BUILDING COMPOSABLE DISTRIBUTED AND ACCELERATED SOFTWARE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Rohan Yadav
June 2026

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Alex Aiken, Primary Advisor

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Fredrik Kjoelstad, Co-Advisor

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

Approved for the Stanford University Committee on Graduate Studies.

Kenneth Goodson, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Abstract

Modern high-performance computing systems are organized as distributed collections of heterogeneous computing elements connected by hierarchical networks. Extracting the peak performance from these machines requires programmers to navigate a deep software stack that spans from programming an individual accelerator to orchestrating communication and synchronization across a cluster. Mastering this multi-level software stack is challenging for computing experts, and almost impossible for the domain scientists who need to use these high-performance computing systems to advance their research.

Abstraction of low-level details and composition of independent modules are the standard tools for controlling software complexity. While this approach is effective in the world of sequential programming, composable distributed and accelerated software is rarely found. Most independently written distributed software cannot be composed without myriad correctness issues. Once correctness has been established, composition is often at odds with achieving the best performance; significant performance is lost at module boundaries, either through the cross-module coordination itself or the missed optimizations across modules.

This thesis describes the Legate library ecosystem and runtime system. The Legate library ecosystem is a collection of high-level libraries that mimic the standard interfaces of libraries like NumPy and SciPy, and seamlessly compose while scaling to clusters of accelerators. The Legate runtime system is a multi-level runtime system that introduces several technologies to support the correct and efficient composition of independently written Legate libraries. The Legate runtime introduces analyses and program representations to support the efficient composition of both distributed

data and distributed computation. These analyses are performed dynamically, and thus can incur overheads that impede scalability and absolute performance; Legate also introduces techniques to control these overheads as they arise in the composition of independent modules.

Lastly, this thesis describes the experience of developing a Legate library using the abstractions exposed by the Legate runtime system. In particular, we describe the development of Legate Sparse, a distributed drop-in replacement for the SciPy Sparse library, enabling the automatic scaling and acceleration of idiomatic sparse applications developed in Python. We show how Legate's abstractions enable both abstraction and composability, and free the developer from the responsibility of reasoning about how the library is used within the context of a larger application.

Altogether, this thesis describes the architecture and analyses that enables the development of composable software that achieves high performance on distributed and accelerated computing systems.

Acknowledgments

While it is my name on this document, this journey through the PhD is the result of the support, encouragement, and intellectual contributions from many different people.

First and foremost, I want to thank my team of advisors, both official and unofficial. Alex Aiken and Fred Kjolstad were my PhD advisors at Stanford. Alex recruited me to Stanford, and has embodied the ideal of a researcher: his foresight, clarity of thought, and breadth of knowledge have guided me through the myriad challenges faced during my PhD. Alex's intuition about the right way to tackle problems and to know exactly what is missing from a talk or paper is razor-sharp, and I hope to approach his level someday. I tell everyone that Alex has never been wrong, and I struggle to think of such an occurrence. Fred and I started at similar times at Stanford, and I began working with Fred during my rotations. Fred has been my model of a successful junior faculty; he has been there for me when navigating the details of design and communication, and he knows exactly when to provide the emotional support I needed when dealing with paper rejections and interview stress. Fred and Alex have always had my back and have gone out of their way to support and promote me and my work. I began a long-standing collaboration with NVIDIA Research in 2022 and have worked with Mike Bauer and Michael Garland there for a majority of my PhD. Working with Mike changed the trajectory of my research, and I have learned so much about how to think about and build systems, how to approach solving problems, and how to communicate my ideas. Michael always helped me to see the bigger picture about my research, and forced me to be able to justify why the problems I wanted to work on really mattered. I thank Kunle Olukotun for serving

on my qualification examination and reading committees. Finally, I would like to thank Keith Winstein and Caroline Trippel for their last-minute flexibility to serve on my oral defense committee.

I would never have considered doing a PhD and becoming involved with research without the formative research experiences as an undergraduate at Carnegie Mellon University. After completing the parallel algorithms course in my sophomore year, I approached Umut Acar to start doing research. He welcomed me into his group and I was able to contribute to the MPL project under Sam Westrick's guidance. Umut and Sam built my foundation for how to think about research and parallel computing, and my experience of working with them convinced me to pursue research as a career.

This thesis discusses the design and implementation of the Legate runtime system. The Legate runtime system is the result of a large collaboration and engineering effort from teams at NVIDIA. I was extremely privileged and lucky to be in a position to do research on and with Legate while it was concurrently being productionized by the engineering team. This arrangement let me draw research problems from the real challenges that I thought would be coming for Legate, and leverage the quality of the engineering done by the team for the results I wanted to collect. Many of the core contributions of this thesis were in collaboration with Wonchan Lee and Manolis Papadakis, who run the Legate project at NVIDIA, and have always been supportive of and receptive to my efforts. I also want to thank Shriram Jagannathan, who took over the management of Legate Sparse and turned my prototype implementation into a production-ready version and drove adoption of the system with real customers. I want to thank the larger Legion community, whose efforts since 2012 have built the technical and engineering foundation that Legate emerged from. The members of the Legion community are the giants whose shoulders I stand upon.

I want to thank all collaborators and co-authors I've had over the years for all the work we've done that has and has not directly contributed to this thesis. The work done during my PhD would not have been the same without conversations and contributions from Rupanshu Soi, Sean Treichler, Maryam Dehnavi, David Zhang, David Broman, Shiv Sundram, Melih Elibol, Taylor Lee-Patti, Thiago Teixeira, Alexandra Henzinger, Rawn Henry, Stephen Chou, and Saman Amarasinghe.

The graduate students in Alex's and Fred's groups have grown and blossomed throughout my PhD into a tight-knit community that is a pleasure both to work with and have fun with outside of the office. I want to thank Olivia Hsu, AJ Root, Chris Gyurgyik, Shiv Sundram, Scott Kovach, Bobby Yan, Rubens Lacouture, James Dong, Haoran Xu, Rupanshu Soi, David Zhang, Matthew Sotoudeh, Anjiang Wei, Thiago Teixeira, Tony Wang, Ben Driscoll, Colin Unger, Samantha Archer, Gina Sohn, Nathan Sobotka, and Konstantin Hossfeld for all the fun conversations and time spent in the office together over the years. I will have fond memories of the seminars, Alpine Inn dinners, ski trips, and hikes throughout the years. Fellow graduate students are the people with whom the most time is spent (not the advisors), and the graduate student community at Stanford has helped me become a better scientist, a better mentor, and a better person.

I want to thank the students whom I have had the privilege of working with throughout my PhD. Working with Rohan Chanani, Joseph Guman, Ahmad Zafar, Gargi Bakshi, and Alexander Waitz has helped me grow and become a better mentor; I hope that I was able to help each of you have a rewarding research experience. Joseph's work both led the initial exploration into the area discussed in Chapter 6 and laid the technical foundations for the approach we ended up taking.

The biggest joys of my time at Stanford were the new friends that I made. Sarah Wu, Jenna Ahn, Roshni Sahoo, Judy Shen, Alex Wang, Ywen Lau, Alex Hwang, Kyle Hsu, and Rohan Taori have been there for me since the early years of my PhD. I have deeply enjoyed all of our trips, concerts, SF visits, brunches, hangs, walks, dinners, drives, dances, games, and parties. I hope that we can stay friends for the rest of our lives.

Through the ever-sunny weather at Stanford, I got back into playing tennis. Through tennis, I was able to play with the club tennis team, make many new friends through the sport, and release the ever-accumulating stress of the PhD. I want to thank Kristy Choi, Nikhil Pandit, Anirudh Jain, Erik Isele, Rajan Aggarwal, Anthony Vento, Praful Vasireddy, Michael James, Nathan Sobotka, Arjun Shah and the club tennis team for all the fun sessions, and for wanting to play with me again even after my temper on the court gets the best of me.

My undergraduate friends (and the friends made through them) have been a continuous source of support and welcome distraction from the PhD. Connor Lin started the PhD with me, and we continued living together at Stanford after several years of doing the same at Carnegie Mellon. Stella Han, Rishov Dutta, Joshua Chao, Clementine Chou, Jason Ma, Lillian Wang, Hung-wei Chuang, Mihir Punji, Nitish Gurrala, Kristin Yin and Joey Yin were endless sources of fun in our hangs all over the Bay Area. Many of my friends from undergrad moved to the East Coast; to Ajay Benno, Rahul Jaisingh and Sneha Palle, I don't get to see you as much anymore, but any time we do get to hang out it feels like the old days. Axel Feldmann and Charles Yuan were among the few others to pursue a PhD after undergrad at Carnegie Mellon, and we spent much time commiserating; Charles and Axel also hosted me on multiple visits to MIT.

Finally, I want to thank my family for their unending and unconditional love and support. My parents, Dhiraj Yadav and Monisha Medhi, have continually sacrificed for the life that I get to live. They have taught me how to be a good person and do the right thing, and nurtured my academic interests from the very beginning. My sister, Riya Yadav, has always supported me and shown me nothing but love. She is embarking on her own journey to become a (real!) doctor, to which I wish her boundless success. I love each of you so much, and dedicate this thesis to you.

Contents

List of Tables

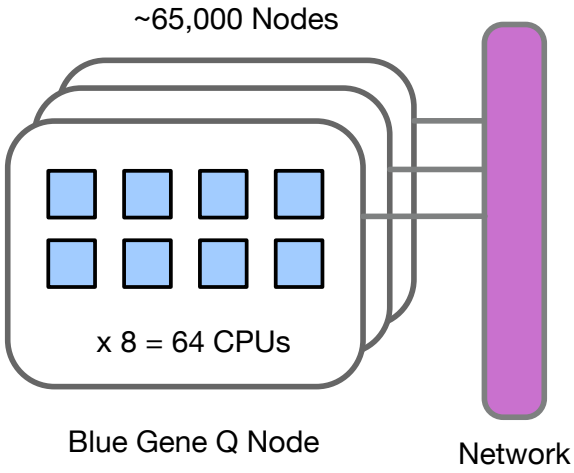
List of Figures

Chapter 1

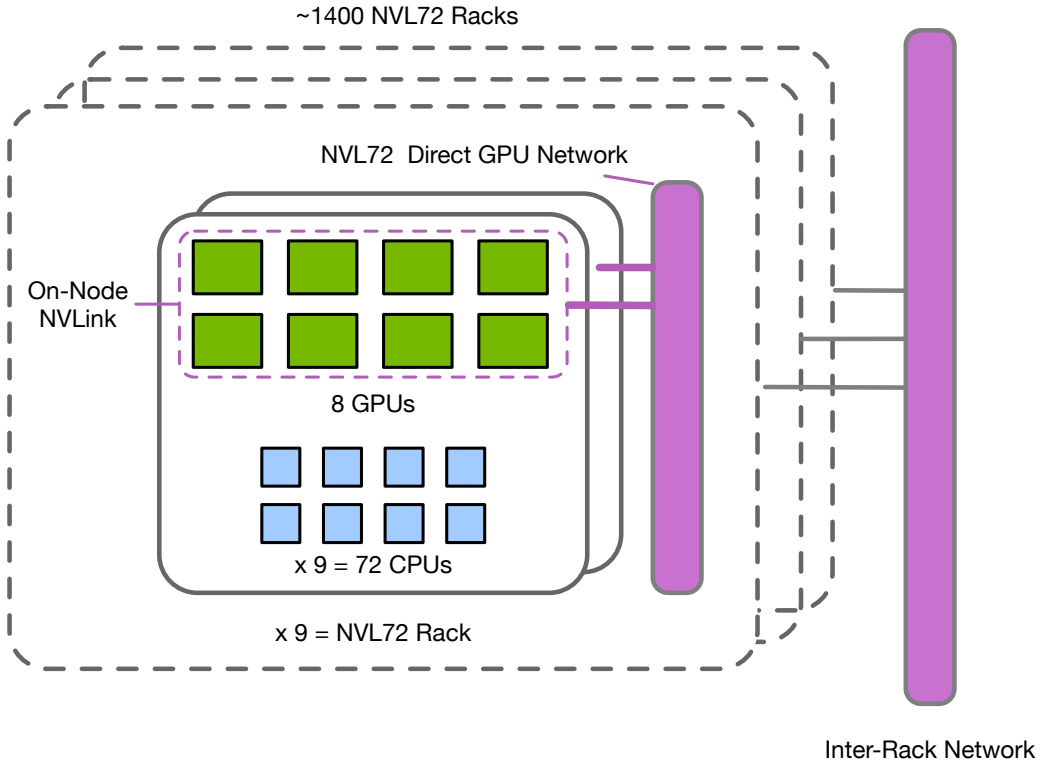
Introduction

High-performance computing systems have continued to deliver increasing performance year over year, offering more computing power as well as additional memory capacity and bandwidth. However, these gains in performance have come with rapid increases in complexity, as the development of software that achieves high performance on modern computers is becoming increasingly challenging. This complexity arises from increasing heterogeneity and hierarchy. Heterogeneity in modern systems arises from the collection of processors available on individual compute nodes, which contain multiple specialized accelerators like GPUs in addition to a general-purpose multi-core CPU. Heterogeneous compute nodes are organized into distributed machines with hierarchical networks for communication, in which links of different latencies and bandwidths connect processors within a node, between nodes in a rack, and across racks. These changes are directly visible in Figure 1.1, which sketches the architecture of the IBM Blue Gene/Q (Figure 1.1a) from 2011 and a modern NVIDIA Blackwell GPU-based supercomputer (Figure 1.1b). Achieving high performance on modern distributed and accelerated systems is a full-stack problem, requiring fast kernels that run on the individual processors, efficient orchestration software that coordinates the fast kernels while moving data across the machine, and algorithms that expose sufficient parallelism and minimize communication.

While high-performance machines are increasing in complexity, enabling developers to program them productively is a critical problem with immediate impact on



(a) IBM Blue Gene Q



(b) NVIDIA B200 NVL72 100,000 GPU Supercomputer

Figure 1.1: Comparison of high-performance machines from the 2010's and now.

scientific progress. High-performance computing is critical to workloads across computational science (such as physics, chemistry, biology, and climate science), machine learning and data analytics. Leveraging the performance improvements that modern machines offer can enable scientists to run larger or higher resolution simulations, train larger machine learning models, or analyze more data collected by physical sensing devices. The increasing complexity of programming modern machines directly decelerates the forwards progression of science. Most experts in scientific domains are not additionally experts in high-performance parallel computing, meaning that leveraging a modern high-performance system is either extremely challenging or even impossible.

This thesis describes how to control the complexity of programming modern high-performance machines through the development of *composable* software that enables end users to develop programs built from separate high-level, domain-specific libraries that, when composed, execute with the performance of a low-level, highly tuned implementation. The artifacts of this thesis are the Legate ecosystem of libraries that provide friendly interfaces to scientists, such as NumPy [68] and SciPy [128], and the Legate runtime system that enables the efficient execution of Legate programs.

Thesis statement: This thesis shows that it is possible to compose independent distributed software with high performance through a combination of the right program representations and analyses.

1.1 Software Composition

The tried-and-true approach to managing complexity in computer science is through abstraction and composition of independent modules. Computer scientists hide complex reasoning and implementation details behind simple interfaces that expose functionality to other software components. All of modern software engineering is based upon the principles of abstraction: correct modules are composed to produce new correct modules. Composition enables large software ecosystems to thrive, such as

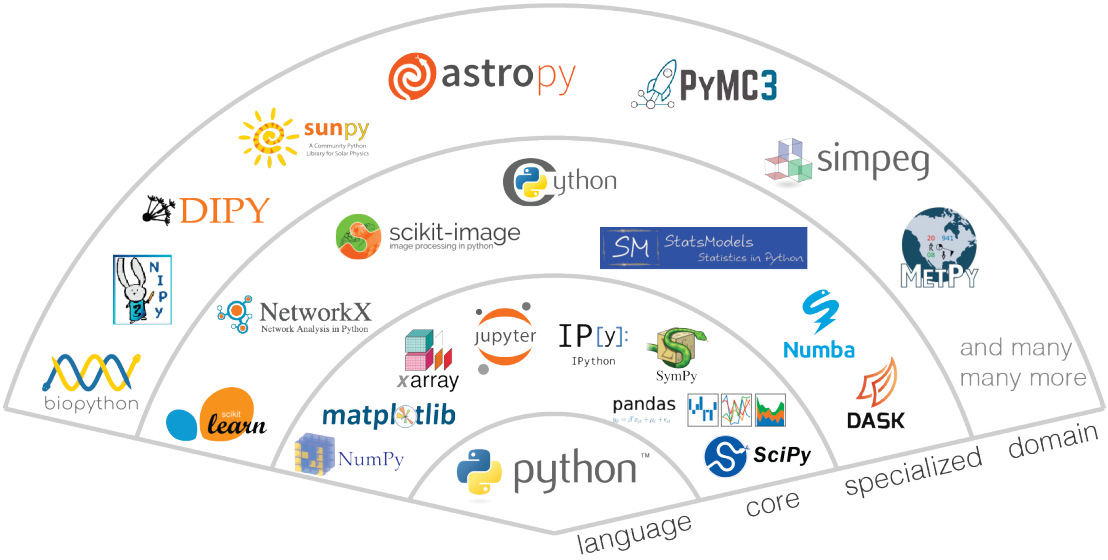


Figure 1.2: The large Python ecosystem of scientific and data processing libraries. Figure extracted from [120].

the large data processing and scientific computing library ecosystems in Python (Figure 1.2). Domain scientists develop sophisticated applications by mixing and matching components from these library ecosystems, leaving the implementation details to computer science experts.

1.1.1 Composition is (Often) Not Performance Preserving

While large library ecosystems exist and enable the development of sophisticated software in the shared-memory programming world, the composition of performant, independent modules often does not yield the same performance as a bespoke implementation tuned for a specific purpose. The performance differences can arise from a variety of sources. In some cases, the differences can arise from inefficient use of hardware features like a tiered memory hierarchy when composing programs from high-level interfaces such as those found in NumPy [98, 16]. The difference in performance can even be asymptotic [80], as additional domain knowledge about the end-to-end application may enable optimizations beyond what is possible within the purview of an individually optimized module. Performance degradation at module

boundaries may be acceptable on small machines, and developers often pay this price to develop prototype programs more quickly.

1.1.2 Composition of Distributed and Accelerated Software

While composition preserves correctness and sometimes sacrifices performance in the shared-memory programming world, neither property is maintained by the current status quo of distributed and accelerated software. Concretely, most independently written modules of distributed and accelerated software are often not even correct under composition, and if correctness is achieved, the end-to-end performance is often far from what is achieved by a hand-tuned implementation. These negative performance effects make it difficult for large library ecosystems that mirror those found in the sequential programming world to exist for distributed and accelerated machines, which, in turn, make it difficult to develop new applications that target high-performance systems.

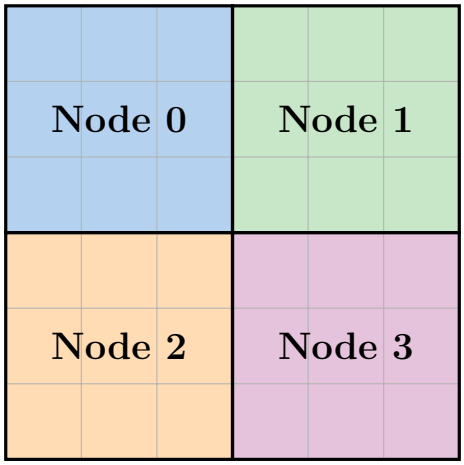
Correctness

Standard distributed and accelerated software built through bulk-synchronous, message-passing interfaces like MPI place the burden of managing correctness during composition on the programmer performing the composition. Similar burdens of coordinating data layout and representation exist in shared-memory machines, but are magnified in distributed and accelerated machines due to the realities of distributed memory. Notions of how data is distributed across the machine pervades the program, and reorganizing data across a distributed machine is significantly more complex than on a shared-memory machine.

Consider the interface to a distributed matrix-matrix multiplication operation within the distributed linear algebra library ScaLAPACK [29], shown in Figure 1.3. The interface accepts a variety of information about the operand distributed matrices, including the backing pointers to the locally held data for each process involved in the operation, as well as descriptors that inform ScaLAPACK about how each subcomponent relates to the globally distributed matrix. While the interface is sensible, it

```
1  void pdgemm(
2    // Transpose information.
3    char *transa, char *transb,
4    // Global matrix sizes.
5    int *m, int *n, int *k,
6    double *alpha,
7    // Locally-stored matrix A information.
8    double *a, int *ia, int *ja,
9    // Distribution descriptor for A.
10   int *desca,
11   // Locally-stored matrix B information.
12   double *b, int *ib, int *jb,
13   // Distribution descriptor for B.
14   int *descb,
15   double *beta,
16   // Locally-stored matrix C information.
17   double *c, int *ic, int *jc,
18   // Distribution descriptor for C.
19   int *descc
20 );
```

(a) ScaLAPACK `PDGEMM` API.



(b) Sample tiled data distribution.

Figure 1.3: Example of ScaLAPACK distributed matrix-multiplication interface.

exposes several unsavory aspects around correctness when it is composed into a larger application. First, the descriptors understood by ScaLAPACK expect only certain data distributions of the underlying matrices (1-D or 2-D block-cyclic distributions). Users must be able to either correctly describe their distributed data coming from another source in this manner, or are responsible for coordinating the transformation of their data into the distributed orientation required by ScaLAPACK, necessitating synchronization and communication over the network. Performing this transformation correctly is a burden on the user, who then must also transform the distributed data back into whatever distributed orientation is required by downstream operations. Second, by speaking in raw pointers, the user is subject to race conditions due to potential concurrent interactions in the larger application: other components of the application may be concurrently reading from or writing to the operand matrices, or pending communication intended to populate the matrices has not yet completed. The kinds of concurrent interactions can become more complex when applications utilize asynchronous accelerators like GPUs, which operate on streams of control separate from the CPU and networking operations.

Performance

Moving beyond correctness in the composition of independent modules in distributed and accelerated programs, achieving high performance from the composition of individually optimized components is challenging. Unlike when running on small machines or developing prototypes, achieving high performance on distributed and accelerated machines is critical. High-performance machines are expensive and a limited resource; they must be fully utilized to justify their use and to avoid wasting resources.

