Index Launches: Scalable, Flexible Representation of Parallel Task Groups

Rupanshu Soi
BITS Pilani - Hyderabad Campus
India
f20180294@hyderabad.bits-pilani.ac.in

Michael Bauer
NVIDIA
USA
mbauer@nvidia.com

Sean Treichler
NVIDIA
USA
sean@nvidia.com

Manolis Papadakis
NVIDIA
USA
mpapadakis@nvidia.com

Wonchan Lee
NVIDIA
USA
wonchanl@nvidia.com

Patrick McCormick
Los Alamos National Laboratory
USA
pat@lanl.gov

Alex Aiken
Stanford University
USA
aiken@cs.stanford.edu

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

ABSTRACT

It's common to see specialized language constructs in modern task-based programming systems for reasoning about groups of independent tasks intended for parallel execution. However, most systems use an ad-hoc representation that limits expressiveness and often overfits for a given application domain. We introduce *index launches*, a scalable and flexible representation of a group of tasks. Index launches use a flexible mechanism to indicate the data required for a given task, allowing them to be used for a much broader set of use cases while maintaining an efficient representation. We present a hybrid design for index launches, involving static and dynamic program analyses, along with a characterization of how they're used in Legion and Regent, and show how they generalize constructs found in other task-based systems. Finally, we present results of scaling experiments which demonstrate that index launches are crucial for the efficient distributed execution of several scientific codes in Regent.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**; **Distributed computing methodologies**.

KEYWORDS

index launches, task-based parallelism, runtime systems, regions

ACM Reference Format:
Rupanshu Soi, Michael Bauer, Sean Treichler, Manolis Papadakis, Wonchan Lee, Patrick McCormick, Alex Aiken, and Elliott Slaughter. 2021. Index Launches: Scalable, Flexible Representation of Parallel Task Groups. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3458817.3476175

1 INTRODUCTION

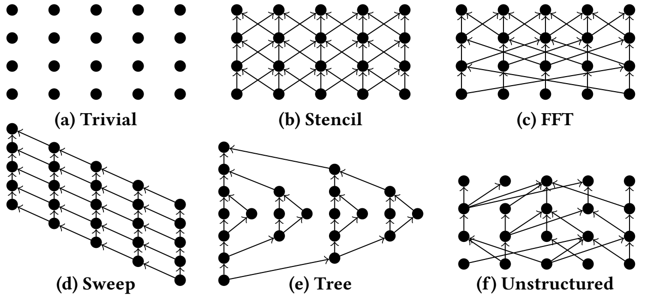
Modern supercomputers are large distributed machines with deep and complex memory hierarchies and heterogeneous processors. Explicitly parallel programming models are traditionally used to program these machines [4, 11, 15, 27]. However, since most of these systems require the programmer to reason about low-level memory and execution details of the target machine, writing large and complex scientific applications is a major challenge even for expert programmers [14]. Recently, a variety of implicitly parallel programming models [2, 3, 5, 7, 9, 16, 24, 32] have presented a solution to this problem by abstracting away details of the machine from the programmer and therefore allowing them to focus primarily on writing application logic.

One design question that must be answered by implicitly parallel programming models is how to encode parallelism in the application. Some systems encode this parallelism at a fine-grained level, such as in the composition of pure functions in Spark [32]. Since a naive execution of such fine-grained operations would be inefficient, these systems automatically coarsen groups of operations into *tasks*, or larger operations that execute in parallel on the machine. Alternatively, *task-based* systems deal natively with such coarse-grained tasks, and leave the job of choosing appropriate task granularities to the user [2, 3, 5, 7, 9, 16, 24].

Although details vary, task-based systems work by constructing a *task graph*—a directed acyclic graph (DAG) representing tasks and their dependencies. Tasks may be enumerated statically or dynamically, but in either case represent an intended program order. The task graph relaxes this order to identify the inherent parallelism that exists in the application. Figure 1 shows examples of task graph patterns that are representative of applications commonly found in high-performance scientific and data analysis workflows. In the figure, time flows from top to bottom along the vertical dimension,

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476175



(a) Trivial (b) Stencil (c) FFT (d) Sweep (e) Tree (f) Unstructured

Figure 1: Examples of common task graph patterns in scientific and data analysis applications.

		Horizontal (Parallelism) Collapsed	
		Yes	No
Vertical (Time) Collapsed	Yes	PaRSEC PTG [9]	TensorFlow [31]
	No	Index Launch	PaRSEC DTD [16] StarPU [3] Dask [24]

Table 1: Task-based systems by dimension(s) of the task graph collapsed.

with arrows identifying dependencies on previously executed tasks. Tasks that are mutually unreachable in the graph may be executed in parallel. By convention, parallel tasks are arrayed along the horizontal dimension.

Whether task graphs are statically or dynamically generated, the manner of generation, as well as the in-memory representation, directly impacts the efficiency and scalability of the overall system. A naive representation of a full task graph scales as a function of $O(\mathcal{P}\mathcal{T})$ where $\mathcal{P}$ is the degree of parallelism (which manifests as the width of the task graph) and $\mathcal{T}$ is the graph's height. To improve scalability, many systems adopt runtime mechanisms that permit one or both dimensions of the task graph to be collapsed in the resulting representation. In general, systems can be categorized by how many (and which) dimensions they collapse, as shown in Table 1.

Parameterized task graphs (PTGs) [9] capture a static, algebraic description of a task graph that can be expanded in an efficient manner at runtime. In this way, a static, $O(1)$ representation of the program is possible—effectively collapsing both dimensions of the dynamic graph. However, since the language for describing such graphs is necessarily restricted, this prevents the representation of a wide variety of task graphs. Most notably, any form of dynamic task generation cannot be represented. Therefore, while the task graph representation is maximally dense, the expressiveness of the system is limited.

TensorFlow [31] collapses the time (vertical) dimension by explicitly capturing control-flow constructs like loops in the task graph. The parallelism (horizontal) dimension is still fully materialized, allowing TensorFlow to support arbitrary dependencies between tasks with full generality. Assuming the application is iterative, the cost of the resulting task graph is $O(\mathcal{P})$, effectively amortizing the cost of wide task graphs over the duration of the program. However, if the program is non-iterative or contains control flow constructs not understood by the underlying system, the technique fails and system falls back to the unoptimized, fully expanded task graph.

An alternative to collapsing the time dimension is available in systems that support dynamic task enumeration. If the runtime is permitted to execute tasks as they are being enumerated and reclaim storage associated with tasks as they are completed, a breadth-first enumeration of the task graph requires only $O(\mathcal{P})$ storage for the unexecuted frontier of the graph being enumerated, even when arbitrarily complicated control flow is present. Note that these benefits are limited to the time dimension—the sliding window of enumerated but not yet executed tasks must necessarily include every element of a set of tasks that is expected to run in parallel.

It is well-known that identical tasks can be broadcast to N processors in $O(\log N)$ time, since N identical tasks can be represented in $O(1)$ space, effectively collapsing the horizontal (parallelism) dimension of the task graph. Our key contribution is to extend this intuitive result to non-identical tasks, particularly tasks with non-trivial and even dynamic data requirements. Our technique integrates with an efficient distributed program analysis [6] to generate an efficient, scalable execution.

Finally, a number of systems do not collapse either dimension of the task graph [3, 16, 24]. Among these systems, the two that achieve high scalability do so by exposing the user to the SPMD semantics of the underlying machine execution model. The program is executed on N nodes and each node is permitted to prune the task graph to exclude tasks not directly dependent on those executed by that node. However, in order to do so, the user must effectively compute the task graph on their own, placing an additional burden on the user and breaking the implicitly parallel abstraction of the original programming model.

