users to create *futures* containing arbitrary data. Futures are immutable, so no coherence algorithm is required. Distributed collections such as arrays and dataframes are partitioned into sets of futures; updates mandate the creation a new futures for mutated data.

Spark's resilient distributed data (RDD) types are implicitly distributed immutable collections of data [29]. When necessary, Spark implicitly *shuffles* elements between nodes to perform computations such as joins and group-bys. A shuffle is an expensive operation that blocks other uses of an RDD until it completes, limiting task parallelism. The immutable nature of RDDs also prevents in-place updates and necessitates making a new RDD to mutate a collection.

10 Conclusion

We have shown an unexpected reduction of the problems of dynamic dependence analysis and distributed coherence to the visibility problem. Using this connection, we've demonstrated that algorithms from computer graphics can serve as the basis for a general class of solutions to these problems. Similar to the evolution of computer graphics algorithms for visibility, ray casting performs the best in our experiments.

Acknowledgments

This research was supported by the Exascale Computing Project (17- SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and funding from Department of Energy Office of Science, Office of Advanced Scientific Computing Research under the guidance of Drs. Laura Biven and Hal Finkel. Experiments on the Piz Daint supercomputer were supported by the Swiss National Supercomputing Centre (CSCS) under project ID d108.

References

[1] 2016. Piz Daint - CSCS. http://www.cscs.ch/computers/piz_daint.

[2] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. 2016. *Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model.* Technical Report. Inria.

[3] Michael Bauer and Michael Garland. 2019. Legate NumPy: Accelerated and Distributed Array Computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 23, 23 pages. https://doi.org/10.1145/3295500.3356175

[4] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. 2021. *Scaling Implicit Parallelism via Dynamic Control Replication.* Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/3437801.3441587

[5] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Supercomputing (SC)*.

[6] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18 (1975), 509–517.

[7] Robert L. Bocchino Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*.

[8] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering* 15, 6 (2013), 36–45.

[9] James H. Clark. 1976. Hierarchical Geometric Models for Visible Surface Algorithms. *Commun. ACM* 19, 10 (oct 1976), 547–554. https://doi.org/10.1145/360349.360354

[10] Fredo Durand. 2000. A Multidisciplinary Survey of Visibility.

[11] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *SC*.

[12] Charles R. Ferenbaugh. 2014. PENNANT: an unstructured mesh mini-app for advanced architecture research. *Concurrency and Computation: Practice and Experience* (2014).

[13] James D. Foley, Richard L. Phillips, John F. Hughes, Andries van Dam, and Steven K. Feiner. 1994. *Introduction to Computer Graphics.* Addison-Wesley Longman Publishing Co., Inc., USA.

[14] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. 2017. Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems* (Denver, Colorado) *(ScalA '17)*. ACM, New York, NY, USA, Article 6, 8 pages. https://doi.org/10.1145/3148226.3148233

[15] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. 2018. Dynamic Tracing: Memoization of Task Graphs for Dynamic Task-Based Runtimes. In *Supercomputing (SC)*.

[16] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A Distributed Framework for Emerging AI applications. In *Operating Systems Design and Implementation (OSDI)*. 561–577.

[17] M. E. Newell, R. G. Newell, and T. L. Sancha. 1972. A Solution to the Hidden Surface Problem. In *Proceedings of the ACM Annual Conference - Volume 1* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 443–450. https://doi.org/10.1145/800193.569954

[18] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4, Article 32 (jul 2012), 12 pages. https://doi.org/10.1145/2185520.2185528

[19] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked Algorithms and Task Scheduling. In *Python in Science Conference (SciPy)*. Citeseer.

[20] SIGGRAPH. 2000. *Course Notes: Visibility: problems, techniques, and applications.* Association for Computing Machinery. https://books.google.com/books?id=k0aGzgEACAAJ

[21] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: A High-Productivity Programming Language for HPC with Logical Regions. In *Supercomputing (SC)*.

[22] Elliott Slaughter, Wonchan Lee, Sean Treichler, Wen Zhang, Michael Bauer, Galen Shipman, Patrick McCormick, and Alex Aiken. 2017. Control Replication: Compiling Implicit Parallelism to Efficient SPMD with Logical Regions. In *Supercomputing (SC)*.

[23] Sean Treichler, Michael Bauer, and Alex Aiken. 2013. Language Support for Dynamic, Hierarchical Data Partitioning. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[24] Sean Treichler, Michael Bauer, and Alex Aiken. 2014. Realm: An Event-Based Low-Level Runtime for Distributed Memory Architectures. In *Parallel Architectures and Compilation Techniques (PACT)*.

[25] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. 2016. Dependent Partitioning. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 344–358.

[26] Rob F. Van der Wijngaart and Timothy G. Mattson. 2014. The Parallel Research Kernels. In *HPEC*. 1–6.