Many critical optimizations in a large distributed and accelerated application require reasoning across different logical components of the program, and discordant decisions in the individual components can result in significant performance penalties. Some of these decisions are optimizations of their own, while others are choices that interplay between correctness and performance. I now discuss some of the performance opportunities lost at module boundaries in distributed and accelerated software, grouped into opportunities for maintaining a coherent parallelism strategy and opportunities for efficiently coordinating logically separate application components.

When independently written modules are optimized, the developers make decisions about a parallelization strategy. When these strategies are chosen without knowledge of the larger application context within which a module is used, a locally optimal strategy can become globally sub-optimal. Consider the parallelization strategy described earlier used by ScaLAPACK, which accepted 2-D distributions of dense matrices. If this module is composed with a separate module that independently decides it requires 1-D distributions of matrices, a significant cost is paid to shuffle data around the system between the distributions. This cost is completely unnecessary if later computations could be rewritten or reformulated to accept the 2-D distribution as is. Alternatively, consider a module that prefers using CPUs instead of GPUs for a particular computation — in the context of a larger application that keeps data resident on the GPUs, a locally slower GPU execution would avoid the costs of moving data before and after a CPU implementation of the module. Finally, a bespoke implementation may fuse the compute kernels of logically independent application components to better utilize processor memory hierarchies, but a modular application with separate implementations forgoes the possibility of this optimization.

The composition of independent modules can then lead to inefficient handling of the application at module boundaries, which often arises from the lack of information exposed by individual modules and the limited flexibility those modules provide. For example, consider again the ScaLAPACK matrix-multiplication implementation: if this implementation launched work onto asynchronous accelerators, the user may be responsible for ensuring that other modules only read the results when the accelerators complete, and often insert overzealous synchronization to ensure application correctness. Finally, well-tuned distributed and accelerated applications heavily overlap compute and communication to find available work and keep machine resources busy at all times. In existing distributed software, finding and exploiting opportunities for overlap between and across independent modules is the responsibility of the end user, and is difficult to realize without breaking down the abstractions provided by the modules themselves.

The goal of this thesis is to support the correct and efficient composition of independent software on distributed and accelerated machines. We focus on the composition of high-level, collection-based libraries like NumPy and SciPy which form the basis of the very popular scientific computing ecosystem in Python. Scientific and data analysis workloads developed in this ecosystem have plentiful dynamic behavior, including dynamic control flow and data-dependent behavior, which render them incompatible with the advances in machine learning infrastructure [100, 35] that support domain scientists at scale on distributed and accelerated machines.

1.2 Dissertation Overview

To demonstrate the correct and efficient composition of independent software on distributed and heterogeneous machines, we have developed the Legate ecosystem of composable distributed libraries and the Legate runtime system, which contains the technical pieces that enable the composition of Legate libraries. Chapter 2 begins by providing an overview of both the Legate library ecosystem and Legate runtime system, and enumerates the many components of the deep Legate technical stack. The remainder of the thesis walks through this stack of technology and discusses the

key ideas that enable efficient and correct composition of Legate libraries. Chapter 3 discusses analysis and data representations within the Legate runtime system that enable the efficient sharing of distributed data across independent modules. Chapter 4 discusses program analyses and program representations that fuse computations across independent Legate libraries. Chapter 5 and Chapter 6 focus on different components of controlling the overheads that various layers of the Legate runtime system impose to enable efficient composition. Chapter 7 describes the process of building a Legate library (Legate Sparse [141]) using the abstractions provided by the Legate runtime system. Finally, we describe related work in Chapter 8 and draw conclusions along with discussing avenues for future work in Chapter 9.

1.3 Collaborators and Publications

This thesis discusses ideas that were the focus of several publications [141, 142, 138, 140]. Work in this thesis was done in collaboration with Michael Bauer, Wonchan Lee, Manolis Papadakis, Shiv Sundram, Melih Elibol, Taylor Lee-Patti, Sean Treichler, Joseph Guman, David Broman, Michael Garland, Fredrik Kjolstad, and Alexander Aiken. Many of the experimental results were achieved by building on the production-grade Legate implementation developed by the Legion, Legate, and Realm teams at NVIDIA.

1.4 Large-Language Model Usage Disclosure

Large language model (LLM) tools were used to proofread this document for grammar issues and to assist in the development of TikZ figures. No text or ideas present in this document were generated by LLMs.

Chapter 2

Legate Overview

In this chapter, I give an overview of the Legate library ecosystem and Legate runtime system. I discuss the many different components involved in the Legate technical stack, as well as describing the different program representations used by each layer. These representations and the transformations between them are discussed in detail in later chapters.

2.1 Legate Library Ecosystem

The Legate library ecosystem is a collection of high-level distributed libraries that act as drop-in, distributed replacements to popular Python libraries in the scientific computing, data analytics and machine learning ecosystems. An explicit goal for libraries within the Legate ecosystem is composition; Legate libraries should correctly and efficiently compose just like their sequential counterparts. An overview diagram of the Legate ecosystem is shown in Figure 2.1. Legate libraries are developed on top of the Legate runtime system, which enables their efficient composition as well as scaling from machines containing a single GPU to multi-node multi-GPU machines.

The Legate ecosystem has expanded far beyond the original prototype research libraries, and now has a team of engineers at NVIDIA that maintains existing libraries, develops new libraries, and engages with customers to help them utilize Legate. While I personally contributed to the development of a subset of these Legate libraries,

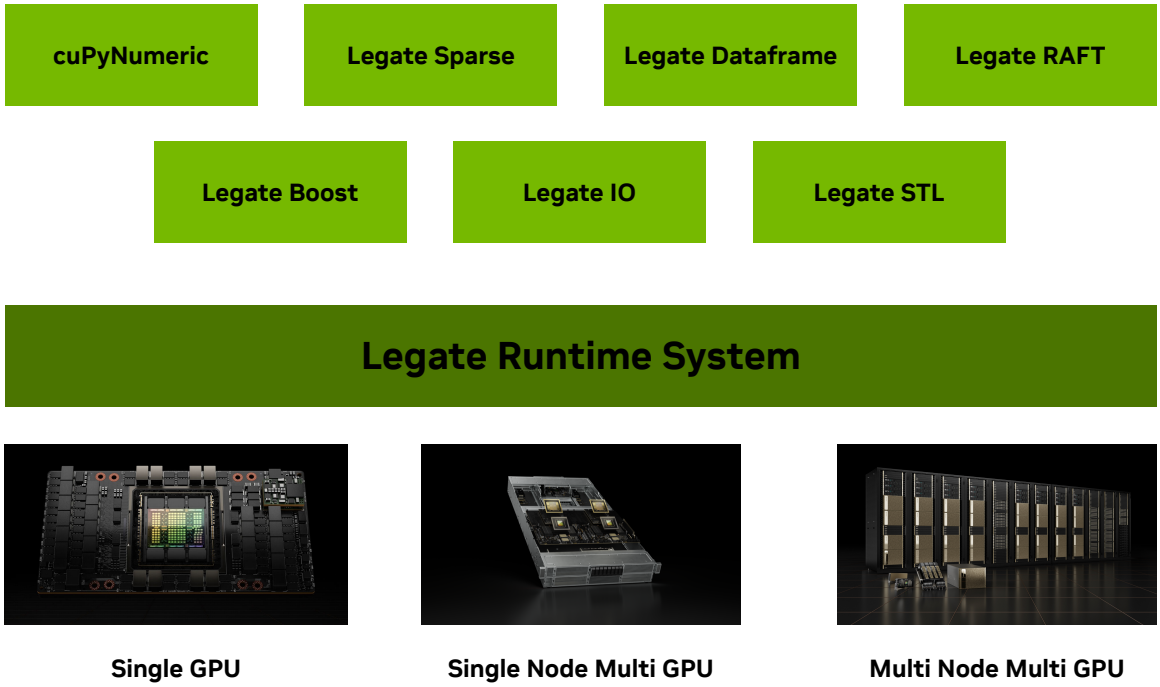


Figure 2.1: Overview of the Legate Ecosystem.

others were developed by independent groups aiming to scale their own workloads. The first Legate library was cuPyNumeric [21] (originally Legate NumPy), developed by Michael Bauer, and is a drop-in replacement for NumPy. I developed Legate Sparse [141], a distributed implementation of SciPy Sparse, that acts a companion to the distributed dense array programming provided by cuPyNumeric. All of the Legate application results presented later in this thesis contain programs built from cuPyNumeric and Legate Sparse.

Beyond dense and sparse array programming, NVIDIA engineers have developed Legate libraries that focus on data analytics and machine learning. Legate Dataframe [105] provides a Pandas-like interface to performing distributed dataframe and relational operations. Legate RAFT [106] provides computational primitives and algorithms for machine learning and information retrieval. Legate Boost [104] delivers a distributed implementation of gradient boosting for training machine learning classifiers. Legate IO [95] provides distributed file I/O support from scientific file formats like HDF5. Finally, Legate STL [45] is a lower-level C++ library that uses Legate to provide distributed implementations of collection operations found in the

C++ standard template library.

2.2 Legate Runtime

Legate libraries are developed on top of abstractions provided by the Legate runtime system, which enables Legate libraries to compose efficiently and scale to large machines. The multiple layers of the Legate runtime system are shown in Figure 2.2; these layers consist of the Legate front-end, the Legate middle-end, Legion [25], and Realm [122]. The contributions of this thesis will span across each of these layers.

At a high-level, the Legate runtime system exposes a *task-based* programming model with a strong *data model*. Legate's data model exposes distributed data as *stores*, which are multi-dimensional arrays. Libraries like Legate Sparse map distributed data structures onto collections of stores. Computations are defined through *tasks*, which are user-defined functions that operate on (subsets of) stores. Tasks declaratively specify the stores (or subsets of stores) that they will access, and perform arbitrary computation on their argument stores, such as executing GPU kernels. The different layers of Legate perform analysis and lower these abstractions onto a physical execution of tasks and data movement across the target machine. Legate's task-based programming model undergoes several lowering steps from the user-facing Legate front-end down to Realm programs that execute on the target machine.

The Legate front-end exposes a programming model that is *implicitly-parallel*, *scale-free*, and *implicitly partitioned*. Concretely, this means that programmers express computations in a sequential manner (logically, a single thread of control issues tasks into Legate), and Legate is responsible for extracting parallelism from this stream of tasks. Legate automatically partitions stores across the machine based on constraints that tasks declare on the stores they access (implicitly partitioned), and task descriptions do not specify concrete parallelization strategies (such as how many GPUs a task should be partitioned across). The flexibility exposed by the Legate front-end representation allows the Legate runtime system to make key decisions around data partitioning and parallelization that are coherent across multiple distributed libraries, which is critical for performance. These analyses are discussed

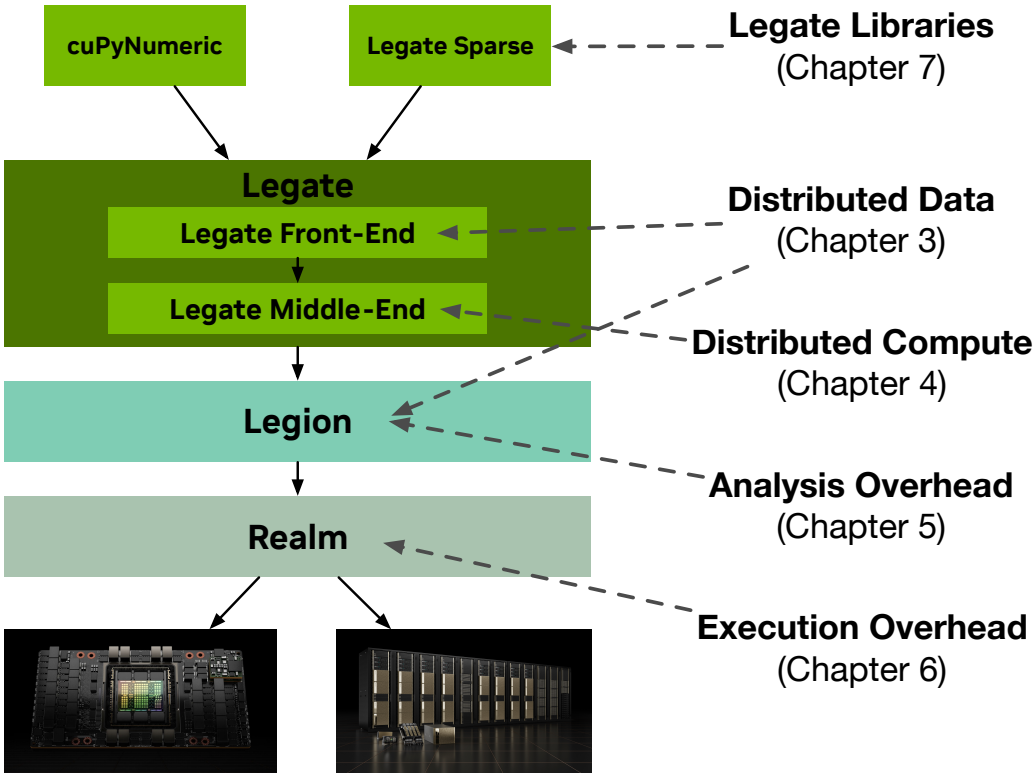


Figure 2.2: Overview of the Legate runtime system software stack. The concepts and relevant components of the runtime discussed by each chapter are shown on the right.

further in Chapter 3.

The Legate front-end representation is then lowered into a second intermediate representation, referred to as the Legate middle-end, that is implicitly-parallel, but is *explicitly partitioned* with *symbolic* scale. In this representation, tasks are associated with concrete partitions of stores to be accessed, and parallelization strategies have been assigned by Legate; the parallelization and partitioning strategies are represented in a compact manner, discussed more in Chapter 4. This representation enables the composition of computation across different distributed libraries, in particular the fusion of operations across library boundaries.

The Legate middle-end program representation is then dynamically translated into a Legion [25] program. Legion is an implicitly-parallel task-based representation with explicit partitioning and explicit scale. Legion performs dynamic analysis to extract parallelism from a sequential input stream of tasks from Legate. Legion determines what tasks are safe to execute in parallel and inserts dependencies between tasks based on the stores each task accesses. In addition to inserting dependencies between tasks, Legion also inserts the data movement operations required to maintain coherence between stores as they are accessed by tasks running on different processors on different nodes. The exact mechanisms used by Legion's analysis are out of the scope of this thesis and explored in prior work [25, 22]. However, this thesis contributes a component to Legion that enables these analyses to be performed efficiently in the context of a Legate program manipulating multiple independent libraries (Chapter 5).

Finally, as the result of dependence analysis, Legion generates task-based programs that target the Realm [122] runtime system. Realm is an explicitly-parallel, explicitly-partitioned task-based programming model with explicit scale, providing higher-level systems with a high-degree of control over how programs execute across a distributed and accelerated machine. Realm is the final layer of this software stack, and is responsible for actually executing computations on target processors, performing the desired data movement, and ensuring that dependencies specified by higher layers of the system are respected. As with Legion, much of Realm is prior work [122] and developed by others. This thesis contributes an analysis and compilation strategy for Realm programs that dramatically lowers the overheads of executing Realm

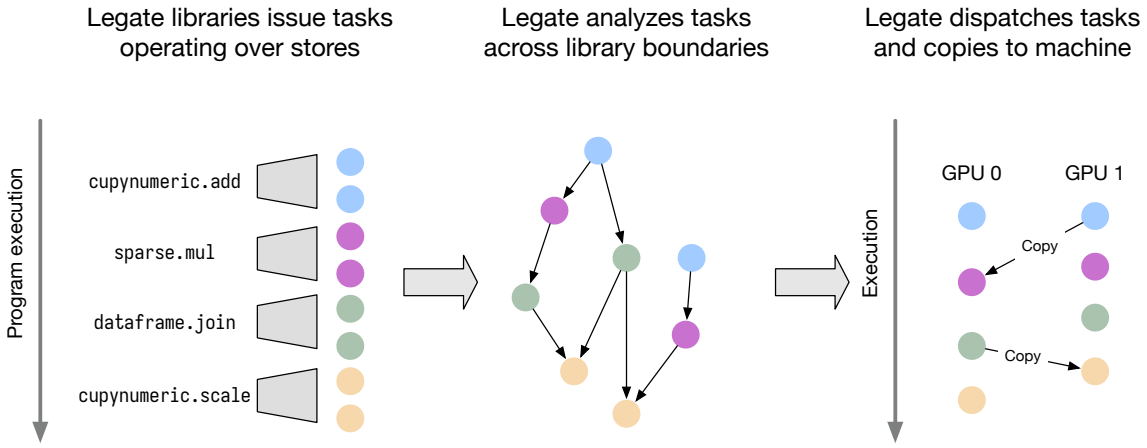


Figure 2.3: The Legate runtime analyzes and interleaves components from multiple independent libraries into a coherent and optimized execution.

programs, which improves their absolute efficiency and strong scaling, bubbling up into future improvements for the whole Legate stack (Chapter 6).

The combination of Legate's user-facing programming model and analysis pipeline enables independent libraries to launch implicitly parallel tasks over stores, and the runtime system weaves these independent tasks into a single coherent execution (Figure 2.3). Legate co-partitions the distributed data required by these implicitly parallel tasks, maps stores used by different libraries into consistent physical allocations, fuses computations across library boundaries, discovers the necessary dependencies and inserts communication to maintain the illusion of sequential execution, and finally executes an optimized graph of computations on the target machine. The ability to see through library boundaries and optimize components from independent distributed modules within the context that those modules are used is the "magic" that allows Legate to efficiently compose distributed software.

The remainder of this thesis discusses these individual layers of the Legate software stack in depth, but in the context of how the choices of representation and analysis enable composition of different performance critical components of high-performance applications. Chapter 3 discusses how representation and analysis in the Legate front-end and Legion enable independent modules to share distributed

data efficiently. Chapter 4 discusses how the representation of the Legate middle-end enables fusion optimizations that efficiently compose distributed computations across modules. Finally, Chapter 5 and Chapter 6 discuss optimizations that control the overheads imposed by the dynamic analysis at various layers of this analysis stack; controlling these overheads enables strong and weak scaling comparable to hand-optimized implementations.

Chapter 3

Composing Distributed Data

This chapter discusses program representations and dynamic analysis within both the Legate front-end and Legion runtime system that enable distributed data structures to be effectively shared across multiple independently-written distributed libraries. We first focus on the Legate front-end, and introduce Legate's developer-facing programming model, which leverages a *constraint-based* representation and analysis to co-partition distributed data across multiple libraries. We then move into components of the Legion runtime system that support the correct usage of distributed data across library boundaries, as well as the sharing of physical data allocations.

Prior Work Declaration. This chapter is based on material in the publication "Legate Sparse: Distributed Sparse Computing in Python" [141]. This chapter also discusses integration with components ofthe Legion runtime system, which is prior work and detailed in Mike Bauer's thesis [20].

3.1 Legate Front-End

As discussed in Chapter 2, the Legate front-end programming model is implicitly-parallel, scale-free, and implicitly-partitioned. The Legate front-end exposes a *data model* as well as a *computational model* to describe distributed programs; we describe each in turn. A formal grammar containing the full front-end language (describing

Syntax

Unique ID	id		
Shape	s	::=	$(\mathbb{Z}, \ldots)$
Store	S	::=	$\mathsf{Store}(id, s)$
Partition Constraint	C	::=	$\mathsf{equal}(S, S) \mid \mathsf{broadcast}(S, \mathbb{Z}\ \mathsf{list}) \mid \mathsf{image}(S, S) \mid \ldots$
Privilege	Pr	::=	Read | Write | Reduce | Read-Write
Task	T	::=	$\mathsf{Task}((S, Pr)\ \mathsf{list}, C\ \mathsf{list})$
Program	$Prog$	::=	T stream

Figure 3.1: Legate front-end program representation.

both components of the model) is shown in Figure 3.1, and is described in more detail in Sections 3.1.1 and 3.1.2.

3.1.1 Data Model

Distributed data is represented in Legate through *stores*, which are multi-dimensional arrays. As shown in Figure 3.1, each store has an (internal) unique identifier and a multi-dimensional shape. Libraries like cuPyNumeric directly represent NumPy's arrays as stores, while libraries like Legate Sparse use collections of stores to represent distributed sparse matrices. Notably, concrete partitions of distributed data are absent from the data model; the language does not directly support the construction of the subsets of a store that would be needed by each processor for a distributed computation. Instead, the language provides *partitioning constraints* that describe a set of potential possible concrete partitioning choices. This construct is discussed in significant detail in Section 3.1.3, and is a critical design choice to enable the composition of distributed data. The production implementation of the Legate front-end additionally contains operations that perform logical shape transformations on stores, such as shifting indices, or changing the dimensionality of the store. These transformations are necessary for a fully functional implementation, but we elide discussion of them in this thesis.

3.1.2 Computational Model

Distributed computation is represented in Legate through *tasks*, which are user-defined functions. Tasks explicitly declare the stores that they will access, and additionally declare how they will access the stores through *privilege annotations*. In addition to privilege annotations on argument stores, tasks also describe a list of partitioning constraints over the argument stores. These partitioning constraints describe assumptions inside the implementation of the task that require specific partitioning relationships between the task's store arguments. Once the partitioning constraints of a task are resolved into concrete partitions (Section 3.1.3), the task is broken up into *sub-tasks* that execute on each piece of partitioned data.

The representation of a complete program in the Legate front-end is a stream of tasks issued to the system by the application (possibly from independent libraries). The Legate front-end is implicitly-parallel, as the application issues a sequential stream of tasks, and Legate is responsible for extracting parallelism from this stream while satisfying the semantics of the sequential program. The representation is also implicitly-partitioned, as only partitioning constraints are provided over stores (discussed in Section 3.1.3), and Legate is responsible for choosing the concrete manner in which stores are partitioned across the target distributed machine for different operations. Finally, the representation is scale-free, as no constructs in the language are tied to the size of the physical machine. Legate can freely choose over how many processors to distribute a computation over, and the *size* of this representation does not scale with the size of the target machine.

