Scaling Implicit Parallelism via Dynamic Control Replication

Michael Bauer
NVIDIA

Wonchan Lee
NVIDIA

Elliott Slaughter
SLAC National Accelerator Laboratory

Zhihao Jia
Carnegie Mellon University

Mario Di Renzo
Sapienza University of Rome

Manolis Papadakis
NVIDIA

Galen Shipman
Los Alamos National Laboratory

Patrick McCormick
Los Alamos National Laboratory

Michael Garland
NVIDIA

Alex Aiken
Stanford University

Abstract

We present dynamic control replication, a run-time program analysis that enables scalable execution of implicitly parallel programs on large machines through a distributed and efficient dynamic dependence analysis. Dynamic control replication distributes dependence analysis by executing multiple copies of an implicitly parallel program while ensuring that they still collectively behave as a single execution. By distributing and parallelizing the dependence analysis, dynamic control replication supports efficient, on-the-fly computation of dependences for programs with arbitrary control flow at scale. We describe an asymptotically scalable algorithm for implementing dynamic control replication that maintains the sequential semantics of implicitly parallel programs.

An implementation of dynamic control replication in the Legion runtime delivers the same programmer productivity as writing in other implicitly parallel programming models, such as Dask or TensorFlow, while providing better performance (11.4X and 14.9X respectively in our experiments), and scalability to hundreds of nodes. We also show that dynamic control replication provides good absolute performance and scaling for HPC applications, competitive in many cases with explicitly parallel programming systems.

CCS Concepts • **Software and its engineering** → *Runtime environments*; • **Computing methodologies** → *Parallel and distributed programming languages*;

Keywords implicit parallelism, scalable dependence analysis, dynamic control replication, Legion, task-based runtime

PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8294-6/21/02...$15.00
https://doi.org/10.1145/3437801.3441587

1 Introduction

Implicitly parallel programming models, where apparently sequential programs are automatically parallelized, are increasingly popular as programmers endeavor to solve computationally difficult problems using massively parallel and distributed machines. Many of these programmers have limited experience writing explicitly parallel code and instead choose to leverage high productivity frameworks—such as TensorFlow [7], PyTorch [38], and Spark [53]—that hide the details of parallelism and data distribution.

In implicitly parallel programming models, distinguished functions, often called *tasks*, are the units of parallelism. Tasks may consume data produced by other tasks preceding them in program order. Thus, to discover implicit parallelism, programming systems must perform a *dependence analysis* to discover a legal partial order on task execution. In the worst case, dependence analysis has a computational complexity that is quadratic in the number of tasks: each task must be checked against all its predecessors in the program for dependences. Since the number of tasks to be analyzed is generally proportional to the size of the target machine, dependence analysis is expensive at scale, and its apparently sequential nature can make it difficult to parallelize. Therefore, all existing systems make tradeoffs limiting the scalability of their implementation, the precision of their analysis, and/or the expressivity of their programming model to avoid or reduce the costs of a fully general dependence analysis.

One such class of systems performs dependence analysis at compile-time, thereby eliminating run-time overhead. Sequoia [20], Deterministic Parallel Java [13], and Regent [43] are compiler-based systems that statically infer data dependences to transform a program into an explicitly parallel code that runs across many distributed nodes. However, common idioms such as data-dependent control flow, or potentially aliased data structures, can defeat static analysis, resulting in failed compilation or generated code incapable of scaling to large node counts. Some systems limit the expressivity of the programming model to ensure static analysis is successful

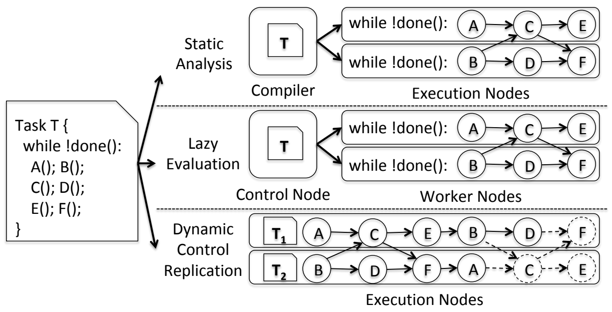


Figure 1. Approaches to implicit parallelism: static analysis and lazy evaluation centralize analysis and distribute partitioned programs to workers for execution. DCR executes a replicated program, dynamically assigns tasks to shards, and performs an on-the-fly distributed dependence analysis.

(e.g., Sequoia). The compile-time approach works best for the subset of programs amenable to precise static analysis.

To avoid the limitations of static analysis, programming systems such as Spark and TensorFlow use *lazy evaluation* to perform dependence analysis at run-time. A single node executes the program's control where all tasks are issued, but tasks are not immediately executed. Instead the control node performs a dynamic dependence analysis on the set of launched tasks to build a representation of the program. At certain points, the system dynamically compiles and optimizes this representation and then distributes it to *worker* nodes for execution. The main difficulty is that the control node can limit scalability, either as a sequential bottleneck in distributing tasks to worker nodes, or as a limitation on the size of the program representation that can be stored in memory. Both Spark and TensorFlow take steps to mitigate these risks. To avoid having the centralized controller become a sequential bottleneck, these systems either memoize repeated executions of code (Spark) or amortize the cost of dependence analysis by explicitly representing loops with some (but not arbitrary) control flow (TensorFlow) [52]. Therefore, programs with large repeated loops of execution can be compiled, optimized, and partitioned for distribution to worker nodes efficiently. Furthermore, to avoid overflowing memory, these systems limit the expressible dependence patterns. Consequently, Spark and TensorFlow work well for programs with minimal dynamic control flow and simple symmetric dependence patterns.

For programs with complex control flow, arbitrary data dependences, and/or the need to run arbitrary tasks on any node, a different approach is needed. The dependence analysis should be distributed to ensure that no node is a bottleneck. In particular, there is little difficulty if a task launches a small, constant number of subtasks; these can be analyzed in essentially constant time and with small overhead in current tasking systems. The problem is with tasks that launch a number of subtasks proportional to the size of the machine, where the dependence analysis quickly becomes a bottleneck when scaling up to large clusters.

We introduce *dynamic control replication* (DCR), a parallel and distributed run-time dependence analysis for implicitly parallel programming models that can scale to large node counts and support arbitrary control flow, data usage, and task execution on any node. Figure 1 gives an example program consisting of a top-level task with a loop that launches six subtasks A-F with some dependences between them, along with an illustration of static, lazy evaluation, and dynamic control replication approaches to the program's execution. In the static approach, to target *n* nodes, the tasks are partitioned into *n* loops with explicit cross-node synchronization to enforce dependences. The compiled program is similar to an MPI implementation [45], with each node explicitly communicating and synchronizing with other nodes. The lazy approach attempts to achieve a similar effect at run-time, with all of the dependence analysis carried out by the control node, and then just-in-time schedules of tasks sent to each worker. These schedules can be cached on the workers to avoid repeating the analysis if possible, but in general when the program's behavior changes, the control node must carry out another lazy gathering of tasks, dependence analysis, and schedule creation for the workers.

In DCR, illustrated at the bottom of Figure 1, the dependence analysis is itself spread across all the nodes of the system. The execution of task T is replicated into *shards*; each shard $\mathtt{T}_\mathtt{i}$ executes a copy of T that replicates all of the control structures of the task T, but only performs the dependence analysis for a subset of the subtasks launched by T. In the figure, shard $\mathtt{T}_1$ handles subtasks A, C, and E for the first loop iteration, while shard $\mathtt{T}_2$ handles subtasks B, D and F. The shards cooperate to discover and enforce any *cross-shard dependences*, in this case the dependences between B and C and between C and F. The assignment of subtasks of T to shards $\mathtt{T}_\mathtt{i}$ is done by the runtime system using a *sharding function* f. The only requirements of f are that it be a function (each subtask is assigned to one shard) and total (every subtask is assigned to some shard). For performance, the sharding function should also balance assignments across shards. Because the sharding is computed dynamically, the sharding function can assign different instances of a subtask to different shards; in Figure 1 the shards of T swap roles on the second iteration, with $\mathtt{T}_1$ handling subtasks B, D and E and shard $\mathtt{T}_2$ handling the rest. Crucial to the scalability of DCR, is the observation that consecutive independent tasks, such as A and B, can be aggregated into *group tasks* that can be launched and analyzed more efficiently as a single operation. We formally introduce group tasks in Section 2.

In DCR the execution of a task is replicated, but collectively the shards behave as a single logical task. By performing dependence analysis on the fly, DCR can easily react to data-dependent control flow. Since the dependence analysis is distributed, each node is only responsible for its subset of the overall analysis and storing a subset of the program representation, yielding an inherently scalable approach.

Each subsequent section presents a contribution: We formalize parallel dynamic dependence analysis and prove this algorithm is equivalent to sequential dependence analysis (Section 2). DCR introduces another new issue: the replicated portion of the program must behave identically in all copies and therefore cannot exhibit non-determinism; we show how to dynamically verify this property of *control determinism* (Section 3). We describe our implementation of DCR (Section 4) and important optimizations required to achieve good performance in practice. Our evaluation demonstrates that DCR delivers the same productivity with better performance and scalability for widely-used data analysis and machine learning programs, and provides good absolute performance and scalability for HPC applications (Section 5).

2 Foundations of Control Replication

Dynamic control replication improves the performance of run-time dependence analysis by enlisting the parallel resources of a distributed machine. To this end, DCR partitions the analysis work, enabling one or more shards of a replicated task, running on multiple nodes, to handle a subset of the full analysis. DCR requires distinguishing where analysis can be done in parallel, and where coordination between the shards is required. A key observation is that programs typically launch tasks in groups of independent tasks (i.e., tasks with no dependences between them); shards can analyze tasks in each *task group* in parallel. A *program* is modeled as a sequence of task groups:

$$\begin{aligned} p &\in \mathit{Program} = \mathit{TaskGroup}^* \qquad p = \epsilon \mid tg; p \\ tg &\in \mathit{TaskGroup} = \{\mathit{Task}\} \\ t &\in \mathit{Task} \end{aligned}$$

This definition of a program is equivalent to a top-level task that launches a sequence of task groups that do not launch further subtasks (e.g., the example in Figure 1). In our implementation, we allow any task to be replicated with DCR, and subtasks launched in replicated tasks may launch (optionally replicated) subtasks of their own, but this simplified model is sufficient to formalize the key ideas.

