Scaling Implicit Parallelism with Index Launches

Undergraduate Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Rupanshu Soi
ID No. 2018A7TS0294H

Under the supervision of

Dr. Elliott Slaughter
&
Prof. R Gururaj

SLAC NATIONAL ACCELERATOR LABORATORY
&
BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

January 2022

Certificate

This is to certify that the report titled "*Scaling Implicit Parallelism with Index Launches*" submitted by Rupanshu Soi, ID No. 2018A7TS0294H, in partial fulfillment of the requirements of BITS F421T Thesis embodies work done by him under my supervision.

Supervisor	*Co-Supervisor*
Dr. Elliott SLAUGHTER	Prof. R GURURAJ
Staff Scientist,	Associate Professor,
SLAC National Accelerator Laboratory,	BITS Pilani - Hyderabad Campus,
USA	India
Date:	Date:

Abstract

Task-based programming systems are now widely used to program modern supercomputers. Since these systems need to achieve efficient scalable execution while being suitable for a wide variety of application domains, their core abstraction, the task graph, needs to have a scalable but expressive representation.

We present *index launches*, a technique, embedded in a general task-based framework, to collapse a task graph to obtain a more efficient representation. Leveraging a hybrid program analysis, we show how a compiler can support index launches by providing safety guarantees in virtually every case of practical importance.

In addition to the support for forall-style parallelism, we extend index launches in two primary directions. First, we introduce *cross products*, a novel abstraction for nested data partitioning, and describe how to incorporate them in index launches. Second, to enable index launches to represent limited forms of pipeline parallelism, we present *ordered launches*, and develop the program analysis required for an efficient implementation.

Our implementation of index launches is in Regent, a high-productivity programming language for distributed computing. Using the Piz Daint supercomputer, we evaluate the performance of several scientific applications written in Regent, and show that index launches lead to improved performance at the scales of contemporary and upcoming high-performance machines.

Acknowledgements

I must begin by thanking Elliott Slaughter, my advisor, without whom this work was impossible. Elliott's resilience in listening to my bad ideas and proposing clever ones in turn is extraordinary.

I am grateful to Prof. Gururaj and my department at BPHC for their support, and for letting me pursue what I am passionate about.

My thesis builds upon a foundation laid by several extremely talented people over the course of a decade. Michael Bauer and Sean Treichler implemented (and continue to maintain) the Legion and Realm runtime systems. Elliott Slaughter is the primary author of the Regent compiler, which constitutes a large chunk of my own work. Wonchan Lee and Manolis Papadakis helped with several applications that we use for performance measurements. And, from the beginning, the driving force behind this research has been Alex Aiken's vision for the future of HPC. I am responsible for all other aspects of this thesis, parts of which previously appeared in [25].

In the end, everyone is a product of their environment, more so than they'd like to publicly admit. My parents, Roopak and Anjali, have been a pillar of support every single day of my life. Everything I have ever achieved, or will achieve, I owe completely to them.

"If you can't solve a problem, then there is an easier problem you can solve: find it."

— George Pólya, *How to Solve It*

Contents

List of Figures

Chapter 1

Introduction

Modern supercomputers are large, often heterogeneous machines with deep memory hierarchies. At the time of writing, the fastest supercomputer in the world is RIKEN Center for Computational Science's Fugaku [26]. It has $158,976$ nodes (one node is made up of a single CPU) spread over 432 racks and connected by 900 km of fiber optic cable. Each CPU has multiple levels of cache and is connected to a multi-layered external file system.

Fugaku is not heterogeneous (it has only one kind of processor), unlike the three main upcoming supercomputers by the US Department of Energy: El Capitan [16], Frontier [12] and Aurora [3]. As a testament to the vast design space of high-performance computational platforms today, it is interesting to note that all four supercomputers have completely different architectures from each other. Their choice of CPUs, hardware accelerators and network backends are all different, and consequently, come with their own strengths and weaknesses.

How do we effectively program these machines? Traditionally, MPI+X programming models have been used for this task. In MPI+X, MPI [24] is used for inter-node data transfers, communication, and synchronization, and X is used for intra-node parallelism. A common choice for X is OpenMP [10] due to its shared-memory programming model, or CUDA [9], if the application wants to target GPUs.

However, these programming models have proven to be insufficient for effectively programming modern supercomputers, for several reasons. A central issue is programmer productivity. In MPI+X, users need to specify several low-level execution details of the program, including extracting parallelism from the application, moving data, overlapping computation and communication, and load balancing. At best, large programs written in these programming models are brittle and hard to maintain, with typical scenarios often exhibiting hard-to-fix concurrency bugs like race conditions. At worst, however, these programming models keep non-expert programmers from being able to effectively utilising modern machines.

Task-based programming systems offer a viable solution to these problems. They are popular in large scale data analytics [31, 21, 17], scientific applications [4, 23, 2, 13], as well as machine learning [1, 22]. In a task-based programming model, programs are written as a set of tasks; tasks are just functions whose dependencies can be reasoned about by the programming system. The system[1] then constructs a *task graph* to represent a program as its tasks and their dependencies. Since this task graph captures the data-flow of the program, the system can automatically analyze it to infer which tasks are safe for simultaneous execution, along with any required data movement and synchronization.

Task-based systems are often categorized as either *explicit* or *implicit*. In explicit task-based systems, the task graph must be explicitly created by the user by specifying a partial order on the execution of tasks. On the other hand, in implicit task-based systems, the programming system is clever enough to infer the task graph based on data dependency information in the program. In this sense, implicit systems exist at a higher level of abstraction than explicit systems; they increase programmer productivity by transferring the responsibility of specifying an order on the execution of tasks from the user to the system. But this increase in productivity comes at the cost of additional static or dynamic analysis, which is either paid for in programming model flexibility (if static), or in runtime overhead (if dynamic).

In implicit task-based programming systems, since the system is entirely responsible for the creation, management and eventual execution of the task graph, it is crucial for the task graph to have an efficient yet expressive representation. Commonly, implicit systems try to collapse their task graph (without losing dependency information) to obtain a more efficient representation. An *index launch* is one such technique for collapsing a task graph.

In an index launch, a set of parallel tasks is collapsed down to a single node in the task graph—they provide a constant size representation of a set of $\mathcal{P}$ parallel tasks. As a result, the system can reason about the entire set of tasks as a single, bulk unit of work, resulting in performance improvements in the system's dynamic analysis.

Index launches exist within a general and expressive data model, which allows each task to easily specify the data it needs for execution. Leveraging this data model, we show how a compiler can

[1]We use the terms programming system and system interchangeably.

prove the correctness of an index launch by performing an efficient and sophisticated hybrid program analysis. This hybrid design allows us to combine the efficiency of static analysis with the generality of dynamic analysis, to produce a system which is both sound and complete, and dramatically improves performance at scale, with negligible overhead in the dynamic cases.

Furthermore, we extend index launches in two primary directions. First, we describe the program analysis for supporting *cross products*, a novel data abstraction for nested partitioning, in index launches. Second, by developing *ordered launches*, we relax the constraint that index launches may only represent sets of forall-style parallel tasks.

Our implementation is in Regent [23], an implicitly parallel programming language for the Legion [4] programming model. Legion is implemented as a runtime system and can be used either via Regent or directly via a C++ API. Although index launches can be implemented completely dynamically (as is done in pure Legion applications), a separation of concerns between the compiler and runtime is one of the foundation stones of our design, and enables an efficient correctness analysis, along with an easy to use syntax that resembles sequential for-loops.

Index launches are such an important optimization for scalability that they are used in every substantial Regent application. As an example, a Regent implementation of the discrete ordinates method used in Soleil-X [27], developed at Stanford University, relies on our dynamic analysis to implement a periodic launch domain. In addition, our cross product work is slated to be used by S3D-Legion [29], a major solver developed by Sandia National Laboratories.

