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.

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.
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 Tᵢ 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 T₁ handles subtasks A, C, and E for the first loop iteration, while shard T₂ 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 T₁ 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 T₁ handling subtasks B, D and E and shard T₂ 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:

\[ p \in \text{Program} = \text{TaskGroup}^* \quad p = \epsilon \ | \ t g; p \]

\[ t g \in \text{TaskGroup} = \{ \text{Task} \} \]

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 \( t_1 \rightarrow t_2 \) to indicate tasks \( t_1 \) and \( t_2 \) are independent. Tasks in a task group must be pairwise independent: \( \forall t_1, t_2 \in t g \cdot t_1 = t_2 \lor t_1 \rightarrow t_2 \). We write \( t_1 \Rightarrow t_2 \) if task \( t_2 \) depends on \( t_1 \), i.e., \( t_1 \) occurs after \( t_2 \) in the program and \( \neg(t_1 \leftrightarrow t_2) \). The dependence analysis’s output is a task graph \( G = (T, D) \), 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 \( t g(k) := \{ t^k \mid t^k \in t g \} \) of a task group \( t g \).

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 \( S = (s_1, \ldots, s_n) \) of states (one for each shard), and a global task graph \( G \) shared by all shards. Each shard \( i \)’s state \( s_i = (p_i, c_i, d_i) \) 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 \( \text{DEP}^\text{rep} \) for the parallel dependence analysis. Each inference rule corresponds to a single analysis step, which starts with an initial state \( S = ((p, \emptyset, \emptyset), \ldots, (p, \emptyset, \emptyset)) \) where the program is replicated in the shards’ states and finishes when all shards have no remaining task groups to analyze. In the rules \( \text{Ta} \) and \( \text{Te} \), we write \( T \rightarrow T' \) to denote a set of dependences between tasks in \( T \) and those in \( T' \) (obtained by querying the oracle):

\[ T \rightarrow T' \triangleq \{(t, t') \mid t \in T \land t' \in T' \land t \Rightarrow t' \} \]

The analysis of a task group in each shard is done in two steps: First, a shard \( i \) identifies dependences for the subset \( t g(i) \) of the task group \( t g \) being analyzed and records them locally in the set \( d_i \) of outstanding dependences (rule \( \text{Ta} \)). 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 \( \text{Tb} \)); to check this condition, each shard \( i \) maintains a set \( c_i \) of tasks whose analysis is completed. The rule \( \text{Tc} \) handles the case where the tasks being analyzed have no dependences, allowing those tasks to be registered to the task graph immediately. In rules \( \text{Ta} \) and \( \text{Tc} \), shard \( i \) compares \( t g(i) \) with \( c_i \) and not with the tasks in the task graph, as shards can make progress.
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.

Theorem 1. For a program \( p \), the following holds:

\[
(p, (\varnothing, \varnothing)) \xrightarrow{\text{seq}} (e, G_s) \land (S_N^N, (\varnothing, \varnothing)) \xrightarrow{\text{rep} \ast} (S_N^{G_r}, G_r)
\]

where

\[
S_N^N = \langle (p, \varnothing, \varnothing), \ldots, (p, \varnothing, \varnothing) \rangle,
\]

\[
S_N^{G_r} = \langle (e, p, \varnothing), \ldots, (e, p, \varnothing) \rangle
\]

and \( R^\ast \) 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 can be reordered so that they simulate an analysis of the same program with the system DEP seq. A detailed proof of Theorem A 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.

### 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.

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 for 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 \( \_\text{demand}(\_\text{replicable}) \) annotation on line 29 indicates that \( \text{main}() \) 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 \( j \)th 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 \( \text{add_one}(\text{owned}[id(\cdot)]) \) over the \( \text{tiles} \) 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.

\(^2\)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 f111 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 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.

A coarse stage dependence from group $G_2$ to $G_1$ requires that the fine stage analysis of all individual tasks in $G_2$ finishes before the fine stage analysis of any tasks in $G_1$. 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.
The second group task launch for owned the same, the first dependence is between two different partitions as well as on the flux dependences on the cells. A fence on the dependence on the fill for the cells is added. Additionally, we insert a cross-shard fence on the cells.state for the pipeline simultaneously.

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 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.
Scaling Implicit Parallelism via Dynamic Control Replication

Figure 12. Scaling of 2D Stencil Benchmark.

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 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.
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 of the HTR solver. HTR with Dynamic Control Replication

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 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...
Scaling Implicit Parallelism via Dynamic Control Replication

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.