We assume that there is a separate dependence analysis, which we model as an oracle, that determines whether two tasks are independent (our implementation similarly reuses the existing pairwise task dependence analysis of the underlying runtime system). We write $t_1 * t_2$ to indicate tasks t_1 and t_2 are independent. Tasks in a task group must be pairwise independent: $\forall t_1, t_2 \in tg. t_1 = t_2 \vee t_1 * t_2$. We write $t_1 \Rightarrow t_2$ if task t_2 depends on t_1, i.e., t_2 occurs after t_1 in the program and $\neg(t_1 * t_2)$. The dependence analysis's output is a task graph $G = \langle T, D\rangle$, a DAG of tasks T whose directed edges D are dependences.

The first step in DCR is to apply a *sharding function*, which determines the owner shard for each task (see Section 4). We assume that a sharding function is already applied to the tasks in a program to be analyzed, and we write t^k if task t is owned by shard k. Each shard k analyzes only the subset $tg(k) \triangleq \{t^k \in tg\}$ of a task group tg.

$$\boxed{(S, G) \xrightarrow{\text{rep}} (S, G)}$$

$$\textsc{Ta}\ \frac{\begin{gathered} S = \langle s_1, \dots, (tg; p_i, c_i, \varnothing), \dots, s_n\rangle \quad d'_i = c_i \overset{\Rightarrow}{\times} tg(i) \neq \varnothing \\ S' = (s_1, \dots, \langle tg; p_i, c_i, d'_i\rangle, \dots, s_n) \end{gathered}}{(S, G) \xrightarrow{\text{rep}} (S', G)}$$

$$\textsc{Tb}\ \frac{\begin{gathered} S = \langle s_1, \dots, (tg; p_i, c_i, d_i), \dots, s_n\rangle \\ d_i \neq \varnothing \quad \forall (t^k, t) \in d_i. t^k \in c_k \\ S' = (s_1, \dots, \langle p_i, c_i \cup tg, \varnothing\rangle, \dots, s_n) \\ T' = T \cup tg(i) \quad D' = D \cup d_i \end{gathered}}{(S, \langle T, D\rangle) \xrightarrow{\text{rep}} (S', \langle T', D'\rangle)}$$

$$\textsc{Tc}\ \frac{\begin{gathered} S = \langle s_1, \dots, (tg; p_i, c_i, \varnothing), \dots, s_n\rangle \quad c_i \overset{\Rightarrow}{\times} tg(i) = \varnothing \\ S' = (s_1, \dots, \langle p_i, c_i \cup tg, \varnothing\rangle, \dots, s_n) \quad T' = T \cup tg(i) \end{gathered}}{(S, \langle T, D\rangle) \xrightarrow{\text{rep}} (S', \langle T', D\rangle)}$$

Figure 2. Parallel dependence analysis DEP_{rep}

Once the owner shard is determined for each task, a program is analyzed by shards in parallel. The analysis state (S, G) consists of a tuple $S = \langle s_1, \dots, s_n\rangle$ of states (one for each shard), and a global task graph G shared by all shards. Each shard i's state $s_i = (p_1, c_i, d_i)$ consists of a program p_i, a set of tasks that the shard has finished analyzing c_i, and a set of outstanding dependences d_i,

Figure 2 gives an operational semantics DEP_{rep} for the parallel dependence analysis. Each inference rule corresponds to a single analysis step, which starts with an initial state $S = \langle(p, \emptyset, \emptyset), \dots, (p, \emptyset, \emptyset)\rangle$ where the program is replicated in the shards' states and finishes when all shards have no remaining task groups to analyze. In the rules Ta and Tc, we write $T \overset{\Rightarrow}{\times} T'$ to denote a set of dependences between tasks in T and those in T' (obtained by querying the oracle):

$$T \overset{\Rightarrow}{\times} T' \triangleq \{(t, t') \mid t \in T \wedge t' \in T' \wedge t \Rightarrow t'\}$$

The analysis of a task group in each shard is done in two steps: First, a shard i identifies dependences for the subset $tg(i)$ of the task group tg being analyzed and records them locally in the set d_i of outstanding dependences (rule Ta). Second, the outstanding dependences of a task t are registered to the global task graph if all of t's dependent predecessors have been analyzed by their owner shards (rule Tb); to check this condition, each shard i maintains a set c_i of tasks whose analysis is completed. The rule Tc handles the case where the tasks being analyzed have no dependences, allowing those tasks to be registered to the task graph immediately. In rules Ta and Tc, shard i compares $tg(i)$ with c_i and not with the tasks in the task graph, as shards can make progress

$$\frac{\boxed{(P,G) \xrightarrow{\text{seq}} (P,G)} \quad T' = T \cup tg \quad D' = D \cup T \overset{\Rightarrow}{\times} tg}{(tg;p,\langle T,D\rangle) \xrightarrow{\text{seq}} (p,\langle T',D'\rangle)}$$

Figure 3. Sequential dependence analysis DEP_{seq}

at different rates and therefore the task graph may not yet contain all the dependent predecessors of $tg(i)$.

The key property of DEP_{rep} is that it produces the same task graph as a sequential analysis; Figure 3 shows a system DEP_{seq} that sequentially analyzes dependences.

Theorem 1. *For a program p, the following holds:*

$$(p,\langle\varnothing,\varnothing\rangle) \xrightarrow{\text{seq}\,\star} (\epsilon, G_s) \wedge (S_\iota^N, \langle\varnothing,\varnothing\rangle) \xrightarrow{\text{rep}\,\star} (S_\varnothing^N, G_r) \implies G_s = G_r,$$

where $$S_\iota^N = \overbrace{\langle(p,\varnothing,\varnothing),\ldots,(p,\varnothing,\varnothing)\rangle}^{N}, \quad S_\varnothing^N = \langle(\epsilon,p,\varnothing),\ldots,(\epsilon,p,\varnothing)\rangle$$

and $R^\star$ denotes the reflexive transitive closure of a relation R.

Theorem 1 is a direct consequence of a lemma stating that any consecutive transitions A and B, in two different shards, can be reordered to B followed by A if the task group analyzed in A appears later than that analyzed in B in the original program order. From this lemma, we derive that the transitions for an analysis of a program using the system DEP_{rep} can be reordered so that they simulate an analysis of the same program with the system DEP_{seq}. A detailed proof of Theorem 1 is included in Appendix A.

For simplicity, we have described dependence analysis as if it always compares each task with all predecessors. In practice, we can minimize comparisons because *transitive* dependences are redundant: if the system already has dependences $t_1 \Rightarrow t_2$ and $t_2 \Rightarrow t_3$ in the task graph, then $t_1 \Rightarrow t_3$ does not further constrain the scheduling of tasks [11].

3 Control Determinism

Our semantics (Section 2) and our implementation (Section 4) rely on all shards analyzing task groups in the same order. Any deterministic program run with the same input in all shards will satisfy this requirement. In practice, however, there can be sources of "input" that differ across shards. The state of the memory allocator, the presence of profile-guided garbage collectors, address space randomization, the choice of hash functions in an interpreter, and hardware clocks for timing are all examples of data sources that may be visible to the program and different in each shard.

Fortunately, dynamic control replication does not require all data sources be identical across shards. A program is *control deterministic* if all shards make the same the sequence of API calls into the runtime system. We illustrate three violations of control determinism we have encountered.

```
import random
...
if (random.random() < 0.5):
   run_algorithm0()
else:
   run_algorithm1()
```

Figure 4. Branching on a random number.

```
future = launch_task0()
if future.is_ready():
    value = future.get_value()
    run_task1_inline(value)
else:
    launch_task1(precondition=future)
```

Figure 5. Branching on a timing-dependent value.

```
regions = set()
for i in range(num_regions):
    regions.add(create_region())
for region in regions:
    launch_task(region)
```

Figure 6. Iterating a data structure with undefined order.

Figure 4 selects different algorithms to run based on the result of a random number generator. To ensure shards produce the same random number sequences, we provide a pseudo-random number generator backed by a parallel counter-based generator [40].

Figure 5 branches on a value whose result depends on timing, in this case an optimization: If the future is unresolved, rather than block on the future's value a task is launched with the resolution of the future as a precondition. The future may resolve at different speeds in different shards, and so some shards may launch a subtask while others do not.

Figure 6 launches one task per set element. In Python, a `set` is ordered by hash values of the items. For security, the hashing is randomized, so shards will launch the same tasks but likely in different orders. Such situations are easily fixed by using a data structure with a defined order, such as a list.

We enforce control determinism using a dynamic analysis. For each runtime API call from a shard of a replicated task (and only for such calls), we compute a 128-bit hash that captures the API call and all its actual arguments. An all-reduce collective (see Section 4.2) checks that the hashes from all shards are identical, indicating that (assuming no hash conflicts) the program is control deterministic to that point in its execution. The all-reduce is performed asynchronously to hide its latency and many applications run without performance degradation even with the check enabled if there is unused communication bandwidth (see Section 5.5). If a check fails, the runtime system aborts with an error listing the operation that failed to be control deterministic, which we have found sufficient for debugging.

For completeness, two additional sources of input that must be handled by any system implementing dynamic control replication are files and interactions with garbage collectors. We expect the support for these sources of input to be implementation specific; we discuss how our implementation of DCR handles both cases in Section 4.3.

4 Dynamic Control Replication in Legion

We have implemented dynamic control replication in the Legion runtime [11]. We work through the execution of the sequential 1D stencil program in Figure 7 with DCR on a small machine with two nodes, each with one CPU and two GPUs. This program is written in Regent [42], a high-level programming language that compiles down to calls to the Legion runtime system. To explain how DCR works with this program, we need to introduce some Legion terminology to explain specifically how Legion expresses task groups.

Legion tasks operate on *regions*, which are tables created from a set of points called an *index space* (which in this case is 1D) that name the table's rows, and a set of typed *fields* naming the table's columns; see lines 1-3 (which defines two field names), line 33 (which creates a 1D index space), and line 35 (which defines a region).