Chapter 2

Task Graphs and their Representations

Index launches are, in essence, a lossless compression algorithm for task graphs. A task graph is a directed acyclic graph (DAG) of tasks and their dependencies. Tasks are represented as nodes in the graph, and a directed edge is added between two tasks to denote a data dependence: the edge (u, v) represents that task v must execute after task u.

In drawing our task graph representation, we arrange parallel tasks along the horizontal dimension, such that all tasks at the same vertical level are free to execute in parallel, and time along the vertical dimension, with execution beginning at the bottom-most row of tasks.

As an example, consider the following piece of JavaScript, intended to be representative of a typical map-reduce operation.

```
var C = [1, 2, 3, 4, 5];
C = C.map(function(i) { return 2 * i; })
C.reduce(function(total, i) { return total + i; })
```

The task graph for this entire operation might look like Figure 2.1. Due to our representation, this graph makes two facts apparent: first, that each call to to the mapping function can execute in parallel, and second, that `bar` must execute after each `foo` has finished, because it has a data dependence by way of `C`.

Let's look at a more practical example. Iterative stencil loops (ISLs) are a class of scientific applications in which array elements are updated according to a fixed pattern, generally based

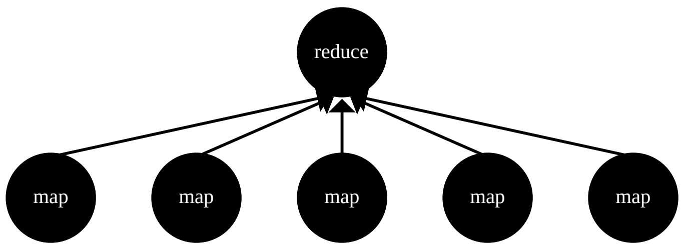


FIGURE 2.1: Task graph for a map-reduce operation.

on a 1-D, 2-D or 3-D grid, called a stencil. Here is an example of an ISL where the stencil pattern is $(i-1, i, i+1)$, this time in Regent syntax.

```
for t = 0, T do
  for i = 0, P do
    -- read from C, write to D
    compute(C[i-1], C[i], C[i+1], D[i])
  end
  swap(C, D)
end
```

The outer `t` loop iterates over timesteps, and the inner loop launches the task `compute` for each point in the domain. `compute` reads from `C` and writes its output to `D`. At the end of each timestep, we swap `C` and `D` to implement double buffering.

The task graph for this stencil computation is shown in Figure 4.2, with the name of the task omitted. Each row contains the calls to `compute` performed in a single timestep ordered left-to-right by increasing value of `i`. The bottom-most row corresponds to timestep `t = 0`.

Due to the stencil pattern, this task graph is highly symmetric in its data dependencies. This property is not unique to ISLs—it is common for practical scientific applications to have highly regular and symmetric task graphs.[1]

Because large scientific applications today can have huge task graphs (S3D-Legion [29] can execute up to 10^6 tasks per second), task-based programming systems attempt to leverage this

[1]Not all scientific applications yield task graphs that are so symmetric, and the symmetry is not a necessary condition for index launches. In fact, one of the strengths of our technique is that we allow a great deal of complexity behind a fairly uniform interface. Granted, index launches are always composed of a set of parallel tasks. But the implementation of these tasks, as well as the dependencies between tasks in different index launches can be largely data dependent, allowing the user to implement a myriad of dynamic behaviours.

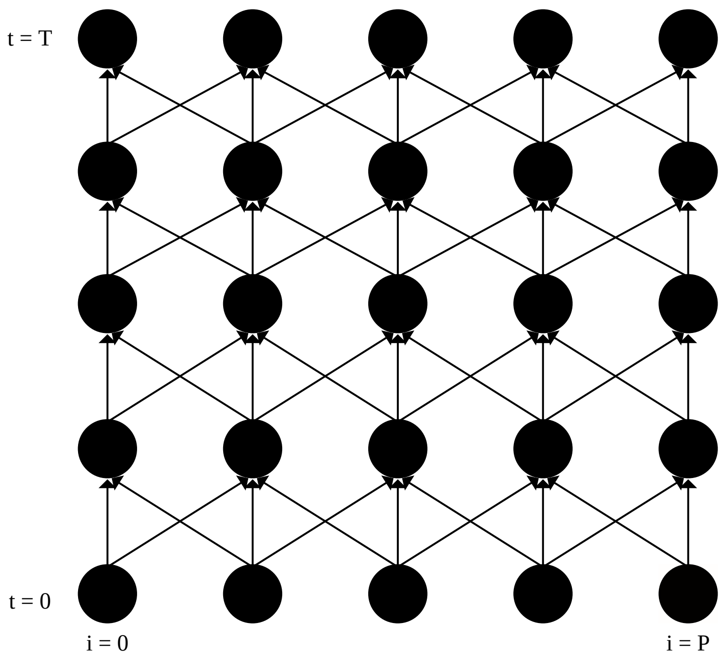


FIGURE 2.2: Task graph for an ISL code.

symmetry to collapse their task graph to obtain a more efficient representation. Beginning with a straightforward task graph representation, we will look at three alternative representations that exploit this symmetry. One of these representations relies on index launches.

2.1 Naive Representation

We call the task graph representation in which both time and parallelism dimensions are kept completely materialised the naive representation, visualized in Figure 4.2. This representation has complexity $\mathcal{O}(\mathcal{PT})$, and is used by Dask [21], Spark [31], StarPU [2], and PaRSEC DTD [13], among others.

2.2 Collapsing both Parallelism and Time

PaRSEC PTG [7] creates a static, algebraic description of the task graph. Shown in Figure 2.3, it collapses an $\mathcal{O}(\mathcal{PT})$ task graph down to an $\mathcal{O}(1)$ representation. For instance, it can express our stencil dependency pattern as: task (x, y) depends on tasks $(x-1, y-1)$, $(x, y-1)$, and $(x+1, y-1)$.[2] Due to the algebraic nature of this representation, the programming system does not need to lookup any data structures at runtime to determine a task's dependencies, making this representation the most concise.

But there is a trade-off. Because this representation is maximally dense, it must sacrifice expressiveness. For example, it cannot express forms of dynamic task generation where a task's dependencies are not completely known at compile time.

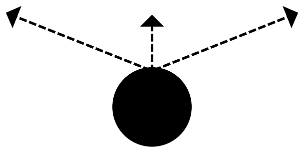


FIGURE 2.3: Task graph collapsed along both dimensions.

2.3 Collapsing Time

Another technique of collapsing a task graph is to collapse its time dimension. TensorFlow [1] is one popular system that has adopted this approach in the domain of large-scale machine learning.

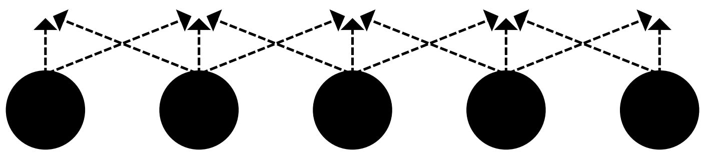


FIGURE 2.4: Task graph collapsed along the time dimension.

In TensorFlow, nodes of the task graph don't only represent tasks. Instead, TensorFlow represents the control-flow of a program *in-graph*, by using pre-defined nodes [30]. For example, TensorFlow

[2]For clarity, we have omitted the necessary bounds checking.

has a task graph node to represent a branch in the application, or a set of nodes to represent a while-loop. TensorFlow attempts to compile the entire source program down to these nodes.

While this approach has been very successful for machine learning workloads, it is not without its limitations. First, to be able to collapse the time dimension, TensorFlow needs to see the entire task graph at once. This is impossible if the program is submitted incrementally to TensorFlow. Second, this technique relies on the compiler's ability to analyse the program's control-flow. If the compiler does not understand a program's control-flow, then it will be unable to compile it down to these pre-defined nodes.