3.1.3 Co-Partitioning Distributed Data

The previous discussions of the Legate front-end's models for distributed data and computation emphasized the freedom for the Legate runtime to make decisions about how data and computation are partitioned and distributed across the machine. To understand the importance of this freedom, we first describe the current state of developing distributed software.

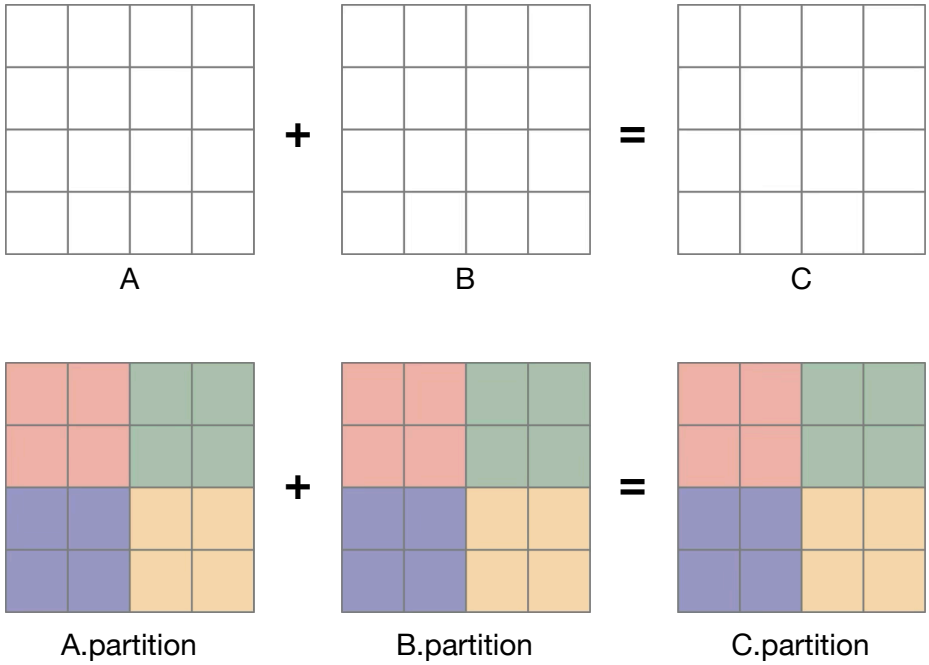


Figure 3.2: Example of an element-wise array addition operation in a hypothetical distributed array programming library. In an explicitly partitioned approach, this library must commit to a particular partitioning strategy, and maintains partitions of each array to use.

The Problems With Explicit Partitioning

The standard and widespread approach to building distributed libraries is through *explicit partitioning*, meaning that the choice of how distributed data is partitioned is explicitly specified and embedded in the target program. Explicit partitioning is separate from the underlying class of programming model, such as bulk-synchronous (MPI) or task-based (Legion); any explicitly-partitioned model is susceptible to challenges that inhibit efficient composition. Because the choice of how distributed data is partitioned is embedded in the source program, these decisions are made at the time at which a distributed library is *developed*, rather than *when the library is used*. These decisions can be locally optimal within the context of an individual distributed library, but suboptimal in the larger context of the application that the library is used within.

To make this composition problem concrete, consider the case of building a distributed NumPy library (like cuPyNumeric), and implementing the element-wise addition operation on two distributed arrays, as shown in Figure 3.2. An explicitly-partitioned system must commit to an arbitrary choice of data partitioning strategy For example, an MPI library would maintain each array in a particular partition across each rank, or a Legion library would maintain an internal `LogicalPartition` object of a particular structure to use for each array. Suppose the arbitrary partitioning choice was to maintain a 2-D tiling of each array. Problems arise when this hypothetical library is used in a larger application that first invokes a computation from a different library on the same distributed data before invoking the elementwise addition operation. If the previous library independently chose a different partitioning strategy for the same distributed data, then either the user or the underlying runtime system is responsible for shuffling the data between the two orientations of the distributed data, as shown in Figure 3.3. However, element-wise addition does not require a specific partitioning, and any partitioning strategy that aligns the partitions of the input and output arrays is sufficient for correctness. In this case, the resulting shuffles of distributed data cause a large amount of unnecessary data movement, and a significant cost to performance.

Implicit Partitioning With Constraints

Instead of forcing libraries to commit to partitioning strategies at library definition time, Legate's front-end representation uses a *constraint-based* parallelism strategy to enable the *late-binding* of partitioning decisions to the context within which a library is used. As discussed in Section 3.1.1, task definitions in the Legate front-end representation declare partitioning constraints over their input stores, instead of concrete partitions. The Legate runtime system then solves this system of constraints at program execution time to discover store partitions that align with how the stores are already partitioned within the context of the larger application. This enables distributed data to be reused in place whenever possible, and allows for the developer reasoning about implementations of tasks within libraries and individual functions to be independent of the surrounding context. A visualization of the constraint-based

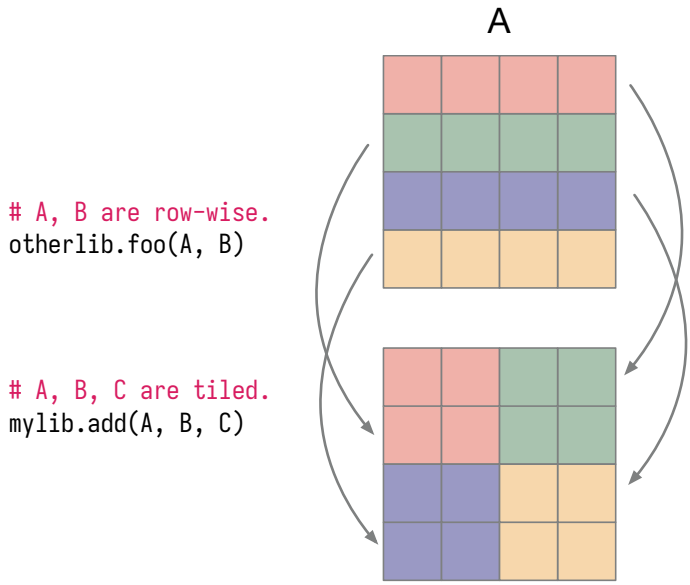


Figure 3.3: Explicit partitioning can result in inefficient execution in the context of a larger program. Each color refers to the node holding a piece of data, and the arrows indicate the data movement required between nodes to switch between the two partitioning strategies.

scheme is shown in Figure 3.4.

Constraints. The syntax presented in Figure 3.1 describes three kinds of partitioning constraints, which are sufficient to describe non-trivial partitioning strategies. As discussed above, the full partitioning implementation with the production implementation of Legate contains more partitioning constraints along with shape manipulation operations, but these are not necessary to discuss the technical core of the work. The $\mathsf{equal}(S_1, S_2)$ constraint captures partition equality between two stores, requiring the same partition choice to be made across both stores S_1 and S_2. The $\mathsf{broadcast}(S, I)$ constraint requires the dimensions in I (an integer list) to not be partitioned in a valid partitioning of S, allowing for partial replication of data. The final constraint is $\mathsf{image}(S, D)$, which relates the partitions of two stores through dynamically valued indirections. Intuitively, an image relationship between two stores projects a partition of the source store through the indirection values within the source, which contain pointers into the destination store. More formally, let a partition P be a

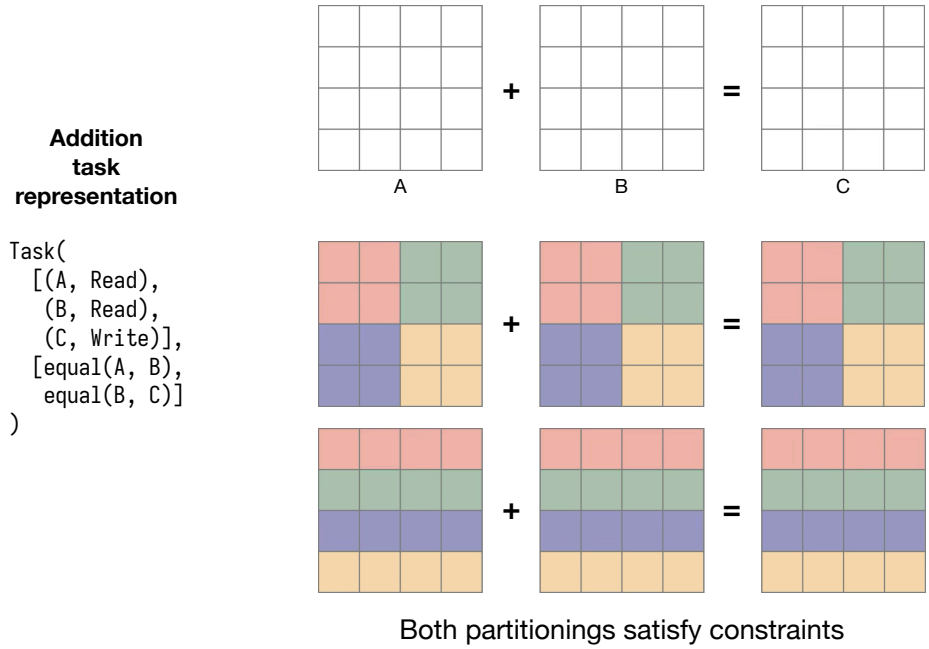


Figure 3.4: Constraint-based partitioning allows for the same logical computation to be used with multiple concrete partitioning strategies based on the larger context.

mapping from a set of *colors* to subsets of indices within a store, and let the elements of S be sets of indices in D. Then the $\mathsf{image}(S, D)$ constraint requires the partitions P_S and P_D of S and D, respectively, to satisfy the following relationship: $\forall c \in P_S, \forall i \in P_S[c], S[i] \subseteq P_D[c]$. The image partitioning operation itself was introduced in prior work [124], and is useful to support a wide variety of data-dependent partitioning patterns that occur in sparse matrix processing [141, 137, 144] and simulation [124, 14, 55].

As discussed previously, tasks declare a set of partitioning constraints over their argument stores, and delegate control to the Legate runtime system to solve the constraints for a concrete set of partitions. The more flexibility the application developer provides, i.e. declaring a partitioning constraint set with the largest possible number of solutions, the more options the Legate runtime system has to satisfy the constraints while minimizing the data movement at task boundaries. In-depth examples of using partitioning constraints are described in Chapter 7, where I describe the implementation of Legate Sparse using the Legate front-end.

Solving. Partitioning constraints are specified at task launch time, and thus must be solved by a dynamic analysis. Becuase the analysis is dynamic (and on the critical path of task launching), we currently favor speed of resolution over global optimality in the choice of solution. The Legate runtime system maintains as metadata with each store a *key partition*, which refers to the last partition that was used to write to the store with. The key partition represents an approximation of the current orientation of a store as it is distributed across the target machine; it is only an approximation because read-only operations over different partitions may have left additional copies of the store in other orientations across the target machine. The constraint resolution strategy uses these key partitions as anchors to avoid unnecessary movement of distributed data.

Legate's constraint resolution algorithm is shown in Algorithm 1. The constraint solver first builds equivalence classes between output partitions based on equality constraints. Then, the solver constructs a dependence graph between the equivalence classes corresponding to dependencies between the constraints implied by constraints like images. For example, in the set of constraints $\{\mathsf{image}(S_1, S_2), \mathsf{equal}(S_2, S_3)\}$, the partition of S_1 is an independent constraint while the partitions of S_2 and S_3 are dependent. The solver will first construct a concrete partition of S_1 before using that partition to construct concrete image partitions of S_2 and S_3.

When constructing partitions of independent equivalence classes, the solver examines if an existing key partition satisfies all constraints over the independent equivalence class. If so, the key partition can be reused and all dependent partitions derived from the independent equivalence class can be recomputed. Otherwise, the solver selects an arbitrary new partition for the independent equivalence class that satisfies the considered constraints. Each independent equivalence class may contain multiple stores, such as the partition constraint set $\{\mathsf{equal}(S_1, S_2), \mathsf{equal}(S_2, S_3)\}$. In this case, the solver selects a key partition to use (if one exists) from the three stores that minimize the total amount of data movement.

The algorithm presented in Algorithm 1 prioritized the speed of constraint resolution over global optimality, avoiding a backtracking search over the space of possible partitions in favor of a more greedy approach. More aggressive constraint solving

Algorithm 1: Legate's partitioning constraint resolution algorithm.

```
1  SolveConstraints (C)
       /* Build equivalence classes between partitions based on equality
          constraints.  */
2      E ← BuildEquivalenceClasses(C)
       /* Construct a dependence graph between equivalence classes based on
          dependent constraints like images.  */
3      G ← BuildEqClassDependenceGraph(E)
       /* Equivalence classes with no incoming edges in G are independent.  */
4      I ← IndependentEquivalenceClasses(G)
5      D ← DependentEquivalenceClasses(G)
       /* Initialize partitioning assignments.  */
6      P ← {}
       /* For each independent equivalence class, construct a find or construct
          a key partition that minimizes data movement.  */
7      foreach EqC ∈ I do
8          P ← FindOrConstructKeyPartition(EqC, P)
       /* For each dependent equivalence class, compute the partition from the
          independent equivalence classes and other dependent equivalence
          classes.  Dependent equivalence classes must be traversed in
          topological order as they can depend on other dependent equivalence
          classes.  */
9      foreach EqC ∈ D do
10         P ← ComputePartition(EqC, P)
11     return P
```

strategies are possible, such as solution strategies discussed by Lee et al. [86] that solve for partitions over an entire graph of tasks at once. Investigating such approaches in the context of iterative Legate computations is interesting future work. However, we do not currently have non-contrived applications where more global partitioning strategies are required.

3.2 Legion

In the previous section, I discussed the Legate front-end's program representation, which used implicit scale and implicit partitioning to avoid unnecessary data movement caused by incoherent partitioning decisions made by independent libraries. The Legate front-end representation is critical in enabling independent libraries to compose with high performance. I now skip the Legate middle-end representation (discussed next in Chapter 4) to focus on the Legion layer of the runtime stack, which enables both the correct and efficient composition of independent distributed libraries. This thesis does not focus on the internals or implementation of Legion, which are detailed in components of separate publications [25, 23, 24] and altogether in Mike Bauer's thesis [20]. Instead, this thesis focuses on the abstractions exposed by Legion, and how Legate uses these abstractions to enable efficient composition.

3.2.1 Legion Program Representation

As discussed in Chapter 2, Legion exposes an implicitly-parallel task-based programming model with explicit partitioning and explicit scale. Figure 3.5 presents a grammar that describes a simplified version of the Legion programming model. The structure of the Legion programming model is similar to the Legate front-end, as both layers represent programs as streams of tasks operating over stores. However, the differences arise in the representation of parallelism in computation and data.

Legion represents partitions as explicit sets of subsets of stores[1]. Partitions are represented as maps from a set of *colors* (points in a multi-dimensional index space) to

[1]Stores are referred to as *regions* in Legion-only publications.

Syntax

Unique ID	id		
Point	p	::=	$(\mathbb{Z}, \ldots)$
Rect	r	::=	$[p, p]$
Store	S	::=	$\mathsf{Store}(id, r)$
Partition	P	::=	$\mathsf{map}(p, p\ \mathsf{set})$
Privilege	Pr	::=	Read | Write | Reduce | Read-Write
Index Task	T	::=	$\mathsf{IndexTask}(p\ \mathsf{set}, (S, Pr, P)\ \mathsf{list})$
Program	$Prog$	::=	$T\ \mathsf{stream}$

Figure 3.5: Legion program representation.

a set of multi-dimensional points that represent a subset of a store. These partitions can be viewed as the output of the partitioning constraint solver in Section 3.1.3, though in reality they go through an additional lowering process before conversion into Legion partitions (Chapter 4). These subsets may be arbitrary, allowing for disjoint/aliased and complete/incomplete partitions. Scale is also represented explicitly in Legion. Tasks in Legion are expressed as *index tasks*, which refer to a group of *point tasks*, where an index task contains one point task for each point in the argument set. The data each point task accesses is described by the point's index into the corresponding partition of each argument store. For the rest of this chapter, we use task to interchangeably mean index task or point task; we specify only when the semantic difference is important. Despite this lower-level program representation (where decisions about scale and data partitioning are concrete), parallelism is still implicit; Legion is responsible for extracting parallelism from tasks in the stream while maintaining sequential semantics between tasks in the stream.

3.2.2 Composition Through Implicit Parallelism

Legion's implicitly-parallel program representation and sequential semantics offer critical capabilities for correct and efficient composition of independent libraries, and logically isolates the development of these libraries. To extract parallelism from an

implicitly-parallel, distributed program while maintaining sequential semantics, Legion is responsible for identifying and enforcing dependencies between all currently executing tasks and a newly launched task, and inserting communication to ensure the data a task views is the most up-to-date version of the data.

By performing these two analyses, Legion acts as the "glue" between independent Legate libraries. Legate libraries issue tasks independently of each other, only declaring the stores they access and how they are accessed. The Legate front-end selects partitions of stores for these tasks, but thanks to Legion, can focus only on the *policy* aspect of partitioning constraint resolution. The Legate front-end can select arbitrary choices of partitions to minimize data movement, and Legion is responsible for actually inserting the necessary data movement and synchronization (possibly across library boundaries) that result from those partitioning choices. Additionally, Legion can exploit parallelism available between libraries by finding pieces of independent work between libraries and running them concurrently, or overlapping the communication required to prepare data for one library with independent computation from a separate library. These optimizations are key pieces of tuned HPC applications that run efficiently on modern supercomputers; these applications utilize every available resource at all times to avoid exposing bottlenecks.

The kind of efficient composition enabled by Legion's implicit representation of parallelism and communication is difficult to achieve in explicitly-parallel programming models that commit to communication and synchronization with only the local program view. Since an explicitly-parallel library must describe at program development time the synchronization and communication patterns, these libraries are unable to adapt to the larger context a library is used in. For example, an MPI-based library that internally synchronizes after message passing would be unable to automatically overlap independent work from an unrelated library, or might send more data than necessary if the desired data was already available on the current node.

3.2.3 Sharing Physical Data With Composable Mapping

Independent Legate libraries describe computations over distributed data through tasks operating on stores. The Legate runtime system is responsible for actually executing these tasks on concrete processors and placing the partitioned pieces of stores into concrete memories for tasks to access. This process is called *mapping* within the Legion ecosystem; Legion exposes this functionality through user-defined *mapper* objects, which the Legion runtime system queries to answer policy decisions around where tasks should run and stores should be placed. The Legate runtime system implements a mapper and registers it with Legion. The mapper is extensible by Legate libraries, allowing some policy decisions to be exposed to and controlled by individual libraries (for example requesting that certain tasks be run on certain kinds of processors, or that stores be mapped into certain kinds of memories), while others are coordinated across library boundaries through the shared mapping infrastructure. Concretely, Legate controls the exact processors that tasks are mapped to, as well as the mapping of stores to physical allocations in concrete memories in the machine. Centralized control over the exact processors used by tasks avoids thrashing of data between different processors assigned to process the same partitions of data.

The more difficult aspect of composing mapping decisions across libraries is the mapping of stores to memories on the machine. The key challenge involved in mapping stores is how to share allocations of distributed and partitioned data between libraries. Because interactions between libraries are unknown until chosen by the partitioning constraint solver, the partitions of stores and the aliasing patterns of those partitions are also not known until program execution. To minimize data movement and memory usage, the mapping strategy must reuse and resize physical allocations across individual operations and library boundaries.

Legate maintains a record of all store allocations made on each node, and reuses physical allocations across as many tasks as possible. However, a strategy that only reuses allocations that exactly match the desired subset of a store is not sufficient to achieve good performance, as tasks from different libraries may use multiple views of the same underlying store. As an example, consider a stencil computation, where a task reads from multiple tiles offset around a center tile. An efficient mapping for this

computation would coalesce all offset tiles with the center tile into a single, larger allocation, reducing the total amount of memory and increasing cache locality.

To efficiently map multiple partitioned pieces of a store into a single allocation, Legate employs a *coalescing* step before allocating. When selecting an allocation for a store, Legate examines the existing allocations for other partitions of the same store. If another store's partition has an intersection with the partition being allocated, then Legate may merge the two views of the data into a single, larger allocation with enough space for both views of the store. We use a heuristic to drive coalescing decisions, where store partitions are coalesced if the size of their overlapping components is sufficiently larger than their non-overlapping components.

3.2.4 Execution Example

Figure 3.6 visualizes the execution of a Legate program that combines operations from cuPyNumeric (vector norm and division) and Legate Sparse (sparse matrix-vector multiplication) on a two GPU machine. An efficient implementation of this main loop with global knowledge only performs one element halo exchanges of the x vector between GPUs, and no other data movement. This figure demonstrates how the design of the Legate runtime system enables the efficient sharing of partitions and physical allocations of distributed data across library boundaries to achieve this property in the resulting composed software.

The left part of Figure 3.6 contains the source program along with an example sparse matrix A and the stores used to represent it (see Chapter 7 for more details about distributed sparse matrices and their layouts). The right part of the figure is the execution; the top half depicts partitioning and launching of tasks in Legate, while the bottom half shows the Legion-level execution on the physical machine. In the right part of the figure, each store entry is labeled with the coordinate of the entry, rather than the data value at that entry. Throughout the figure, we refer to the versions of the vector x at each iteration i of the main loop with x_i. For example, the initial vector x is denoted x_0, and the resulting x after the first $x = A \ @ \ x$ operation is denoted as x_1.