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 memorizes 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 DE-NA0002373-1. 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.


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 DEP_seq and DEP_rep (Lemmas 1 and 2). We also establish an equivalence between a single transition in DEP_seq and multiple transitions in DEP_rep (Lemma 4).

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

\[
\begin{align*}
T' &= T \cup tg \\
D' &= D \cup T \xrightarrow{\text{seq}} \text{tg}
\end{align*}
\]

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 $R^*$ for the reflexive transitive closure of a relation $R$.

Definition 1. A task graph $⟨T, D⟩$ is valid iff

\[\forall t, t'. (t \in T \land t' \in T \land t' \Rightarrow t') \implies (t, t') \in D\]

Definition 2. An analysis state in DEP_seq

\[⟨p', c, (T, D)⟩\]

is consistent for a program $p$ iff

1. $⟨T, D⟩$ is valid
2. $c; p' = p \land T = c$

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

\[⟨p', c, G⟩ \xrightarrow{\text{seq}} ⟨p'', c', G'⟩\]

Proof. The proof follows from the definition of the transition relation $\xrightarrow{\text{seq}}$. \hfill $\Box$

Definition 3. An analysis state in DEP_rep

\[⟨⟨p_1, c_1, d_1⟩, \ldots, (p_n, c_n, d_n)⟩, (T, D)⟩\]

is consistent for a program $p$ iff

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

\[c_i; p_i = p \land (d_i = \emptyset \lor p_i = tg; p'_i \land d_i = c_i \xrightarrow{\text{tg}} tg(i))\]

Lemma 2. If an analysis state $⟨S, G⟩$ is consistent for a program $p$, then there exists another consistent state $⟨S', G'⟩$ such that

\[⟨S, G⟩ \xrightarrow{\text{rep}} ⟨S', G'⟩\]

Proof. The proof follows from the definition of the transition relation $\xrightarrow{\text{rep}}$. \hfill $\Box$

Definition 4.

\[\text{dist}(s) \doteq |c| \text{ where } s = (p, c, d)\]

Lemma 3.

\[
\begin{align*}
\langle \ldots, s_i, \ldots, s_j, \ldots \rangle, G &\xrightarrow{\text{rep}} \langle \ldots, s_i', \ldots, s_j', \ldots \rangle, G' \\
\langle \ldots, s_i', \ldots, s_j', \ldots \rangle, G'' &\xrightarrow{\text{rep}} \langle \ldots, s_i'', \ldots, s_j'', \ldots \rangle, G''' \\
\end{align*}
\]

where all analysis states are consistent for a program.

Proof. The proof proceeds by case analysis on the transition relation $\xrightarrow{\text{rep}}$. 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 $(tg_i; p_i, c_i, \emptyset)$ and $(tg_j; p_j, c_j, \emptyset)$, respectively, and suppose $G = ⟨T, D⟩$. From the definition of $Tc$, we have $G'' = (T \cup tg_i(i) \cup tg_j(j), D)$ and $G''' = (T \cup tg_j(j), D)$, 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 $s'_i = (p_i, c_i \cup tg_i, \emptyset)$. To be able to reorder two transitions as in the consequent, it must be true that for all $t'$ such that $(t', t) \in d_i$, 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 $t'$ is included in $c_j$. Because $\text{dist}(s_i) \geq \text{dist}(s_j)$, 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, $t'$ 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 $t'$ such that $(t'^i, t) \in d_i$, we have $t'^i \in c_j \cup tg_j$, which is trivial because we already have $t'^i \in c_j$ for all $t'$ from the premise of $Tb$. Therefore, the lemma holds. \hfill $\Box$

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

\[
\begin{align*}
\langle \ldots, p', c, G \rangle &\xrightarrow{\text{seq}} \langle p', c, tg; G' \rangle \\
\langle \ldots, p', c, G \rangle &\xrightarrow{\text{rep}} \langle (p', c \cup tg, G') \rangle \\
\end{align*}
\]

where all analysis states are consistent for $p$.

Proof. We first prove the implication from the LHS to the RHS. Let $G = ⟨T, D⟩$. From the definition of $\xrightarrow{\text{seq}}$, we have
\[ G' = \langle T', D' \rangle \text{ where } T' = T \cup tg \text{ and } D' = D \cup T \times tg. \]

The sets \( T' \) and \( D' \) can be expanded as follows:

\[
\begin{align*}
T' &= T \cup tg(1) \cup \ldots \cup tg(N) \\
D' &= D \cup T \times tg(1) \cup \ldots \cup T \times tg(N)
\end{align*}
\]

\( = D \cup c \times tg(1) \cup \ldots \cup c \times tg(N) \)

\( (\because \text{ the consistency of the analysis state}) \)

where \( tg = \bigcup_{1 \leq i \leq N} tg(i) \). (We assume the number of shards is \( N \).