2.4 Collapsing Parallelism

Finally, we come to our solution. Index launches collapse the task graph along the horizontal, parallelism dimension—an entire row of tasks is captured as a single index launch. An index launch can be seen as an $\mathcal{O}(1)$ representation of $\mathcal{P}$ tasks. Consequently, using index launches reduces the complexity of the task graph from $\mathcal{O}(\mathcal{PT})$ to $\mathcal{O}(\mathcal{T})$.

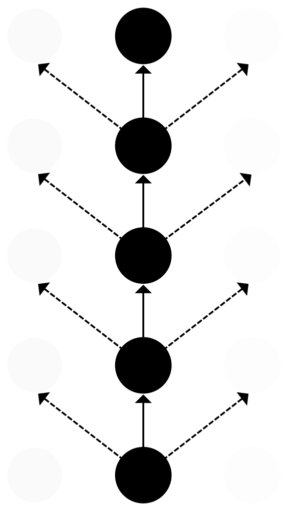


FIGURE 2.5: Task graph collapsed along the parallelism dimension.

A task-based programming system that is not completely static must expose an API. This API will be used by either a human programmer or a higher-level system (e.g. a compiler) that translates from a higher-level description of the task graph down to this API. Whatever the case may be[3], the final program must make API calls to *register* each task for execution with the programming system's runtime. Then, from the runtime's perspective, an index launch transforms a loop that makes $\mathcal{P}$ API calls (one to register each task) into a single API call for an index launch. As a result, using an index launch allows the programming system to reason about the entire group of tasks as a single, bulk unit of work, as opposed to reasoning about the $\mathcal{P}$ tasks individually. This reduction in the runtime's work is the primary source of performance improvement afforded by using index launches.

However, for this transformation to be possible, the task graph must satisfy some constraints—it is impossible to collapse an arbitrary, asymmetric group of tasks into a single unit. The primary constraint on index launches is that they are a representation for a group of tasks to be executed in parallel.[4] In other words, an index launch has no implied ordering of its tasks—the system is free to execute the tasks in any order, as well as concurrently. Therefore, an index launch is *safe* only when no two tasks in it have a data dependence between them.

In our design, users are not responsible for annotating loops to be converted into index launches. Instead, we provide a fully automatic compiler analysis which is sound and complete, and therefore guaranteed not to miss any optimization opportunities in the program. Users may optionally provide annotations on loops they believe to be eligible, but this is purely as a measure of defensive programming, to ensure that the code is not unexpectedly deoptimized if the compiler deems an index launch unsafe. In any case, the compiler will not permit any unsafe launches to occur.

2.5 Other Techniques

Collapsing the task graph is not only the way means to achieve efficient execution in a task-based system. In the absence of such a collapsing mechanism, systems primarily rely on one of two techniques:

1. *Dynamic unrolling.* Instead of storing the entire task graph in memory, a system may choose to store only the frontier of computation, along with a constant number of tasks on either side of this frontier. This reduces the memory footprint of the task graph, but does

[3]As described in Chapter 1, although not necessary for index launches, our design greatly benefits from separating the language (Regent) from the runtime system (Legion) it targets.

[4]This constraint can be relaxed in certain cases to permit pipeline parallelism. See Chapter 5.

not reduce the execution complexity. Legion [4], StarPU [2] and PaRSEC [13] employ this strategy, among several others.

2. *Pruning.* Some systems, notably StarPU and PaRSEC, are designed to be able to recover the correct task graph in a distributed manner without pulling all of it onto any single node. In particular, they allow the user to manually prune the task graph such that only the tasks executing on a node, plus the tasks that *directly* depend on those tasks, are visible to that node. However, because this technique requires the user to escape out of the implicitly parallel abstraction, we do not consider it as a representation of implicit task graphs in our work.

Chapter 3

Index Launches

In this chapter, we present our programming and data models and use them to formalize index launches. Since our actual implementation is only in Regent and Legion, in an effort to make our technique applicable to other languages and systems, we leave several details implementation defined, focusing instead on only those that are essential to develop our idea of index launches.

3.1 Programming Model

Our programming model is a general framework for describing a task-based programming system. A program is structured as a series of tasks. The dependence analysis, scheduling, and load balancing of the system happens at the granularity of tasks. The code within a task can largely be executed sequentially, with the exception of any nested subtasks launched by the task, which in most cases execute in parallel to the parent (except where blocking is necessary to avoid data races).

3.2 Data Model

Data in our model is organized via *collections*. A collection is any data structure that contains objects, and are often indexed in multiple dimensions. For example, a 2-D velocity vector field can be implemented as a 2-D collection where each element stores the x and y components of the velocity vector at that point.

To exploit parallelism while optimizing locality, collections can be *partitioned*. A partition is a division of a collection into *sub-collections*, which are just subsets of data of the parent collection intended to be used together in a task. Like collections, partitions can be indexed to access their sub-collections.

Furthermore, a partition in which all sub-collections are disjoint sets of data is called a *disjoint* partition. Otherwise, the partition is *aliased*. Note that our usage of the word "partition" is unlike its more common mathematical sense: mathematical partitions are disjoint by definition.[1]

Partitioning a collection is one of the most common ways to extract parallelism from an application, because tasks that compute on disjoint sub-collections can execute in parallel, even if they write to the data. For instance, to implement matrix-matrix multiplication, we might partition the output matrix and launch a task for each sub-collection. In this case, the partitioning will enable the programming system to recognise that each task can execute in parallel.

There are several techniques available for partitioning [15, 28]. However, since the exact method is unimportant for developing index launches, we leave this detail unspecified.

The only property of a partition that is necessary for our data model is its disjointness. All other properties can be dynamic, but whether a partition is disjoint or aliased must be known at compile time.

Finally, our data model has the concept of *privileges*. Every task must declare privileges on each collection it receives as input. A privilege is one or more of `read`, `write`, or `reduce` (the last with a specific commutative operator).

The programming system utilizes all three aspects of our data model to determine if two tasks are safe to execute in parallel. For instance,

- tasks accessing mutually disjoint collections only cannot have a data dependence, or
- tasks with `read` privileges on the same collection cannot have a dependence due to that collection, or

[1]For mathematically inclined readers, a precise definition of collections and partitions follows. A collection is a function $C : I \to F$, where $I \subset \mathbb{Z}^n$ and F is a finite set. A partition P of C is a function $P : I' \to 2^C$, where $I' \subset \mathbb{Z}^m$, 2^C is the power set of C and $\bigcup_{i \in I'} P(i) \subset C$. P is an aliased partition if $\exists c \in C$ and $\exists i_1, i_2 \in I'$ such that $c \in P(i_1)$, $c \in P(i_2)$ and $i_1 \neq i_2$. Otherwise, P is a disjoint partition.

- tasks that declare `write` privileges on two sub-collections of an aliased partition might have a WAW (write after write) dependence (depending on the exact aliasing, which is not known at compile time), so the compiler must conservatively assume that there may be a data dependence.

As an example of the syntax, here is a task that receives two collection arguments and declares `read` privileges on the first argument and `write` on the second.

```
task distribute_currents(C, D) where reads(C) writes(D)
do
  -- computation
end
```

3.3 Formal Description

An index launch with n partition arguments is written as,

$$\text{forall}\,\big(D, T; (P_1, f_1), \ldots, (P_n, f_n)\big)$$

where,

- D is the launch domain: the index launch will have one invocation of T corresponding to each point in D,
- T is the task: the index launch will be a set of invocations of T (each invocation corresponds to a single point in D), and T must declare privileges on n collection arguments,
- P_i is the i^{th} partition argument: the i^{th} argument of T will be a sub-collection of P_i, and
- f_i is the i^{th} projection functor: f_i is an arbitrary pure function from D to the set of sub-collections of P_i. It will be used to index into P_i to obtain a sub-collection to pass to T.