We present *index launches*, a scalable and flexible representation of a group of tasks intended for parallel execution. An index launch is a compression algorithm for a naive task graph that compresses an $O(\mathcal{P})$ subset of the task graph representing parallel tasks down to $O(1)$, i.e., it collapses a task graph along the horizontal (parallelism) dimension. The resulting task graph is $O(\mathcal{T})$, where $\mathcal{T}$ is the original task graph's height, though of course, this need not be represented in memory all at once as only the frontier of executing tasks is materialized. Intuitively, this provides additional flexibility in the time dimension: programs need not be iterative to take advantage of index launches. Our technique supports completely arbitrary and dynamic control flow.

The flexibility of index launches is further bolstered by a very general data model. Each task in an index launch may specify the *collections* of data which it requires to operate. We use collections here to refer to any group of data objects used together by a task. The important property of collections is that they may be *partitioned.* Partitions name subsets of objects in a collection, and for the purposes of our technique these subsets may be arbitrary, though it is commonly desirable for them to be disjoint (i.e., each object is contained in at most one sub-collection). Such collections and data partitions permit tasks to identify precisely which data is required to execute.

One of the challenges of index launches is to identify which sub-collections of data a task requires to execute. Rather than limiting the programming model to predefined operators (as in Spark [32]), we permit tasks to select arbitrary sub-collections. To maintain a compact representation of the index launch, we encode this selection as a *projection functor*, a function that maps from a task's index within a launch to an arbitrary sub-collection. Listing 1 presents two examples of index launches with their associated tasks (foo and bar) and arguments (p[i] and q[f(i)]). In the example, the projection functors are λi.i and λi.f(i) (where f is an opaque function), respectively.

```
for i = 0, N do -- parallel
  foo(p[i])
end

for i = 0, N do -- parallel
  bar(q[f(i)])
end
```

Listing 1: Index launches with a trivial and non-trivial projection functor, respectively.

For an index launch to be valid (i.e., all tasks to be able to run in parallel), a compiler or runtime must be able to prove either that accesses are disjoint (no two tasks request overlapping sub-collections), or that the usage of data by tasks is read-only. Facilitated by our data model, there are several ways a compiler or runtime system can prove the safety of an index launch. As a simple example, assuming the partition is disjoint, index launches are trivially safe if the identity projection functor is used (as in the first case above), because, even in the presence of writes, no data accesses can overlap. However, in many cases, the final verdict of safety depends on the projection functors (an index launch might have multiple projection functors, one for each task argument) and how they map collection subsets to tasks. Of the two main approaches for analyzing the projection functor, a static analysis can be done at compile time and will not incur any runtime overhead, but will be inherently limited in scope. On the other hand, a dynamic analysis can easily handle a much larger class of expressions at the cost of runtime performance. We present a hybrid design, in which we first attempt to statically guarantee the safety of the index launch. However, if the static analyzer is unable to fully analyze the projection functor, we generate a dynamic check that analyzes any remaining arguments at runtime. This dynamic check has cost $O(\mathcal{P})$, but we show that the constant factors are small enough to make this check economical at all but the most extreme scales. Additionally, this check can be disabled (if desired) for production runs to eliminate any overheads; i.e., correct execution of the program does not rely on the result of the safety analysis (assuming the program is valid).

We present an evaluation of index launches on up to 1024 nodes of Piz Daint, a heterogeneous supercomputer [1]. Our results demonstrate that index launches improve scalability and performance, even in the presence of an already-distributed and optimized task scheduling infrastructure. Furthermore, the dynamic checks added for safety in our hybrid analysis carry negligible cost at scales relevant to known current and future supercomputers.

Overall, this paper makes the following contributions:

- We characterize the space of abstractions for coarse-grained, implicitly parallel tasks and show that efficient execution requires collapsing one or both dimensions (of time and parallelism).
- We demonstrate how index launches are more expressive, general and orthogonal than existing abstractions found in task-based programming models.
- We describe a hybrid design of index launches leveraging both static and dynamic program analyses.
- We present an implementation of this hybrid design in the Legion runtime system and Regent, a language which targets Legion.
- We evaluate the impact of index launches and show that they are critical for achieving efficient distributed execution of task-based programs, even in the presence of a distributed task scheduling infrastructure. Furthermore, our hybrid analysis incurs cost that is amenable for usage at the scales of all known current and future supercomputers.

2 PROGRAMMING MODEL

Index launches are intended to be embedded in a larger task-based programming model. In this section we describe the relevant features of a core programming model largely based on the Legion runtime system. Note that not all these features are required for index launches to function or be useful, but in many cases, the flexibility afforded by them increases the generality and expressiveness that index launches can provide.

The unit of program execution in our model is a task, which is just a function marked for parallel execution by the user. Even though there is an implicit program order, the compiler and runtime system are free to execute tasks in any relaxed order that is consistent with the original sequential semantics.

Data is organized into collections, which, to a first approximation, can be any data structure containing sets of objects. The exact nature of objects is unimportant for our work. Generally speaking, collections are indexed, and may represent single- or multi-dimensional arrays, maps, sets, and so on. For example, a stencil code might use a collection consisting of a 2-D grid with objects at each point in the domain containing the relevant physical variables (pressure, velocity, etc.).

Collections are the primary way to pass large data to tasks. Notably, in the Legion runtime collections are not fixed in a specific memory but may be copied and migrated between distributed memories in the machine. While not mandatory for an implementation of index launches, this eases many subsequent implementation details, especially on distributed machines.

The most important property of collections, for the purpose of this paper, is that they can be partitioned. Partitions divide collections into subsets of objects intended to be used together. Many partitions are *disjoint* (i.e., the subsets do not overlap), but they can also be *aliased*. Taking the stencil example from earlier, two useful partitions might consist of dense blocks of the domain (the points computed by each task, a disjoint partition) and the "halos" around each block (an aliased partition). Collections can be partitioned multiple times, and the different subsets are views onto the same underlying data.

The exact method for determining partitions is left unspecified, though there are a number of approaches that might be used, such as language-based techniques [29] or automatic graph partitioners [18, 25]. We assume that the compiler and runtime have a procedure for determining the disjointness of partitions, but otherwise do not need to consider it in detail.

To correctly analyze the effects of a task prior to its execution, tasks must declare *privileges* on each collection they receive as input. A privilege can be one of *read*, *write*, *read-write*, or a reduction on a commutative operator (+, ×, etc.). These privileges are required not only to check the safety of an index launch, but also to ensure that the correct data dependencies are computed. A dependency occurs when tasks in subsequent index launches read data written (or reduced) by a previous launch. In these cases Legion automatically copies the data to make it available (if the dependent task executes on a different node in the machine). Privileges can be verified at compile time to ensure that tasks obey their declarations [26].

3 INDEX LAUNCHES

In general, an index launch can be seen as a map from points in a domain D to instances of invocations of a task T. For example, an index launch of a task with n collection arguments might be written as

$$\mathrm{forall}(D, T, \langle P_1, f_1\rangle, \ldots, \langle P_i, f_i\rangle, \ldots, \langle P_n, f_n\rangle)$$

where P_i is the i^{th} partition and f_i is the corresponding projection functor. For simplicity, we ignore non-collection arguments, which are simply passed to the task by value.

We can immediately see that an index launch is an $\mathcal{O}(1)$ representation of a task group of $|D|$ tasks, and the projection functor f_i controls how sub-collections of P_i are assigned to different instances of the task T. We also note that $\mathcal{P} = |D|$, i.e., the domain determines the degree of parallelism.

Index launches enable the compiler and runtime to efficiently determine if two tasks are safe to run in parallel. In general, tasks are safe to run in parallel when they are *non-interfering*, that is, when no task accesses (with any privilege) data written by another task in the same launch. This requirement can be expressed as two checks:

Self-checks. For each argument $\langle P_i, f_i\rangle$, either

- the privilege must be read (or a reduction), or
- the partition P_i must be disjoint and the projection functor f_i injective over launch domain D.

Cross-checks. For every pair of arguments $\langle P_i, f_i\rangle$ and $\langle P_j, f_j\rangle$, either

- the privileges must be both read, or both a reduction (with the same reduction operator),
- P_i and P_j must be disjoint (i.e., partitions of collections that are themselves disjoint), or
- $P_i = P_j$, P_i must be disjoint and the images of the respective projection functors on D must also be disjoint sets (i.e., no two projection functors may select the same sub-collection).