Regions can be *partitioned* into arbitrary subregions [49, 50]; partitions are used like arrays of subregions (lines 44, 47, and 50). Subregions can be further partitioned, and in general, programs construct *region trees* by recursively partitioning a root region. Figure 8 depicts the region tree used by the stencil program (the partitioning code is omitted from Figure 7 for space reasons but can be found online [6]). An important property of region trees is that any region in the tree is a superset of all the regions in its subtree. Thus, any set of regions can be over-approximated by a region r, or more generally a partition of r, that is their least common ancestor in the tree. We will make use of this property in our analysis of the dependences between task groups.[1] In this example three different partitions of the cells are created, representing the cells owned by each node, the cells in the interior of the domain within each owned subregion, and the owned regions plus ghost cells on either side.

The `__demand(__replicable)` annotation on line 29 indicates that `main` is a control deterministic task and therefore eligible for DCR. Most applications do not require further changes, except for minor tweaks to interact correctly with external side effects (see Section 4.3).

Regardless of whether this program is executed with DCR or not, the Regent compiler transforms inner loops of independent task launches into group task launches [42]. To enable each task in a group task launch to have different arguments, the task calls have the form $t(p[f(i_j)])$ where t is the task, $i_1, \ldots, i_n$ is an index space, p is a partition containing the subregions accessed in the original loop, and f is a function that returns the subregion index of p for the i_jth call of t.[2] Lines 43-45, 46-48, and 49-51 are converted to group task launches, which define task groups in the sense of Section 2. For example, the loop on lines 43-45 is replaced by the group task launch `add_one(owned[id(·)]` over the `tiles` index space, where `id` is the identity function. While

[1]Note that this property of region trees is used in our implementation of DCR in Legion and is not a requirement of DCR in general.

```
1   fspace Cell {
2     state : double,
3     flux : double,
4   }
5
6   task add_one(cells: region(ispace(int1d),Cell))
7   where reads writes(cells.state) do
8     for i in cells do
9       cells[i].state = cells[i].state + 1
10    end
11  end
12
13  task mul_two(cells: region(ispace(int1d),Cell))
14  where reads writes(cells.flux) do
15    for i in cells do
16      cells[i].flux = cells[i].flux * 2
17    end
18  end
19
20  task stencil(owned: region(ispace(int1d),Cell),
21               ghost: region(ispace(int1d),Cell))
22  where reads writes(owned.flux),
23              reads(ghost.state) do
24    for i in owned do
25      owned[i].flux = owned[i].flux +
26        0.5*(ghost[i-1].state + ghost[i+1].state)
27    end
28  end
29
30  __demand(__replicable)
31  task main()
32    -- Parse args ncells, ntiles=4, nsteps, init
33    var grid = ispace(int1d, { x = ncells })
34    var tiles = ispace(int1d, { x = ntiles })
35    var cells = region(grid, Cell)
36    -- Create partitions: owned, interior, ghost
37    -- Code omitted for space, see Figure 8
38    fill(cells.{state, flux}, init)
39    for t = 0, nsteps do
40      for i in tiles do
41        add_one(owned[i])
42      end
43      for i in tiles do
44        mul_two(interior[i])
45      end
46      for i in tiles do
47        stencil(interior[i], ghost[i])
48      end
49    end
50  end
```

Figure 7. A 1-D stencil program written in Regent [42].

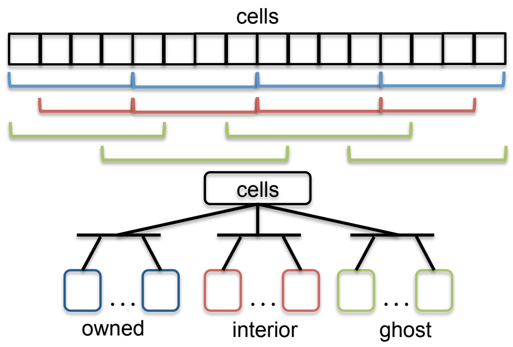


Figure 8. A region tree with three partitions to describe different subsets of cells used by different tasks in the stencil example. Ranges of sub-regions are depicted visually at top.

[2]For region trees with multiple levels of partitioning, a more general form of this function can choose any subregion in the subtree of p [11].

DCR works correctly for individual tasks and operations (such as the `fill` to initialize data on line 40), the scalability of dependence analysis hinges on the efficiency of analyzing programs consisting primarily of group launches.

In our Legion implementation we do not attempt to decide automatically when to use DCR; instead we expose this decision in the Legion *mapping interface* [11], an API for application- and machine-specific policies that affect performance. A client of the mapping interface is a *mapper*. Our mapping interface extensions enable mappers to specify which task(s) to dynamically control replicate, the number of shards, and on which processors shards should execute. For the example program, the mapper (not shown) requests one shard of `main` to execute on a CPU of each node. We note that there is nothing the prevents the use of DCR from being automated by heuristics in the runtime system to decide when to use it; we have simply chosen to expose it through an API so users can decide when best to deploy DCR.

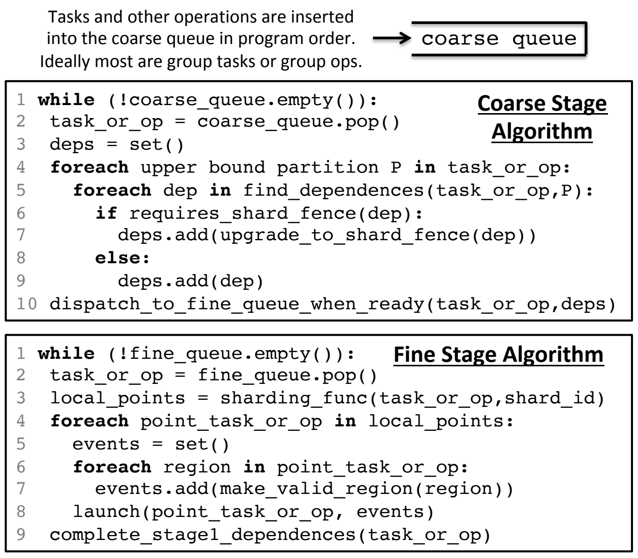
When a DCR task executes, Legion queries mappers to select a *sharding function* for each subtask launch. Sharding functions are pure functions that map individual tasks or tasks in a group launch to shards. Each shard is responsible for the dependence analysis of the tasks assigned to it. A good sharding function assigns tasks near where they will execute, while a poor choice may require significant movement of meta-data by the runtime. In the example, the *cyclic* sharding function (ID `0`) round-robins tasks across the shards. Because sharding functions are pure, we can memoize their results, which reduces their dynamic invocation overhead.

The implementation described here is specific to Legion, but DCR is not. The essential requirements for any implicitly parallel programming system to support DCR are task groups, a way to indicate a task is control deterministic, and sharding functions. While our Legion implementation exposes the implementation of sharding functions, allowing customization for additional performance, this is not essential; a fixed sharding function could also be implemented within the runtime system.

4.1 Dependence Analysis

In a straightforward implementation of DCR, each shard would analyze all subtasks launched by the replicated task, which would be problematic because the number of subtasks is often proportional to the number of compute nodes and dependence analysis costs would scale with node count. To improve scalability, our implementation uses a hierarchical analysis that achieves $O(\log N)$ overhead in the average case (where N is the number replicated shards), and appears to execute with $O(1)$ cost if there is sufficient task parallelism.

In our approach, each shard implements the dependence analysis in Figure 9, which consists of a *coarse stage* and a *fine stage*. The coarse stage analysis discovers dependences at the granularity of task groups; i.e., for two task groups G_1 and G_2, if there are any tasks $t_1 \in G_1$ and $t_2 \in G_2$ such that t_2 depends on t_1, then there will be a coarse-stage dependence between groups G_1 and G_2. Importantly, these group-level dependences are discovered without enumerating the individual tasks within a group. Consider a task with one argument. Conceptually, we construct a single task $t(r)$ that is representative of an entire task group tg by taking r to be an upper bound in the region tree of all the region arguments of tasks in tg. We then query the dependence oracle for any dependences between the task representative t and representatives of other task groups. In Figure 9, since the call to a task group already provides an upper bound partition that covers all possible region arguments in the task group, we use the partition and do not calculate another upper bound. For a task with multiple arguments, we repeat this construction for each argument. All shards perform dependence analysis for all task groups in the coarse stage, so every shard is aware of all dependences between task groups.


Tasks and other operations are inserted into the coarse queue in program order. Ideally most are group tasks or group ops.
coarse queue
Coarse Stage Algorithm
Fine Stage Algorithm

```
1  while (!coarse_queue.empty()):
2   task_or_op = coarse_queue.pop()
3   deps = set()
4   foreach upper bound partition P in task_or_op:
5     foreach dep in find_dependences(task_or_op,P):
6       if requires_shard_fence(dep):
7         deps.add(upgrade_to_shard_fence(dep))
8       else:
9         deps.add(dep)
10  dispatch_to_fine_queue_when_ready(task_or_op,deps)
```

```
1  while (!fine_queue.empty()):
2   task_or_op = fine_queue.pop()
3   local_points = sharding_func(task_or_op,shard_id)
4   foreach point_task_or_op in local_points:
5     events = set()
6     foreach region in point_task_or_op:
7       events.add(make_valid_region(region))
8     launch(point_task_or_op, events)
9   complete_stage1_dependences(task_or_op)
```

Figure 9. Pseudo-code for the dependence analysis algorithm. Each stage operates independently and continuously on every shard for the lifetime of a DCR program.

A coarse stage dependence from group G_1 to G_2 requires that the fine stage analysis of all individual tasks in G_1 finishes before the fine stage analysis of any tasks in G_2. As coarse stage dependences are satisfied, tasks and operations proceed to the fine stage where a precise dependence analysis is performed for the tasks owned by the shard.

Coarse stage dependences are trivially satisfied among tasks owned by the same shard, as shards analyze their tasks in program order. To handle cross-shard dependences, the coarse stage promotes dependences to *cross-shard fences* to enforce a partial order between the fine stage analyses on different shards (lines 6-7). Cross-shard fences behave similarly to scoped memory fences on parts of the region tree, providing ordering of fine stage analysis across shards for tasks and operations that access the fenced regions/partitions.