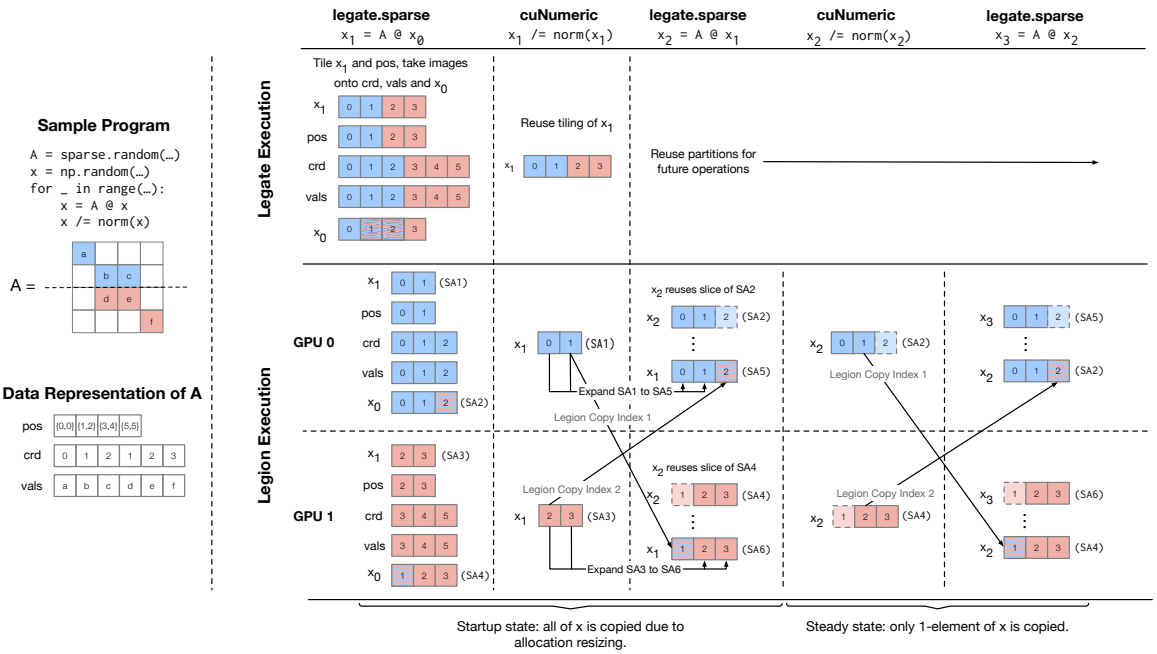


Figure 3.6: Execution of sample Legate program, with control flowing between Legate Sparse and cuPyNumeric.

We first discuss the top half of the figure, which shows how stores are partitioned for distributed execution. Legate Sparse's representation of sparse matrices and constraints used to partition them (discussed in Chapter 7) relate the `pos`, `crd` and `vals` stores with the input and output vectors. Legate solves the constraints to construct partitions (identified with blue and red colors) of each store on the first iteration, and creates a tiled key partition of x_1 and an aliased partition of x_0, based on the data dependent coordinates needed in the sparse matrix. Control flows to cuPyNumeric for the norm and division operations (which we treat as a single operation for illustration purposes). These are elementwise operations without partitioning constraints, so Legate reuses the key partitions created during the solving of Legate Sparse's constraints for cuPyNumeric. These partitioning choices are then reused for all future iterations of the main loop.

We now shift to the bottom half of Figure 3.6, which depicts the execution with Legion, and the mapping of logical operations onto physical resources. For all tasks launched, the Legate Sparse and cuPyNumeric mapping policies assign tasks and

stores to each GPU and the corresponding GPU memory. The key to peak performance in this program is the Legate mapper's choice of allocations for each store.

In the first iteration, Legate creates allocations for each store (denoted with "SA") that correspond to the exact bounds of each store requested by each library's tasks. The choices made in the second iteration of the program stress the importance of the compositional-awareness of Legate's mapping strategy. When mapping the second SpMV operation, the mapper chooses new allocations (SA5 and SA6) for each piece of x_1, resizing the allocations SA1 and SA3 to account for the larger slice of x_1 required by each SpMV task. Resizing SA1 and SA3 requires a full copy of x_1, and a single element halo-copy between GPUs. Next, Legate sees that the store backing x_0 can be deleted, and chooses to reuse the allocations SA2 and SA4 by coalescing them with the requested partition of x_2. Since the allocations have been coalesced, the SpMV tasks only operate on a slice of the allocation, and similarly with the norm and division tasks. The elements outside of these tasks' slices are denoted by a faded color. At the start of the third iteration, the application hits a steady state where the existing allocations are large enough to re-use without additional resizing, causing only the single-element halo-copy to occur. Without the mapper's coalescing step, the full vector copy executed in the first iteration would be executed in each iteration, resulting in a significant loss of performance.

This example demonstrates how the use of constraint-based parallelization enables logically isolated implementations of operations to compose, and how information can be shared during mapping to extract efficient communication patterns.

3.3 Evaluation

The presented strategy and Legate runtime system subset (focusing on composing distributed data efficiently) is sufficient to implement non-trivial applications that achieve good performance and scale to large machines. Chronologically, the Legate Sparse publication [141] was first, and contained the presented core techniques to build independent libraries that efficiently share distributed data. I will discuss the initial applications developed using a combination of Legate Sparse and cuPyNumeric and

evaluated on the (now decommissioned) Summit supercomputer. I will show that the presented techniques for composing distributed data enable complex applications to be built from independent components, while still achieving performance competitive with hand-tuned systems.

Each Summit node has a 40 core dual socket IBM Power9. Each socket has three NVIDIA Volta V100s connected by NVLink 2.0, for a total of six GPUs per node. Each node is connected by an Infiniband EDR interconnect.

Our initial evaluation of Legate consists of Python applications developed using a combination of SciPy Sparse and NumPy, and range from twenty-five to nearly a thousand lines of code. We measure Legate's performance running in both CPU-only and GPU-only settings, allowing us to compare against systems that only support CPUs or GPUs. On a single node, we compare against the standard implementations of SciPy and NumPy for CPUs, and CuPy for GPUs. CuPy provides a drop-in replacement for the SciPy and NumPy APIs, but can only utilize a single GPU. On multiple nodes, we compare (when a hand-tuned baseline exists) against the industry-standard PETSc sparse linear algebra library, which supports both CPUs and GPUs. PETSc provides a C API with high-level linear algebra operations similar to SciPy, but requires users to both specify low-level details about partitioning and distribution and hand-write many distributed NumPy-like array computations. For all experiments, we collect 12 runs of data points, drop the fastest and slowest runs, and then average the results of the remaining 10 runs.

3.3.1 Weak Scaling

In this section, we evaluate the weak-scaling performance of Legate, emulating a usage where users increase the size of their machine to scale to larger data sets. For all benchmarks but the quantum simulation, we compare the performance between one socket of CPUs and the three GPUs connected to that socket. However, we start the weak-scaling at one GPU to compare performance with CuPy. We plot throughput on a log-log plot due to the order-of-magnitude difference in performance between various systems.

CG Solver.

We implemented a conjugate-gradient iterative linear solver for a 2-D Poisson problem, with the results displayed in Figure 3.7. As with the previous experiment, we compare both modes of Legate to the same code run in SciPy and CuPy, and a comparable implementation in PETSc. Legate's CPU mode outperforms SciPy due to being multi-threaded. Legate and PETSc achieve nearly perfect weak scaling on CPUs, with PETSc slightly outperforming Legate, as Legion reserves some CPU resources for runtime work. On GPUs, CuPy, Legate and PETSc have similar performance on a single GPU, with Legate achieving 85% of the performance of PETSc. PETSc and Legate then weak-scale from a single GPU, where PETSc achieves nearly perfect weak scaling, starting to fall off slightly at 192 GPUs. Legate also scales well, but experiences some performance drop-off at 32 nodes due to the fast GPU kernels exposing overheads in Legion's all-reduce implementation[2], causing the dot-product communication in the CG solve to affect Legate's performance at a smaller processor count than PETSc. At 192 GPUs, Legate achieves 65% of PETSc's performance.

Multi-grid Solver.

We implement a two-level geometric multi-grid conjugate gradient solver, which uses the injection restriction operator and a weighted Jacobi smoother[3]. Multi-grid methods are known to be relatively challenging to implement correctly and efficiently on distributed machines — our implementation is 300 lines of Python. We do not have a distributed reference implementation, so we compare Legate's CPU mode to SciPy, Legate's GPU mode to CuPy, and then weak-scale to larger machines. Figure 3.8 contains the weak-scaling results for the geometric multi-grid solver. As with prior experiments, Legate's CPU version significantly outperforms SciPy and has good weak-scaling to 64 sockets. On a single GPU, CuPy is 30% faster than Legate's GPU version. This performance difference is caused by overheads in the Legate library due to its Python implementation. During the V-cycle of the multi-grid method, the application launches several tasks small enough to expose overheads in Legate's task

[2]The Legion developers are aware of this issue, and plan to address it in the future.
[3]This benchmark was inspired by, but is not directly comparable to HPCG [69].

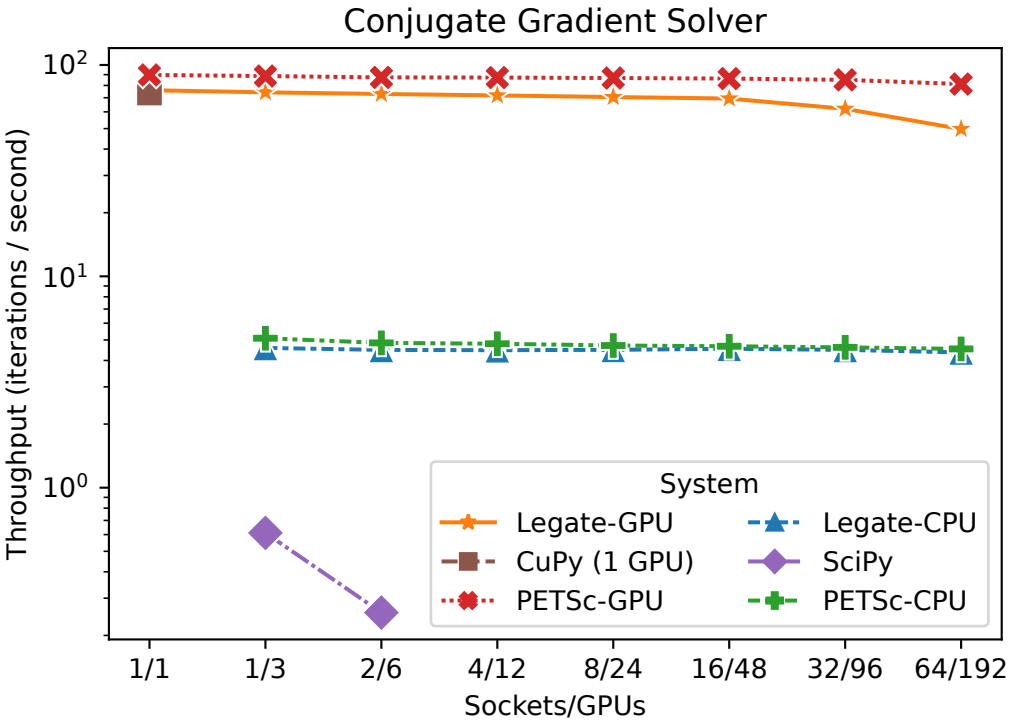


Figure 3.7: Weak-scaling of a Conjugate Gradient solver.

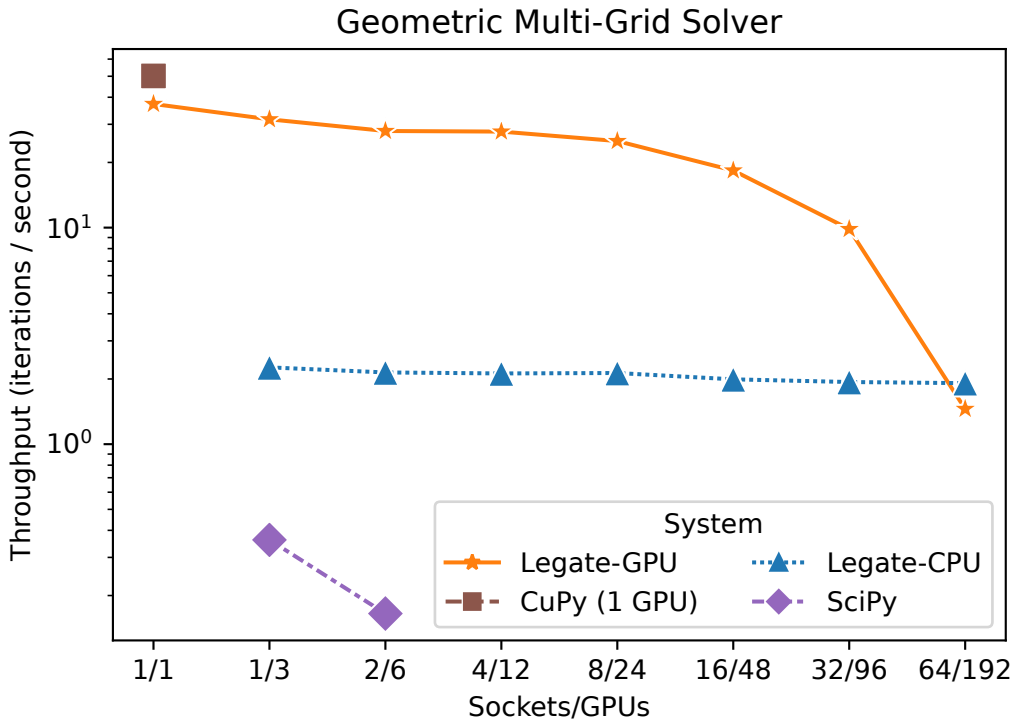


Figure 3.8: Weak-scaling of a Geometric Multi-Grid solver.

launching and metadata management. Legate's GPU version starts off weak-scaling well, but has kernels that run fast enough to expose overheads in Legion, which have been fixed since the original collection of these results. Benchmark results later in this thesis (in Chapter 4) show significantly improved strong scaling of the GMG code. Despite the imperfect weak scaling, Legate is able to execute the Python multi-grid solver on accelerated hardware much faster and on larger problem sizes than SciPy.

Quantum Simulation.

We develop a Legate quantum simulation of Rydberg atom arrays. The simulation can be used to solve Maximum Independent Set (MIS) problems, as pioneered by the group of Mikhail D. Lukin and QuEra Computing [56]. Like previous implementations [102], we significantly reduce the memory footprint of the simulation by including only states that are allowed by the Rydberg blockade mechanism [89]. Likewise, the interactions between states are rather sparse, as they only permit transition

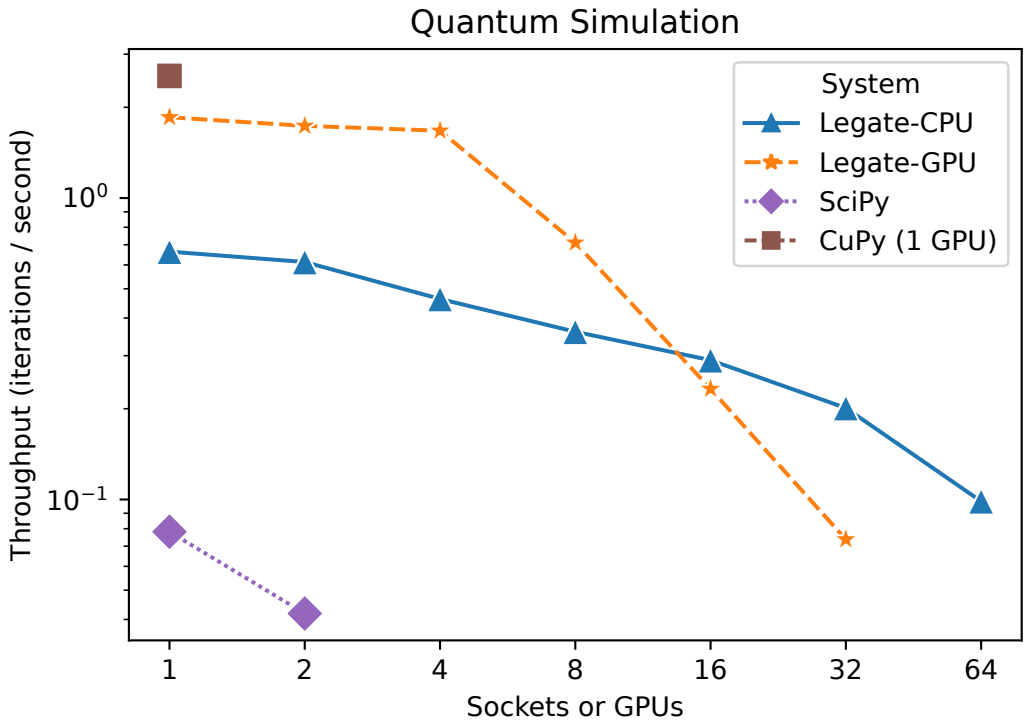


Figure 3.9: Weak-scaling of a Runge-Kutta integration-based Quantum Simulation.

between states in adjacent excitation manifolds and otherwise identical excitation structure. Competing quantum dynamics, namely the energy terms stemming from laser detuning of the system, are inherently sparse due to their diagonal action. Nevertheless, the exponential growth of the quantum state space is only partially stymied by exploiting inherent problem structure, so the simulation remains memory hungry. This application was developed in Python without any expectation that it would be eventually executed in a distributed fashion; the algorithms used in the simulation could be tuned to achieve more scalable performance. We aimed to maximize scale of the Python application as-is, and were able to achieve the exact simulation of the full wave function of up to 50 qubits at time-scales consistent with deep circuits or high entanglement.

The core computational component of this benchmark is an 8th-order Runge-Kutta integration. using the "DOP853" method, which we ported directly from the implementation in SciPy. Despite being written with no expectation of distributed

execution, Legate is able to efficiently parallelize and distribute this program, leveraging the efficient composition enabled by sharing distributed data across Legate Sparse and cuPyNumeric.

Similarly to the GMG benchmark, we compare against SciPy and CuPy. Due to the nature of the application, we were unable to exert fine-grained control over the input size: we could only approximately double the problem size. Therefore, we utilize 4 of the 6 GPUs on each Summit node for this benchmark to directly compare weak-scaling performance between CPUs and GPUs. We stress that Legate can successfully utilize all 6 GPUs per node for standard runs of the simulation.

The weak-scaling results are found in Figure 3.9. As with prior experiments, Legate significantly outperforms the standard implementation of SciPy. On a single GPU, CuPy achieves a 40% speedup over Legate, for a similar reason as the GMG benchmark — several small tasks launched in the integration expose overheads in Legate. The simulation experiences a loss in weak-scaling efficiency as the number of processors increases. This fall-off is expected due to the communication pattern of the application: the sparse matrices that describe the atomic relationships have a very high bandwidth (the coordinates in a row reference a wide range of columns). Our profiling shows that the algorithms used by the application require every processor to exchange tens to hundreds of megabytes of data with at least half of the other processors in the system, almost an all-to-all communication pattern.

At 1 to 4 GPUs, Legate's GPU version significantly outperforms the CPU version, due to utilizing the higher-bandwidth NVLink. Once inter-node communication over Infiniband is required after 4 GPUs, the GPU version has similar performance as the CPU version, even dropping below the CPU performance at 16 GPUs. This drop is due to the ratio of communication to effective bandwidth available between each experiment: at 16 GPUs, Legate's GPU version is utilizing 4 nodes of network hardware, while Legate's 16-socket CPU version is using 8 nodes to exchange the same amount of data, thus having double the network bandwidth available to communicate through. Finally, the large halo regions present in the application result in imperfect weak scaling of the memory usage per processor, causing Legate's 64 GPU version to run out of memory.

3.4 Conclusion

This chapter discussed program representation and analysis within the front-end of the Legate runtime system as well as lower-level components where Legate interacts with Legion below it to enable applications to share distributed data across independent modules. Efficiently and correctly sharing distributed data is a "zero-to-one" jump in the kinds of high-level programs that can be built from composing independent modules, as demonstrated in Section 3.3. User-level programs and the implementations of Legate libraries can now be built while requiring only local reasoning about the functionality and performance of a function or module, and the correctness, communication, synchronization, concrete partitioning and mapping to physical data are all handled in the larger context by the Legate runtime system.

Chapter 4

Composing Distributed Computation

The previous chapter presented the Legate front-end program representation and discussed Legate's use of the Legion runtime system to allow independent libraries to efficiently and correctly share distributed data, and skipped discussion the intermediate layer between the Legate front-end and Legion. This chapter focuses on the program representation and analysis used within the Legate middle-end that enables the fusion of distributed computations across function and library boundaries. We show that careful choice of these representations and placement within the larger Legate architecture enables a scalable, efficient, and precise fusion analysis. Exploiting fusion in distributed computations requires precisely reasoning about aliasing and mutation of the target program's distributed data structures. These analyses are only made more challenging in the presence of data-dependent control-flow and communication patterns, which are common in Legate programs. We approach this problem in Legate by separating the overall problem of fusion into two components: 1) *task fusion*, which determines whether the bodies of tasks can be combined without communication, and 2) *kernel fusion*, which fuses the many computational loops of fused tasks. This analysis and approach to fusion allows for Legate to find optimization opportunities missed by application developers, and allows Legate applications to even exceed the performance of programs developed in specialized frameworks.

Prior Work Declaration. This chapter is based on material in the publication "Composing Distributed Computations Through Task and Kernel Fusion" [142]. The chapter references the MLIR [84] compilation framework, which is prior work and was used to support our implementation.

4.1 Legate Middle End

As discussed in Chapter 2, the Legate middle-end program representation is implicitly parallel (like the Legate front-end and Legion), but is explicitly partitioned with symbolic scale. In this representation, stores are assigned concrete partitioning strategies, and the width of parallelism is represented symbolically. The symbolic representation means that even though data has been partitioned, the size of the program representation itself does not scale with the size of the target machine (unlike in lower-level models like Legion and Realm), enabling a fusion analysis with a cost that is also independent of the scale of the target machine. The Legate middle-end representation is shown in Figure 4.1a, visualized in Figure 4.1b, and discussed in detail next. The data and computational abstractions are then lowered onto the corresponding Legion abstractions after the analyses discussed in Section 4.2.

4.1.1 Data Model

Similarly to the Legate front-end and Legion, distributed data is represented as stores. We refer to the rectangular shape of a store as a *domain*, which is also used to describe the decomposition of data and compute across processors. Stores are partitioned across the target machine into *sub-stores*, which are the subsets of a store.