In practice, checking the privileges requested by T is straightforward. As noted above, the disjointness of a partition P is a function of the partitioning language that is used, and can be assumed to be provided. In many practical cases, these first two conditions are sufficient: either the partitions are disjoint or the privileges are reads or reductions. Hence, the main remaining case that is unresolved is the problem of guaranteeing that the projection functor f_i is injective. We guarantee injectivity via a combination of static and dynamic analysis as described below.

4 COMPILER IMPLEMENTATION

Although our design of index launches from Section 3 can be implemented with only runtime support, performance and usability can be improved in many common cases by performing various compile time checks. Most notably, an approach based on hybrid compiler optimizations enables the automatic generation of index launches from apparently sequential loops such as those in Listings 1 and 2. This section presents an overview of how a compiler can support index launches, including our hybrid design involving static and dynamic program analyses, which we have implemented in Regent [26].

Broadly speaking, the compiler performs a static dependence analysis in-line with the description in Section 3. By default, any loop in the program source whose body contains a task launch and other simple statements (such as variable declarations), and that contains no loop-carried dependencies (other than reductions), is eligible to be executed as an index launch. Let us consider a simple example and walk through the steps the compiler takes to try to prove that the corresponding index launch will be safe.

```
task foo(C1, C2) reads(C1) writes(C2) do
  -- computation
end

for i = 0, 5 do
  foo(p[i], q[i%3])
end
```

Listing 2: A task launch loop with the task definition.

To begin, each partition argument must be checked against itself. In our example, the first argument `p` passes this test as the privilege requested on it by `foo` is `reads`. However, since `foo` requests `write` privileges on the second argument `q`, the compiler must check if `q` is a disjoint partition and that `i%3` is injective over the domain $[0, 5)$. Since the compiler has a method of determining the disjointness of any partition, our problem is either solved (if the partition is not disjoint) or quickly reduced to analyzing an arbitrary expression, i.e., the projection functor, over the launch domain (if the partition is disjoint).

To analyze the projection functor, the compiler first subjects it to a simple static analysis that can recognize *trivial* projection functors like constant (not injective), identity (injective), or the slightly more general affine case (injective, iff it does not degenerate to a constant). The strength of this static analysis is left unspecified in our design; a number of well-known static analysis techniques, such as polyhedral methods, can be used to determine the safety of these loops [8]. The exact level of power and expressiveness of this analysis is less important in our case than in more traditional compiler settings because we augment this static analysis with a precise dynamic analysis that handles all remaining cases.

This dynamic analysis works well in our case because we are dealing with coarse-grained tasks and data: the checks occur at the

```
1  var conflict = false
2  var volume = q.volume() -- size of q
3  var bitmask = malloc(volume)
4
5  for i = 0, volume do
6    bitmask[i] = false
7  end
8
9  var value
10 for i = 0, 5 do -- loop over launch domain
11   -- eval and linearize projection functor
12   value = linearize(i%3)
13   if value >= 0 and value < volume
14   then -- bounds check
15     conflict = bitmask[value]
16     bitmask[value] = true
17     if conflict then
18       break
19     end
20   end
21 end
22
23 if conflict then
24   -- original task loop
25 else
26   -- index launch
27 end
```

Listing 3: Generated code for dynamic projection functor check.

granularity of partitions, and are thus amortized over the elements of the sub-collections of those partitions, no matter how large the collections are. In general the cost of the dynamic analysis is $O(|D|)$ in the size of the launch domain, with only small constant factor differences depending on the precise expressions used. It is important to note that the dynamic analysis is only done if the static analysis fails (i.e., it is a hybrid technique), making it even less expensive.

The dynamic analysis, in essence, is a simple loop that evaluates the projection functor at each domain point and determines if it is injective, i.e., whether it evaluates to the same value twice or not. Despite its simplicity, the analysis is sound and complete with respect to determining injectivity of the projection functor, allowing us to completely support arbitrary projection functors in index launches.

A straightforward implementation of the dynamic analysis is as a program transformation during an optimization pass inside the compiler—it replaces the abstract syntax tree (AST) of the loop with the AST of a check followed by a branch that selects whether to execute the index launch or the usual task loop, depending on the result of the check. The generated AST is functionally equivalent to the code in Listing 3.

The check works by intializing a bitmask and then iterating over the domain of the index launch. At each domain point, we evaluate the projection functor and check if the bitmask is already set for that point. If it is, then we must have already visited it, which proves that our projection functor is not injective, and as a result, the index launch is not safe. Otherwise, we set the bitmask and continue to the next iteration. At the end, we simply choose between an index launch or a usual task launch loop depending on the result of the check.

If the partition under consideration P has dimension $N > 1$, we can't directly use the value of the projection functor to set the bitmask, as the value will be a tuple of dimension N, whereas the bitmask is a linear array. In these cases, we *linearize* the projection functor to obtain a scalar value. This linearization procedure (line 12 in Listing 3) uses the bounds of the partition to bijectively map the N-dimensional projection functor to a scalar.

Astute readers will have already realized that because `i%3` is not injective over the domain $[0, 5)$, the task loop in our example is ineligible to be executed as an index launch. To summarize the compiler's reasoning, we cannot index launch the task loop because:

- `foo` requests `write` privileges on its second argument,
- the projection functor of the second argument is non-injective over the given launch domain,
- therefore, there will exist two simultaneous invocations of `foo` that will receive as input the same underlying piece of data,
- and hence, two tasks may attempt to simultaneously write to the same piece of data, leading to a race condition.

Another interesting case for an index launch is when multiple task arguments use the same disjoint partition, but with different projection functors. If any of the arguments has `write` or `reduce` privileges, it becomes necessary to cross-check the different projection functors against each other. Although a naive pairwise implementation with quadratic time complexity is possible, we achieve linear time by checking arguments on a given partition using a single bitmask. The key insight is that writes to a given sub-collection must be exclusive, while reads need not be (as long as the only other accesses are reads). For simplicity, we consider reductions to be writes for the purposes of these checks. Therefore, using a bitmask per partition, we repeat the main body of the dynamic check (lines 10—21 in Listing 3) for each argument with the following modifications:

- we do not set the bitmask (line 16) for read-only arguments, because we do not want such arguments to trigger conflicts with subsequent reads, and
- we check all write or reduce arguments before any read-only arguments.

This ordering allow us to catch all write-write and write-read conflicts (and similarly for reductions), since the bitmask is set for writes prior to checking reads.

Now, we briefly analyze the asymptotic complexity of our dynamic checks. Checking each argument requires initializing its bitmask, which takes $O(|P_i|)$ time and space, where P_i is the partition passed to the i^{th} argument of the task. Assuming that evaluating the projection functor takes constant time, iterating through the launch domain D will take $O(|D|)$ time, resulting in a $O(|D| + |P|)$