A common source of confusion is between the n partition arguments of the index launch, and the n collection arguments of the task T. Tasks always receive collections[2] as arguments.[3] A

[2]Sub-collections are collections.
[3]We ignore non-collection arguments for now because they can simply be passed to the task by value.

partition argument is a property of the index launch, not of T or an invocation of T. The i^{th} partition argument will be indexed by the i^{th} projection functor to obtain a sub-collection, and this sub-collection will be passed to a specific invocation of T.

As an example, here is the stencil loop from before.

```
for t = 0, T do
  for i = 0, P do
  compute(C[i-1], C[i], C[i+1], D[i])
  end
  swap(C, D)
end
```

Written as a formal index launch, the `P` loop is

$$\text{forall}\left([0,\texttt{P}), \texttt{compute}; (\texttt{C},\texttt{i-1}), (\texttt{C},\texttt{i}), (\texttt{C},\texttt{i+1}), (\texttt{D},\texttt{i})\right)$$

Note that the projection functors are functions of the loop iterator variable and are used to index into the corresponding partition arguments.

3.4 Conditions for Safety

All program optimizations, no matter how effective, must not change the output of the program (for some definition of "output"). Therefore, we must now tackle this important question for index launches. When it is safe to turn a simple loop that launches a group of tasks into an index launch?

Since an index launch is just a representation for a set of tasks meant to be executed in parallel, the only condition we have is that all the tasks in an index launch must be safe to execute in parallel, or, in other words, there can be no data dependence between two tasks in the same index launch.[4]

In several cases of practical interest, this safety can be guaranteed by the compiler by analysing the nature of the partitions involved (whether they are disjoint or aliased), combined with analysing the privileges declared by the task on its collection arguments. Our execution and

[4]We will relax this constraint in Chapter 5.

data model are designed to allow a compiler to easily compute these properties: both the nature of a partition and the privileges requested by a task are explicitly stated by the user.

As a result, the compiler can determine safety in many common situations. For example, an index launch in which

- T declares `read` privileges on all its arguments is safe, or
- T declares `write` privileges on a collection argument derived from an aliased partition is unsafe.

However, by far the most common (and interesting) case is when we have a disjoint partition with write privileges declared on it. In this case, to guarantee that no two tasks will exhibit a data dependence, we must guarantee that no two tasks will receive the same sub-collection as input. Since the projection functor controls how sub-collections are assigned to calls to the task, the safety of the index launch in this case depends solely on the projection functor. As an example, consider the following two *potential* index launches:

```
for i in D do
  flow(D[0])
end
```

```
for j in D do
  flow(D[j])
end
```

where `flow` requests write privileges on its sole collection argument, and `D` is a disjoint partition.

The only difference between these task launch loops is the projection functor: the `i` loop (on the left) has a constant projection functor, and the `j` loop (on the right) has the identity projection functor. Recall that an index launch is safe when no two tasks in it have a data dependence on each other. Keeping this in mind, it is easy to see that the `i` loop is unsafe to index launch, because each task in it[5] receives the sub-collection `D[0]` as input, which means all invocations of `flow` will attempt to write to the same piece of data simultaneously, resulting in a race condition. On the other hand, the identity projection functor in the `j` loop guarantees that each call to `flow` will receive a distinct sub-collection of `D`, and because `D` is disjoint, this loop is safe to index launch.

The general condition for safety here, in the case of a disjoint partition and write privileges, is that the projection functor must not assign two different invocations of T the same sub-collection. In other words, the projection functor must be an injective function over D, the launch domain.

[5]The exact number of tasks will be equal to $|D|$, the size of the launch domain.

```
input : Launch domain $D$, partition $P$, and corresponding projection functor $f$
bitmask ← malloc($|P|$)
for $i \in |P|$ do
    bitmask [$i$] = False
end
conflict ← False
value ← 0
for $i \in |D|$ do
    value ← linearize($f(i)$)
    if $0 \leq$ value $< |P|$ then
        conflict = bitmask [value]
        bitmask [value] = True
        if conflict then
            break
        end
    end
end
if conflict then
    // original task launch loop
else
    // index launch
end
```

Algorithm 1: Dynamic analysis for projection functor self-check.

3.5 Analyzing Projection Functors

How do we guarantee this property for projection functors in the compiler? The compiler can perform static analysis of the projection functor to determine its injectivity. However, a static analysis is inherently limited in scope, and even the most powerful static analysis will not be complete.[6] Therefore, our design uses a hybrid approach. First, the compiler attempts a straightforward static analysis of the projection functor. This analysis can recognize trivial projection functors (constant and identity), which occur in many common cases. If this analysis is inconclusive, the compiler falls back to a dynamic analysis that is sound and complete for verifying the injectivity of the projection functor over the launch domain.

Dynamically analysing the injectivity of a function over a finite domain is straightforward. We evaluate the function at each point and check for duplicates using a bitmask. Our dynamic analysis algorithm for *self-checks*, i.e. checking a partition against itself, is given in Algorithm 1.[7]

[6]A more sophisticated static analysis is certainly possible, based perhaps on the polyhedral model [6], but is unnecessary. In practice, our dynamic analysis is efficient enough, even at the most extreme scales, to cover any cases not handled by our static analysis.

[7]The algorithm shown represents the compiler's output, not the actual implementation. Because we are implementing a program analysis, we manually create an AST inside the compiler that, when passed through the code-generation phase, results in Algorithm 1.

The `linearize` method on line 8 is used to handle partitions that are indexed in $N > 1$ dimensions. Such partitions are indexed by projection functors that are N-D tuples. Since our bitmask is a linear array, `value` must be a scalar, which means we have to map the output of the projection functor to a scalar before we can assign it to `value`. The `linearize` method is used to bijectively map an N-D tuple to a scalar.

Our algorithm uses $\mathcal{O}(|D| + |P|)$ time and $\mathcal{O}(|P|)$ space. Since $|P|$ refers to the number of sub-collections in P, it follows that the complexity is independent of the total number of elements in the parent collection that P partitions. Furthermore, since the number of sub-collections is typically bounded by the size of the machine the cost of our analysis is as well. In other words, our dynamic analysis is independent of problem size—it is only as costly as the number of nodes the application runs on.

3.5.1 Cross-checks

Our analysis for partitions is sound but incomplete. In particular, it does not handle index launches in which two or more partitions may interfere with each other. This *cross interference* can occur if an index launch uses two different partitions of the same underlying collection, or the same partition multiple times (with a different projection functor each time). The former cannot be handled efficiently at compile time because it depends on the exact partitioning. If it occurs, the compiler will conservatively assume that the corresponding index launch will be unsafe. Nevertheless, the latter is also a common pattern, and we describe a sound and complete dynamic analysis to handle it.

As an example, consider the following index launch where P supplies two write arguments to T,

$$\text{forall}\big(D, T; (P, f), (P, g)\big)$$

Again, to guarantee safety, we must ensure that no two invocations of T access the same piece of data. This condition is satisfied iff,

1. f is injective on D,
2. g is injective on D, and
3. $\nexists i, j \in D$ such that $f(i) = g(j)$.

Unlike our analysis for a single partition, where read privileges guaranteed safety, a partition that supplies a read-only sub-collection can still interfere with other partitions. Consider the same index launch as above, but now T declares write privileges on its first argument and read privileges on its second. Safety in this case can be guaranteed by conditions (1) and (3) alone; we no longer need g to be injective on D because the sub-collection it supplies is never modified by T.