[27] John Warnock. 1969. A Hidden Surface Algorithm for Computer Generated Half-Tone Pictures. *IEEE Transactions on Reliability - TR* (06 1969), 35.

[28] Turner Whitted. 1980. An Improved Illumination Model for Shaded Display. *Commun. ACM* 23, 6 (jun 1980), 343–349. https://doi.org/10.1145/358876.358882

[29] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. *HotCloud* 10 (2010), 10–10.

A Artifact Description

This paper includes software artifacts required to run and reproduce the results presented in Figures 12–17. The artifact includes three versions of the Legion runtime, three application codes, and scripts needed to download and build all dependences.

Obtain the artifact at: https://zenodo.org/record/7332228

A.1 Account Access

This paper requires the use of a supercomputer to replicate the results. Due to the number of variables involved, the only reasonable way to replicate these results is to use the same supercomputer we used in the original experiments. You will need to obtain access to Piz Daint, a supercomputer at CSCS. The experiments use GPUs, so you will need access to the GPU partition. You can find out more about Piz Daint allocation schemes on their website.

Like all machines, supercomputers have an expected lifetime and will eventually be decommissioned. If you come to this paper after Piz Daint is no longer available, it may be substantially more difficult to replicate these results. At the bottom of this section, we provide some thoughts and suggestions on how to go about this. However note that we cannot provide any guarantee that the exact results will be replicable on any machine other than Piz Daint.

A.2 Setup and Compilation

We have provided a master script to build and run all versions of the experiments. After unpacking the artifact, run:

```
./setup_all.sh
```

Note that this will take some time (estimated 30 minutes to 1 hour). The script is not interactive, and should complete successfully if you leave it to run.

A.3 Running Basic Tests

To run a basic set of experiments, do:

```
./run_first.sh PROJECT_ID
```

Where `PROJECT_ID` is the ID of the project assigned to you by CSCS for use on Piz Daint.

This will run all of our tests on 1 and 2 nodes. Note that this command queues the jobs in the system, the jobs may take some time to complete (usually on the order of 1 hour to 1 day, depending on how many other users are on the system).

To check the status of your jobs in the queue, run:

```
squeue -u $USER
```

When this reports an empty set of pending jobs, all of your runs are complete. You can of course check on the job output before this point, but the output may be incomplete. The command above can also be used to get an estimate of when your jobs will run.

A.4 Checking the Basic Tests

You can check the results of any runs either before *or* after those jobs have completed. However, please note that if you run this prior to the completion of your jobs, you may see incomplete output from the following commands.

There are nine directories that contain output from the runs in this paper. They are:

```
equiv-ppopp23-neweqcr-regent/language/stencil.run1
equiv-ppopp23-neweqcr-regent/language/circuit.run1
equiv-ppopp23-neweqcr-regent/language/penant.run1
equiv-ppopp23-oldeqcr-regent/language/stencil.run1
equiv-ppopp23-oldeqcr-regent/language/circuit.run1
equiv-ppopp23-oldeqcr-regent/language/penant.run1
equiv-ppopp23-paint-regent/language/stencil.run1
equiv-ppopp23-paint-regent/language/circuit.run1
equiv-ppopp23-paint-regent/language/penant.run1
```

The directories named `neweqcr` refer to experiments named `RayCast` in the paper. Those named `oldeqcr` refer to `Warnock`, and the ones called `paint` refer to `Paint` in the paper. There are three applications, Stencil, Circuit and Pennant.

To check results, `cd` to one of the directories above, and run:

```
../ppopp23_scripts/parse_results.py
```

For example, from the following directory:

```
equiv-ppopp23-neweqcr-regent/language/stencil.run1
```

The command above should produce something that looks like the following. (The output has been cleaned up slightly for presentation.)

```
system nodes procs_per_node rep init_time elapsed_time
neweqcr_dcr      1    1    0    0.063    1.668
neweqcr_dcr      1    1    1    0.063    1.669
neweqcr_dcr      1    1    2    0.063    1.669
neweqcr_dcr      1    1    3    0.063    1.668
neweqcr_dcr      1    1    4    0.062    1.669
neweqcr_dcr      2    1    0    0.068    1.691
neweqcr_dcr      2    1    1    0.063    1.689
neweqcr_dcr      2    1    2    0.065    1.690
neweqcr_dcr      2    1    3    0.067    1.692
neweqcr_dcr      2    1    4    0.063    1.690
neweqcr_nodcr    1    1    0    0.063    1.668
neweqcr_nodcr    1    1    1    0.062    1.667
neweqcr_nodcr    1    1    2    0.062    1.669
neweqcr_nodcr    1    1    3    0.062    1.668
neweqcr_nodcr    1    1    4    0.063    1.668
neweqcr_nodcr    2    1    0    0.063    1.691
neweqcr_nodcr    2    1    1    0.064    1.689
neweqcr_nodcr    2    1    2    0.065    1.690
neweqcr_nodcr    2    1    3    0.064    1.689
neweqcr_nodcr    2    1    4    0.064    1.691
```

This output is a TSV-formatted table. In most cases you can copy-and-paste this directly into a spreadsheet editor for analysis (as we do below).