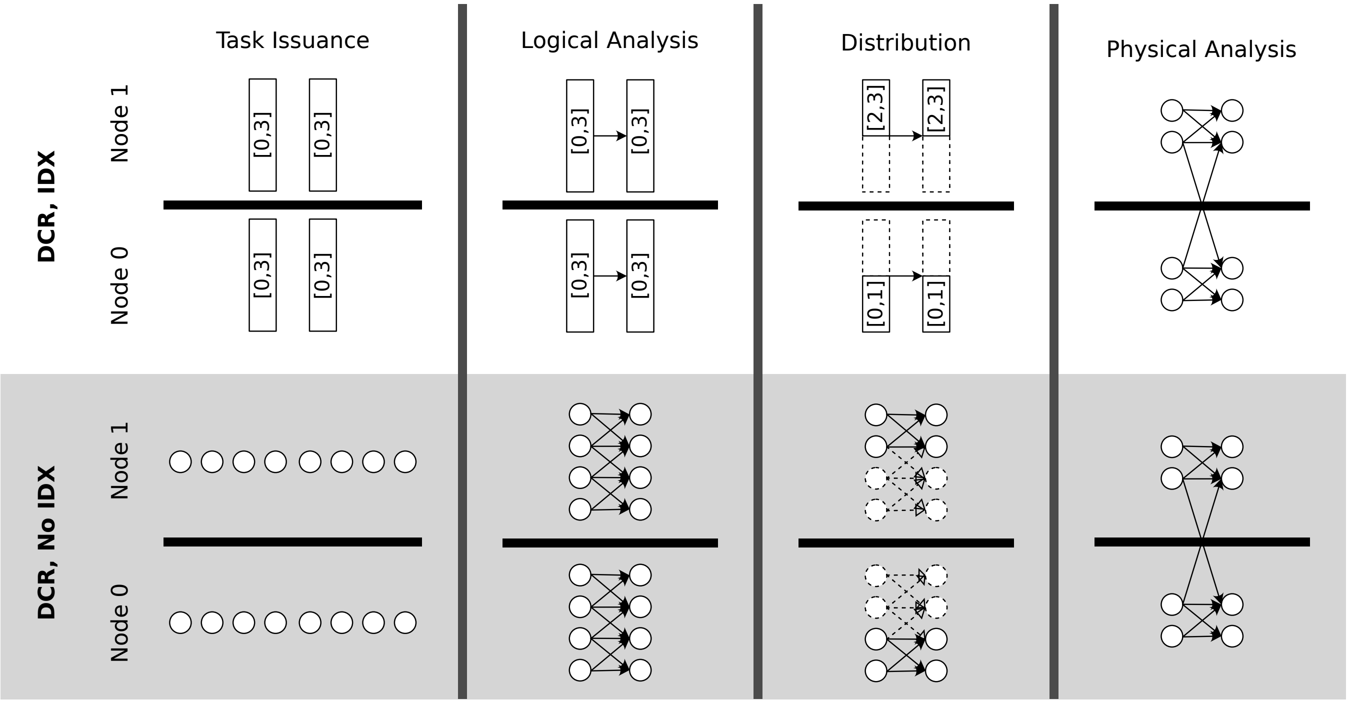


Figure 2: Illustration of runtime pipeline stages for a set of two index launches or eight single tasks, with DCR.

runtime complexity for checking a single argument. Therefore, if a task has n arguments and each uses a separate partition, the worst-case complexity is $O(n|D| + \sum_{i=1}^{n} |P_i|)$. On the other hand, if all n arguments use the same partition P, the complexity reduces to $O(n|D| + |P|)$, because we only need to initialize the bitmask once. Note that n is a static property of the program, thus for the purposes of scalability $O(n|D| + |P|)$ reduces to $O(|D| + |P|)$. Moreover, since $|P|$ is often in $O(|D|)$, $O(|D| + |P|)$ can be further reduced to $O(|D|)$. Finally, using $\mathcal{P} = |D|$ from before, our overall complexity can be written as $O(\mathcal{P})$.

However, as described earlier, the dynamic analysis code emitted by the compiler is a pure safety check, and the correct execution of the program does not rely on it. Therefore, it can be disabled (after the program has been verified at least once) to eliminate this cost, leaving the remaining space and time complexity of the index launch as $O(1)$.

Overall, the main effort of the compiler is to identify candidates for being executed as an index launch, and then verifying that it is safe to do so. Due to the generality of our hybrid approach, we are able to support arbitrary dependencies with negligible runtime cost, even at extreme scales (see Section 6).

5 RUNTIME IMPLEMENTATION

In contrast to the compiler, which is primarily concerned with the safety of index launches, the Legion runtime is additionally responsible for executing them efficiently. Index launches are a compact representation of a set of tasks, so they must be expanded to be executed. The goal is then to expand them in a manner that is efficient and not overly eager, so that one node is not wholly responsible for all tasks in the launch. A second responsibility of the runtime is to efficiently identify dependencies between tasks in different index launches (tasks within a launch cannot have dependencies because they are parallel). Note that the runtime assumes that safety checks have already been performed in a previous stage, either statically by the compiler or dynamically in code emitted by the compiler, and therefore operates under the assumption that these launches are valid.

Legion provides two modes of analysis: a newer, distributed, mode called dynamic control replication (DCR) [6], and the original, centralized mode [7]. Index launches interoperate with both of these modes, though the mechanisms are different.

The runtime uses an internal pipeline to process tasks. The relevant phases of this pipeline, from the perspective of index launches are task issuance, logical analysis, distribution, and physical analysis. Once physical analysis is completed, dependencies have been computed and task execution (and any necessary data movement) can begin.

Figures 2 and 3 show an illustration of the four pipeline stages on a sample program that contains eight tasks arranged either into two index launches (in rows marked IDX), or eight individual task launches (marked No IDX). The execution is distributed between two nodes, labeled node 0 and node 1. In the figure, individual tasks are indicated by circles, and index launches by rectangles. The domain of an index launch is marked on each index launch. Note that the runtime's in-memory representation of an index launch is a fixed size, regardless of how many tasks are represented.

Task issuance is the frontend phase of the runtime. The Regent compiler ultimately compiles programs into a series of API calls to the Legion runtime. In the case of an index launch, a set of tasks can be issued with a single runtime call. Therefore, with index launches

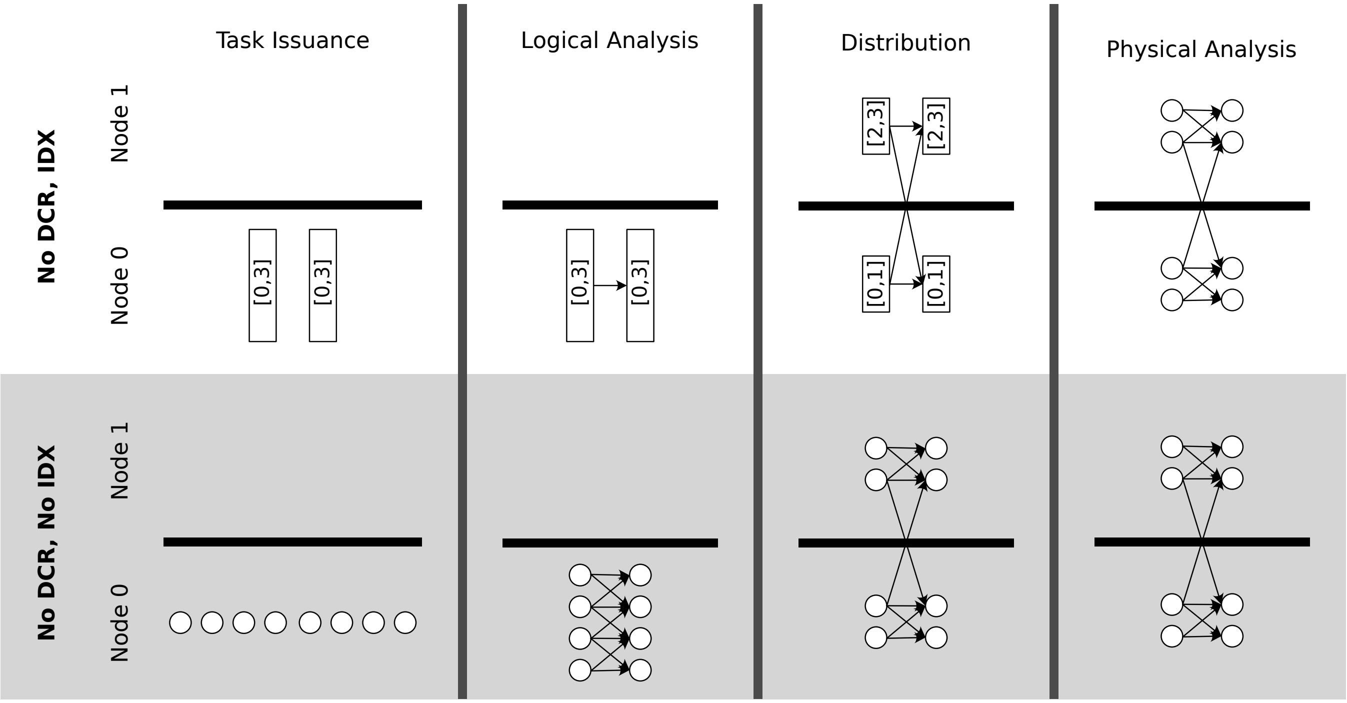


Figure 3: Illustration of runtime pipeline stages for a set of two index launches or eight single tasks, without DCR.

enabled, the runtime receives the task group as a single bulk unit of work. This process is similar whether or not DCR is enabled, except that with DCR, all nodes in the machine simultaneously issue identical index launches to the runtime. As a result, in DCR, all instances of the runtime (of which there is one per node) become aware of the task simultaneously and without any communication. This process is shown in Figure 2 as the identical index launches that appear in the issuance phase on both nodes 0 and 1.