In the general case, we want to *cross-check* N partitions, P_1 through P_N, each with read, write or read-write privileges.[8] Instead of checking every pair individually, we group all potentially interfering partitions. We allocate (and initialize) a single bitmask for each group, and test whether the projection functor associated with each partition in this group is injective using the same bitmask. In other words, we repeat lines 7-16 from Algorithm 1 for the projection functor associated with each partition in a group. However, we make two important modifications to that algorithm:

- we don't set the bitmask (line 11 in Algorithm 1) for read-only partitions, because we only require read-only partitions to not interfere with a write partition, and
- we check all write (or read-write) partitions before any read-only partition. Otherwise, we might miss a potential conflict between a read-only and write partition, because the bitmask has no "memory" of projection functors of read-only partitions (due to the previous bullet point).

By preprocessing partitions and only slightly modifying Algorithm 1, we have generalized it to handle N partitions in linear, as opposed to quadratic, time. The interested reader can verify that this newer algorithm exactly checks the mathematical conditions for correctness described earlier.

3.6 Runtime Implementation

While the compiler is primarily concerned with proving the safety of index launches, a runtime system is responsible for actually executing them. After all, the entire utility of index launches stems from reducing the work done by such a runtime system.

[8]Reduce and write privileges are treated identically.

As we noted in Chapter 1, our implementation is within the Legion runtime [4]. Legion has an internal pipeline to process tasks, involving four main stages: task issuance, logical analysis, distribution, and physical analysis. Recall that index launches collapse a task graph for a more efficient representation. As a result, they allow Legion to maintain an efficient, collapsed representation for several of these stages.[9] This collapsed representation greatly reduces the work done by Legion, because it allows Legion to reason about the group of tasks in the index launch as a single unit of work, as opposed to reasoning about each task individually.

For a more detailed discussion of the interplay between index launches and the Legion runtime, please see [25].

[9]The task graph representation at the final stage must be independent of index launches to ensure correctness.

Chapter 4

Performance Evaluation

We evaluate our design and implementation of index launches on three scientific applications in Regent. Our tests consist of strong and weak scaling runs on up to 1024 nodes of the Piz Daint supercomputer [18]; its details are presented in Table 4.1. 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.

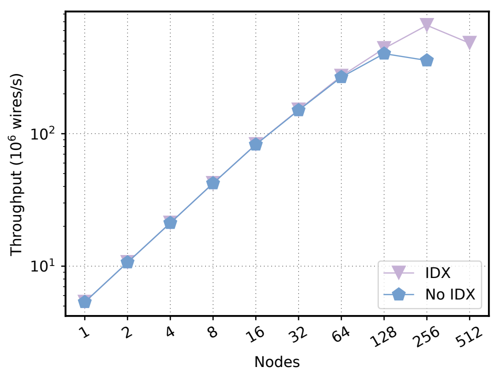
A more complete discussion of the performance impact of index launches can be found in [25].

4.1 Circuit

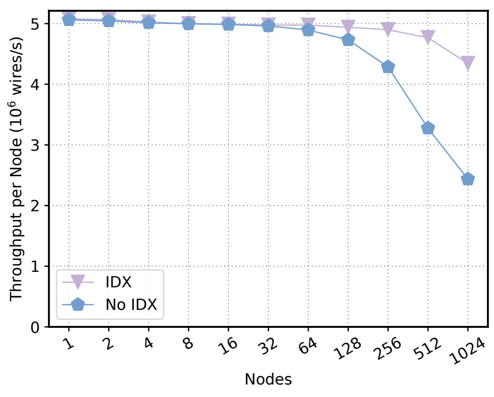
Circuit is a simulation of an electrical circuit on an unstructured grid, taken from [5]. Figure 4.1 presents strong and weak scaling results for Circuit. In both cases, we see that best performance is achieved with index launches enabled. Strong scaling shows a 1.8× speed-up with index launches, and weak scaling shows a similar 1.7× performance improvement.

Model	Cray XC50
CPU	Intel Xeon E5-2690 v3
GPU	NVIDIA Tesla P100
Interconnect	Cray Aries
Nodes	1024

TABLE 4.1: Piz Daint specifications.

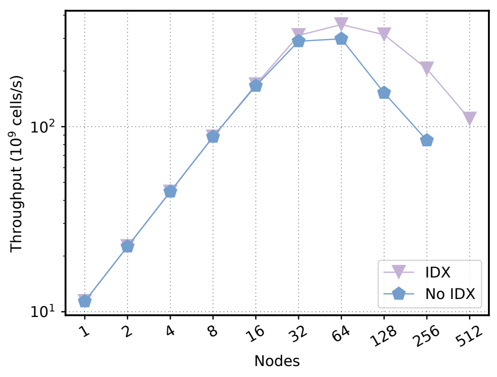


(A) Strong scaling

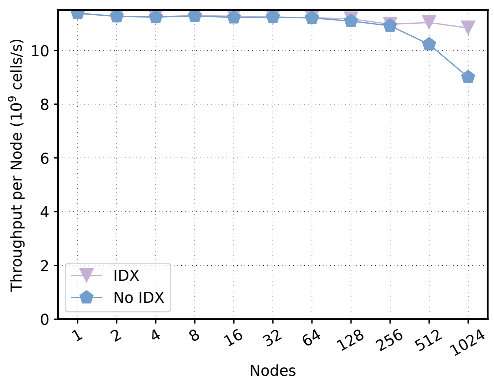


(B) Weak scaling

FIGURE 4.1: Circuit performance measurements.



(A) Strong scaling



(B) Weak scaling

FIGURE 4.2: Stencil performance measurements.

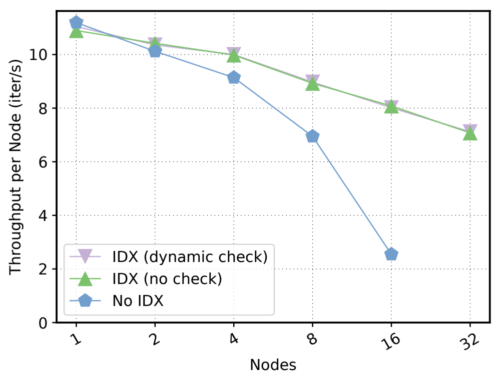
4.2 Stencil

Stencil is a 2-D stencil benchmark (an ISL from Chapter 2) previously implemented for [5]. Figure 4.2 present strong and weak scaling results. Again, we observe that index launches enable the best performance at scale. In particular, strong scaling shows a 1.2× speed-up at 32 nodes,[1] and weak scaling shows a similar speed-up at 1024 nodes.

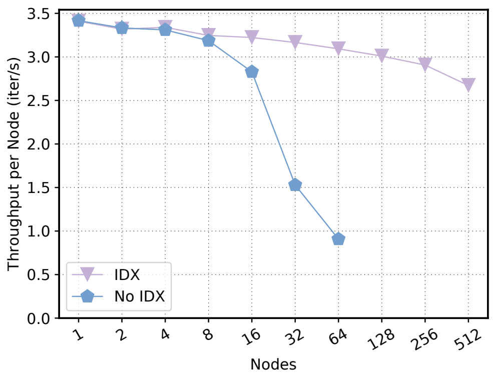
4.3 Soleil-X

Soleil-X [27] is the most substantial application that we tested. It is a fluid, particle and radiation solver developed as part of PSAAP-II at Stanford University. The radiation module in Soleil-X

[1]We are only concerned with the maximum speed-up in strong scaling, regardless of the number of nodes it occurs on.



(A) Weak scaling with fluid, particle and radiation



(B) Weak scaling, fluid only

FIGURE 4.3: Soleil-X performance measurements.

uses our dynamic analysis due to its sophisticated use of projection functors. Before our dynamic analysis was implemented, this module was unable to utilize index launches.

Like Circuit and Stencil, we observe significant performance improvements with index launches enabled. Figure 4.3 (A) shows a 3.4× speed-up on using index launches, and (B) shows a 3.1× speed-up. (A) also compares performance with the dynamic analysis included or elided.[2] We see that its cost is negligible at these scales, further bolstering our design choice of a hybrid program analysis.