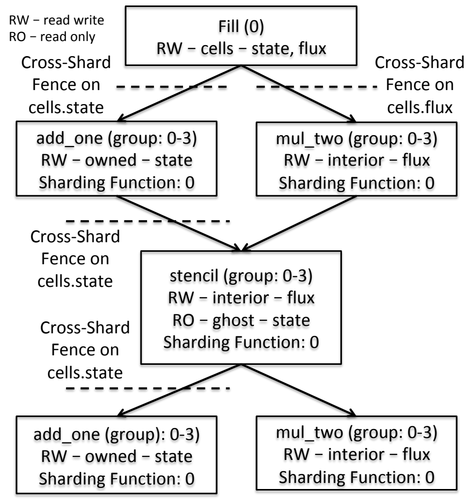


Figure 10. Coarse stage dependence analysis computed by every shard for the program in Figure 7.

Any operation can begin the fine stage once all its incoming dependences from the coarse stage analysis have been satisfied (line 2). The fine stage analyzes individual tasks or operations (line 4). For each individual task or operation, it performs any data movement (line 7), gathers event preconditions (lines 5,7), and dispatches execution to the lowest layer of Legion [48]. On line 3 we use our local shard ID to evaluate the sharding function for each task or operation to determine the set of individual tasks or operations owned by the local shard. Note that both stages of the analysis operate asynchronously with respect to each other and we exploit pipeline parallelism to hide the latency of the on-the-fly analysis; different tasks and operations can be in both stages of the pipeline simultaneously.

We illustrate how the two-stage algorithm analyzes the program in Figure 7. Figure 10 shows the result of the dependence analysis for the beginning of the program. After the fill is issued to both fields of the region, the next operation analyzed is the `add_one` group task launch over the domain $0-3$. This task uses the `state` field of the `owned` partition (see Figure 8) and therefore depends on the fill so a dependence is added. Additionally, we insert a cross-shard fence on the `cells` region because the fill will be performed on shard 0, whereas the cyclic sharding function (ID `0`) for `add_one` maps individual tasks 1 and 3, that depend on the fill, to shard 1. The second group task launch for `mul_two` has a similar dependence on the fill for the `flux` field and therefore inserts a fence on the `cells` region for the `flux` field.

Next to be analyzed is the `stencil` task group, which has dependences on the `state` field of the previous `add_one` task as well as on the `flux` field of the previous `mul_two` task. The coarse analysis requires a cross-shard fence for the first but not the second dependence. While the sharding functions are the same, the first dependence is between two different partitions of the same region: `owned` and `ghost`. We cannot easily

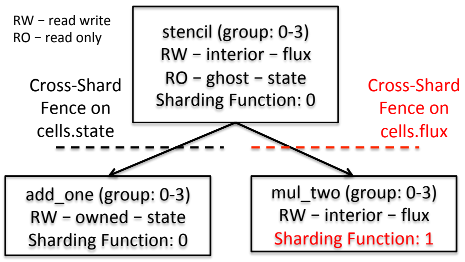


Figure 11. Changed dependence analysis (in red) from an alternate selection of sharding function for `mul_two`.

discern the aliasing relationship between the sub-regions of these two partitions, so we must conservatively insert the fence to handle any cross-shard dependences. For the second dependence we know all individual tasks on each shard access the same subregions of the same partition (`interior`) on each shard because they have the same sharding function. Therefore all the individual task dependences will be *shard-local* and we can safely elide the cross-shard fence because the fine stage is guaranteed to analyze individual task dependences on local shards in program order. If the mapper picked a different sharding function for `mul_two`, as in Figure 11, we would need a cross-shard fence because there may be dependences between individual tasks owned by different shards. The tasks from later iterations are analyzed similarly.

Finally, for Legion we implement the dependence oracle for two task calls $t_1(r_1)$ and $t_2(r_2)$ by checking first whether regions r_1 and r_2 share any index points—if they are disjoint then the tasks are independent. If they share points then we check whether they are accessing at least one field in common. If the tasks are accessing the same points of at least one field, we lastly check to see if either task writes its region argument; if at least one is writing then a dependence is required. This procedure is the standard dynamic dependence analysis carried out by the Legion runtime [11] with no modifications required for DCR.

We make three observations. First, the coarse analysis stage does not require enumerating individual tasks for task groups. The cost of the coarse stage is independent of the size of the group launches, which are usually proportional to the size of the machine. This independence from machine size makes the coarse stage scalable.

Second, in the common case of data parallel operations, we can prove that all dependences are shard-local and therefore the cross-shard fences can be elided, which avoids unnecessary synchronization (line 6 of the coarse stage in Figure 9). Currently, we do this proof symbolically by analyzing the index space of the group task launches, the regions or partitions that are upper bounds for task groups or operations, and names of the projection and sharding functions.

Third, when cross-shard fences are required, we insert these on specific regions or partitions. This design is a middle ground between full analysis barriers that create all-to-all dependences between preceding and succeeding tasks, and direct shard-to-shard dependences that could be computed with an expensive inversion of the functions that determine which subregions are used by individual tasks. Our cross-shard fences are implemented using a collective (see Section 4.2) which has cost $O(\log N)$ in latency across the shards. Performing the coarse-grained analysis discovers coarse task parallelism that can be exploited during the fine-grained analysis stage to hide the cost of the fences. In practice, most applications we have observed have sufficient task parallelism to hide these latencies.

4.2 Collectives

Our DCR implementation uses a set of *collective* primitives for performing cooperative work between shards: broadcast from one shard to all shards, reduce from all shards to one shard, all-gather from all shards to all other shards, and all-reduce that produces a single value for all shards. As an example, our implementation of the cross-shard dependence fences from Section 4.1 is performed with an all-gather collective with no data payload. The collectives are implemented using standard tree or butterfly communication networks with $O(\log N)$ latency.

4.3 Side Effects

Control replicated programs must interact correctly with the external world. We have focused on two aspects: file I/O and support for languages with garbage collectors. To support file I/O, we provide implementations of Legion's *attach* and *detach* operations [26] that execute correctly in dynamically control replicated contexts. Attach operations allow applications to associate external memory allocations (e.g., from an MPI program [45]) or files (e.g., HDF5 [46]) with existing regions in a program; detach operations flush any changes (e.g., write back any file updates to disk) and remove these associations. With dynamic control replication, we provide support for sharding *attach* and *detach* operations just like any other kind of operation. Normal files are read and written by a single owner shard; group variants of attach and detach provide support for parallel file I/O.

We have also added support for using DCR with languages that rely on garbage collectors (GC) such as Lua and Python. Finalizers [2] that make runtime API calls can violate control determinism because collections can occur at arbitrary times in each shard. In practice, we have seen finalizers perform detach operations as well as deleting regions and fields (see Section 5.4). To handle such cases without violating control determinism, we provide an option to *delay* detach operations and deletions of Legion resources. The runtime periodically polls the shards to see if they have all observed the same delayed operation. The polling is done with exponential back-off so that it can quickly handle cases when GC is active, but incurs minimal overhead when GC is absent. Once the shards concur, the operation is inserted into the same location in the dependence analysis of each shard.

5 Evaluation

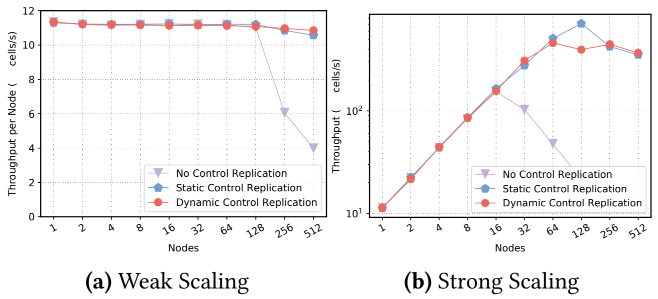
We evaluate DCR against a wide variety of applications drawn from HPC, data analytics and machine learning, written in Legion without DCR, Regent [43], MPI [45], TensorFlow [7], and Dask+NumPy [39]. We show that compared to other implicit parallel models (TensorFlow, Dask+NumPy) we achieve much better performance, and that for HPC applications, our performance is often competitive with explicit parallel programming models such as MPI. Using DCR with these applications required marking the top-level task as replicable and writing sharding functions, all of which were one or two lines of code (e.g., round-robin or tiled sharding of tasks). We used either one shard per node or one shard per GPU in all experiments. To underscore the portability of our approach, our experiments are run on a diverse collection of the world's top supercomputers including Summit (#2), Sierra (#3), Piz-Daint (#10), and Lassen (#14) [5], among others, with a mix of heterogeneous processor types.

5.1 Benchmarks

We begin by benchmarking DCR against both Legion without control replication, as well as the implementation of static control replication in Regent.[3] Recall that static control replication, when it applies, has no runtime overhead because it is a compile-time program transformation. Furthermore, because Regent is also built on the Legion runtime, and its static control replication approach is known to perform well [43], this comparison provides a measure of the end-to-end overheads of our dynamic control replication implementation. Our first benchmark is an implicitly parallel 2D stencil code [6], similar to Figure 7, that requires the system to recognize a nearest-neighbors communication pattern in multiple dimensions. For this benchmark, Figure 12 gives both weak and strong scaling results using the GPUs on Piz-Daint. Without control replication, the overhead of a centralized controller node dominates once the cost of analyzing and distributing all tasks eclipses the execution time of tasks assigned to a single processor. DCR weak scales nearly as well as static control replication, with only a 2.5% slowdown at 512 nodes. For strong scaling on the chosen problem size, static and dynamic control replication scale to 128 and 64 nodes, respectively, before degrading. Unsurprisingly, DCR has higher runtime overheads than static control replication, but they are within a factor of two.

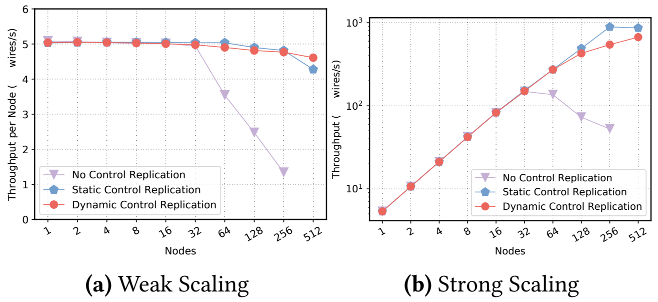
Our next benchmark is a circuit simulation that iteratively updates currents on wires and voltages on nodes in a graph of circuit components. The partitioning of the graph is done dynamically, so the communication pattern must also be established at runtime. Figure 13 gives weak and strong scaling results, both of which are significantly better with DCR

[3]There are approaches to scaling task-based programs based on using explicit communication and synchronization in a SPMD-like structure [12]. Our evaluation focuses on programs written purely using implicit tasking.