Logical analysis is the same with or without DCR. The benefit of index launches is that they permit *whole-partition* reasoning at runtime. An index launch on partition P and an index launch on partition Q can be seen to be independent if P and Q are partitions of distinct collections C_1 and C_2, and C_1 and C_2 are themselves disjoint. This phase thus identifies bulk dependencies between index launches, but does not attempt to identify precisely which tasks in a given launch depend on which others. This can be seen in Figures 2 and 3 as there is only a single arrow (representing the logical dependency) between the index launch boxes on each node. In DCR, this phase is simply performed identically on every node, such that all nodes have the dependency information (again without requiring communication).

Distribution is the phase where an index launch is expanded into individual tasks, and those tasks are assigned to nodes. This phase is critical because if performed too early, or in an inefficient manner, we lose the asymptotic benefit of having an $O(1)$ representation for $|D|$ tasks, which means that the runtime can no longer reason about these tasks as a single unit of work. This phase also shows the largest differences between DCR and non-DCR executions.

In either case, distribution in Legion is entirely under the control of the end user. Users supply (or choose a system-provided default) *mapper* which controls all performance decisions that the runtime makes. At the distribution phase of the execution pipeline, this means selecting the assignment of tasks to nodes.

When DCR is employed, the user can supply a sharding functor which determines, for each point in the domain of an index launch, which node owns that point. The selection process can be seen in Figure 2, where the solid portion of the index launch shows that the sub-domain $[0, 1]$ has been selected to execute on node 0 and $[2, 3]$ has been selected to execute on node 1. (Note that the dashed portion, which has not been selected, is shown simply to indicate that the launch is still a single logical unit, and takes no additional space in memory.) Sharding functors are pure functions, which permit this mapping to be memoized for efficiency. As a result, the set of tasks within an index launch that are assigned to a single shard can be computed with complexity $O(|D|_{\text{local}})$ where $|D|_{\text{local}}$ is the number of local tasks assigned to a node. As with previous phases, this is also performed without communication.

Without DCR, users control the assignment of tasks to nodes via a slicing functor. In contrast to the sharding functor, slicing can be performed recursively, allowing tasks to be distributed to nodes in a broadcast tree-like manner. Figure 3 shows that two slices have been created from each launch, and slices representing the sub-domain $[2, 3]$ have been moved to node 1. Unlike DCR, this phase requires communication as slices are sent around the machine. Dependencies are copied during this stage, but not otherwise expanded. This stage therefore occurs in $O(\log |D|)$ time as the fixed-size representation of the index launch is broadcast across the machine, with only changes to the sub-domain of each slice as necessary. Once slices arrive at their final destinations, the runtime expands them into individual tasks. This expansion is not shown in the figure for clarity about how the slices are moved, and happens immediately after the distribution phase.

Finally (for the purposes of this discussion), the runtime performs physical analysis. In this stage of the pipeline, dependencies are identified on specific tasks. The process is similar with or without DCR, and in either case, communication is required. To compute dependencies, Legion tracks the last tasks to have read, written or reduced to a given sub-collection of a partition. This metadata is distributed around the machine in order to facilitate efficient analysis. Thus, with or without DCR this phase of the pipeline proceeds in a distributed manner, with complexity that is proportional to $O(|D|_{\text{local}} \log |P|)$ where $|D|_{\text{local}}$ is the number of local tasks per node and $|P|$ is the size of partitions (i.e., the number of sub-collections per partition). Legion uses a distributed bounding volume hierarchy to perform this check in logarithmic time with respect to partition size.

Critically, there is no single node in either case (with or without DCR) that needs to process the entire set of tasks launched by the user. Because distribution occurs in a communication-less, constant time manner—or with a tree broadcast—the index launch is distributed efficiently. Even though projection functors can be dynamic and request arbitrary sub-collections, the final physical analysis is distributed and uses optimized, logarithmic-time data structures to compute dependencies. Thus index launches delay the expansion of the task representation, and facilitate efficient distributed analysis of dependencies.

6 EVALUATION

We evaluate the impact of index launches by conducting a set of strong and weak scaling runs of three codes on up to 1024 nodes of the Piz Daint supercomputer, a heterogeneous Cray XC50 machine with one Intel Xeon E5-2690 v3 (12 physical cores) and one NVIDIA Tesla P100 per node, and a Cray Aries interconnect [1]. We used the system default installation of the Cray programming environment version 2.7.3. Legion was compiled with GCC 8.3.0 and CUDA 11.0.2, and was configured to use GASNet 2021.3.0 as a network portability backend. Regent programs were compiled with LLVM 3.8.1 as the code generation backend.

In our experiments, we consider three programs: a circuit simulation [6], a 2-D stencil from the PRK suite [30], and a multi-physics solver code that includes fluid, particle and discrete ordinates radiation modules [28]. We present both strong and weak scaling runs, and measure the impact of index launches by comparing the results with and without this optimization enabled, as well as with and without Legion's distributed task scheduling mode enabled [6]. Of the three codes, Soleil-X, which is also the most complicated, takes advantage of non-trivial projection functors, and we also analyze the impact of the cost of these checks on scaling and efficiency. In all of the experiments below, each data point is the average of 5 runs, in order to reduce noise due to interference from other jobs on the machine.

We also provide a more fine-grained analysis of the cost of the hybrid analysis presented in Section 4. Using a set of benchmark codes with a variety of trivial and non-trivial projection functors, we time the cost of the dynamic check component of our hybrid analysis. This setup allows us to extrapolate beyond the scale of today's supercomputers to estimate the impact on anticipated future machines.

6.1 Test Programs

We present strong and weak scaling results from three programs: Circuit, Stencil, and Soleil-X.

Circuit is a simulation of an electrical circuit on an unstructured graph from [6], where it was previously optimized. The code uses only trivial projection functors, and therefore is verified entirely by Regent's static checker and does not incur any runtime cost to verify projection functors. Circuit takes advantage of Regent's built-in CUDA code generation to automatically generate kernels appropriate for use on Piz Daint's GPUs.

Stencil is a 2-D stencil benchmark adapted from the PRK suite [30] and previously optimized in [6]. As with Circuit, the code uses trivial projection functors and takes advantage of Regent's CUDA code generation.

In both Circuit and Stencil we used configurations identical to the experiments in [6]. Circuit used 5.1×10^6 wires for strong scaling and 2×10^5 wires per node for weak scaling. Stencil used 9×10^8 grid cells for strong scaling and 9×10^8 grid cells per node for weak scaling. As in [6], task granularities for all applications were chosen to generate a task per GPU per computational stage of the application.

Soleil-X is the most substantial of the codes tested in our evaluation. Developed as part of the NNSA PSAAP II center at Stanford University, it is a multi-physics code with modules for turbulent fluid flow, particles, and radiation (via the discrete ordinates method) [28]. All three modules take advantage of Regent's CUDA code generation to target GPUs. Of the three modules, the DOM radiation model uses non-trivial projection functors, and was not previously optimized to use index launches. Our hybrid approach allows the entire Soleil-X code to make use of index launches. To evaluate the impact of these checks, we also include results that have all optimizations enabled and additionally have the dynamic checks disabled.