4.4 Dynamic Analysis

As further proof of the negligible cost of our dynamic analysis, Tables 4.2 and 4.3 present performance results for several common configurations to evaluate the runtime overhead of our dynamic analysis for projection functors. Each data point is the average of five runs to remove any inaccuracies due to other programs on the machine. In both tables, reading each

TABLE 4.2: Elapsed times (in µs) for the dynamic analysis for several 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

[2]The dynamic analysis is only a safety check. After the program has been verified once, it can simply be elided for performance runs.

Table 4.3: 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

row left-to-right, we observe that the cost of the analysis scales linearly with the size of the launch domain, as expected from our complexity analysis. In Table 4.3, reading each column top-to-bottom, we see that the cost of the dynamic analysis increases linearly with the number of arguments.

Chapter 5

Cross Products & Ordered Launches

In this chapter, we extend index launches in two primary directions. First, we introduce cross products, a new data abstraction built over partitions to enable nested partitioning, and discuss how we can support cross products as arguments in an index launch. We describe the required program analysis, its implementation and limitations. Second, we present ordered launches, a generalization of index launches in which tasks with limited dependencies can be represented, enabling programs with pipeline parallelism to be expressed in our technique. We present practical use cases that can benefit from ordered launches, and explore the program analysis required for an implementation.

5.1 Cross Products

Cross products are an addition to our data model from Chapter 3. If an application computes over a grid partitioned in two or more different ways, a cross product allows the programmer to easily access data included in both partitions. Cross products can also be used as a form of nested data partitioning.

A cross product is defined by the cartesian product of two or more partitions of a single parent region, resulting in a tree of nested partitions defined by the intersections of the corresponding sub-collections. A general cross product of N partitions $P_1, \ldots, P_N$ is written,

$$X = \text{cross\_product}(P_1, \ldots, P_N)$$

Like partitions, cross products can be indexed in multiple dimensions. However, unlike partitions, a cross product defined by N partitions can be indexed at least $N+1$ times. Continuing our example, a general indexing of a cross product $X[i_1]\dots[i_k]$, for $k < N$, gives a cross product,

$$X' = \text{cross\_product}(Q, P_{k+1}, \dots, P_N)$$

where Q is a partition that contains the intersection of the corresponding sub-collections of all previous partitions,

$$Q[i] = P_1[i_1] \cap \dots \cap P_k[i_k] \quad \forall i \in |Q|$$

For $k = N$, the indexing returns a sub-collection that is formed by taking the intersection of the corresponding sub-collections of all N partitions,

$$X[i_1]\dots[i_N] = P_1[i_1] \cap \dots \cap P_N[i_N]$$

Lastly, for an indexing of a cross product to be valid, we require that each index i_k be valid for the corresponding partition P_k.

As a concrete example, consider a collection C such that $C[i] = i$ and $|C| = 10$. To define a cross product, we need two or more partitions of C. Consider equal[1] partitions P and Q of C, where $|P| = 2$, and $|Q| = 5$. Let's define a cross product,

$$X = \text{cross\_product}(P, Q)$$

Now, for instance,

$$X[0][0] = P[0] \cap Q[0] = [0, \dots, 4] \cap [0, 1] = [0, 1]$$

$$X[1][0] = P[1] \cap Q[0] = [5, \dots, 9] \cap [0, 1] = \phi \text{ (the empty set)}$$

$$X[1][4] = P[1] \cap Q[4] = [5, \dots, 9] \cap [8, 9] = [8, 9]$$

5.1.1 Cross Products in Index Launches

Let us begin by extending our formal definition from Chapter 3. In the context of index launches, the main difference between a partition and a cross product is that a cross product can have multiple projection functors associated with it. Consequently, we now denote a general index

[1]A partition with all sub-collections of the same size is called an equal partition.

launch as,

$$\text{forall}\left(D, T; (P_1, f_1), \ldots, (P_a, f_a), (X_1, f_{1,1}, \ldots, f_{1,m}), \ldots, (X_b, f_{b,1}, \ldots, f_{b,n})\right)$$

where the task T receives a total of $a + b$ collection arguments: a derived from partitions, and b derived from cross products. The relative ordering of the partition and cross product arguments is only a convenience for our formal notation; in the actual implementation partitions and cross products can be listed in any order. Moreover, by looking at the number of projection functors, we can tell that cross product X_1 is defined by m partitions (and similarly for X_b). For brevity, we say that the *dimension* of X_1 is m.

5.1.2 Conditions for Safety

Armed with this (more) general definition, we can now consider the question of correctness. Since we are still analyzing index launches, the same rule applies: all tasks must be safe to execute in parallel. If the task requests write privileges on its data, then, like before, we need to ensure that no two invocations of the task have access to the same underlying piece of data. What analysis, static or otherwise, do we need to do to guarantee that such an index launch is safe?

To begin, let's look at an index launch with a single 2-D cross product argument.

$$\text{forall}\left(D, T; (X, f_1, f_2)\right)$$

Recall from our analysis of partitions that an index launch with write privileges on an aliased partition is always unsafe, regardless of its projection functor. We can make an analogous claim for cross products if we notice the following general property: for any cross product X,

$$X[i_1] \ldots [i_k] \subset X^{(1)}[i_1]$$

where $X^{(1)}$ denotes the first partition in the definition of X. In other words, any indexing of X is always a subset of the corresponding sub-collection of $X^{(1)}$. If $X^{(1)}$ was an aliased partition, then without having access to the exact aliasing pattern (which in general is only available at runtime), there is no way we can guarantee that two sub-collections derived from X are disjoint. Consequently, our first rule for ensuring safety in our example is that $X^{(1)}$ be a disjoint partition.

This condition on $X^{(1)}$ is not sufficient to ensure correctness, so now we must analyze the projection functors. Recall that X is indexed by both its projection functors to obtain a sub-collection: $X[f_1(i)][f_2(i)]$. If f_1 was injective over D, then $X[f_1(i)]$ would return a distinct sub-collection for each $i \in D$, which is enough to guarantee that the final sub-collections obtained from X would be disjoint, regardless of f_2. Therefore, the second condition to ensure safety[2] in our example is that f_1 be an injective function over the finite launch domain D.

Interestingly, this analysis for X is identical to what we would have done if the index launch had $X^{(1)}$ as its sole argument. This observation makes the implementation much easier: we retain our code for partitions, and just treat a cross product like its first partition. Moreover, it implies that the complexity analysis for dynamically checking a cross product is identical to that for checking a partition, and does not depend on the dimension of the cross product.

5.1.3 Loop Invariant Prefixes

Our analysis for cross products is sound but not complete. One case where it fails is when a prefix of the indexing expression is loop invariant. For example, our analysis will give a false negative for $X[0][i]$, because the projection functor associated with its first partition is not injective.

To handle these cases, we need to check whether the indexing expression has a prefix that is loop invariant. If it does, we do our usual cross product analysis on the index (and corresponding partition) following the prefix.

Concretely, given a general indexing $X[i_1]\dots[i_N]$, we look for a value of k such that $1 \leq k \leq N$ and that the prefix $i_1, \dots, i_k$ is loop invariant. If such a value of k exists, we do our analysis for cross products on i_{k+1}, i.e., the index following the prefix, and the partition associated with it.

To aid the actual implementation, before coming to the analysis, we do a separate loop invariant code motion pass over these expressions, where we hoist any loop invariant prefixes out of the loop. As a result, we no longer need to change our analysis code to handle this case, because after hoisting the prefix out of the loop, the cross product is again treated as its first partition.

[2]We use the terms safety and correctness interchangeably.

5.2 Ordered Launches

The primary constraint over index launches is that they can only represent a set of parallel tasks. However, there are several classes of scientific applications, wavefront computations being a notable example, that can benefit from the ability to launch large numbers of dependent tasks. Ordered launches target this use case. They are a generalization of index launches that can represent tasks with limited forms of dependencies.