(a) Weak Scaling (b) Strong Scaling

Figure 12. Scaling of 2D Stencil Benchmark.



(a) Weak Scaling (b) Strong Scaling

Figure 13. Scaling of Circuit Simulation Benchmark.

than without, and we can see that DCR adds no noticeable overhead at the smaller node counts. DCR slightly underperforms static control replication in weak scaling to 256 nodes with a maximum slowdown of 2.7%. At 512 nodes, though, DCR is 7.8% better, as it better analyzes the increasingly complex inter-node communication patterns of the small-diameter graph partitioned across many nodes.

To compare our performance against MPI, we implemented the Pennant mini-application [21] for Lagrangian staggered grid hydrodynamics on unstructured meshes in Legion. For comparison, we use an independently developed and optimized version of Pennant written using MPI and CUDA by an NVIDIA tuning engineer. Figure 14 shows the weak scaling performance of the Legion dynamic control replication running on GPUs against three configurations of the MPI version: one run only with CPUs, one with CUDA acceleration, and a third with CUDA and GPUDirect acceleration [37]. These experiments were run on a cluster of NVIDIA DGX-1V nodes, each with 8 GPUs per node and an Infiniband EDR interconnect. The implementation with DCR outperforms the MPI+CUDA version by 2.3X on 256 GPUs because we are able to run one process per node and leverage intelligent sharding functions to maintain data locality and consequently leverage NVLink interconnect bandwidth for communication. MPI with GPUDirect recovers NVLink performance and also benefits inter-node GPU transfers that Legion cannot match because GASNet [51], Legion's underlying transport layer, does not support GPUDirect. The version of Pennant without DCR scales poorly. The drop in parallel efficiency for the two fastest implementations is due to a global collective for computing the next iteration's time step; this collective

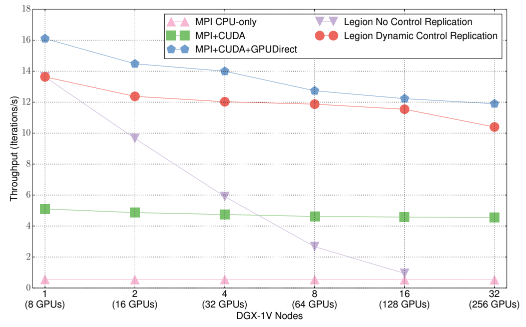


Figure 14. Weak scaling of Pennant compared to MPI.

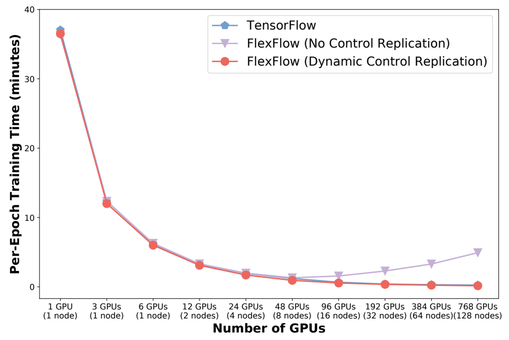


Figure 15. Training of ResNet-50 compared to TensorFlow.

blocks all downstream work and incurs additional latency with increased processor counts. Overall, Legion with DCR is only 14% slower than MPI+CUDA+GPUDirect.

Finally, we perform a scale-out experiment training the popular ResNet-50 [22] convolutional neural network on Summit with the ImageNet dataset [18]. We use FlexFlow (see Section 5.3) for training on top of Legion with DCR, and compare with TensorFlow [7] (r1.14) running with Horovod [41]. We train ResNet-50 to the same accuracy using the same parallelization strategy (data-parallelism) and a per-GPU batch size of 64 for both systems. Figure 15 shows the per-epoch training time of ResNet-50 as we increase the number of GPUs. The performance of training with DCR on Legion is nearly identical to the performance of using TensorFlow out to 128 nodes with 768 GPUs with DCR, but stops scaling at 48 GPUs without DCR.

5.2 Regent with Dynamic Control Replication

DCR complements the static program analyses and transformations in the Regent programming language in ways particularly suited to writing complex task-based systems. Regent has always supported the program transformation for converting sequential task launches into the group task launches needed to make the dependence analysis for dynamic control replication efficient [42] (recall Section 4). However, Regent's auto-parallelizer, which synthesizes both partitions and group task launches from sequential code [30], often results in programs that cannot be handled by static control replication (SCR)—including the two applications discussed in this subsection.

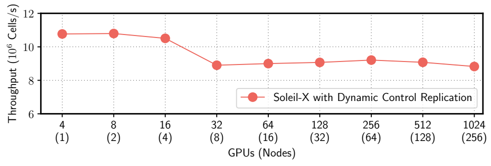
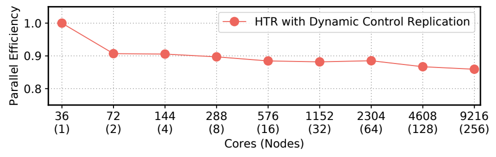
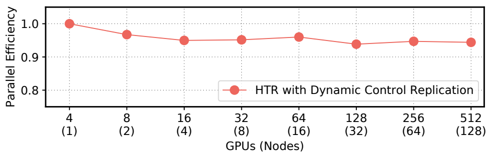


Figure 16. Throughput of Soleil-X on Sierra.



(a) Quartz (36 cores/node)



(b) Lassen (4 GPUs/node)

Figure 17. Weak scaling of the HTR solver.

Soleil-X is a multi-physics solver that models three separate but coupled physical phenomena: fluid flow, Lagrangian particle dynamics, and thermal radiation transport [47]. Different solvers and partitions of data are used for modeling each of the three physical processes. At various points in the simulation information is shared between each of the different physical representations, necessitating complex dependence patterns and control flow. Soleil-X is implemented entirely in Regent and uses Regent's auto-parallelizer to aid developer productivity. Soleil-X requires a number of partitions (to capture independent work in wavefront computations) that cannot be fixed statically, but SCR requires a statically known number of partitions; this led us to use DCR. Figure 16 shows weak scaling of Soleil-X running with all three physics modules on Sierra. On 1024 GPUs, Soleil-X achieves a weak-scaling parallel efficiency of 82% using DCR; the full 3D communication pattern is only reached at 32 nodes, which explains the drop in efficiency at that point.

Another complex and very different Regent code is the Hypersonic Task-Based Research (HTR) solver [19]. The HTR solver performs multi-physics simulations of hypersonic flows at high enthalpies and Mach numbers, such as those that occur upon the reentry of spacecraft into the atmosphere. HTR has complex control flow for which SCR's analysis is too conservative and thus requires the dynamic dependence analysis of DCR. Figure 17 shows weak scaling performance of the HTR running on Quartz and Lassen. Using DCR, HTR achieves parallel efficiencies of 86% on 9216 cores on Quartz and 94% on 512 GPUs on Lassen.

5.3 FlexFlow

FlexFlow is a framework for training deep neural networks that exploits hybrid data and model parallel approaches on different layers of the network [27]. FlexFlow performs a

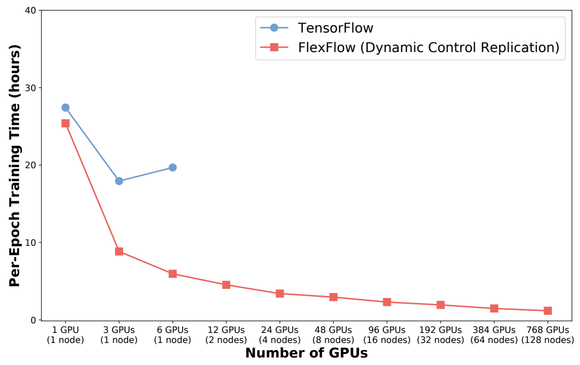


Figure 18. Training CANDLE Multi-Layer Perceptron on Summit with DCR (FlexFlow) and TensorFlow.

search to discover the fastest parallel strategy for a network, where each layer can have its own strategy. Parallelizing the many layers of a network in different ways leads to extremely complex partitioning and dependence patterns, which FlexFlow delegates to Legion to implement. For multi-node runs, FlexFlow relies on DCR for achieving scalability.

To underscore the need for DCR, we use FlexFlow with DCR to train the largest (`pilot1`) network from the CANDLE initiative [3] using the Uno dataset [4]. CANDLE is a collection of deep learning models for precision medicine. The network we train takes drug and cell features as inputs and predicts the response between a drug and cell pair. TensorFlow uses data parallelism, keeping a replica of the model weights on each GPU, and performs collective reductions across GPUs using Horovod [41]. However, the large number of weights (768M) makes scaling with data parallelism difficult, because the communication cost to synchronize model weight gradients across GPUs dominates performance.

Unlike Section 5.1, FlexFlow does not take a data-parallel approach. Instead, FlexFlow's search discovers a hybrid data- and model-parallel approach with a more sophisticated dependence pattern that reduces communication costs by 20X. Figure 18 shows the training performance comparison between TensorFlow and FlexFlow on CANDLE on Summit. The more complicated dependence pattern of the hybrid approach, supported by DCR, allows FlexFlow to scale to 768 GPUs over 128 nodes and to improve per-epoch training time by 14.9X over TensorFlow. We note that FlexFlow provides an interface with the same API calls as PyTorch and TensorFlow, enabling equivalent programmer productivity but with higher performance.

5.4 Legate NumPy

Legate NumPy is a drop-in replacement for Python's NumPy package [36] that allows unmodified NumPy programs to be run at scale [10]. Legate NumPy performs a dynamic translation of NumPy programs to the Legion programming model: NumPy `ndarray` types are backed by individual fields in Legion regions, and NumPy API calls are performed by launching one or more tasks to perform computations. Legate

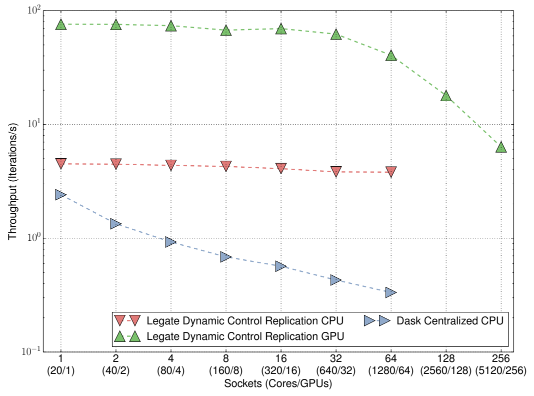


Figure 19. Logistic Regression in Legate NumPy.