6.2 Strong and Weak Scaling Results

We present strong and weak scaling results for each of the three applications below.

6.2.1 Circuit. Strong and weak scaling results for Circuit can be seen in Figures 4 and 5, respectively. In each of the figures, use of Legion's distributed task scheduling mode [6] is marked with DCR, and use of index launches is marked by IDX. Thus we present four configurations with the cartesian product of each option enabled or disabled, respectively.

We can see that the best performance is achieved with both optimizations enabled. When strong scaling, this configuration achieves a 1.6× speedup over the next best configuration, with DCR enabled but index launches disabled. Similarly, when weak scaling, the configuration with both optimizations enabled is able to run at a scale 4× larger (1024 vs 256 nodes) with better parallel efficiency (85% vs 84%). Though the analysis with DCR is distributed, the inherent cost of issuing $O(\mathcal{P})$ tasks, and reduced efficiency due to the lack of whole-partition analysis in the runtime, reduces performance when index launches are disabled.

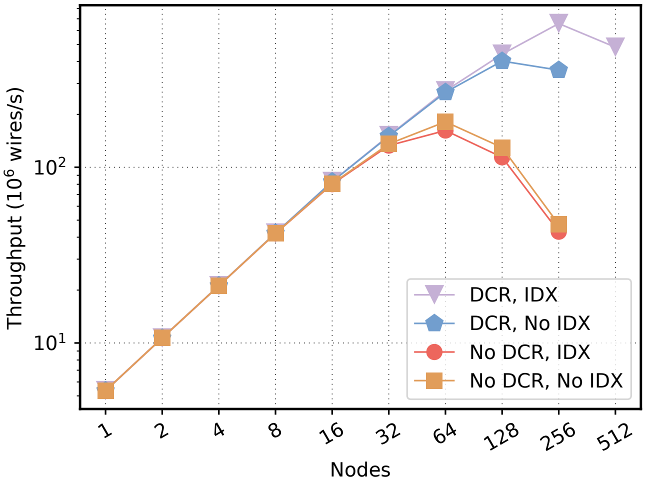


Figure 4: Circuit strong scaling.

When DCR is disabled, we see a slight reversal of this trend; index launches result in a slight decrease in performance due to interference with an unrelated tracing optimization in the Legion runtime. Legion relies on tracing to amortize expensive runtime analysis of the task graph [20]. In Legion's current design, when DCR is disabled, the tracing occurs prior to distribution (i.e., in the second column of Figure 3). Because tracing works fundamentally at the level of individual tasks (because its job is to reconstruct the task graph without repeating the analysis of task dataflow), this forces the expansion of the task graph prior to distribution, and effectively undoes the asymptotic benefit of index launches.

To demonstrate that this effect is a result of tracing, we perform an additional experiment, shown in Figure 6. This experiment is identical to Figure 5 except that tracing is disabled, and the tasks are overdecomposed (such that there are 10× as many tasks for the same problem size). Without tracing, index launches provide a benefit by deferring the expansion of the task graph to after distribution, as shown in Figure 3. This effect is magnified by the overdecomposition, due to the additional savings when moving tasks in bulk. Thus, in this configuration, index launches show a benefit relative to the same configuration without index launches, whether or not DCR is used.

As future work, we plan to investigate a deeper integration with Legion's tracing feature to enable tracing to work with bulk task launches, such that the benefits of index launches can be enjoyed, even without DCR. Note that even with this support, our experiments in the highest-performance configuration (with DCR) demonstrate that index launches are a critical optimization for efficient execution at extreme scales.

6.2.2 Stencil. Figures 7 and 8 show the strong and weak scaling results for Stencil, respectively. Overall we observe similar, but less dramatic, results as compared to Circuit. Strong scaling with all optimizations achieves a 1.2× speedup over the next best configuration, which is again DCR without index launches. The weak scaling configuration shows divergence between DCR with and without index launches beginning at around 512 nodes, which grows with node count.

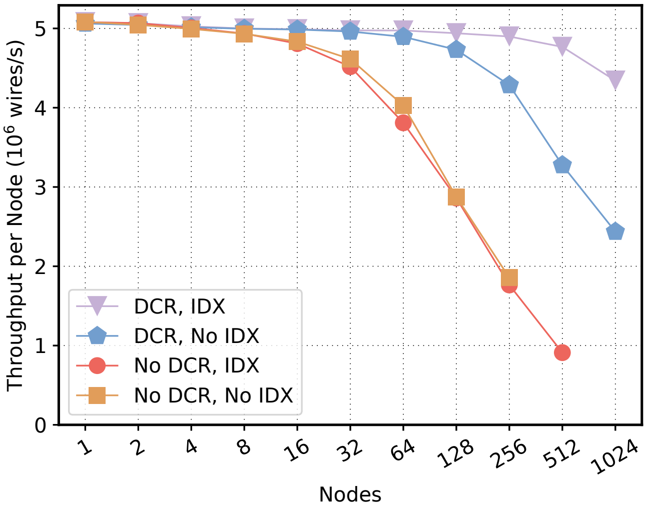


Figure 5: Circuit weak scaling.

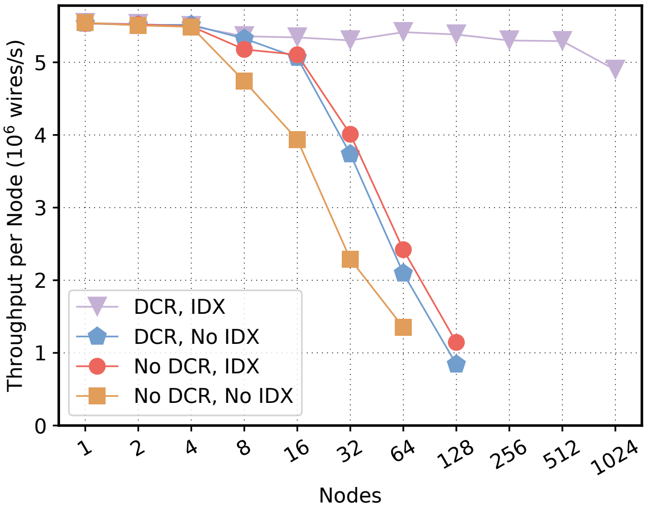


Figure 6: Circuit weak scaling, overdecomposed, no tracing.

6.2.3 Soleil-X. Figure 9 and 10 shows weak scaling results for Soleil-X in two configurations: with fluid flow only; and with fluid, particles and radiation (DOM). In both configurations, index launches improve parallel efficiency and enable the code to scale to higher node counts. In the pure fluid case, we observe 78% parallel efficiency at 512 nodes, while in the full simulation case, we observe 64% efficiency at 32 nodes. Note that the second case, which includes the DOM radiation model, is expected to scale more poorly because of the inherent properties of the DOM algorithm, which involves sweeps rather than forall-style parallelism.

The DOM code in Soleil-X makes use of non-trivial projection functors. The algorithm consists of sweeps over a grid, one from each corner. Thus the launch domains of each index launch vary, and consist of 3-D diagonal slices. The projection functors in the launch project these diagonal slices into a 2-D plane used for the exchange data. This projection is safe only when the launch domain contains no duplicate (x, y), (y, z) or (x, z) pairs. While it could be

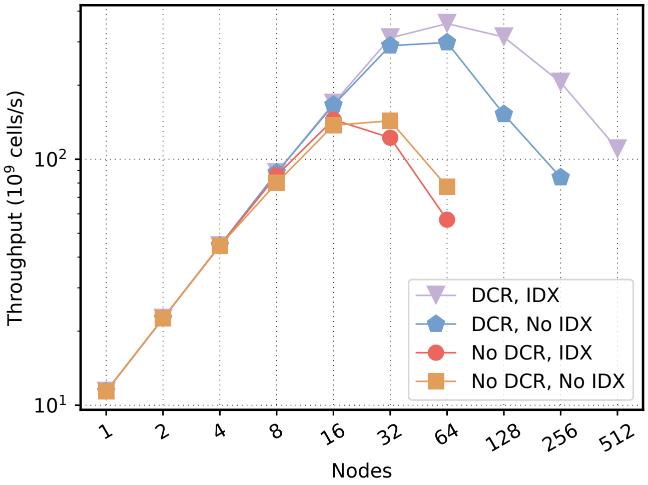


Figure 7: Stencil strong scaling.