Partitions of stores are first-class objects in the Legate middle-end, unlike the constraint-based representation in the Legate front-end and the set-of-sets representation in Legion. A *partition* is a mapping from points in a domain to sub-stores, where each point in the domain represents a processor. The mapping encoded by a partition is represented in a structured manner, breaking different kinds of partitions into different syntactic groups. For simplicity of presentation, we consider a

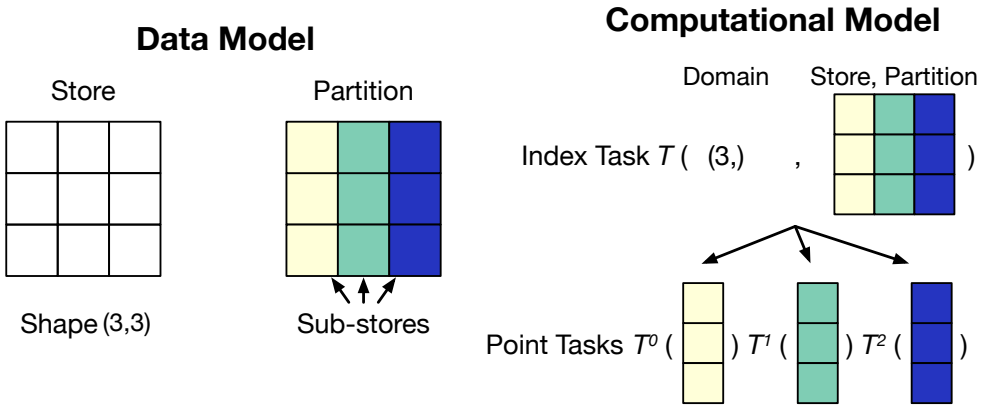
Syntax

$$
\begin{array}{rlll}
\textit{Unique ID} & id & & \\
\textit{Shape} & s & ::= & (\mathbb{Z}, \ldots) \\
\textit{Point} & p & ::= & (\mathbb{Z}, \ldots) \\
 & & & \\
\textit{Store} & S & ::= & \mathsf{Store}(id, s) \\
\textit{Projection Function} & F & ::= & \mathsf{Projection}(id, \mathsf{Point} \rightarrow \mathsf{Point}) \\
\textit{Partition} & P & ::= & \mathsf{None} \mid \mathsf{Tiling}(s, p, F) \mid \mathsf{Image}(S, P, S) \\
 & & & \\
\textit{Privilege} & Pr & ::= & \mathsf{Read} \mid \mathsf{Write} \mid \mathsf{Reduce} \mid \mathsf{Read\text{-}Write} \\
\textit{Index Task} & T & ::= & \mathsf{IndexTask}(p, (S, Pr, P)\ \mathsf{list}) \\
\textit{Program} & Prog & ::= & T\ \mathsf{stream} \\
\textit{Task Window} & W & ::= & \mathsf{prefix}(Prog)
\end{array}
$$

Constructs for Reasoning

$$
\begin{array}{rlll}
\textit{Sub-Store} & S^p & \triangleq & \mathsf{SubStore}(S, P, p) \\
\textit{Point Task} & T^p & \triangleq & \mathsf{PointTask}((S^p, Pr)\ \mathsf{list})
\end{array}
$$

(a) Abstract syntax of the Legate middle-end.



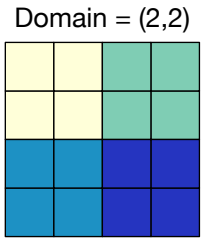
(b) Visualization of Legate middle-end constructs.

Figure 4.1: Legate's middle-end program representation and visualization.

limited set of partition kinds, sufficient to explore Legate's analyses. The production implementation of Legate supports more partition kinds with no additional technical insights. The main requirement on partitions is that two partitions of the same kind can be checked for inequality without examining each sub-store within each partition. This requirement is critical for a scalable analysis, as discussed in Section 4.2.

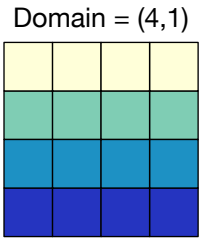
The first partition kind None represents the replication of a store, where all points in the partition's domain are mapped to the entire store. The second partition kind Tiling represents an n-dimensional affine tiling of a store. A Tiling contains an n-dimensional tile shape and an offset from the origin, which are used to compute the sub-store associated with each point in the partition's domain. For example, Figure 4.2a shows a tiling of a two-dimensional store using 2x2 tiles over a 2x2 domain, while Figure 4.2b shows a row-based tiling (i.e., tiles of size 1x4) of the same store over a 4x1 domain. Figure 4.2c shows a partition of a subset of the store beginning at the point $(1, 1)$. Tiling partitions also contain a *projection function* that applies a transformation to each point in the partition's domain before computing the subset with the tile size and offset. Projection functions enable Tiling partitions to express aliased and replicated data. For example, Figure 4.2d shows a vector tiled over a two-dimensional domain by a projection function that discards the second dimension of each point in the partition's domain, resulting in a partially aliased partition. The formula that defines the sub-store bounds for each point of a Tiling partition is shown in Figure 4.2e. The final partition kind is Image, which as discussed previously, relates a source and destination store through a dynamically valued indirection. The representations of each partition are scale-free, as the mapping of points to sub-stores is implicitly described by the structure of the partition, rather than explicitly storing a set-of-sets of points describing each sub-store within the partition. These partition kinds are the concrete partitions synthesized by the partitioning constraint resolution process described in Section 3.1.3.

To reason about the sub-store referenced by each point of a partition, we include an explicit $\mathsf{SubStore}(S, P, p)$ construct, representing the sub-store associated with point p of store S using partition P. As a short-hand, we let $S[P, p] = \mathsf{SubStore}(S, P, p)$, and refer to S as the *parent* store of $S[P, p]$. The indices contained within the sub-store



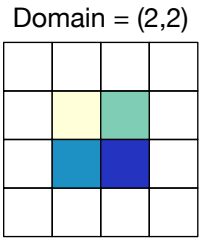

```
Tiling(
  shape=(2,2),
  offset=(0,0),
  proj=id,
)
```

(a) 2x2 tiling of a 4x4 store.



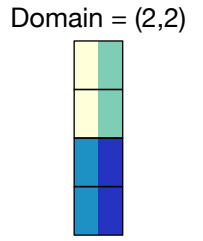

```
Tiling(
  shape=(1,4),
  offset=(0,0),
  proj=id,
)
```

(b) 1x4 tiling of a 4x4 store.




```
Tiling(
  shape=(1,1),
  offset=(1,1),
  proj=id,
)
```

(c) Offset 1x1 tiling of a 4x4 store.




```
Tiling(
  shape=(2,),
  offset=(0,),
  proj=fn p -> (p[0],),
)
```

(d) Aliased blocking of a size 4 store.

$$\textsf{sub-store-bounds}(\textsf{Tiling}(\textsf{shape}, \textsf{offset}, \textsf{proj}), p) =$$

$$[\textsf{proj}(p) * \textsf{shape}, \textsf{proj}(p+1) * \textsf{shape}) + \textsf{offset}$$

(e) Function that computes a bounding-box within the store that a Tiling partition maps point p to.

Figure 4.2: Examples of Tiling partitions. Partitions maps points in the denoted domain to sub-stores. Each color refers to the sub-store associated with each point in the domain.

$S[P, p]$ are directly computable in cases when P is None or Tiling, but may depend on runtime values held by stores when P is Image. Our later definitions assume that it is possible to find the intersection between two sub-stores, but our fusion algorithm in Section 4.2 does not require explicit computation of these intersections.

4.1.2 Computational Model

The Legate middle-end represents programs as a stream of tasks. Like Legion, the Legate middle-end represents task groups through the *index task* construct, which compactly describes a group of parallel tasks. An $\mathsf{IndexTask}(d, A)$ represents a group of parallel tasks over points in a rectangular domain d, referred to as the *launch domain*. Stores are passed to tasks with privilege annotations just like in other layers of the Legate runtime system, and each parallel task within the group reads from, writes to, or reduces to the sub-stores referred to by the stores and partitions at each point. This representation is explicitly parallel as tasks are annotated with their launch domain and partitions of distributed data structures. However, the representation is scale-free as the size of the representation is independent of the degree of parallelism — only the symbolic size of the launch domain increases. This representation is in contrast with Legion's index tasks, which explicitly materializes the set of points describing sub-tasks in the group.

Similar to sub-stores, Legate's middle-end representation has an explicit notion of a *point task*, which is one point in an index task's launch domain. Given an index task $T = \mathsf{IndexTask}(d, A)$, let T^p be the point task at point $p \in d$, operating on the list of stores $[(S[P, p], pr) : \forall\ (S, pr, P) \in A]$. Point tasks operate on the sub-stores corresponding to their index point.

We define the predicates $\mathsf{R}(T, (S, P))$, $\mathsf{W}(T, (S, P))$ and $\mathsf{Rd}(T, (S, P))$ to be true when the task T correspondingly reads from, writes to, or reduces to the store S using partition P. When (S, P) has the privilege Read-Write, both $\mathsf{R}(T, (S, P))$ and $\mathsf{W}(T, (S, P))$ are true. We also overload these predicates for point tasks and sub-stores, where $\mathsf{R}(T^p, S)$ is true when point task T^p reads sub-store S.

4.2 Distributed Task Fusion

Legate leverages the middle-end representation to fuse distributed computations through task fusion, which then enables the fusion of computational kernels within the fused tasks (Section 4.4) yielding end-to-end speedup through reduced runtime overheads and faster kernels. Legate buffers the stream of issued tasks into a *window* of tasks to be analyzed before translating and forwarding them to Legion. A distributed task fusion algorithm finds a fusible prefix of tasks in the window, and replaces the prefix with a fused task. To be fusible, the prefix of index tasks must be executable in sequence without cross-processor communication. We define when communication may occur between index tasks and describe when a sequence of index tasks is fusible. We then give an algorithm for finding fusible index task sequences.

4.2.1 Dependencies

Communication is required between point tasks that have a dependence. The dependence implies synchronization and potentially data movement between the point tasks. A dependency exists between two point tasks that access the same data unless both tasks read from or reduce to the data with the same associative and commutative operator. Recall that for an index task T, we refer to the point task at point p as T^p. We define $\mathsf{dep}(T_1^p, T_2^{p'})$ to be true if $T_2^{p'}$ depends on T_1^p.

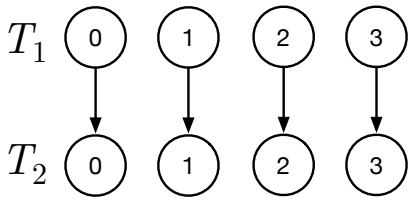
Definition 1. Given point tasks T_1^p, $T_2^{p'}$ where index task T_1 is issued before index task T_2, $\mathsf{dep}(T_1^p, T_2^{p'})$ if $\exists$ sub-stores S, S' with the same parent such that $S \cap S' \neq \emptyset$ and either

true-dep: $\mathsf{W}(T_1^p, S) \wedge \Big(\mathsf{R}(T_2^{p'}, S') \vee \mathsf{W}(T_2^{p'}, S') \vee \mathsf{Rd}(T_2^{p'}, S')\Big)$

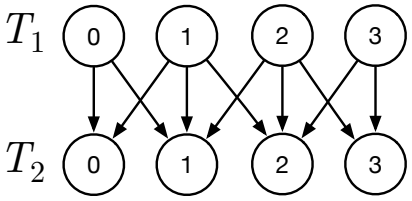
anti-dep: $\mathsf{R}(T_1^p, S) \wedge \Big(\mathsf{W}(T_2^{p'}, S') \vee \mathsf{Rd}(T_2^{p'}, S')\Big)$

reduction-dep: $\mathsf{Rd}(T_1^p, S) \wedge \Big(\mathsf{R}(T_2^{p'}, S') \vee \mathsf{W}(T_2^{p'}, S')\Big)$.

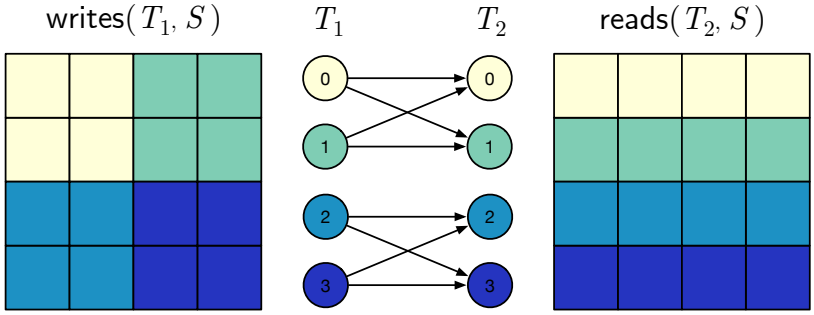
The dependencies between two index tasks T_1 and T_2 are defined by the pairwise dependencies of their point tasks. We capture these dependencies through a mapping



(a) Point-wise dependence map: $\mathcal{D}(T_1, T_2)[p] = \{p\}$



(b) Stencil dependence map: $\mathcal{D}(T_1, T_2)[p] = \{p-1, p, p+1\}$



(c) $\mathcal{D}(T_1, T_2)$ of writing to, then reading from different partitions.

Figure 4.3: Visualization of dependence maps $\mathcal{D}(T_1, T_2)$.

between the points of T_1 and T_2 that represents all of the point tasks in T_2 that depend on point tasks in T_1. Figure 4.3 shows different dependence maps over the launch domain $(4,)$.

Definition 2. For two index tasks T_1 and T_2, the *dependence map* $\mathcal{D}(T_1, T_2)$ is a function of type $\mathsf{domain}(T_1) \to \mathcal{P}(\mathsf{domain}(T_2))$, where $\forall p \in \mathsf{domain}(T_1), \mathcal{D}(T_1, T_2)[p] = \{p' \in \mathsf{domain}(T_2) : \mathsf{dep}(T_1^p, T_2^{p'})\}$.

Having characterized the dependencies between two distributed index tasks T_1 and T_2, we can now define when fusion of T_1 and T_2 is valid. T_1 and T_2 may be fused if the only dependencies that exist between their point tasks are at most point-wise, as the processor that executes each point task does not need to communicate with any other processors.

Definition 3. Index tasks T_1 and T_2 are fusible if $\forall p, \mathcal{D}(T_1, T_2)[p] \subseteq \{p\}$.

While Definition 3 admits a simple dependency structure, there are several subtleties in what tasks are fusible and the identification of fusible tasks. First, tasks

with at most point-wise dependencies is a broader set than just tasks that perform point-wise array operations. Point-wise dependencies allow for simultaneous reads and writes of different stores (Section 5.1.1) and multiple reductions to the same store. While task dependencies may be at most point-wise, the computations within the tasks are arbitrary computations that may be more complex than point-wise operations. Next, identifying when at most point-wise dependencies exist between two index tasks is non-trivial as tasks operate on arbitrarily aliasing distributed data. Legate defines a framework, discussed next, to reason about fusion in this setting, allowing for fusion to be performed between components within and across libraries.

4.2.2 Fusion Algorithm

A naïve algorithm for fusion might fully materialize $\mathcal{D}(T_1, T_2)$ to check that the condition in Definition 3 holds. However, the computation required to materialize $\mathcal{D}(T_1, T_2)$ scales with the number of processors. Even lower-level systems like Legion do not materialize all of $\mathcal{D}$, but instead leverage sophisticated algorithms to compute only the portion of $\mathcal{D}$ needed by each node [24]. However, a key insight in Legate is that to perform distributed task fusion effectively, our analysis only needs to rule out cases where $\exists\, p, \mathcal{D}(T_1, T_2)[p] \not\subseteq \{p\}$. Legate's middle-end representation enables this analysis to be performed irrespective of the scale of the target machine. Our algorithm for distributed task fusion identifies when index tasks have point-wise dependencies through greedy application of a set of *fusion constraints* to identify a fusible prefix of the task window. We then build a fused task from the identified prefix. We describe each of these components in turn, and then sketch a correctness proof.

Fusion Constraints.

We define four constraints to identify when communication may occur between distributed index tasks, i.e., when $\exists p, \mathcal{D}(T_1, T_2)[p] \not\subseteq \{p\}$. These fusion constraints are sound, but not complete—for example, leveraging application knowledge could result in fusion opportunities that are out of scope. Figure 4.4 presents each of the constraints used by Legate by defining when a provided sequence of tasks satisfy the

$$
\begin{aligned}
&\mathsf{launch\text{-}domain\text{-}equivalence}([T_1, \ldots, T_n]) = \\
&\quad \forall i, \mathsf{domain}(T_i) = \mathsf{domain}(T_1) \\
&\mathsf{true\text{-}dependence}([T_1, \ldots, T_n]) = \\
&\quad \forall\ T_i \text{ s.t. } \mathsf{W}(T_i, (S, P)), \\
&\quad \nexists T_j \text{ s.t. } (\mathsf{R}(T_j, (S, P')) \vee \mathsf{W}(T_j, (S, P'))) \ \wedge i < j \wedge P \neq P' \\
&\mathsf{anti\text{-}dependence}([T_1, \ldots, T_n]) = \\
&\quad \forall T_i \text{ s.t. } \mathsf{R}(T_i, (S, P)), \\
&\quad \nexists T_j \text{ s.t. } \mathsf{W}(T_j, (S, P')) \ \wedge i < j \wedge P \neq P' \\
&\mathsf{reduction}([T_1, \ldots, T_n]) = \\
&\quad \forall T_i \text{ s.t. } \mathsf{Rd}(T_i, (S, P)), \\
&\quad \nexists T_j \text{ s.t. } (\mathsf{R}(T_j, (S, P')) \vee \mathsf{W}(T_j, (S, P'))) \ \wedge i \neq j
\end{aligned}
$$

Figure 4.4: Fusion constraints employed by Legate to identify potential communication between index tasks.

constraint.

Launch Domain Equivalence. The first constraint checks that the tasks to be fused have the same launch domain. The partitioning constraint resolution process may decompose computations across different launch domains, or applications may scope their computation directly to different subsets of the machine, requiring different launch domains. Data movement is generally required between these different decompositions.

True Dependence. The next constraint utilizes the partitions of stores and the privileges with which they are accessed to identify communication between index tasks caused by read-after-write dependencies. The true-dependence constraint checks that if a task T_i writes to a store S through partition P, then it cannot be followed by a task T_j that reads or writes to S with an aliasing partition P', as T_j requires communication of the updated values written by T_i. However, operating on the same partition P is permitted, preserving point-wise dependencies between T_i and T_j.

Our analysis relies on the ability to check whether two partitions alias, which Legate does through a constant-time equality check between partitions. Constant-time alias checking is possible through the Legate middle-end representation's symbolic representation of scale and the syntactic grouping of partitions into structured kinds. Legate does not need to compute pairwise intersections of the sub-stores accessed by the point tasks of considered index tasks, a computation that scales

quadratically with the number of processors. Additionally, the alias analysis does not depend on the structure of the partitions, as the constraints are defined without knowing the syntactic kinds of each partition. Finally, this aliasing check is not too coarse, since partitions of different syntactic kinds nearly always alias in practice.

Anti-Dependence. The anti-dependence constraint ensures that $\mathcal{D}$ does not contain write-after-read dependencies. The constraint enforces that if a task T reads a store S, then any tasks that write to S must write to the same distributed view as the read to be fused with T. Thus, a fused task may read from multiple different distributed views of a store, but then cannot write to any of the views, as such an operation would require communication of the written data.

Reduction. The reduction constraint makes sure that viewing a partially reduced value is not allowed. It does not permit a task that reads from or writes to a store to be fused with a task performing a reduction to any view of the same store.

Fused Task Construction.

Legate's fusion algorithm greedily applies the fusion constraints on the input task window to find its longest fusible prefix. The true-dependence and anti-dependence constraints are verified through a forwards dataflow analysis on the task window. The analyses iterate through the candidate prefix of tasks, and track the effects that each task applies to its argument stores. Once a suitable prefix of the task window has been identified, Legate builds a fused task that has all store arguments and executes the same computation as the identified prefix of tasks. The fused task's store arguments are constructed by reading all stores read by tasks in the prefix, and similarly for the written to and reduced to stores. Stores that are both read from and written to are promoted to the Read-Write privilege. Legate constructs the body of the fused task by composing the bodies of each task in the prefix in program order—we further discuss this process in Section 4.4.

4.2.3 Proof of Correctness

We now show that our algorithm correctly fuses sequences of distributed index tasks. We prove the following statement:

Theorem 1. Given a window of tasks $[T_1, \ldots, T_n]$, our task fusion algorithm identifies a prefix $[T_1, \ldots, T_f]$ and produces a fused task F such that

1. $[T_1, \ldots, T_f]$ are fusible, and
2. F preserves all dependencies of the task sequence $[T_1, \ldots, T_f]$.

We provide a proof sketch for each component of the theorem. To prove that $[T_1, \ldots, T_f]$ are fusible, we must show that for each pair of tasks $T_i, T_j, i < j$ in $[T_1, \ldots, T_f]$, $\forall\ p,\ \mathcal{D}(T_i, T_j)[p] \subseteq \{p\}$. The launch-domain-equivalence constraint ensures that the dependence map is between points of the same dimensionality. For the sake of obtaining a contradiction, suppose $\exists\ p, p'$ such that $p \neq p'$ and $\mathsf{depends}(T_i^p, T_j^{p'})$. Then one of the three dependencies in Definition 1 must exist. Suppose that the condition for true-dep is satisfied, meaning that $\exists S, P, P'$ such that $S[P, p] \cap S[P', p']$ and $\mathsf{W}(T_i, (S, P))$ and one of $\mathsf{R}(T_j, (S, P'))$, $\mathsf{W}(T_j, (S, P'))$ or $\mathsf{Rd}(T_j, (S, P')$ is true. $\mathsf{R}(T_j, (S, P'))$ or $\mathsf{W}(T_j, (S, P'))$ are contradictions, as the true-dependence constraint would disallow fusion. $\mathsf{Rd}(T_j, (S, P'))$ is a contradiction due to the reduction constraint. Similar logic can be applied to other dependence cases. Here, we show that our algorithm is sound by identifying cases where fusion is possible—we do not claim completeness by proving the converse.

We have shown that all dependencies between index tasks are at most point-wise, so any T_j^p can only depend on T_i^p, where $i < j$. Since the fused task body is the composition of each task in $[T_1, \ldots, T_f]$ in program order, all dependencies in $[T_1, \ldots, T_f]$ are preserved.

4.2.4 Discussion

Performing fusion at the Legate layer between the application and Legion is key for a domain-agnostic analysis, and for analysis scalability as the size of the machine

increases. We compare against fusion on high-level domain-specific libraries, and against fusion within lower-level runtime systems like Legion.