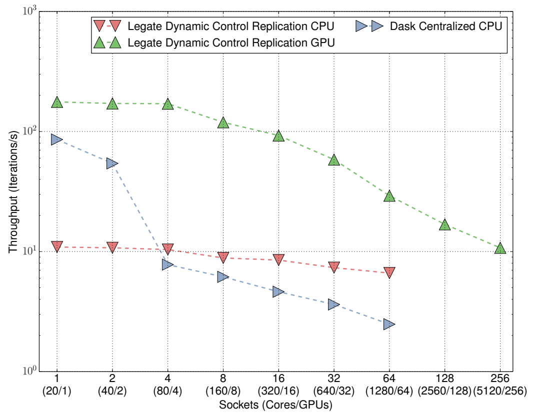


Figure 20. Preconditioned CG Solver in Legate NumPy.

NumPy also decides on-the-fly how to partition arrays and when to convert NumPy API calls into group task launches. When running on multiple nodes, Legate NumPy leverages DCR to replicate the execution of the NumPy program in the top-level task, which avoids the sequential bottleneck in other distributed Python systems [35, 39, 53].

To illustrate the benefits that DCR provides Legate NumPy, we show how it aids scaling compared to the `dask.array` library built on top of the Python-based Dask runtime [39]. The Dask runtime executes programs on one node and distributes tasks to worker nodes. Like Legate NumPy, `dask.array` translates API calls into task launches, and provides a nearly drop-in replacement for NumPy with the exception that users must choose how to partition arrays. For our experiments, we use a brute-force tuning algorithm to find the best partitioning of arrays for Dask; Legate needs no such tuning as it automatically selects chunk sizes for users implicitly.

Figures 19 and 20 show the weak scaling performance of Legate NumPy and Dask array programs for logistic regression and a preconditioned conjugate gradient (CG) solver respectively running on a cluster of NVIDIA DGX-1V nodes with 8 GPUs per node and an Infiniband network. Legate NumPy programs are run with CPU-only execution (red), as well as GPU execution (green). Dask array programs run only on the CPU. The two programs are identical except for the chunk sizes on array creations in the Dask programs.

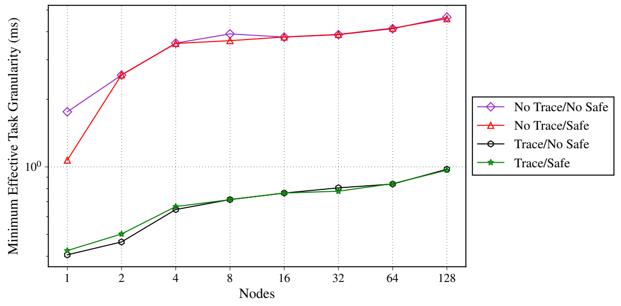


Figure 21. METG(50%) overhead of control determinism checks, both with and without tracing (lower is better).

DCR scales better than Dask's centralized approach, even where Dask's single-node performance is better. Using DCR with CPU-only execution, Legate NumPy is 11.4X faster than Dask at 32 nodes (1280 cores) on logistic regression and 2.7X faster running the preconditioned CG solver. The drop in performance for Legate NumPy at higher node counts is due to application communication unrelated to DCR.

5.5 Overhead of Control Determinism Checks

To measure the overhead of the dynamic checks for control determinism (Section 3), we leverage the `stencil` benchmark from the Task Bench suite [44]. Task Bench uses the metric *minimum effective task granularity* to determine when an application achieves 50% efficiency compared to runtime overheads, abbreviated METG(50%). This metric is better than using strong or weak scaling as it specifically measures the impact of runtime system overhead. Intuitively, the higher the overheads of the runtime system, the higher the METG(50%) will be to reflect how long tasks need to run for the system to perform efficiently. By itself, the `stencil` benchmark has no task parallelism to hide overhead, but by running four independent copies simultaneously, we can simulate an application with a modicum of task parallelism.

Figure 21 plots the METG(50%) of the `stencil` benchmark for two configurations running with *tracing* and without, each with the control determinism checks both enabled (Safe) and disabled (No Safe). Tracing memoizes aspects of the dependence analysis [31], making other overheads, such as the determinism checks, more noticeable. In both cases, we see that METG(50%) increases with node count; this is expected as longer-running tasks are required to hide longer communication latencies. Furthermore, we also notice that control determinism checks have negligible impact on METG(50%).

6 Related Work

A number of task-based systems use dynamic dependence analysis, including OmpSs [15], OpenMP 4.0 [1], PaRSEC (with dynamic task discovery) [25], and StarPU [8]. Of these, only PaRSEC and StarPU support distributed memory execution. PaRSEC and StarPU do not support dynamic control replication, but do permit users to manually prune the set of tasks that must be analyzed on each node. PaRSEC and

StarPU programs execute in a single-program multiple-data (SPMD) fashion where each task is explicitly assigned to a given node to execute. Users may prune any tasks that are not direct dependences or dependents of the tasks executed on the local node. In the absence of such pruning, every node must analyze the tasks on every other node. PaRSEC and StarPU provide a restricted data model in which every region resides on a specific owner node—this restriction simplifies the programming model and also enables a mechanical (if not fully automatic) method to prune the tasks to be executed, based on dynamically checking the ownership of each region. However, due to the cost of these dynamic checks, application and/or domain-specific knowledge may still be required to achieve optimal scalability.

DCR automatically exploits SPMD-style parallelism in task-based programs, which is distinct from the recursive divide-and-conquer style of parallelism that task-based systems also naturally support. It is possible that optimizations for divide-and-conquer parallelism [23] could be combined with DCR. There is also a literature on the parallel construction of task graphs in a shared memory setting (e.g., [24, 32]).

As mentioned in Section 1, many systems use a compile-time approach to extract implicit parallelism [13, 14, 20]. The closest to DCR is static control replication in Regent [43]. Static control replication is a compile-time analysis that transforms programs into explicitly parallel programs. Our dynamic analysis avoids the limitations that static analysis imposes on control flow and partitions of regions.

Also discussed in Section 1, TensorFlow [7], PyTorch [38], and Spark [53] provide implicit parallelism using lazy evaluation. These programming systems run a program on a single control node and then distribute bulk work to remote nodes efficiently. For programs that can be easily partitioned to require minimal inter-node communication and synchronization (e.g., trivially parallel tasks), this approach can achieve competitive performance. However the use of a centralized controller can limit scalability. Some approaches have explored mitigating this bottleneck by replaying the dependence analysis for repeated sequences of tasks [33]. However, for more general programs, DCR makes it easier to express, distribute, and scale a program.

There is a large collection of explicitly parallel programming models and systems for distributed memory architectures [16, 17, 29, 45, 51]. These systems are very diverse, but generally provide a path to performance by exposing low-level mechanisms for synchronization, data movement, and computation and avoid runtime system overheads associated with implicit parallelism such as dependence analysis. Explicit approaches place the burden of performing the equivalent of dependence analysis on the programmer: The programmer must choreograph computation and data to ensure correctness. In another class of system the programmer constructs dependence graphs explicitly [28, 34, 35, 39, 48]. Our experience is that manually constructing both correct and optimized dependence graphs is difficult: correctness bugs occur when dependences are missed, and performance bugs occur when unnecessary dependences are specified. For complex applications, such as those presented in Section 5, building, and especially maintaining, dependences by hand would be challenging in explicitly parallel systems.

Control determinism is related to structural correctness in SPMD programs [9]. Both require that replicated control code must behave the same on all shards. Structural equivalence is a stronger condition, as it captures the core of a static analysis for checking explicitly parallel SPMD programs for incorrect synchronization patterns.

7 Conclusion

We have presented dynamic control replication, a novel execution technique and run-time program analysis for computing dependences on-the-fly for implicitly parallel programs run on large scale machines. We described an asymptotically scalable algorithm for implementing dynamic control replication and detailed its implementation in the Legion runtime. Using our implementation, we have demonstrated that dynamic control replication is competitive with existing programming systems on common benchmarks, and can also provide better performance on challenging programs that are not easily handled by existing implicitly parallel programming systems. Most importantly, we have shown that dynamic control replication can be leveraged to productively construct large scale programming systems for a wide range of problems, ranging from high performance computing to machine learning and deep learning. We believe that scaling implicit parallelism with dynamic control replication will be crucial for delivering high performance to a large class of users that require massive parallelism to tackle computationally difficult problems, but lack the expertise needed for programming large-scale machines.

Acknowledgments

The idea for dynamic control replication originated in conversations with Sean Treichler. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of ASCR, under the guidance of Dr. Laura Biven, under the contract number DE-AC02-76SF00515 and 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, under prime contract DE-AC05-00OR22725, and UT Battelle subawards 4000151974 and 89233218CNA000001. Los Alamos National Laboratory provided support under Department of Energy award DENA0002373-1. Lawrence Livermore National Laboratory supported experiments on Lassen and Sierra, Oak Ridge National Laboratory supported experiments on Summit, and the Swiss National Supercomputing Centre (CSCS) supported experiments on Piz Daint under project ID d80.

References

[1] 2013. OpenMP Application Program Interface. http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf.

[2] 2013. Safe Object Finalization in Python. https://www.python.org/dev/peps/pep-0442/.

[3] 2019. CANDLE: Exascale Deep Learning and Simulation Enabled Precision Medicine for Cancer. https://candle.cels.anl.gov/.

[4] 2019. Uno: Predicting Tumor Dose Response across Multiple Data Sources. https://github.com/ECP-CANDLE/Benchmarks/tree/master/Pilot1/Uno.

[5] 2020. June 2020 Top 500 Supercomputers. https://www.top500.org/lists/top500/2020/06/.

[6] 2020. Regent Stencil Example. https://gitlab.com/StanfordLegion/legion/-/blob/master/language/examples/stencil.rg.

[7] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[8] 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.

[9] Alex Aiken and David Gay. 1998. Barrier Inference. In *Proceedings of the Symposium on Principles of Programming Languages.* 342–354.