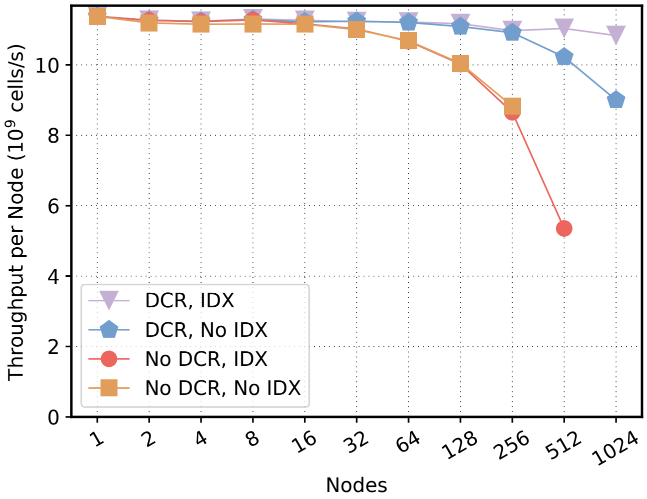


Figure 8: Stencil weak scaling.

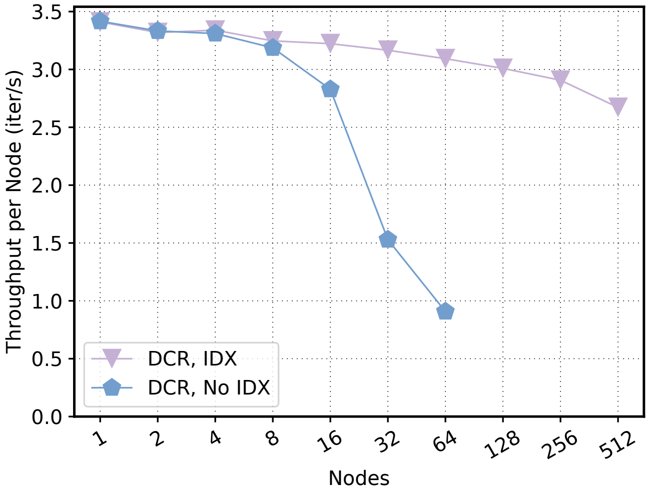


Figure 9: Soleil-X (fluid-only) weak scaling.

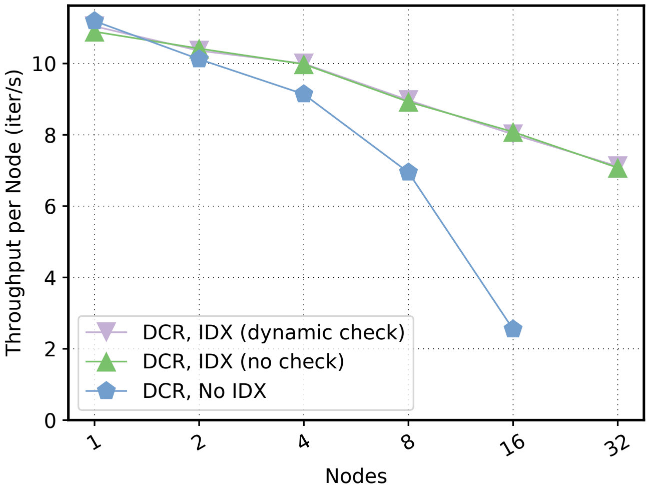


Figure 10: Soleil-X (fluid, particle and DOM) weak scaling.

challenging for a static compiler to verify that no duplicate pairs exist, a dynamic check can verify this trivially. Figure 10 compares the peformance with these checks included or elided. We see that the cost of these checks is negligible at this scale.

6.3 Dynamic Projection Functor Checks

Tables 2 and 3 show timing results for the dynamic analysis for various common practical cases. Each data point is the average of 5 runs and shows the elapsed time in microseconds. Column headings represent the size of the launch domain, which was equal to the number of sub-collections in Table 2 and half the number of sub-collections in Table 3. Note that the launch domain is independent of problem size since the checks are performed at the coarse granularity of partitions, rather than individual objects within collections. A single partition was used for both tables—in Table 2 to measure the cost of self-checks, and in Table 3 to measure the cost of cross-checks (i.e., verifying that the images of two projection functors are disjoint sets). The algorithms for both are described in Section 4. Since the analysis exits as soon as it finds a duplicate value, we were careful to choose valid projection functors and launch domains so that this early exit does not occur. In other words, all index launches corresponding to these tests would have been safe.

It is a common idiom in scientific applications in Regent to set the partition size equal to the number of nodes the application will run on. Therefore, both tables are a good proxy for understanding how the analysis will behave for large node counts beyond the size of current supercomputers. We can see that even for extreme values of the size of the launch domain (and partition size), the dynamic analysis never takes longer than 3 ms, which is approximately the same as the overhead of launching a task in Regent/Legion at these scales. We found that the measurements scale linearly with the size of the launch domain in both tables (reading each row in both tables from left to right), and linearly in the number of arguments in Table 3 (reading each column from top to bottom), confirming that our implementation indeed runs in linear time.

In any case, the checks can be executed in parallel with the runtime analysis and tasks themselves, so the exact cost of a check

Table 2: Elapsed times (in µs) for the dynamic self-checks for various safe projection functors. Column headings are the size of the launch domain.

Projection functor		10^3	10^4	10^5	10^6
Identity	i	1	10	141	1314
Linear	$ai + b$	1	15	148	1396
Modular	$(i + k) \bmod N$	3	20	128	1258
Quadratic	$ai^2 + bi + c$	2	19	238	2389

Table 3: Elapsed times (in µs) for the dynamic cross-check for multiple arguments on the same partition, showing linear scaling with the number of arguments. Column headings are the size of the launch domain.

Number of arguments	10^3	10^4	10^5	10^6
2	1	11	153	1480
3	1	16	186	1831
4	2	18	221	2118
5	2	23	256	2527

is unimportant as long as it is less on average than the application's task granularity.

7 RELATED WORK

It is well-known that a broadcast tree can be constructed to distribute N copies of an identical message in $O(\log N)$ time. This idea predates computing, but has found use in a number of parallel algorithms such as MPI collectives `MPI_Bcast` and `MPI_Scatter`, among others [27]. Index launches leverages this same general intuition, but applies it within the context of representations of parallel task groups.

Programming models for distributed computing frequently provide abstractions to describe sets of parallel tasks. Among the explicitly parallel programming models, for example, the ranks in a SPMD program (such as MPI) can themselves be seen as a form of task group, that simply start up at the beginning and run for the duration of a job. In Charm++ [17], chare arrays serve a similar purpose, spawning a number of chares (actors) in parallel and optionally specifying a distribution of chares around the machine. Chare arrays are also used in Charm++ to perform collective operations such as reductions.

Chapel has a `coforall` loop construct which can be used to launch parallel work [10]. Though the loops are written out as if sequentially launched, they can be optimized by a compiler to perform an $O(\log |D|)$ broadcast tree. However, the closure which is broadcast across the machine need not include all of the data closed over in the loop body, as the Chapel language supports remote references and the remote tasks can fetch data as needed at a later time (but with all the usual pitfalls of explicitly parallel programming).