The columns of this table are, respectively, the configuration of the runtime (visibility algorithm, and whether or not DCR has been enabled), the number of nodes, number of processors per node (always 1, because Piz Daint has one GPU per node), the repetition number (every job is run 5 times), the initialization time in seconds, and the elapsed time in seconds.

To begin with, you should `cd` into each of the nine directories and confirm there is some output. For this basic configuration we expect to see either 5 or 10 rows, depending on the algorithm you are running. The `paint` algorithm should have 5 rows (because there is no DCR configuration for this algorithm), while the other two should have 10. When the jobs are complete, you should see all either 5 or 10 rows as specified above, and none of the time entries should contain `ERROR`. See below for details on troubleshooting any problems.

A.5 Troubleshooting

If any part of `setup_all.sh` fails, the first thing to try is to rerun `setup_all.sh`. The script may report that there is some intermediate build output left from the previous build. If it does, it is safe to completely remove that directory (e.g., any of `language/gasnet`, `language/llvm`, or `language/terra.build`) and try again. The script tracks what has already been built and only rebuilds the parts that require rebuilding.

If you see `ERROR` in the output of `parse_results.py`, this may indicate a number of issues at runtime, either a crash of some sort, OOM, or a timeout. We do not expect this to happen with any of our applications or configurations, but if you do, an initial step would be to try rerunning the job (you can see `run_all.sh` to see how to launch the appropriate job). If that still fails, feel free to contact the authors for assistance.

A.6 Submitting the Full Set of Jobs

Our experiments in the paper go up to 512 nodes. To submit the remaining jobs, run:

```
./run_rest.sh PROJECT_ID
```

Where again, `PROJECT_ID` is the ID assigned to you by CSCS.

This will submit jobs for 4, 8, … 512 nodes. As above, running these jobs may take time. In our experience, this will take at least a couple of hours, and maybe up to 2–3 days for the largest jobs. However, if you submit the jobs during an especially busy period, queue times can be longer.

A.7 Reproducing Graphs in the Paper

Section 8 contains 6 graphs: two for each application, one for initialization time and one for weak scaling performance.

As before, the raw data for these graphs can be obtained by going to each of the directories for the experiments (listed above) and running:

```
../ppopp23_scripts/parse_results.py
```

This produces nine tables of raw results that must be analyzed to produce the final graphs. To make this easier, we provide our versions of these tables, including our raw results and calculations to generate the tables. We performed our analysis in Google Sheets. You can access our spreadsheet at the link below, and copy it into your own account for your own use: https://docs.google.com/spreadsheets/d/173C8yQMZALJqNXw4wJFJFy-RdTqLMNsW4fwPzfSYaMo

Because Google Sheets does not offer the ability to archive a spreadsheet, we also provide a copy of this document in Excel format, which is included in the artifact tarball. *We strongly recommend using the Google Sheets version.* Google Sheets is not fully compatible with Excel, and the graphs and pivot tables in particular may not work properly.

In order to use the spreadsheet to obtain results, what you mainly need to do is copy-and-paste the raw results into the tables. There are nine sheets in the document, three with raw results (`Circuit Weak`, `Stencil Weak`, and `Pennant Weak`) and six with derived data (pivot tables and graphs). You only need to touch the tables with raw data.

A.8 Future-Proofing These Instructions

All supercomputers will eventually be decommissioned. If you come to this paper after Piz Daint is unavailable, it may require substantially more work to replicate.

For best results, we recommend picking a machine as similar to the target platform as possible. Intel (or AMD) CPUs (x86-64) and NVIDIA GPUs would be ideal. However, we cannot guarantee that GPUs and versions of CUDA produced after the publication of this paper will work with the archived version of Regent. Similarly, supercomputer interconnects evolve over time and newer interconnects may not work with the version of GASNet archived in the artifact. If you encounter this situation, you may be able to make progress by using an upstream version of Regent; however, that will prevent you from evaluating any version of the algorithms in this paper other than the latest (Ray Casting).

The scripts in the artifact assume that a Cray programming environment is available. If this is not the case, then the scripts may need to be modified to use conventional compilers (e.g., GCC) instead.

If the target machine uses the SLURM job scheduler, the run scripts may work with some modifications specific to the site. One item to pay attention to is the number of GPUs per node: if it is greater than one, then to make the configuration as similar to Piz Daint as possible it will likely be best to run with one rank per GPU (rather than one rank per node).