[10] 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)*. ACM, New York, NY, USA, Article 23, 23 pages. https://doi.org/10.1145/3295500.3356175

[11] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Supercomputing (SC)*.

[12] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2014. Structure slicing: Extending logical regions with fields. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 845–856.

[13] 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 *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 97–116. https://doi.org/10.1145/1640089.1640097

[14] 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.

[15] Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. 2013. Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (Eugene, Oregon, USA) *(ICS '13)*. ACM, New York, NY, USA, 359–368. https://doi.org/10.1145/2464996.2465017

[16] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (Aug. 2007), 291–312. https://doi.org/10.1177/1094342007078442

[17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. ACM, New York, NY, USA, 519–538. https://doi.org/10.1145/1094811.1094852

[18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition.* IEEE, 248–255.

[19] Mario Di Renzo, Lin Fu, and Javier Urzay. 2020. HTR solver: An open-source exascale-oriented task-based multi-GPU high-order code for hypersonic aerothermodynamics. *Computer Physics Communications* 255 (2020), 107262. https://doi.org/10.1016/j.cpc.2020.107262 (In Press).

[20] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing.*

[21] Charles Ferenbaugh. 2016. The PENNANT Mini-App. https://github.com/lanl/PENNANT/blob/master/doc/pennantdoc.pdf.

[22] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.

[23] Nikhil Hegde, Qifan Chang, and Milind Kulkarni. 2019. D2P: From recursive formulations to distributed-memory codes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–22.

[24] Stephen T Heumann, Alexandros Tzannes, and Vikram S Adve. 2015. Scalable task scheduling and synchronization using hierarchical effects. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 125–137.

[25] 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

[26] Z. Jia, S. Treichler, G. Shipman, M. Bauer, N. Watkins, C. Maltzahn, P. McCormick, and A. Aiken. 2017. Integrating External Resources with a Task-Based Programming Model. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. 307–316. https://doi.org/10.1109/HiPC.2017.00043

[27] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2018. Beyond Data and Model Parallelism for Deep Neural Networks. In *SysML 2018.*

[28] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models* (Eugene, OR, USA) *(PGAS '14)*. ACM, New York, NY, USA, Article 6, 11 pages. https://doi.org/10.1145/2676870.2676883

[29] L.V. Kalé and S. Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93*, A. Paepcke (Ed.). ACM Press, 91–108.

[30] Wonchan Lee, Manolis Papadakis, Elliott Slaughter, and Alex Aiken. 2019. A Constraint-based Approach to Automatic Data Partitioning for Distributed Memory Execution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. ACM, New York, NY, USA, Article 45, 24 pages. https://doi.org/10.1145/3295500.3356199

[31] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Dallas, Texas) *(SC '18)*. IEEE Press, Piscataway, NJ, USA, Article 34, 13 pages. http://dl.acm.org/citation.cfm?id=3291656.3291702

[32] Jonathan Lifflander and Sriram Krishnamoorthy. 2017. Cache locality optimization for recursive programs. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1–16.

[33] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. 2017. Execution Templates: Caching Control Plane Decisions for Strong Scaling of Data Analytics. In *USENIX Annual Technical Conference (USENIX ATC)*.

[34] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, Min Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo. 2016. The Open Community Runtime: A runtime system for extreme scale computing. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2016.7761580

[35] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *CoRR* abs/1712.05889 (2017). arXiv:1712.05889 http://arxiv.org/abs/1712.05889

[36] NumPy 2019. NumPy v1.16 Manual. https://docs.scipy.org/doc/numpy/.

[37] NVIDIA 2019. GPUDirect. https://developer.nvidia.com/gpudirect.

[38] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic Differentiation in PyTorch. In *NIPS Autodiff Workshop*.

[39] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked algorithms and Task Scheduling. In *Proceedings of the 14th Python in Science Conference*, Kathryn Huff and James Bergstra (Eds.). 130 – 136.

[40] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. 2011. Parallel Random Numbers: As Easy As 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (Seattle, Washington) *(SC '11)*. ACM, New York, NY, USA, Article 16, 12 pages. https://doi.org/10.1145/2063384.2063405

[41] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *CoRR* abs/1802.05799 (2018). http://arxiv.org/abs/1802.05799

[42] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. 2015. Regent: A High-productivity Programming Language for HPC with Logical Regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Austin, Texas) *(SC '15)*. ACM, New York, NY, USA, Article 81, 12 pages. https://doi.org/10.1145/2807591.2807629

[43] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. ACM, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/3126908.3126949

[44] E. Slaughter, W. Wu, Y. Fu, L. Brandenburg, N. Garcia, E. Marx, K.S. Morris, Q. Cao, G. Bosilca, S. Mirchandaney, W. Lee, S. Treichler, P. McCormick, and A. Aiken. 2020. Task Bench: A Parameterized Benchmark for Evaluating Parallel Runtime Performance. In *Proceedings of the International Conference on Supercomputing*.

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

[46] The HDF Group. 1997-2020. Hierarchical Data Format, version 5. http://www.hdfgroup.org/HDF5/.

[47] Hilario Torres, Manolis Papadakis, Lluis Jofre, Wonchan Lee, Alex Aiken, and Gianluca Iaccarino. 2019. Soleil-X: Turbulence, Particles, and Radiation in the Regent Programming Language. In *Proceedings of PAW@SC 2019: Parallel Applications Workshop, Held in conjunction with SC19: The International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, Colorado, USA, November 16-22, 2019*. ACM.

[48] S. Treichler, M. Bauer, and Aiken A. 2014. Realm: An Event-Based Low-Level Runtime for Distributed Memory Architectures. In *Parallel Architectures and Compilation Techniques (PACT)*.

[49] S. Treichler, M. Bauer, and A. Aiken. 2013. Language Support for Dynamic, Hierarchical Data Partitioning. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[50] S. Treichler, M. Bauer, Sharma R., Slaughter E., and A. Aiken. 2016. Dependent Partitioning. In *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[51] Katherine Yelick, Dan Bonachea, Wei-Yu Chen, Phillip Colella, Kaushik Datta, Jason Duell, Susan L. Graham, Paul Hargrove, Paul Hilfinger, Parry Husbands, Costin Iancu, Amir Kamil, Rajesh Nishtala, Jimmy Su, Michael Welcome, and Tong Wen. 2007. Productivity and Performance Using Partitioned Global Address Space Languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation* (London, Ontario, Canada) *(PASCO '07)*. ACM, New York, NY, USA, 24–32. https://doi.org/10.1145/1278177.1278183

[52] Yuan Yu, Martin Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. 2018. Dynamic Control Flow in Large-Scale Machine Learning. In *Proceedings of EuroSys 2018*. https://arxiv.org/pdf/1805.01772.pdf

[53] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

A Formalization of Dynamic Control Replication

This appendix gives the remaining details of the proof of Theorem 1. As preliminaries we define consistency of analysis states (Definitions 2 and 3) that are maintained by transitions both in $\mathsf{DEP}_{\mathsf{seq}}$ and $\mathsf{DEP}_{\mathsf{rep}}$ (Lemmas 1 and 2). We also establish an equivalence between a single transition in $\mathsf{DEP}_{\mathsf{seq}}$ and multiple transitions in $\mathsf{DEP}_{\mathsf{rep}}$ (Lemma 4).

For simplicity, we extend each analysis state in $\mathsf{DEP}_{\mathsf{seq}}$ with a prefix c of task groups that have been analyzed so far. With the extension, the system $\mathsf{DEP}_{\mathsf{seq}}$ is defined as follows:

$$\boxed{\frac{T' = T \cup tg \quad D' = D \cup T \overset{\Rightarrow}{\times} tg}{(tg; p, c, \langle T, D\rangle) \xrightarrow{\mathsf{seq}} (p, c; tg, \langle T', D'\rangle)}}$$

We also treat a set c_i of completed tasks in the state of each shard i as a sequence of task groups whenever necessary.

In the rest of this appendix, we write $R^\star$ for the reflexive transitive closure of a relation R.

Definition 1. *A task graph $\langle T, D\rangle$ is valid iff*

$$\forall t, t'.(t \in T \wedge t' \in T \wedge t \Rightarrow t') \iff (t, t') \in D$$

Definition 2. *An analysis state in* $\mathsf{DEP}_{\mathsf{seq}}$

$$(p', c, \langle T, D\rangle)$$

is consistent for a program p iff

1. *$\langle T, D\rangle$ is valid*
2. *$c; p' = p \wedge T = c$*

Lemma 1. *If an analysis state (p', c, G) is consistent for a program p, then there exists another consistent state (p'', c', G') such that*

$$(p', c, G) \xrightarrow{\mathsf{seq}} (p'', c', G')$$

Proof. The proof follows from the definition of the transition relation $\xrightarrow{\mathsf{seq}}$. □

Definition 3. *An analysis state in* $\mathsf{DEP}_{\mathsf{rep}}$

$$(\langle (p_1, c_1, d_1), \ldots, (p_n, c_n, d_n)\rangle, \langle T, D\rangle)$$

is consistent for a program p iff

1. *$\langle T, D\rangle$ is valid*
2. *$T = \bigcup_i \{tg(i) \mid tg \in c_i\}$*
3. *for all i,*

$$c_i; p_i = p \wedge (d_i = \varnothing \vee p_i = tg_i; p'_i \wedge d_i = c_i \overset{\Rightarrow}{\times} tg_i(i))$$

Lemma 2. *If an analysis state (S, G) is consistent for a program p, then there exists another consistent state (S', G') such that*

$$(S, G) \xrightarrow{\mathsf{rep}} (S', G')$$

Proof. The proof follows from the definition of the transition relation $\xrightarrow{\mathsf{rep}}$. □

Definition 4.

$$\mathsf{dist}(s) \triangleq |c| \textit{ where } s = (p, c, d)$$

Lemma 3.

$$\begin{aligned}
&(\langle \ldots, s_i, \ldots, s_j, \ldots\rangle, G)\\
&\xrightarrow{\mathsf{rep}} (\langle \ldots, s'_i, \ldots, s_j, \ldots\rangle, G')\\
&\xrightarrow{\mathsf{rep}} (\langle \ldots, s'_i, \ldots, s'_j, \ldots\rangle, G'')\\
&\wedge\ \mathsf{dist}(s_i) \geq \mathsf{dist}(s_j)\\
&\Longrightarrow\\
&\exists G'''.(\langle \ldots, s_i, \ldots, s_j, \ldots\rangle, G)\\
&\xrightarrow{\mathsf{rep}} (\langle \ldots, s_i, \ldots, s'_j, \ldots\rangle, G''')\\
&\xrightarrow{\mathsf{rep}} (\langle \ldots, s'_i, \ldots, s'_j, \ldots\rangle, G'')
\end{aligned}$$

where all analysis states are consistent for a program.