If two tasks have a dependency between them, one of them must execute completely before the other. Given two such tasks, ordered launches need a way to tell which one to execute first. The *ordering functor* provides this disambiguation mechanism by assigning an integer to each task. Formally, the ordering functor is a function $g : D \to \mathbb{Z}$ with the following semantics: for two points $i, j \in D$, if $g(i) < g(j)$ then the the task corresponding to i may execute before the task corresponding to j, but, more importantly, if $g(i) = g(j)$, the two tasks *must* be free to execute simultaneously.

The ordering functor is what makes ordered launches more general. It is helpful to consider two extreme cases to visualize its effect on an ordered launch: the ordering functor $g(i) = i$ implies that the ordered launch may be executed completely sequentially, whereas $g(i) = k$ (for some constant k) turns the ordered launch back into a normal (forall-style parallel) index launch.[3]

Syntactically, an ordered launch extends the usual syntax for writing an index launch with an ordering functor,

```
__demand(__ordered_launch)
for i in D by g do
  foo(p[f(i)])
end
```

The formal definition can be extended similarly,

$$\text{forall}\,\big(D, T, g; (P, f), \dots\big)$$

[3]The ordering functor provides only an upper bound on the dependencies between tasks in an ordered launch. Tasks with unequal values of the ordering functor will still execute in parallel if they don't have an actual dependence to enforce the implied ordering. For instance, the programming system will always try to execute an ordered launch with read-only tasks completely in parallel, even if the user supplies $g(i) = i$ as the ordering functor.

As with index launches, a compiler's ability to guarantee safety for a general ordered launch is quite limited. Building on the hybrid approach from Chapter 3, we outline a static analysis that can provide this safety guarantee for some cases, and supplement it with a sound and complete dynamic analysis. However, due to the added generality of ordered launches, both analyses are significantly more complicated than the ones in Chapter 3; notably, the dynamic analysis no longer takes linear time.

5.2.1 Use Cases

The benefit of ordered launches is not only in being able to describe a task group with dependencies. By virtue of the ordering functor, they also partially order a multi-dimensional iteration space.

Consequently, a GEMM operation can now be described in one ordered loop rather than three unordered loops in Regent. The `ispace(...)` notation is Regent syntax for declaring a 3-D launch domain from $(0, 0, 0)$ to (m, n, k). The reduction expression should occur within a task, but we show it inside the loop for brevity.

```
__demand(__ordered_launch)
for i in ispace(int3d, {m, n, k})
by i.z do
  C[i.x, i.y] += A[i.x, i.z] * B[i.z, i.y] -- inside a task
end
```

The discrete ordinates method [27] is another nice example. We have a 3-D launch domain and we want to iterate on planes where $x + y$ is constant.

```
__demand(__ordered_launch)
for i in ispace(int3d, ...)
by i.x + i.y + i.z do
  radiate(p[{i.x, i.y}])
end
```

5.2.2 Conditions for Safety

Consider a simple ordered launch with domain D, partition P, projection functor f, and ordering functor g. When is such an ordered launch safe? Observe that the ordering functor creates a mathematical partition over the launch domain D. It creates a set of *equivalence classes* which are mutually exclusive, and whose union is D. Each equivalence class is a set,

$$\{d \mid d \in D \text{ such that } g(d) = k \text{ for some constant } k \in \mathbb{Z}\}$$

The significance of equivalence classes is that tasks in an equivalence class can be executed in parallel. Therefore, since we cannot have a dependence between two tasks in the same equivalence class, to guarantee the safety of an ordered launch, we simply apply our analysis for index launches individually within each equivalence class.

However, we have another requirement. To allow the runtime to cheaply identify dependencies in an ordered launch, it requires the user to supply an *inverse* of the projection functor f.[4] This function[5], which we denote f^{-1}, must map a sub-collection of P back to its preimages under f in D. Additionally, f^{-1} must return these points in sorted order according to g.

In applications that directly target the Legion runtime, the user is responsible for implementing f^{-1}. However, ideally, we would like the Regent compiler to automatically generate this inverse given f and g. If we are not concerned about runtime performance, the compiler can emit a straightforward dynamic implementation of f^{-1} by creating a hash table of values of f and sorting each bucket by g.

This dynamic approach has worst-case complexity $\mathcal{O}(|P| \log |P|)$, which is expensive enough to warrant investigation into a static analysis capable of handling at least some common and easy cases. A sensible constraint on f and g for performing static analysis is to limit them to affine expressions of the loop variable. To figure out exactly what analysis is required, let's look at a small concrete example.

Consider an ordered launch over a 3-D domain where the partition is 2-D. Let's say we have $f(i) = (2i_x + 3, i_z + 1)$. Note that even though i is a 3-D vector, the output of f is 2-D since it is used to index into the partition associated with f. Due to the affine nature of f, we can invert it by equating $(k_x, k_y) = (2i_x + 3, i_z + 1)$ and solving for i. We get $i_x = \frac{k_x - 3}{2}$ and $i_z = k_y - 1$, which gives,

$$f^{-1}(k) = D \cap \left\{ (\frac{k_x - 3}{2}, \lambda, k_y - 1) \mid \forall \lambda \in \mathbb{Z} \right\}$$

[4]The runtime could compute dependencies even without this inverse, but it would be quite expensive. Therefore, we chose this interface between the compiler and runtime, and the compiler is responsible for solving this piece of the problem.

[5]Technically, f^{-1} is a multifunction since f is not necessarily a bijection.

The output of f^{-1} is a set because the projection functor was projecting from 3-D down to 2-D. We need to intersect this set with D because universal quantification over λ results in extraneous points. In practice, the intersection can be reduced to a set of symbolic constraints over k. For instance, if $D = [(0,0,0), \ldots, (a,b,c)]$ then we can remove the intersection by additionally requiring that,

$$(0,0,0) \leq (\frac{k_x - 3}{2}, \lambda, k_y - 1) \leq (a,b,c)$$

or,

$$3 \leq k_x \leq 2a+3 \quad \text{and} \quad 0 \leq \lambda \leq b \quad \text{and} \quad 1 \leq k_y \leq c+1$$

Furthermore, we need output of the inverse to be ordered by the ordering functor. Continuing the same example, let's say our ordering functor was $g(i) = i_z$. Since the range of f^{-1} is compatible with the domain of g, we can plug one into the other,

$$g(f^{-1}(k)) = k_y - 1$$

to obtain the expression which is used to order the output of f^{-1}.

If we are able to write a symbolic loop, consisting of the loop variable i and the bounds of D, that enumerates the members of $f^{-1}(k)$ ordered by $g(f^{-1}(k))$, we are done. However, at the time of writing, it is not clear whether this can be done efficiently. In the near future, we plan to determine whether this approach is feasible in a compiler. Nevertheless, although it hurts performance, our dynamic implementation of f^{-1} is sound, complete, and places no additional constraints[6] on f and g.

[6]However, as in index launches, both f and g must still be pure functions.

Chapter 6

Related Work

Programming models targeting distributed platforms commonly provide abstractions to describe sets of parallel tasks. For instance, ranks in an MPI [24] application can be seen as a task group involving tasks that last the entire duration of the program. Chare arrays in the actor model of Charm++ [14] serve a similar purpose; they spawn a number of chares (actors) around the machine in parallel. Chapel [8] has a `coforall` loop construct that can be used to launch parallel work. However, Chapel relies on remote references instead of privileges like us, and therefore comes with all the usual pitfalls of an explicitly parallel programming model.

Task fusion is provided by several implicitly parallel systems, but it typically only provides a constant factor of improvement. On the other hand, as discussed in Chapter 2, index launches provide an asymptotic improvement in the complexity of the task graph representation: from $\mathcal{O}(\mathcal{P}\mathcal{T})$ to $\mathcal{O}(\mathcal{T})$.