Domain-specific algorithms for fusion [35, 136, 137, 2, 125] are effective optimizations for individual distributed libraries. Approaches that perform fusion on a set of domain-specific computations use algorithms and analyses that are tied to the domain of computations being optimized, especially analyses related to distributed memory. As a result, these techniques do not readily generalize across libraries. Legate targets fusion in the more general case after computations have been decomposed into tasks in a domain-specific manner, enabling domain-agnostic analyses to find optimizations across function and library boundaries. We expect that domain-specific techniques may be used in conjunction with Legate's analysis.

While generality is lost when fusing operations within individual libraries, scalability becomes a concern when analyzing lower-level program representations. A key design decision in the Legate middle-end representation is its use of symbolic scale, as the representation of parallel task groups and partitions of distributed data are independent of the degree of parallelism. This design enables Legate to symbolically compute a conservative estimate of the aliasing relationships between distributed data structures through constant-time queries, which are heavily used when defining the fusion constraints in Figure 4.4. In contrast, Legion's lower-level representation of partitions that explicitly maps points to sets of indices scales with the number of pieces the data is partitioned into. These representations are more flexible than the Legate middle-end, but result in the aliasing relationship queries needed by a fusion algorithm to scale with the degree of available parallelism.

4.3 Task Fusion Optimizations

Having described our algorithm for task fusion, we now describe optimizations necessary for a practical implementation. We show how to eliminate temporary distributed data structures (Section 4.3.1) and how to memoize the fusion analysis (Section 4.3.2). Temporary elimination and memoization are widely applied optimizations; we discuss how to perform these optimizations in a distributed, task-based setting.

```
1 import cupynumeric as np
2 x, y = np.zeros(n), np.ones(n)
3 flush_window()
4 z = 2.0 * x
5 w = y + z
6 v = w ** 2
7 norm = np.linalg.norm(w[len(w)//2:])
8 del x, y, z, w
9 flush_window()
```

(a) cuPyNumeric code fragment.

```
1 # Partitions and launch
2 # domains excluded.
3 ---
4 MULT([(x, R), (z, W)])
5 ADD([(y, R), (z, R, (w, W)])
6 POW([(w, R), (v, W)])
7 ---
8 NORM([(w[len(w)//2:], R), (norm, Rd)])
```

(b) Emitted task stream.

Figure 4.5: Example of distributed temporaries.

4.3.1 Temporary Store Elimination

Once Legate identifies a fusible prefix of tasks, stores that fusion has made temporary may be promoted into task-local data. Conversion of distributed data into task-local data is critical for realizing the benefits of fusion, as task-local data can then be optimized away (Section 4.4) to maximize reuse.

To introduce when a store is temporary, consider the cuPyNumeric program in Figure 4.5a and the resulting task stream in Figure 4.5b. This example introduces some new operations, specifically `flush_window`, which sends all pending tasks through Legate's analysis into Legion, and the Python `del` operator, which drops references. The program creates the stores `x`, `y`, `z`, `w`, and `v`. Consider the program state after line 9: the tasks that initialize `x` and `y` have executed, as the first `flush_window` call sent those tasks through Legate. We note that there are no pending tasks outside the window, and future tasks are ones the application may launch once the call to `flush_window` returns. The fusion algorithm determines that the tasks issued by lines 4–6 can be fused, while the final `norm` must be excluded. First, `v` is not temporary because the application holds a reference to it, meaning that it could launch a task that reads `v` after the call to `flush_window()`. Next, while the application has deleted its reference to `w`, the `norm` task reads a piece of `w` and is still pending after the fused task, and thus must observe any effects performed on `w`, meaning that `w` is not temporary. The stores `x` and `y` are only read by the fused task, and thus are not temporary. Only `z` is temporary because it is produced entirely within the fused task and is not visible to the application or pending tasks. We formalize this intuition as constraints that must be satisfied for a store to be temporary.

Definition 4. Given tasks $[T_1, \ldots, T_f, \ldots, T_n]$, a store S is *temporary* in the fusion of $[T_1, \ldots, T_f]$ if

1. If $\exists\, T_j, P$ s.t. $\mathsf{R}(T_j, (S, P))$, $\exists\, T_i$ such that $i < j \wedge \mathsf{W}(T_i, (S, P)) \wedge \mathsf{covers}(S, P)$
2. $\nexists\, T_k, P$ s.t. $k > f \wedge \mathsf{R}(T_k, (S, P)) \vee \mathsf{Rd}(T_k, (S, P))$
3. S has no live application references.

The function $\mathsf{covers}(S, P)$ is true when the partition P contains all points in the store S. The first two constraints check that the store's contents are entirely created within the fused task and not used by any other existing task; these conditions are checked through a forwards dataflow analysis of the task stream. The third constraint ensures that the application can no longer view any effects on a store, checked through a split reference counting scheme on stores in the Legate runtime. The split reference counting scheme separates references held by the application from references held by the Legate runtime. Temporary stores are demoted from a distributed allocation into a task-local allocation, as described in Section 4.4.

4.3.2 Memoization of Fusion Analysis

The final component of our distributed task fusion pipeline is memoization analysis and code generation (Section 4.4). The key challenge in memoization is allowing for the analyses to be replayed on *isomorphic* task streams rather than identical task streams. Consider the streams of tasks in Figure 4.6a, where partitions and launch domains are excluded.

Legate may reuse the analysis results from the left stream in Figure 4.6a on the middle stream, as the pattern of stores among tasks is isomorphic. In contrast, the right task stream in Figure 4.6a has a different pattern of stores across tasks, particularly the use of `S7` in `T3`. We observe that this problem is identical to *alpha-equivalence*, where each store argument is a bound variable. We identify when two task streams are isomorphic within Legate through a conversion to and comparison on a canonical, De Bruijn index-like representation. This representation is shown

```
T1([(S1,Read),(S2,Write)])   T1([(S5,Read),(S6,Write)])   T1([(S5,Read),(S6,Write)])
T2([(S2,Read),(S1,Write)])   T2([(S6,Read),(S5,Write)])   T2([(S6,Read),(S5,Write)])
T3([(S1,Read),(S3,Write)])   T3([(S5,Read),(S7,Write)])   T3([(S7,Read),(S7,Write)])
T4([(S3,Read),(S1,Write)])   T4([(S7,Read),(S5,Write)])   T4([(S7,Read),(S5,Write)])
```

(a) Two isomorphic task streams and one differing task stream.

```
T1([(0,Read), (1,Write)])                    T1([(0,R), (1,W)])
T2([(1,Read), (0,Write)])                    T2([(1,R), (0,W)])
T3([(1,Read), (2,Write)])                    T3([(2,R), (2,W)])
T4([(2,Read), (0,Write)])                    T4([(2,R), (0,W)])
```

(b) Canonical representations of isomorphic and differing streams.

Figure 4.6: Example of task stream memoization.

in Figure 4.6b. A similar technique has previously been used to avoid enumerating instruction sequences equivalent up to register renaming [17].

4.4 Kernel Fusion

The final component of fusion in the Legate middle-end is a compilation stack to optimized fused tasks. A high-level program representation is required to both perform optimizations like loop fusion and to lower to different backends like GPUs and multi-threaded CPUs. We leverage the MLIR compiler stack, which is extensible and is pre-packaged with many common compiler analyses. We first provide background on MLIR, and then describe the code generation process and optimizations performed. We then discuss how the Legate architecture enables the separation of reasoning about distributed programs from the optimization of nested loops.

4.4.1 MLIR Background

We leverage MLIR [84] to build a JIT compiler for Legate. MLIR is an extension of LLVM [83] that provides compiler infrastructure for program analyses on higher-level languages than three-address code. The most relevant component of this infrastructure to our work is the notion of a *dialect*, which is an intermediate representation that has user-defined semantics. A key aspect of dialects in MLIR is that a single MLIR program can contain types and operations from multiple dialects, enabling the composition of dialects with different semantics. Compilers built using the MLIR

framework run passes over programs that either optimize the operations within a single dialect, or convert between dialects to perform progressive lowering. Legate leverages community-developed dialects and passes to optimize and lower task bodies into CPU and GPU code.

4.4.2 Generator Functions

Legate library developers implement tasks by providing task bodies that run on processors like CPUs or GPUs. To opt into Legate's kernel fusion, developers register a *generator* function for the implementation of a task that produces an MLIR fragment describing the task's computation. We found the integration effort of adding these generator functions to be modest, requiring 50–100 lines of C++ code per operation. Only library developers, not end users of Legate, must develop MLIR kernels for tasks. Additionally, the integration effort was incremental—as more tasks were implemented with MLIR generators, Legate could exploit more kernel fusion. An example generated MLIR fragment by cuPyNumeric for an element-wise addition operation is shown in Figure 4.7b.

The generated MLIR fragment in Figure 4.7b contains multiple dialects: 1) stores are mapped onto the `memref` dialect, which provides stronger aliasing guarantees than raw pointers; 2) dense iteration is mapped onto the `affine` dialect, a target for polyhedral compilation [32]; and 3) the computation itself is mapped onto the `arith` dialect, containing arithmetic operations. Using MLIR, other dialects can be used to express higher level operations, like dense and sparse tensor algebra with the `linalg` and `sparse_tensor` dialects.

4.4.3 Compilation Pipeline

When Legate identifies that a sequence of tasks may be fused, it invokes each task generator and constructs an MLIR module containing the body of each task in the original program order. Figure 4.7c shows a fused task for the cuPyNumeric computation `c = a + b; e = c + d`, where all variables represent distributed vectors. This program originally has two index tasks (one for each add operation) which are fused

```
class Task {
  // Standard implementation.
  void gpu_variant(
    vec<pair<Store, Priv>> args,
  );
  // Opt-in MLIR task body generator.
  mlir::func::FuncOp generator(
    vec<pair<StoreDesc, Priv>> args
  );
};
```

(a) API to define a Legate task amenable to fusion.

```
func.func @kernel(
  %a: memref<?xf64>,
  %b: memref<?xf64>,
  %c: memref<?xf64>) {
  %dim = memref.dim %c, 0
  affine.for %i = 0 to %dim {
    %0 = affine.load %a[%i]
    %1 = affine.load %b[%i]
    %2 = arith.addf %0, %1
    affine.store %2, %c[%i]
  }
}
```

(b) MLIR generated for an element-wise addition.

```
func.func @fused_kernel(
  %a: memref<?xf64>,
  %b: memref<?xf64>,
  %c: memref<?xf64>,
  %d: memref<?xf64>,
  %e: memref<?xf64>) {
  %dim = memref.dim %e, 0
  affine.for %i = 0 to %dim {
    %0 = affine.load %a[%i]
    %1 = affine.load %b[%i]
    %2 = arith.addf %0, %1
    affine.store %2, %c[%i]
  }
  affine.for %i = 0 to %dim {
    %0 = affine.load %c[%i]
    %1 = affine.load %d[%i]
    %2 = arith.addf %0, %1
    affine.store %2, %e[%i]
  }
}
```

(c) Initial body of fused task.

```
func.func @fused_kernel(
  %a: memref<?xf64>,
  %b: memref<?xf64>,
  %d: memref<?xf64>,
  %e: memref<?xf64>) {
  %dim = memref.dim %e, 0
  %c = memref.alloc %dim
  affine.for %i = 0 to %dim {
    %0 = affine.load %a[%i]
    %1 = affine.load %b[%i]
    %2 = arith.addf %0, %1
    affine.store %2, %c[%i]
  }
  affine.for %i = 0 to %dim {
    %0 = affine.load %c[%i]
    %1 = affine.load %d[%i]
    %2 = arith.addf %0, %1
    affine.store %2, %e[%i]
  }
}
```

(d) After temporary elimination.

```
func.func @fused_kernel(
  %a: memref<?xf64>,
  %b: memref<?xf64>,
  %d: memref<?xf64>,
  %e: memref<?xf64>) {
  %dim = memref.dim %e, 0
  affine.par %i = 0 to %dim {
    %0 = affine.load %a[%i]
    %1 = affine.load %b[%i]
    %2 = arith.addf %0, %1
    %3 = affine.load %d[%i]
    %4 = arith.addf %2, %3
    affine.store %2, %e[%i]
  }
}
```

(e) Fully optimized fused task.

Figure 4.7: Fused MLIR kernel for three way element-wise addition traversing the compilation pipeline. The initial kernel is created by sequentially composing two of the generated task bodies in Figure 4.7b.

into a single index task where the original task bodies (the MLIR in Figure 4.7b) appear sequentially in the fused task. Before optimization of the task body, Legate first promotes distributed data into task-local allocations, resulting in Figure 4.7d.

After elimination of temporary stores, we apply passes that fuse and parallelize nested loops, and remove task-local temporary allocations to yield the optimized code in Figure 4.7e. The generated kernel is the ideal implementation for the original program: the separate loops of the original task bodies have been combined into a single pass over the vectors, and the temporary `c` has been eliminated. The optimized kernel is then lowered to GPU kernel launches or OpenMP parallel regions.

In this work, we leverage polyhedral optimizations [32, 33] to perform fusion and parallelization of loops in kernels. However, with higher level dialects in MLIR, various domain-specific kernel fusion techniques could be leveraged within a fused task body. We consider the exact kernel fusion techniques used to be orthogonal to our work.

4.4.4 Qualitative Benefits

We note several qualitative benefits of Legate's architecture in contrast to approaches that attempt to optimize distributed programs entirely through analysis of imperative code. A key design decision of Legate is to leverage a distributed data model in a dynamic representation of computation with symbolic scale to enable cheap dependence analysis between distributed computations. Separating out the reasoning about distributed computation avoids intertwining loop optimizations with distributed communication analyses, allowing the loop optimizations to remain unaware of the distributed context. This separation also allows for information gained during the distributed analysis phase to be used in code generation: properties such as array non-aliasing are provided to the MLIR optimization passes to generate better code. Finally, the separation of distributed computation into tasks means that Legate does not need to identify optimizable fragments of static source code from the larger application source. Through the process of building a Legate library, developers extract the key computational kernels from code that does not need to be considered for

analysis and optimization.

4.5 Experimental Results

We now evaluate the performance that the fusion of computation enabled by the Legate middle-end representation and analysis over the initial performance achieved by the data composition techniques in the rest of Legate. We performed these experiments on the (now decommissioned) NVIDIA Selene supercomputer, which is a cluster of NVIDIA A100 DGX SuperPOD nodes. Each Selene node has 8 A100 GPUs with 80GB of memory, connected by NVLink and NVSwitch connections, and a dual socket, 128 core AMD 7742 Rome CPU with 2TB of memory. Each node is connected via an InfiniBand connection through 8 NICs.

For each experiment, we perform a weak-scaling study, and report the throughput achieved per processor. Each reported value is the result of performing 12 runs, dropping the fastest and slowest runs, and then computing the average of the remaining 10 runs. In the weak-scaling experiments (Section 4.5.1), we exclude a set of warmup iterations from timing to measure the steady-state performance with and without fusion by Legate. We separately evaluate the overhead that Legate's fusion imposes due to compilation in Section 4.5.2.

Overview.

We evaluate fusion in Legate on unmodified cuPyNumeric and Legate Sparse applications that range from tens to thousands of lines of Python. An overview of each application is in Table 4.1. We compare each application's performance when run with and without fusion — no changes to the application are necessary. For some applications, a suitable baseline written in the industry-standard PETSc [13] library for distributed sparse linear algebra already exists, and we compare against those baselines. For other applications, we compare against manually optimized implementations by the original authors. However, some full cuPyNumeric applications have no baseline other than when run without fusion—these applications are sufficiently complex that developing an independent high-performance distributed, multi-GPU

Benchmark	Tasks per Iteration	Tasks per Iteration (Fused)	Avg Task Length (ms)	Window Size
Black-Scholes	67	1	5.3	70
Jacobi	3	2	5.3	5
CG	12.1	4.1	1.9	10
BiCGSTAB	27.1	8.1	1.7	15
GMG	24.1	11.1	1.8	15
CFD	378	40.7	1.1	30
TorchSWE	276.5	152.8	1.4	30

Table 4.1: Tasks per iteration with and without fusion. Task count is not always whole as iterations may launch different tasks, or fusion occurs across iteration boundaries. Reported task granularities are from unfused single-GPU executions. Window size was selected by Legate.

implementation is not feasible. We show that when fusion opportunities are available, Legate can exploit them to find speedups in unmodified, distributed applications. Legate enables high-level programs to equal, and in many cases improve on, the performance of hand-optimized code.

We do not ablate on the optimizations in Section 4.3, as temporary elimination is essential for speedup with kernel fusion and memoization is a requirement for a practical implementation. The window sizes shown in Table 4.1 were selected automatically by the Legate runtime through a process that increases the window size when all tasks in the current window size were fused. As a result, these window sizes enable the maximum amount of fusion possible in each application.

4.5.1 Weak Scaling Experiments

Black-Scholes. The Black-Scholes option pricing benchmark is a trivially-parallel micro-benchmark that contains a sequence of 67 data-parallel, and thus fusible, operations. It is a micro-benchmark that provides a reference point on potential improvement when the entire application is amenable to fusion. Figure 4.8 shows that Legate achieves a 10.7x speedup over the unfused program on 128 GPUs, as the fused program is a single task containing a single GPU kernel making one pass over the data, greatly increasing the arithmetic intensity of the computation.

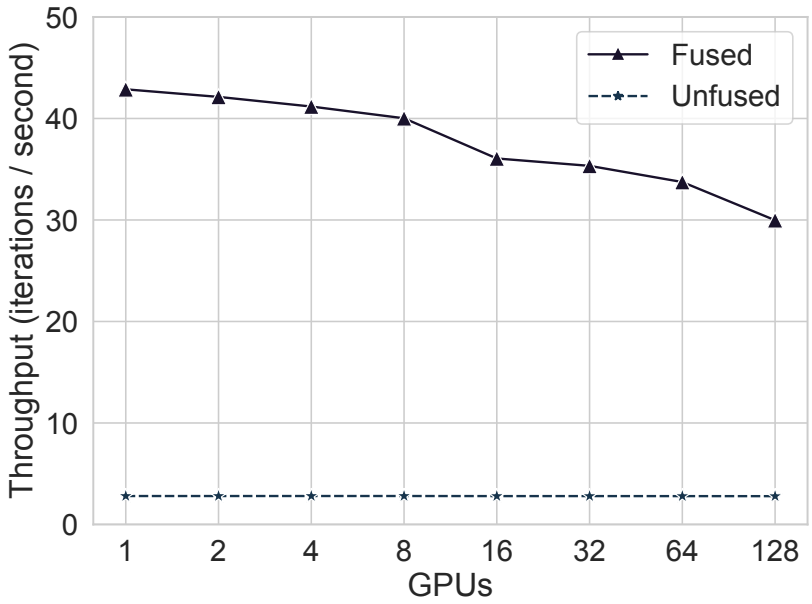


Figure 4.8: Weak scaling of Black-Scholes.

Dense Jacobi Iteration. Unlike Black-Scholes, dense Jacobi iteration has negligible potential benefit from fusion. Jacobi iteration consists of a dense matrix-vector multiplication and two fusible vector operations that are negligible in runtime. This benchmark shows that our analyses do not have a significant negative impact on performance when there is no fusion. Legate achieves 0.93–1.08x of the performance of the unfused Jacobi iteration in Figure 4.9, where we believe the slight improvement is due to experimental variability.

Sparse Krylov Solvers. We evaluate sparse Krylov solvers implemented with cuPyNumeric and Legate Sparse, namely Conjugate Gradient (CG) and Bi-Conjugate Gradient Stabilized (BiCGSTAB). The PETSc benchmark implementations are implemented in MPI+C using PETSc's API. To perform a controlled comparison against PETSc, we modify Legate Sparse to perform a similar optimization as PETSc, where the non-zero coordinates in each sparse matrix partition are stored as 32-bit integers instead of 64-bit integers.[1] We note that this restriction has been lifted in current

[1]PETSc stores coordinates in 32-bit integers even when 64-bit integers are requested at build time, affecting the performance of the SpMV kernel.

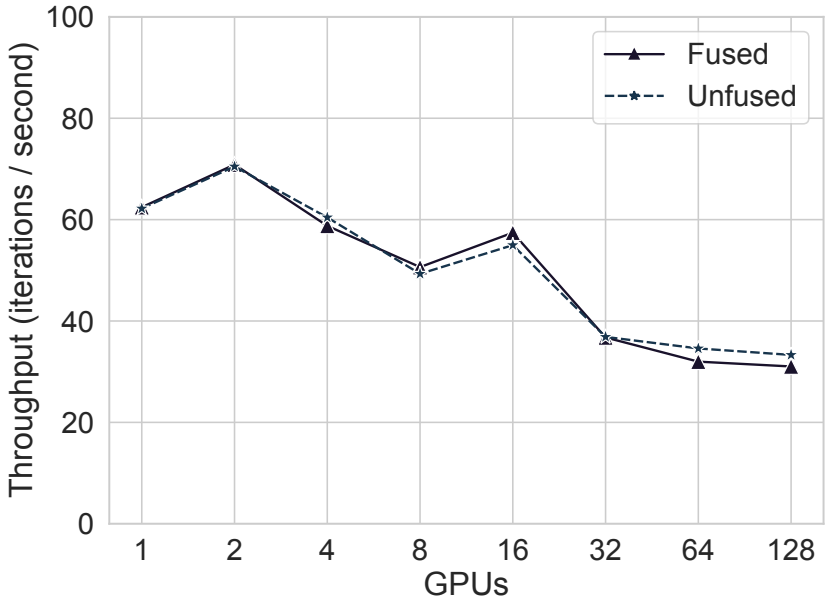


Figure 4.9: Weak scaling of Jacobi Iteration.

versions of PETSc, but was not fixed at the time these experiments were collected.

The original implementation of CG in Legate Sparse (as presented in Chapter 3) had been optimized manually to perform many of the optimizations that Legate does automatically with fusion. These optimizations were performed opaquely in the original implementation, as SciPy Sparse exposes the CG solve through a high-level `linalg.cg` interface. As a result, the internal implementation of `linalg.cg` no longer resembled the high-level description of CG. We compare against this manually fused implementation, a naturally written implementation using cuPyNumeric and Legate Sparse, and PETSc. Figure 4.10 shows that Legate automatically optimizes the naturally written CG so that it runs faster than both the manually optimized version and PETSc. Legate finds additional fusion opportunities by fusing AXPY's and dot-products from different iterations.