Then, we can make transitions with \( \rightarrow_{rep} \) 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 rule \( Tn \), depending on whether the set \( T \times tg(i) \) of dependences is empty or not.

The rule \( Tn \) 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 \( Tn \) 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

\[ G_i = \langle T \cup tg(i), D \cup c \times tg(i) \rangle \]

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

\[ G' = \langle T \cup \bigcup_{1 \leq i \leq N} tg(i), D \cup \bigcup_{1 \leq i \leq N} c \times tg(i) \rangle \]

\( = \langle T \cup tg, D \cup c \times tg \rangle \)

\( (\because \text{ the consistency of the analysis state}) \)

Therefore, the consequent (the LHS) holds.

\[
\text{Lemma 5.} \quad (p, c, G) \rightarrow_{seq}^* (p', c', G') \quad \iff (p; p', c, G) \rightarrow_{seq}^* (p'; p'', c', G')
\]

\[
\text{Proof.} \quad \text{The proof proceeds by the induction on } \rightarrow_{seq}^* \text{ and then follows from the definition of } \rightarrow_{seq}, \text{ which only examines the prefix } p \text{ of the program in the analysis state.}
\]

\[
\text{Lemma 6.} \quad \text{Lemma 6.} \quad ((p_1; c_1, d_1), \ldots, (p_n; c_n, d_n), G) \rightarrow_{rep}^* ((p'_1; c'_1, d'_1), \ldots, (p'_n; c'_n, d'_n), G') \quad \iff ((p_1; p', c_1, d_1), \ldots, (p_n; p', c_n, d_n), G) \rightarrow_{rep}^* ((p'_1; p'; c'_1, d'_1), \ldots, (p'_n; p'; c'_n, d'_n), G')
\]

\[
\text{Proof.} \quad \text{The proof proceeds by the induction on } \rightarrow_{rep}^* \text{ and then follows from the definition of } \rightarrow_{rep}, \text{ which only examines the prefix } p_i \text{ of the program in each shard state.}
\]

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

\[
(p, \epsilon, \langle \emptyset, \emptyset \rangle) \rightarrow_{seq}^* (\epsilon, p, G_r) \quad \iff (\epsilon, p, G_r) \rightarrow_{seq}^* (\epsilon, p, G_r)
\]

\[
\begin{align*}
\text{where} & \quad S^N_r = \langle (p, \emptyset, \emptyset), \ldots, (p, \emptyset, \emptyset) \rangle \\
& \quad S^N = \langle (\epsilon, p, \emptyset), \ldots, (\epsilon, p, \emptyset) \rangle
\end{align*}
\]

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

\[
(p; tg, \epsilon, \langle \emptyset, \emptyset \rangle) \rightarrow_{seq}^* (tg, p, G'_t) \rightarrow_{seq}^* (\epsilon, p; tg, G_r)
\]

\[
\text{From Lemma 3, the transitions in DEP}_{rep} \text{ that consume the last task graph } tg \text{ can be reordered as follows:}
\]

\[
\begin{align*}
& \quad \text{Lemma 3.} \quad ((p; tg, \emptyset, \emptyset), \ldots, (p; tg, \emptyset, \emptyset), \langle \emptyset, \emptyset \rangle) \\
& \quad \quad \rightarrow_{rep}^* ((p; tg, \emptyset, \emptyset), \langle \emptyset, \emptyset \rangle, G'_t) \\
& \quad \quad \quad \rightarrow_{rep}^* ((\epsilon, p, \emptyset), \ldots, (\epsilon, p, \emptyset), G_r)
\end{align*}
\]

\[
\text{Then, we can derive } G'_t = G'_t \text{ from the induction hypothesis and the following after substitution:}
\]

\[
(tg, p, G'_t) \rightarrow_{seq}^* (\epsilon, p; tg, G_r)
\]

\[ \text{and} \]

\[
((p; tg, \emptyset, \emptyset), \ldots, (tg, p, \emptyset), G'_t) \\
\rightarrow_{rep}^* ((\epsilon, p, \emptyset), \ldots, (\epsilon, p, \emptyset), G_r)
\]

\[
\text{Finally, from Lemma 4 we conclude } G_z = G_r. \quad \square
\]