A key difference between explicitly and implicitly parallel programming models is that the former are not *strict*, and tasks (or ranks, chares, etc.) can typically communicate (via messages, remote memory access, etc.) after being created. As a result, there is often no explicit notion of privilege or of data being passed to a task as an argument at its initial creation. Index launches, being a feature specifically intended for implicit parallelism, focus on the identification of the work to be performed including all necessary data.

Coarse-grained, implicitly parallel programming models employ a variety of mechanisms to improve efficiency and scalability. For example, OpenMP [12] and OmpSs [13] provide a `taskloop` construct that can be used to launch a group of parallel tasks, one per iteration of an annotated loop. However, as of OpenMP version 5.0, the `taskloop` construct does not support dependencies and thus does not provide the expressive power of index launches. In any case, scalability is less of a concern for OpenMP and OmpSs as they are both used on single-node systems. Index launches, and particularly projection functors, are designed specifically to promote an efficient distributed implementation.

Nimbus uses a centralized controller to orchestrate task execution, but supports execution templates which allow sequences of tasks be replayed with a single command [21]. This approach can improve efficiency despite the lack of a compact representation of tasks, but still relies on a centralized controller which may become a bottleneck at sufficient scale.

The Sequoia programming language uses compiler optimizations to determine an optimal placement of tasks around the machine [19]. Because the parallelism and placement of tasks is computed statically, a compact runtime representation of the work is not required. However, this approach is limited by the compiler's ability to completely determine the dataflow of the program.

Some task-based models provide higher-level abstractions that appear to work at bulk level. For example, Dask [24] provides a NumPy like abstraction that works on distributed arrays. Similarly, Spark [32] offers functional parallel operators for map, reduce, and other common routines. However, these operations are implemented in terms of a series of individual task launches, and do not use a compact task representation.

A number of implicitly parallel systems provide built-in capabilities for task fusion. Similar to index launches, task fusion helps reduce the overhead observed per task. However, task fusion generally provides a constant factor, not an asymptotic, improvement, and the degree of fusion is limited by the number of parallel processors which need to be filled. Index launches are compact by design and have a in-memory representation that is independent of the number of processors.

The identification of parallel work and data dependencies can also occur at a fine-grained level. Polyhedral methods are commonly used to identify parallelizable loops and transformations that can be applied to optimize them. These methods have been been applied to generate distributed codes from sequential programs [8]. As with coarse-grained compile-time methods, however, the key limitation of these approaches is that the language of affine loop nests is quite restrictive and prevents many useful programs from being written. Index launches provide a much more dynamic and flexible capability.

At the opposite end of the spectrum, inspector-executor methods have been used to generate distributed codes from iterative, but otherwise unstructured, or mixed structured/unstructured, sequential programs [22, 23]. These methods provide flexibility which is comparable to index launches, by analyzing loops at runtime and observing their actual data access patterns to discover the available parallelism. However, due to being fine-grained, and the lack of a compact in-memory representation, these approaches can easily run out of memory, limiting their practical applicability at large scales. Index launches strike a useful balance by optimizing for the compactness of the representation, allowing efficient use at the scales of modern supercomputers.

8 CONCLUSION

We have presented index launches, a scalable and flexible technique to denote a set of coarse-grained tasks for parallel execution. We compared index launches to other techniques in popular task-based parallel programming systems by considering how these techniques operate on the task-graph to achieve scalable distributed execution. We embedded index launches in a general programming model with compiler and runtime support, closely based on the actual implementations in the Legion runtime and the Regent compiler. We showed how we are able to support arbitrary dependencies in index launches by using a combination of static and dynamic program analyses. Finally, we demonstrated how index launches aid in the distributed execution of 3 scientific applications in Regent, enabling scalable execution on up to 1024 nodes of the Piz Daint supercomputer.

ACKNOWLEDGMENTS

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

REFERENCES

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

[2] 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, Dan 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. http://tensorflow.org/.

[3] 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.

[4] Ashwin M Aji, Lokendra S Panwar, Feng Ji, Karthik Murthy, Milind Chabbi, Pavan Balaji, Keith R Bisset, James Dinan, Wu-chun Feng, John Mellor-Crummey, Xiaosong Ma, and Rajeev Thakur. 2016. MPI-ACC: Accelerator-Aware MPI for Scientific Applications. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2016), 1401–1414. https://doi.org/10.1109/TPDS.2015.2446479

[5] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience* 23 (Feb. 2011), 187–198. Issue 2.

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

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

[8] Uday Bondhugula. 2013. Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures. In *Supercomputing (SC)*. ACM, 33.

[9] 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.

[10] Bradford L. Chamberlain. 2015. Chapel. In *Programming Models for Parallel Computing*, Pavan Balaji (Ed.). MIT Press, 129–159.

[11] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Partitioned Global Address Space Programming Model*. ACM, 2.

[12] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.* (1998).

[13] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: A Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters* 21, 02 (2011), 173–193.

[14] Richard Gerber, James Hack, Katherine Riley, Katie Antypas, Richard Coffey, Eli Dart, Tjerk Straatsma, Jack Wells, Deborah Bard, Sudip Dosanjh, Inder Monga, Michael E. Papka, and Lauren Rotman. 2018. Crosscut Report: Exascale Requirements Reviews, March 9–10, 2017 – Tysons Corner, Virginia. An Office of Science review sponsored by: Advanced Scientific Computing Research, Basic Energy Sciences, Biological and Environmental Research, Fusion Energy Sciences, High Energy Physics, Nuclear Physics. (1 2018). https://doi.org/10.2172/1417653

[15] D. S. Henty. 2000. Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling. In *Supercomputing (SC)*. https://doi.org/10.1109/SC.2000.10005

[16] 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

[17] Laxmikant V. Kalé and Sanjeev Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA*. 91–108.

[18] G. Karypis and V. Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* (1998).

[19] Timothy J. Knight, Ji Young Park, Manman Ren, Mike Houston, Mattan Erez, Kayvon Fatahalian, Alex Aiken, William J. Dally, and Pat Hanrahan. 2007. Compilation for Explicitly Managed Memory Hierarchies. In *Principles and Practice of Parallel Programming (PPoPP)*. 226–236.

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

[21] 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)*.

[22] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. Distributed memory code generation for mixed Irregular/Regular computations *(PPoPP)*. ACM, 65–75.

[23] Mahesh Ravishankar, John Eisenlohr, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2012. Code generation for parallel execution of a class of irregular loops on distributed memory systems. In *Supercomputing (SC)*.

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

[25] Kirk Schloegel, George Karypis, and Vipin Kumar. 2002. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience* 14, 3 (2002), 219–240.

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

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

[28] Hilario Torres and Gianluca Iaccarino. 2018. Soleil-X: Turbulence, Particles, and Radiation in the Regent Programming Language. *Bulletin of the American Physical Society* 63 (2018).

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

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

[31] Yuan Yu, Martín 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 the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 18, 15 pages. https://doi.org/10.1145/3190508.3190551

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

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

We performed strong and weak scaling experiments for three scientific applications on up to 1024 nodes of the Piz Daint supercomputer. We also performed single-node timing experiments for the dynamic analysis, as described in the paper.

We used the system default installation of the Cray programming environment version 2.7.3. Legion was compiled with GCC 8.3.0 and CUDA 11.0.2 and was configured to use GASNet 2021.3.0. LLVM 3.8.1 was used as the code generation backend for Regent.

We have saved the exact version of Legion and Regent used in a GitLab branch, along with all the scripts used to build and run.

Author-Created or Modified Artifacts:

```
Persistent ID: 10.5281/zenodo.5164404
Artifact name: Index launches software release
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: Piz Daint: Cray XC50 machine with 1 Intel Xeon E5-2690 v3 (12 physical cores) and 1 NVIDIA Tesla P100 per node

Operating systems and versions: SUSE Linux Enterprise Server 15 SP1

Compilers and versions: GCC 8.3.0

Libraries and versions: LLVM 3.8.1, CUDA 11.0.2, GASNet 2021.3.0