We implement an unfused version of BiCGSTAB in cuPyNumeric and Legate Sparse and compare against PETSc. Figure 4.11 shows that Legate accelerates the high-level implementation of BiCGSTAB to outperform the unfused version by 1.31x on average (geo-mean) and PETSc by 1.15x on average (geo-mean). PETSc exposes

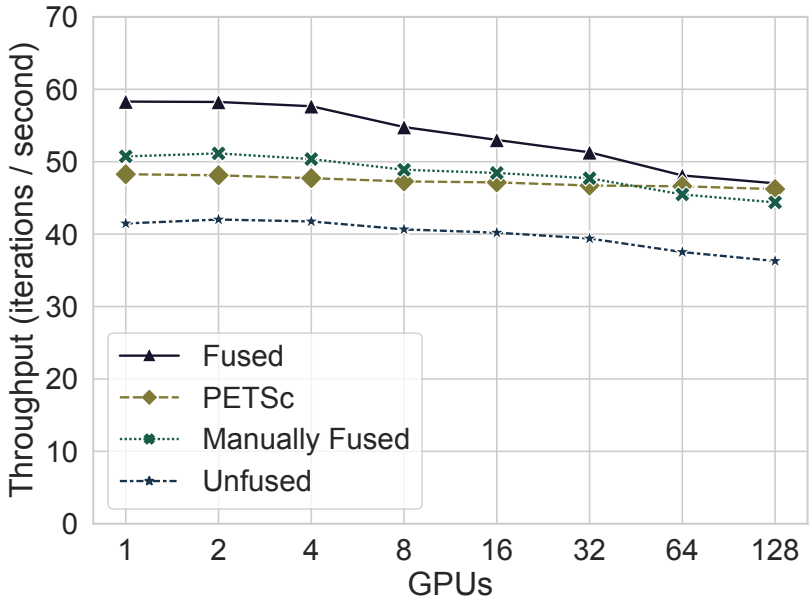


Figure 4.10: Weak scaling of CG.

several fused kernels to users for use in building iterative solvers, but these kernels can quickly become complicated and esoteric[2] In contrast, Legate enables users to write high-level programs in cuPyNumeric and Legate Sparse and then derives optimized kernels for efficient execution.

Geometric Multi-Grid Solver (GMG). Moving from smaller benchmarks to full applications, we apply Legate's fusion to a Geometric Multi-Grid (GMG) solver developed in Legate Sparse. The GMG solver is a CG-based iterative solver with a V-cycling preconditioner, the injection restriction operator, and a weighted Jacobi smoother. As with the previous benchmarks, using Legate with the more complex solver required no changes to user-facing code, and results in a 1.2x speedup over the original implementation, as seen in Figure 4.12. Additionally, unrelated performance challenges within the Legion runtime system were resolved since the original development of Legate Sparse, yielding better weak scaling than originally presented in Chapter 3.

Computational Fluid Dynamics (CFD). CFD is a cuPyNumeric application that

[2]Such as `VecAXPBYPCZ` in BiCGSTAB [1].

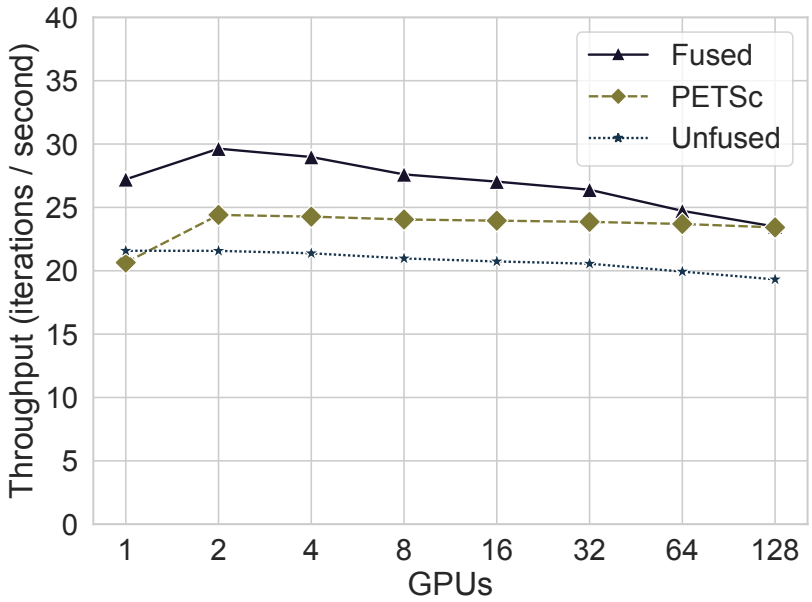


Figure 4.11: Weak scaling of BiCGSTAB.

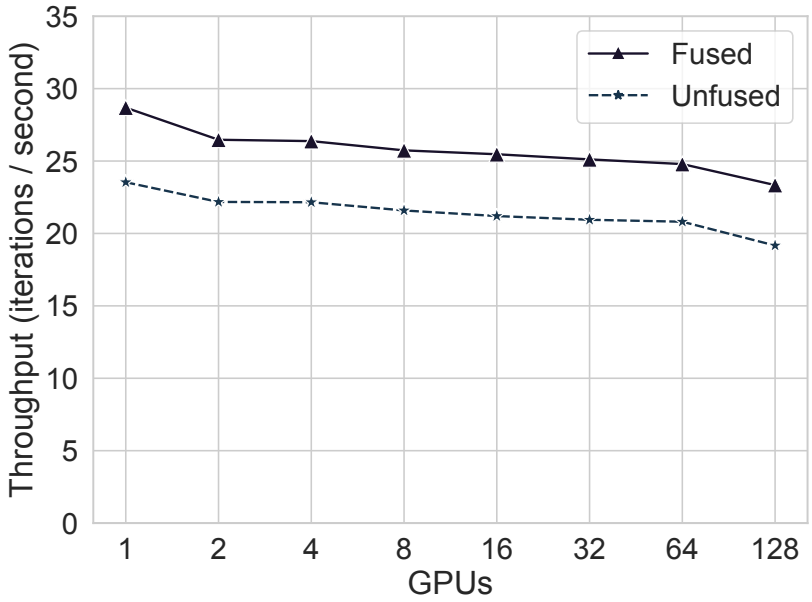


Figure 4.12: Weak scaling of GMG.

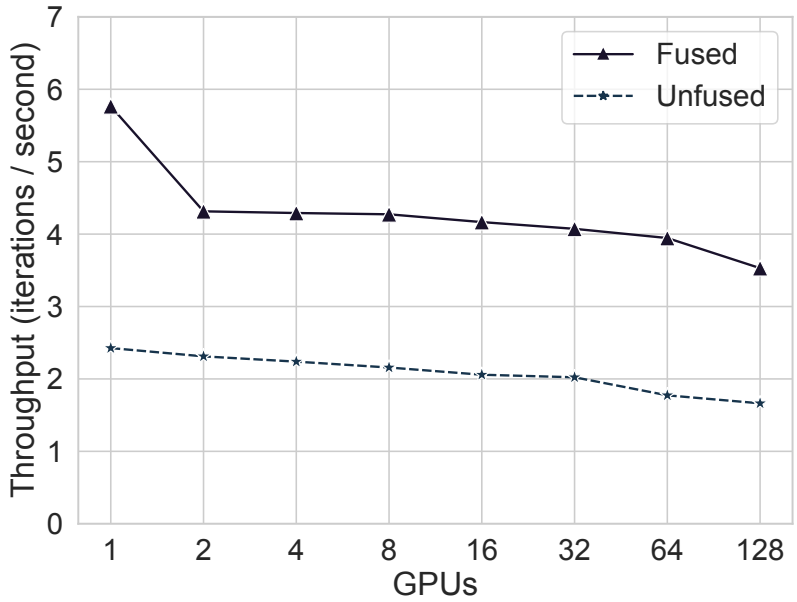


Figure 4.13: Weak scaling of CFD.

solves the Navier-Stokes equations for 2D channel flow [18]. The application performs element-wise operations on aliasing slices of distributed arrays, exposing opportunities for fusion. Legate finds between 1.8x–2.3x speedup over the original implementation, as shown in Figure 4.13. Legate achieves higher speedup on a single GPU than on multiple GPUs. On a single GPU, data is not partitioned, enabling longer sequences of tasks to satisfy fusion constraints. On multiple GPUs, the dependencies caused by aliasing data reduce the opportunities for fusion.

Shallow Water Equation Solver (TorchSWE). Our final benchmark application is also our most complex: the cuPyNumeric port of the TorchSWE shallow-water equation solver [44]. We compare against the original cuPyNumeric port, as well as a version that the cuPyNumeric developers manually optimized using `numpy.vectorize`. The `vectorize` utility JIT-compiles a user-defined element-wise operator, doing some of the optimizations that the Legate middle-end performs automatically. Figure 4.14 shows the performance of TorchSWE with Legate compared to these baselines. Legate achieves a 1.61x speedup on average (geo-mean) over the unfused TorchSWE, and a 1.35x speedup on average (geo-mean) over the manually vectorized version (labeled

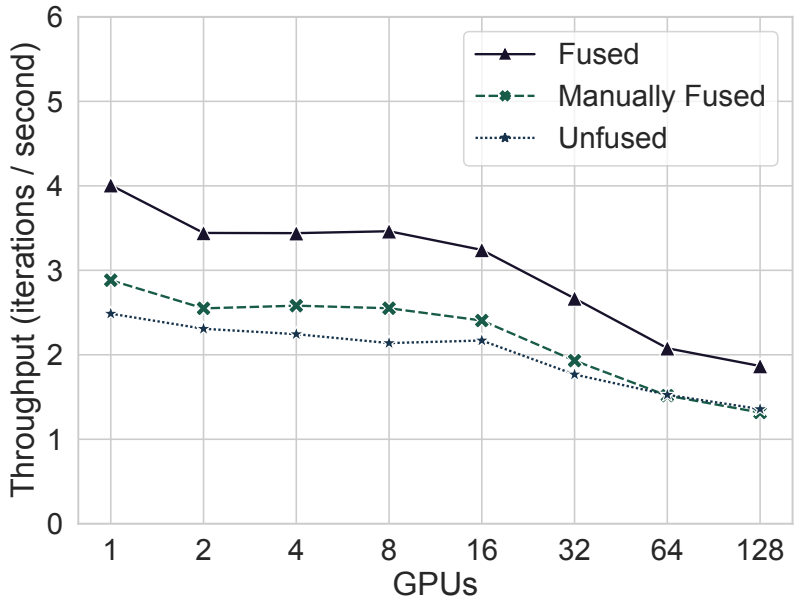


Figure 4.14: Weak scaling of TorchSWE.

with "Manually Fused" in Figure 4.14). Since Legate analyzes the entire application, it finds fusion opportunities missed by developers optimizing the program by hand.

4.5.2 Compilation Time

We measure the overhead that the Legate middle-end's compilation imposes on the overall application runtime. When evaluating our benchmarks, we compute the throughput after warmup iterations have concluded. To measure the effect of compilation, we measure the warmup time with and without compilation, using the window

Benchmark	Standard (s)	Compiled (s)	Breakeven Iterations
Black-Scholes	0.38	0.06	N/A
Jacobi	0.53	0.43	N/A
CG	0.67	1.30	99.44
BiCGSTAB	1.26	2.19	80.43
GMG	0.49	1.38	118.75
CFD	5.10	10.89	25.21
TorchSWE	0.97	8.82	43.88

Table 4.2: Warmup times on 8 GPUs.

sizes reported in Table 4.1. We then compute the number of iterations required for the fused version to be faster than the unfused version of the application when including the warmup compilation time. The results are shown in Table 4.2; Legate's compilation times are modest, requiring 25–119 iterations to amortize the cost of compilation. The fused Black-Scholes computation is so much faster than the unfused version that a single iteration is sufficient to amortize compilation. For Jacobi, compilation time was overlapped with expensive dense matrix-vector multiply kernel, and thus not exposed in the warmup. As seen in Figure 4.9, due to experimental variation, the fused and unfused versions of Jacobi are slightly faster or slower than each other on different GPU counts. These costs are especially reasonable as scientific applications like the ones we evaluated would be run in production for thousands to millions of iterations. In the future, a production-grade implementation of task and kernel fusion in Legate could maintain a cache of compiled kernels on disk, rather than in memory, and pay the compilation cost only the first time the application is run.

4.6 Conclusion

This chapter discussed Legate's middle-end program representation and analyses, which enable the computation of independent libraries and functions within a library to efficiently compose through fusion. Legate separates the problem of fusion into task fusion, which identifies when communication is not required between computations, and kernel fusion, which then fuses the nested loops within application computations. By factoring the fusion problem in this way, Legate is able to find fusion opportunities in real-world programs with dynamic behavior and complex aliasing patterns, in contrast to brittle static analyses. We showed how Legate's fusion analysis can derive optimizations manually implemented by humans, and can find additional optimizations that are only visible through dynamic inspection of the target program.

Chapter 5

Controlling Overheads (Dependence Analysis)

The previous chapters of this thesis focused on the intermediate layers within the Legate runtime system that enable the composition of distributed data and computation across library boundaries. These layers, along with Legion and Realm beneath Legate, perform a variety of dynamic analyses to co-partition stores, fuse computations, perform dependence analysis, and schedule tasks from the source program. While these dynamic analyses are critical for the composed programs to achieve high performance, the cost of performing these dynamic analyses can impact the end-to-end performance of applications, especially as the task granularity of applications decreases. Controlling the impact of these overheads, especially in programs composed from independent pieces, is important to yield end-to-end performance at a variety of target problem sizes. This chapter and Chapter 6 discuss two key techniques in the Legate runtime system to control these overheads. This chapter focuses on amortizing dynamic analysis overheads caused by parallelism extraction and communication insertion in the Legion runtime system automatically, without user input. Legion's analysis is significantly more heavyweight than the corresponding analysis in Legate, and also can scale with the size of the target machine, which is why we focus specifically on analysis amortization within Legion. Then, Chapter 6 focuses on the bottom layer of the Legate runtime stack, tackling overheads at the Realm layer.

Prior Work Declaration. This chapter is based on material in the publication "Automatic Tracing in Task-Based Runtime Systems" [138]. This chapter references the Legion runtime system, which is prior work and detailed in Mike Bauer's thesis [20]. Specifically, the chapter builds on top of Legion's tracing infrastructure, which is also prior work and detailed by Lee et al. [87].

5.1 Background and Motivation

As discussed in Chapter 2 and Chapter 3, Legion exposes an implicitly-parallel task-based programming model. Programs are expressed as a sequential stream of tasks, where tasks describe which stores they access and how the stores will be accessed (with privileges). Legion then performs a *dependence analysis* [25, 22, 24] to identify which tasks depend on each other, extract parallelism from the input sequential stream of tasks, and insert communication between tasks to ensure that tasks observe data consistent with the sequential program interpretation. Chapter 3 describes how this dependence analysis is a critical component for the efficient composition of independent Legate libraries.

Legion's dependence analysis, is the most expensive dynamic analysis performed in the entire Legate runtime stack, which correlates with the level of sophistication within the analysis. This analysis costs roughly 1 millisecond per task in the source program; as tasks drop below this threshold, which is becoming more common with higher performance GPUs, the analysis is unable to be amortized and dominates the performance of the application. To improve the performance, Lee et al. [87] introduced *dynamic tracing*, a memoization framework for Legion's dependence analysis. Tracing records the results of the dependence analysis for an issued sequence of tasks, and then replays the results of the analysis the next time an identical sequence of tasks is issued. Tracing has been shown to yield significant speedups by eliminating the cost of the dependence analysis on iterative programs, and reducing the overheads imposed by Legion to roughly 100 microseconds per task.

A significant limitation of tracing is that it requires the programmer to annotate

repeatedly issued program fragments with stop/start markers for the runtime system. Explicit stop/start markers derail the correctness and performance of Legate programs (and directly in Legion) under composition. The correct placement of trace annotations when composing complex software becomes unclear. Functions defined in independent Legate libraries may contain operations that cannot be traced by a practical tracing implementation, or may issue a different sequence of tasks on each invocation. Each of these cases result in errors, due to the incorrect trace annotations constructing an ill-formed sequence of operations. Furthermore, even simple programs constructed from an individual Legate library can have traces that do not correspond to syntactic loop structures in the source program, making it difficult to correctly place tracing annotations, and thus difficult to take advantage of the performance benefits of tracing. We elaborate on such an example program later in Section 5.1.1.

To both improve programmer productivity and to enable the tracing of high-level programs composed from independent Legate libraries, we argue that implicitly-parallel task-based systems like Legion should automatically identify repeated sequences of tasks, memoize their dependence analysis results, and cheaply replay the analysis as needed. We call this the problem of *automatic tracing*, which is similar to Just-In-Time (JIT) compilation in the context of dynamic language implementations [70, 61, 97]. JIT compilers for dynamic languages interpret bytecode during program startup, and compile bytecode to native instructions as repeatedly invoked program fragments become hot. Following this architecture, Legion should interpret issued operations with a dynamic dependence analysis, and switch to an analysis-free compiled execution once repeated sequences of operations are encountered.

The key challenge of automatic tracing is the *identification* of repeated sequences of tasks produced by the source program. Unlike JIT compilers, the input to Legion is a stream of tasks that lacks information about control flow such as basic block labels or function definitions. As such, Legion cannot rely on these code landmarks or predictable execution flow to identify repeated sequences of operations.

```python
import cupynumeric as np
# Generate random system.
A = np.random.rand(N,N)
b = np.random.rand(N)
# Initialize solution and
# extract diagonal.
x = np.zeros(A.shape[1])
d = np.diag(A)
R = A - np.diag(d)
# Jacobi iteration.
for i in range(iters):
  x = (b - np.dot(R, x)) / d
```

(a) Python source code.

```
DOT(R, x1, t1)
SUB(b, t1, t2)
DIV(t2, d, x2) # Iteration 1
DOT(R, x2, t1)
SUB(b, t1, t2)
DIV(t2, d, x1) # Iteration 2
DOT(R, x1, t1)
SUB(b, t1, t2)
DIV(t2, d, x2) # Iteration 3
DOT(R, x2, t1)
SUB(b, t1, t2)
DIV(t2, d, x1) # Iteration 4
```

(b) Main loop task stream.

Figure 5.1: Example cuPyNumeric program and the stream of tasks issued to the Legate (and then Legion) runtime. An intuitive trace around the main loop does not correspond to a repeated program fragment.

5.1.1 Motivating Automatic Tracing

We now describe how high-level, implicitly-parallel Legate programs can be difficult to add manual tracing annotations to, due to layers of abstraction over the concrete stream of tasks. These challenges arise as a result of the same properties that enable Legate programs to compose in the first place: the Legate runtime reduces operations from independent libraries into a single coherent task stream that can be analyzed and optimized. However, once the program has entered this flattened representation, recovering structure around looping patterns is a challenge.

Figure 5.1a is a small cuPyNumeric program that performs Jacobi iteration. The corresponding stream of tasks issued by the main loop is shown in Figure 5.1b. As before, each cuPyNumeric array maps directly to a Legate store. For each task, we elide privilege annotations on store arguments, and let the first two arguments denote the source-level inputs, while the third argument is the source-level output.

To trace a program, the programmer must issue a `tbegin(id)` call (standing for "trace begin") before and a `tend(id)` call after the fragment. The first time Legion executes a trace with a particular `id`, it records the results of the dependence analysis, and then replays the results when executing the same trace `id` again [87]. For a trace to be valid, the sequence of tasks and their store arguments encapsulated by `tbegin(id)` and `tend(id)` calls must be exactly the same for a given `id`. The same store arguments must be used across trace invocations as the dependence analysis is

affected by the usages of the stores and how they are partitioned.

A natural attempt to trace the program in Figure 5.1a would place the `tbegin` and `tend` around the body of the main `for` loop. However, this annotation results in an invalid trace, due to an implementation detail within cuPyNumeric and the Legate runtime. The problem with this natural annotation is the loop-carried use of the Python variable `x`, which is bound to different cuPyNumeric arrays (stores) at different points of execution. Upon entering loop iteration i, `x` is bound to a store arbitrarily named `x1`, which is used as an argument for the first `dot` operation. As execution proceeds, cuPyNumeric allocates a new store `x2` for the result of the division with `d`, and binds the Python variable `x` to the region `x2`. Therefore, the next iteration $i + 1$ issues a `dot` on `x2`, causing iteration $i + 1$ to issue a different sequence of tasks than iteration i! Issuing a different sequence of tasks with the same trace `id` is a violation of the conditions to use tracing, and Legion may either choose to raise an error or fall back to the expensive dependence analysis.

To correctly trace the program in Figure 5.1a, a programmer must either add trace annotations around every two iterations of the main loop, or use two different trace IDs for each different iteration's repetition pattern. This steady state of groups of two iterations is achieved because when `x` is assigned, the store it refers to can be collected by Legate and immediately returned for reuse by cuPyNumeric. Relying on this steady state is brittle, as the addition of more operations in the main loop or a change in Legate store recycling policy and cuPyNumeric's mapping of stores to NumPy arrays could perturb the way in which the necessary steady state for tracing is achieved. Instead, we argue that a dynamic analysis should analyze the application stream of tasks and automatically discover what fragments of the application should be traced, removing this concern from the programmer.

5.2 What Are Good Traces?

By identifying traces automatically, we aim to reduce the amount of time Legion spends performing dynamic dependence analysis. To maximally reduce this dynamic dependence analysis cost, we must define what are the properties of traces that Legion

should automatically identify.

A simple cost model of Legion's dynamic dependence analysis is that Legion spends time α analyzing each task. The first time a trace is issued, the dependence analysis results are memoized, so Legion spends time α_m (memoization time) on each task in the trace, where α_m is slightly larger than α. Then, on subsequent executions of the trace, there is some constant c amount of overhead to replaying the trace, but every task in the trace only incurs an analysis cost of α_r (replaying time), where $\alpha_r \ll \alpha$.