Proof. The proof proceeds by case analysis on the transition relation $\xrightarrow{\mathsf{rep}}$.

When one of the transitions uses the rule Ta, the proof is trivial as Ta does not change the task graph.

We now prove the lemma when both transitions use the rule Tc. Let s_i and s_j be $(tg_i; p_i, c_i, \varnothing)$ and $(tg_j; p_j, c_j, \varnothing)$, respectively, and suppose $G = \langle T, D\rangle$. From the definition of Tc, we have $G'' = \langle T \cup tg_i(i) \cup tg_j(j), D\rangle$ and $G''' = \langle T \cup tg_j(j), D\rangle$, satisfying the consequent of the lemma.

We now consider cases where the second transition uses the rule Tb. Let s_i and s_j be $(tg_i; p_i, c_i, d_i)$ and (p_j, c_j, d_j), respectively. Whether the first transition uses the rule Tb or the rule Tc, we have $s'_i = (p_i, c_i \cup tg_i, \varnothing)$. To be able to reorder two transitions as in the consequent, it must be true that for all t^i such that $(t^i, t) \in d_j$ we have $t^i \notin tg_i(i)$, where $s_j = (tg_j; p_j, c_j, d_j)$. From the consistency of the analysis state, every t^i is included in c_j. Because $\mathsf{dist}(s_i) \geq \mathsf{dist}(s_j)$, the set c_i of completed tasks in s_i is a superset of the set c_j of completed tasks in s_j. Therefore, t^i cannot appear in tg_i and must exist in c_i, which proves the lemma.

The remaining case is where the first transition uses Tb and the second transition uses Tc. Let s_i and s_j be $(tg_i; p_i, c_i, d_i)$ and $(tg_j; p_j, c_j, d_j)$, respectively. Similar to the previous case, we must show that for all t^j such that $(t^j, t) \in d_i$ we have $t^j \in c_j \cup tg_j$, which is trivial because we already have $t^j \in c_j$ for all t^j from the premise of Tb. Therefore, the lemma holds. □

Lemma 4. *For a program $p = c; tg; p'$ the following equivalence holds:*

$$\begin{aligned}
&(tg; p', c, G) \xrightarrow{\mathsf{seq}} (p', c; tg, G')\\
&\iff\\
&(\langle (tg; p', c, \varnothing), \ldots, (tg; p', c, \varnothing)\rangle, G)\\
&\xrightarrow{\mathsf{rep}}{}^{\star}\ (\langle (p', c \cup tg, \varnothing), \ldots, (p', c \cup tg, \varnothing)\rangle, G')
\end{aligned}$$

where all analysis states are consistent for p.

Proof. We first prove the implication from the LHS to the RHS. Let G be $\langle T, D\rangle$. From the definition of $\xrightarrow{\mathsf{seq}}$, we have

$G' = \langle T', D' \rangle$ where $T' = T \cup tg$ and $D' = D \cup T \overset{\Rightarrow}{\times} tg$. The sets T' and D' can be expanded as follows:

$$\begin{aligned} &T' = T \cup tg(1) \cup \ldots \cup tg(N) \\ &D' = D \cup T \overset{\Rightarrow}{\times} tg(1) \cup \ldots \cup T \overset{\Rightarrow}{\times} tg(N) \\ &= D \cup c \overset{\Rightarrow}{\times} tg(1) \cup \ldots \cup c \overset{\Rightarrow}{\times} tg(N) \\ &(\because \text{the consistency of the analysis state}) \end{aligned}$$

where $tg = \bigcup_{1 \le i \le N} tg(i)$. (We assume the number of shards is N.) Then, we can make transitions with $\xrightarrow{\text{rep}}$ from the initial state on the RHS for each subset $tg(i)$ by applying either the rule Tc or the rule Ta followed by the ruleTb, depending on whether the set $T \overset{\Rightarrow}{\times} tg(i)$ of dependences is empty or not; The rule Tb is always applicable because all shards have c as their sets of completed tasks, on which the dependences of tasks in $tg(i)$ are identified. Therefore, the consequent (the RHS) holds.

We now prove the implication from the RHS to the LHS. Each shard i can apply either the rule Tc or the rule Ta followed by the rule Tb to consume the outstanding task group tg in its transitions. In either case, a graph $G = \langle T, D \rangle$ before the transitions in shard i becomes

$$G_i = \langle T \cup tg(i), D \cup c \overset{\Rightarrow}{\times} tg(i) \rangle$$

after those transitions. Therefore, once all N shards make their transitions, the final graph G' becomes the following:

$$\begin{aligned} &G' = \langle T \cup \textstyle\bigcup_{1 \le i \le N} tg(i), D \cup \bigcup_{1 \le i \le N} c \overset{\Rightarrow}{\times} tg(i) \rangle \\ &= \langle T \cup tg, D \cup c \overset{\Rightarrow}{\times} tg \rangle \\ &= \langle T \cup tg, D \cup T \overset{\Rightarrow}{\times} tg \rangle \\ &(\because \text{the consistency of the analysis state}) \end{aligned}$$

Therefore, the consequent (the LHS) holds. □

Lemma 5.

$$\begin{aligned} &(p, c, G) \xrightarrow{\text{seq}}{}^{\star} (p', c', G') \\ &\iff (p; p'', c, G) \xrightarrow{\text{seq}}{}^{\star} (p'; p'', c', G') \end{aligned}$$

Proof. The proof proceeds by the induction on $\xrightarrow{\text{seq}}{}^{\star}$ and then follows from the definition of $\xrightarrow{\text{seq}}$, which only examines the prefix p of the program in the analysis state. □

Lemma 6.

$$\begin{aligned} &(\langle (p_1, c_1, d_1), \ldots, (p_n, c_n, d_n) \rangle, G) \\ &\xrightarrow{\text{rep}}{}^{\star} (\langle (p'_1, c'_1, d'_1), \ldots, (p'_n, c'_n, d'_n) \rangle, G') \\ &\iff \\ &(\langle (p_1; p', c_1, d_1), \ldots, (p_n; p', c_n, d_n) \rangle, G) \\ &\xrightarrow{\text{rep}}{}^{\star} (\langle (p'_1; p', c'_1, d'_1), \ldots, (p'_n; p', c'_n, d'_n) \rangle, G') \end{aligned}$$

Proof. The proof proceeds by the induction on $\xrightarrow{\text{rep}}{}^{\star}$ and then follows from the definition of $\xrightarrow{\text{rep}}$, which only examines the prefix p_i of the program in each shard state. □

Theorem 1. *For a program p, the following holds:*

$$\begin{aligned} &(p, \epsilon, \langle \emptyset, \emptyset \rangle) \xrightarrow{\text{seq}}{}^{\star} (\epsilon, p, G_s) \land (S^N_\iota, \langle \emptyset, \emptyset \rangle) \xrightarrow{\text{rep}}{}^{\star} (S^N_\emptyset, G_r) \\ &\implies G_s = G_r, \end{aligned}$$

$$\textit{where} \quad \begin{aligned} S^N_\iota &= \overbrace{\langle (p, \emptyset, \emptyset), \ldots, (p, \emptyset, \emptyset) \rangle}^{N} \ \textit{and} \\ S^N_\emptyset &= \langle (\epsilon, p, \emptyset), \ldots, (\epsilon, p, \emptyset) \rangle \end{aligned}$$

Proof. The proof proceeds by induction on the length of the program p. Suppose the theorem holds for a program p. Consider the transitions in DEP_{seq} and DEP_{rep} for a program $p; tg$. With DEP_{seq} the last transition must be as follows:

$$(p; tg, \epsilon, \langle \emptyset, \emptyset \rangle) \xrightarrow{\text{seq}}{}^{\star} (tg, p, G'_s) \xrightarrow{\text{seq}} (\epsilon, p; tg, G_s).$$

From Lemma 3, the transitions in DEP_{rep} that consume the last task graph tg can be reordered as follows:

$$\begin{aligned} &(\langle (p; tg, \emptyset, \emptyset), \ldots, (p; tg, \emptyset, \emptyset) \rangle, \langle \emptyset, \emptyset \rangle) \\ &\xrightarrow{\text{rep}}{}^{\star} (\langle (tg, p, \emptyset), \ldots, (tg, p, \emptyset) \rangle, G'_r) \\ &\xrightarrow{\text{rep}}{}^{\star} (\langle (\epsilon, p \cup tg, \emptyset), \ldots, (\epsilon, p \cup tg, \emptyset) \rangle, G_r) \end{aligned}$$

From Lemmas 5 and 6 we have

$$(p, \epsilon, \langle \emptyset, \emptyset \rangle) \xrightarrow{\text{seq}}{}^{\star} (\epsilon, p, G'_s)$$

and

$$\begin{aligned} &(\langle (p, \emptyset, \emptyset), \ldots, (p, \emptyset, \emptyset) \rangle, \langle \emptyset, \emptyset \rangle) \\ &\xrightarrow{\text{rep}}{}^{\star} (\langle (\epsilon, p, \emptyset), \ldots, (\epsilon, p, \emptyset) \rangle, G'_r) \end{aligned}$$

Then, we can derive $G'_s = G'_r$ from the induction hypothesis and the following after substitution:

$$(tg, p, G'_s) \xrightarrow{\text{seq}} (\epsilon, p; tg, G_s).$$

and

$$\begin{aligned} &(\langle (tg, p, \emptyset), \ldots, (tg, p, \emptyset) \rangle, G'_s) \\ &\xrightarrow{\text{rep}}{}^{\star} (\langle (\epsilon, p \cup tg, \emptyset), \ldots, (\epsilon, p \cup tg, \emptyset) \rangle, G_r) \end{aligned}$$

Finally, from Lemma 4 we conclude $G_s = G_r$. □