Sequoia [11] uses compiler optimizations to determine an optimal placement of tasks around a distributed machine. Being a static approach, it is limited by its compiler's ability to analyze the dataflow of a program.

Some popular task-based models, including Dask [21] and Spark [31], provide coarse grained abstractions for parallelism. However, they don't employ a compact representation, and rely instead on a series of individual task launches.

Index launches identify parallel work at the granularity of tasks, but this identification can also be done at a fine-grained level. Polyhedral methods [6] are a class of compiler techniques that target individual loop nests. By statically analyzing the index expressions and loop bounds, they attempt to modify a loop's iteration space to expose parallelism. But because these methods operate at compile time, they are quite limited. Notably, polyhedral analysis restrict array indices and loop bounds to be affine expressions, leaving out several important scientific applications. For instance, the periodic launch domain from Soleil-X could not have been analyzed by a polyhedral solver, because it uses modulus, a non-affine operation.

Another fine-grained approach to distributed computing is taken by inspector-executor methods [19, 20]. They operate at runtime and observe the actual data access patterns of loops to discover opportunities for parallelism. Although they are more flexible than polyhedral methods, due to being fine-grained, they can easily go out of memory and therefore have limited applicability at extreme scales. Index launches, on the other hand, have a compact representation that is independent of the amount of work they represent, and can easily scale up to the size of today's and tomorrow's supercomputers.

Chapter 7

Conclusions

We have presented index launches, a flexible yet expressive technique for collapsing a task graph to obtain a more efficient representation. By embedding index launches in a general task-based programming and data model, we described how they can be implemented in any task-based programming system that has a notion of collections and partitions. We presented a hybrid program analysis, along with an actual implementation in the Regent compiler, that is sound and complete for determining the safety of converting a simple task launch loop into an index launch.

We conducted rigorous performance experiments on the Piz Daint supercomputer to evaluate the impact of index launches on three representative scientific applications. We quantified the performance overhead of our dynamic analysis, and showed that it was negligible at large scales.

Furthermore, we presented cross products, a novel data abstraction, and discussed the required program analysis for supporting them in index launches. We also presented an initial design for ordered launches, which are a generalization of index launches designed to represent and exploit pipeline parallelism. In the future, we plan to build on this initial work on ordered launches, eventually implementing them in Regent.

Appendix A

Where is the Code?

Legion and Regent are developed as open-source research projects hosted on the Legion GitHub repository, which, ipso facto, also contains the entire implementation of our work.

This research stemmed from a single issue opened several years ago, #413, requesting support for the modulus operator in projection functors to implement a periodic domain in Soleil-X.

Our implementation of the dynamic program analysis for partitions is split between two pull requests: #1054, which added self-checks, and #1097, which added cross-checks. Additionally, #1143 fixed a bug with cross-checks that was reported in #1142.

The entire analysis for cross products, including loop invariant code motion, is implemented in #1151. At the time of writing, this code is yet to be merged, due to failure in a unit test in the HTR solver. Furthermore, #1155 added the `__import_cross_product` keyword, which was necessary for quick adoption of our cross product work, and #14 updated the Regent documentation to include cross products.

Ordered launches were first mentioned in #641. However, our contributions exist only as a design proposal, and therefore have no associated implementation yet.

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* http://tensorflow.org/. 2015.

[2] Cédric Augonnet et al. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In: *Concurrency and Computation: Practice and Experience* 23 (2 Feb. 2011), pp. 187–198.

[3] *Aurora.* https://www.alcf.anl.gov/aurora. Accessed: 2021-12-21.

[4] Michael Bauer et al. “Legion: Expressing Locality and Independence with Logical Regions”. In: *Supercomputing (SC).* 2012.

[5] Michael Bauer et al. “Scaling Implicit Parallelism via Dynamic Control Replication”. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, 105–118. ISBN: 9781450382946. DOI: 10.1145/3437801.3441587. URL: https://doi.org/10.1145/3437801.3441587.

[6] Uday Bondhugula. “Compiling Affine Loop Nests for Distributed-Memory Parallel Architectures”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* SC ’13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503289. URL: https://doi.org/10.1145/2503210.2503289.

[7] George Bosilca et al. “PaRSEC: Exploiting Heterogeneity to Enhance Scalability”. In: *Computing in Science & Engineering* 15.6 (2013), pp. 36–45.

[8] D. Callahan, B. L. Chamberlain, and H. P. Zima. “The Cascade High Productivity Language”. In: *High-Level Parallel Programming Models and Supportive Environments.* 2004, pp. 52–60.

[9] *CUDA Programming Guide 5.5.* http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. 2013.

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

[11] Kayvon Fatahalian et al. “Sequoia: Programming the Memory Hierarchy”. In: *SC*. 2006.

[12] *Frontier*. https://www.olcf.ornl.gov/frontier/. Accessed: 2021-12-21.

[13] Reazul Hoque et al. “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*. ScalA ’17. Denver, Colorado: ACM, 2017, 6:1–6:8. ISBN: 978-1-4503-5125-6. DOI: 10.1145/3148226.3148233. URL: http://doi.acm.org/10.1145/3148226.3148233.

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

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

[16] *LLNL and HPE to partner with AMD on El Capitan, projected as world’s fastest supercomputer*. https://www.llnl.gov/news/llnl-and-hpe-partner-amd-el-capitan-projected-worlds-fastest-supercomputer. Accessed: 2021-12-21.

[17] Philipp Moritz et al. “Ray: A Distributed Framework for Emerging AI applications”. In: *Operating Systems Design and Implementation (OSDI)*. 2018, pp. 561–577.

[18] *Piz Daint — CSCS*. http://www.cscs.ch/computers/piz_daint. Accessed: 2021-12-21.

[19] Mahesh Ravishankar et al. “Code generation for parallel execution of a class of irregular loops on distributed memory systems”. In: *Supercomputing (SC)*. 2012.

[20] Mahesh Ravishankar et al. “Distributed memory code generation for mixed Irregular/Regular computations”. In: PPoPP. ACM. 2015, pp. 65–75.

[21] Matthew Rocklin. “Dask: Parallel Computation with Blocked Algorithms and Task Scheduling”. In: *Python in Science Conference (SciPy)*. 130-136. Citeseer. 2015.

[22] Alexander Sergeev and Mike Del Balso. “Horovod: fast and easy distributed deep learning in TensorFlow”. In: *arXiv preprint arXiv:1802.05799* (2018).

[23] Elliott Slaughter et al. “Regent: A High-Productivity Programming Language for HPC with Logical Regions”. In: *Supercomputing (SC)*. 2015.

[24] M. Snir et al. *MPI-The Complete Reference*. MIT Press, 1998.

[25] Rupanshu Soi et al. “Index Launches: Scalable, Flexible Representation of Parallel Task Groups”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476175. URL: https://doi.org/10.1145/3458817.3476175.

[26] *Specifications - Fugaku : Fujitsu Global*. https://www.fujitsu.com/global/about/innovation/fugaku/specifications/. Accessed: 2021-12-21.

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

[28] Sean Treichler et al. “Dependent Partitioning”. In: *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM. 2016, pp. 344–358.

[29] Sean Treichler et al. “S3D-Legion: An Exascale Software for Direct Numerical Simulation of Turbulent Combustion with Complex Multicomponent Chemistry”. In: *Exascale Scientific Applications*. Chapman and Hall/CRC, 2017, pp. 257–278.

[30] Yuan Yu et al. “Dynamic Control Flow in Large-Scale Machine Learning”. In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys ’18. Porto, Portugal: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190551. URL: https://doi.org/10.1145/3190508.3190551.

[31] Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”. In: *HotCloud* 10 (2010), pp. 10–10.