Using this model of Legion, we derive several properties of traces that we aim to automatically discover. First, the selected traces should maximize the number of traced operations to minimize the number of tasks that contribute an α to the overall analysis cost. Next, the selected traces should be relatively long so that the constant replay cost c does not accumulate. Finally, the set of selected traces should be small, so that Legion does not continually memoize new traces and pay α_m per task in each new trace. Intuitively, the ideal set of traces corresponds to the loops in the target program.

We now concretize the good traces that Legion should find as the solutions of a concrete optimization problem. Consider the sequence of tasks S constructed from a complete execution of the target program. A system for automatic trace identification must construct from S

- A set of traces T, containing sub-strings of S,
- A function $f : T \to \mathsf{interval\ set}$, mapping each $t \in T$ to a set of intervals in S that are *matched* by t,

that maximizes the *coverage* of f, defined by $\mathsf{coverage}(T, f) = \sum_{t \in T} \sum_{i \in f(t)} |i|$, subject to the constraints

1. $\forall t \in T$, t is longer than a minimum length,
2. $\bigcup_{t \in T} f(t)$ is a disjoint set of intervals.

Multiple solutions exist for this problem, so we prefer solutions that first maximize the number of matched intervals ($\sum_{t \in T} |f(t)|$), and then minimize the total number

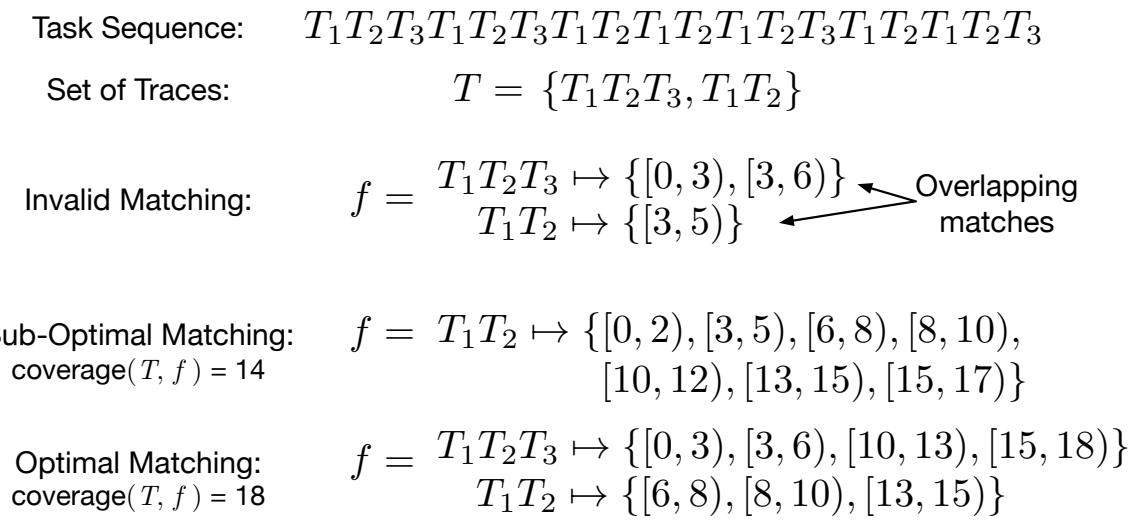


Figure 5.2: Example of a task stream and fixed trace set T with an invalid matching function f, and two matching functions with different coverage(T, f).

of selected traces ($|T|$). Maximizing coverage(T, f) directly minimizes the number of untraced tasks, and selecting a small set of traces that repeats many times minimizes the memoization cost of α_m per task. Finally, a minimum length is placed on traces to ensure that the constant replay cost c can be effectively amortized. We present a concrete problem instance and example solutions in Figure 5.2.

The presented optimization problem precisely defines the properties of traces that Legion should attempt to find, but it does not directly yield an algorithm to discover good solutions. Additionally, the optimization problem is structured in a post-hoc formulation, where an optimal solution is constructed from the results of the entire program execution. In practice, Legion must construct the solution (T, f) in an online manner, using the currently visible prefix of the sequence of tasks launched by the application. In the next section, we discuss algorithms for dynamically finding good solutions to this optimization problem through a set of string processing algorithms.

5.3 Trace Identification

Dynamically finding good traces requires processing information about the tasks seen so far, and then using that information to record and replay traces in the future. An

overview of Legion's dynamic analysis procedure for automatic tracing is sketched in Algorithm 2. Legion has two components that correspond to the targets of the optimization problem in Section 5.2. The *trace finder* constructs the candidate set of traces T by accumulating the tasks issued by the application into a buffer, and asynchronously mining the buffer to find candidate traces. The *trace replayer* then constructs the matching function f by ingesting the candidate traces into a trie, and identifying candidate traces in the application stream by maintaining pointers into the trie that represent potential matches. The automatic tracing analysis intercepts calls to the application-exposed `ExecuteTask` function, and forwards a potentially different set of tasks and trace markers to the runtime. A concrete example of how Legion identifies a trace in an application is shown in Figure 5.3. We now describe each of these components in detail.

5.3.1 A Stream of Tokens

An insight is that automatic trace identification is inherently an online string analysis problem of finding repeated sub-sequences in the application's task stream. As seen in Figure 5.1b, the task stream is not just a list of identifiers—tasks have store arguments that must also be the same across iterations to be used in a trace. To capture all aspects of a task that can affect the dependence analysis, Legion constructs a hash from each task and its store arguments. Converting the input stream of tasks into a stream of hash tokens[1] enables more direct application of string processing techniques, and straightforward handling of traceable operations that are not tasks, such as copies and fills.

5.3.2 Finding Traces With High Coverage

Legion's trace finder records tasks as they are issued by the application into a buffer (we describe a refinement to this scheme in Section 5.3.4). Once the buffer fills up, Legion launches an asynchronous analysis of the buffer to find a set of traces within

[1]Tokens as discussed in the string processing literature as opposed to the term in LLM contexts.

Algorithm 2: Legion's Dynamic Analysis for Automatic Tracing.

```
   /* Initialize token history buffer B and pending async analyses J.  */
1  B, J ← [], []
   /* Initialize trie of candidates C, potential current traces A, and
      pending tasks P.  */
2  C, A, P ← Trie(), [], []
   /* Discussed in Section 5.3.2.  */
3  TraceFinder (H)
4  |  B ← B + [H]
5  |  if ShouldAnalyzeHistory(B) then
   |  |  /* What subset of the history to analyze is discussed in
   |  |     Section 5.3.4.  */
6  |  |  B′ ← GetAnalysisSubset(B)
   |  |  /* Find repeated sub-strings.  */
7  |  |  j ← async FindRepeats(B′)
8  |  |  J ← J + [j]
9  |  |  B ← MaybeClearHistory(B)
   /* Discussed in Section 5.3.3.  */
10 TraceReplayer (T, H)
11 |  if ∃ j ∈ J, j is complete then
12 |  |  IngestCandidates(j, C)
13 |  P ← P + [T]
   |  /* Advance all potential traces by H in the trie if possible.  Remove
   |     impossible traces, and extract fully matched candidates.  */
14 |  A ← AdvanceActiveCandidates(C, A, H)
15 |  A ← FilterInvalidCandidates(C, A)
16 |  D, A ← FilterCompletedCandidates(C, A)
17 |  if |D| > 0 then
   |  |  /* Select one of the pending candidates to replay.  Execute any tasks
   |  |     before it, and issue a trace replay for the candidate.  */
18 |  |  R ← SelectReplayTrace(D, P, A)
19 |  |  P, A ← ExecuteAndReplay(R, P, A)
   /* Applications (or Legate itself) issue tasks through Legion's ExecuteTask
      function.  */
20 ExecuteTask (T)
21 |  H ← Hash(T)
22 |  TraceFinder(H)
23 |  TraceReplayer(T, H)
```

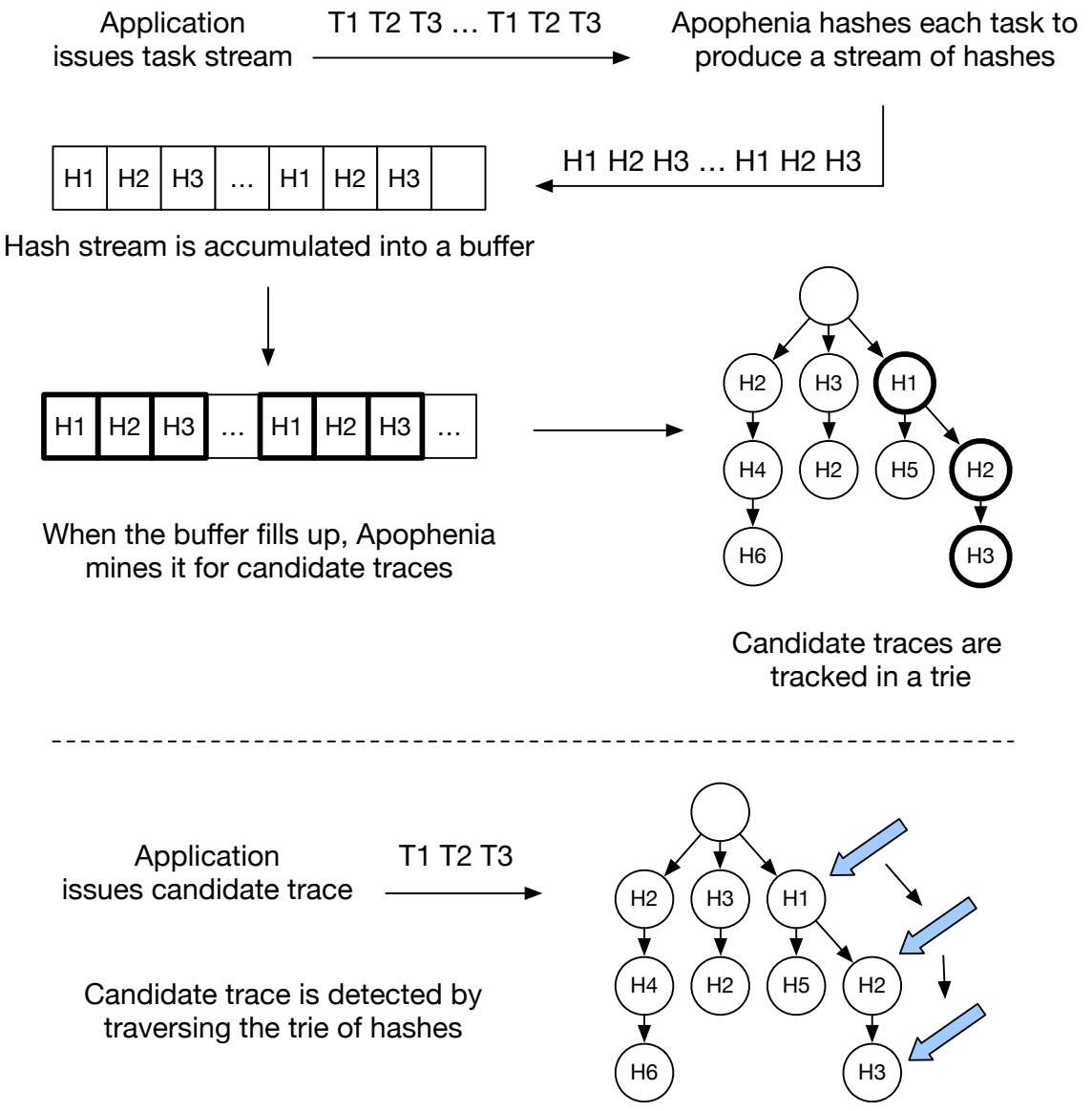


Figure 5.3: Visualization of Legion's automatic tracing analysis.

the buffer that maximize the coverage of the buffer. We discuss previous ideas that are related to this goal, and then describe the solution used in Legion.

Existing Techniques

The Lempel-Ziv family of algorithms use repeated sub-strings for compression. Algorithms like LZ77 [146, 147, 118] maintain a sliding window of previous tokens to search for repeats in when encoding upcoming tokens. The LZW [131] algorithm avoids the use of a sliding window by only increasing the length of any candidate repeat by a single token at a time. While not directly finding a set of repeats with high coverage, similar algorithms that use a sliding window would need to maintain and search in a window the size of the analyzed buffer, resulting in quadratic time complexity. In order to recognize a trace of length n, an LZW-style algorithm would also need to encounter the trace $n-1$ times. We wanted an algorithm that is sub-quadratic in order to scale to large buffer sizes. Real-world applications we discuss in Section 5.6 have traces that contain more than 2000 tasks, requiring token buffers of at least twice that size to detect a single repeat.

The most relevant string processing problem in the bio-informatics community is *motif finding* [48], which is the problem of finding short (5–20 token long), fixed-length repeated strings in a larger corpus. The focus on a short and fixed sub-string length and a tendency to use genomic information to guide the search makes these techniques not applicable to our problem.

Algorithms for document fingerprinting such as Moss [111] have been developed that accurately identify copies between documents. In particular, these techniques are guaranteed to detect if repetitions of at least a minimum size exist across documents. Fingerprinting techniques are useful to detect whether there exist repeated sub-strings, but do not directly aid in finding the sub-strings themselves that have high coverage.

Within the programming languages community, recent work by Sisco et al. [113] used a technique called *tandem repeat analysis* [119] to find loops in the netlists that result from compiling hardware description languages. A tandem repeat is a sub-string α that repeats contiguously within a larger string S, such that α^k is a

sub-string of S, for some k. Despite the success that Sisco et al. had using tandem repeat analysis, we found that even simple real world cuPyNumeric programs did not contain enough tandem repeats for the analysis to reliably identify a trace set with high coverage. The reason is that while these real-world programs tended to have repetitive main loops, there would often be irregularly appearing computations such as convergence checks or statistics calculations that occur infrequently between loop iterations. As such, the strings that represented these programs would not contain tandem repeats, but instead repeated sub-strings separated by other tokens.

A relaxation of tandem repeat analysis is to search for non-overlapping repeated sub-strings, which removes the contiguity requirement on the repeats. Concretely, given the string *ababab*, *abab* is an overlapping repeat, while *ab* is non-overlapping. We could use non-overlapping repeated sub-strings to assemble a set of traces T and a disjoint mapping f that achieves high coverage. While there exist standard suffix-tree algorithms to find repeated sub-strings, we found that the natural extensions of these algorithms to detect non-overlapping repeated sub-strings also resulted in quadratic runtime complexity.

Our Algorithm

We design a repeat finding algorithm that is directly aware of the optimization problem in Section 5.2 and runs in $O(n \log(n))$, where n is the size of the token history buffer. At a high level, our algorithm makes a pass through a suffix array constructed from the input buffer to collect a set of candidate repeats. It then greedily selects the largest repeated sub-strings that do not overlap with any previously chosen sub-strings. Pseudocode for our algorithm is in Algorithm 3[2], which takes a string S and returns a set of sub-strings that achieve high coverage of S. We assume that the reader is knowledgeable about suffix arrays and their structural properties. However, understanding the algorithm in Algorithm 3 is not required to understand its usage in Legion, as discussed in Section 5.3.3 and Section 5.3.4.

As a first step, we construct a suffix array and longest common prefix array from

[2]We also present a standalone implementation of the algorithm available at `https://github.com/david-broman/matching-substrings`.

Algorithm 3: Algorithm to compute non-overlapping repeated sub-strings.

```
1  FindRepeats (S)
2      SA, LCP ← SuffixArray(S)
       /* Candidates are tuples of string length, the repeated sub-string, and
          starting position. */
3      C ← []
4      foreach i ∈ [0, |SA| − 1) do
           /* Extract adjacent suffix array entries and their overlap length.
              */
5          s1, s2, p ← SA[i], SA[i + 1], LCP[i]
6          if [s1 : s1 + p) ∩ [s2 : s2 + p) = ∅ then
               /* S[s1 : s1 + p] and S[s2 : s2 + p] are repeated strings that do not
                  overlap in S, so they are candidates. */
7              r ← S[s1 : s1 + p]
8              C ← C + [(p, r, s1), (p, r, s2)]
9          else
               /* S[s1 : s1 + p] and S[s2 : s2 + p] overlap in S. Assume s2 > s1, the
                  other case is symmetric. In this case, the overlap is a
                  collection of repeats of S[s1 : s1 + d], by the structure of the
                  suffix array. */
10             d ← s2 − s1
               /* Break prefix into two chunks of repeated pieces of
                  S[s1 : s1 + d]. */
11             l ← (p + d)/2
               /* Remove trailing tokens. */
12             l ← l − (l%d)
13             r ← S[s1 : s1 + l]
14             C ← C + [(l, r, s1), (l, r, s1 + l)]
       /* Sort the candidates by decreasing length and by increasing sub-string
          and start position. */
15     Sort(C)
       /* Greedily collect sub-strings that do not overlap with previously
          chosen sub-strings. */
16     I, R ← [], []
17     foreach (l, _, s) ∈ C do
18         if [s, s + l) does not intersect I then
19             I ← I + [[s, s + l)]
20             R ← R + [S[s : s + l]]
21     return R
```

the input buffer of tokens. We then iterate through adjacent pairs of suffixes to construct a set of *candidate repeats*, which are tuples of sub-strings defined by their length, the repeated sub-string, and its starting position in S. These candidates are constructed based on whether or not the shared prefix between adjacent suffix array entries overlap. Once all of the candidates have been constructed, we sort the candidates to greedily select candidates in order of length, and select as many occurrences of a particular sub-string as possible. We only select candidates that do not overlap with any previously selected candidates, and then deduplicate the chosen set of candidates as the result. A sample execution of Algorithm 3 is shown in Figure 5.4.

Our algorithm can be implemented with time complexity $O(n \log(n))$. Linear time algorithms exist for suffix array and LCP array construction [79]. Two candidates are generated for each entry in the suffix array, so sorting the candidates takes $O(n \log(n))$ time. The interval intersection step can be reduced to constant time by leveraging the candidate iteration order, so the entire loop executes in $O(n)$ time. In particular, an array of length $|S|$ can be maintained, and as each candidate is selected, all positions covered by the candidate are marked. Then, as candidates are iterated over in decreasing length and increasing start position order, interval intersection can be checked by checking if the start or end of an interval is marked. Finally, the deduplication can be done by generating a unique ID for each candidate sub-string in the candidate generation phase, and adjusting the candidate representation to be a tuple of length, ID and starting position; using this sort order allows deduplication to be done at each iteration of the candidate selection loop.

Our algorithm aims to find good solutions to the optimization problem in Section 5.2 by identifying long repeated sub-strings and selecting as many as possible that do not overlap with each other. We trade off an optimal solution to the optimization problem to instead find good solutions and maintain a lower asymptotic runtime. There are two such heuristics in our algorithm. First, when adjacent suffix array entries have a repetition, we consider only the maximal length repetition instead of all sub-strings of the repetition. Second, when we select which candidates to keep, we greedily choose the largest candidates instead of performing a bin-packing

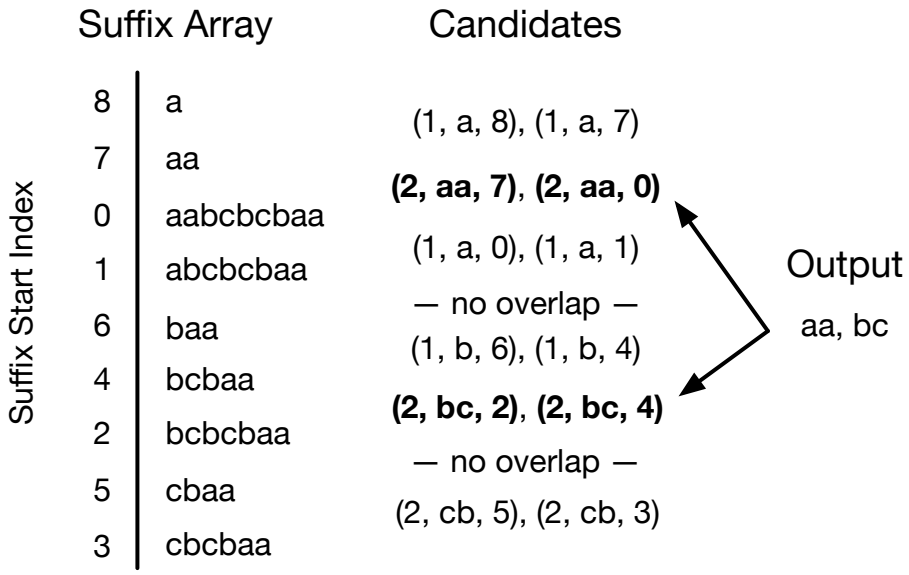


Figure 5.4: Execution of Algorithm 3 on "aabcbcbaa". The candidates for each suffix pair is shown between the pair.

computation. Our algorithm is guaranteed to find the longest repeated sub-string, but due to the second heuristic, we cannot provide theoretical guarantees about the other chosen sub-strings. We show in Section 5.6 that Legion using our algorithm is able to identify good traces in complex, real-world applications.

5.3.3 Recognizing and Replaying Candidate Traces

Legion's trace replayer uses Algorithm 3 to find candidate traces from the application's history of tasks. In this section, we discuss how Legion's trace replayer identifies and selects these candidate traces from the task stream to record and replay. Our design of the trace replayer has two major goals. First, the per-task overhead must be low, as it is imperative for performance for the application to issue as many tasks into the runtime as possible so that the runtime can either replay traces or perform dependence analysis ahead of execution. Slowing down the task launch rate would result in exposed latency from various sources in the runtime. Second, Legion must balance exploration and exploitation when selecting traces. As more information about the application is gained, Legion should switch to better traces as it finds

them. However, Legion should not leave a steady state of replaying a particular trace until it is confident that performance can be improved, as memoization of the dependence analysis for new traces has a cost.

As discussed previously, Legion accumulates a history of tasks launched by the application and asynchronously uses Algorithm 3 to select candidate traces. Asynchronous analysis of task histories is important to avoid stalling the application by waiting for the analysis to finish before accepting the next task from the application.

When an asynchronous analysis completes, Legion ingests the results into a trie that maintains the current set of candidate traces. Along with this trie, Legion maintains a set of pointers into the trie that represent potential matched traces. As tasks are issued, Legion updates the set of pointers by creating new pointers for each new task, stepping any existing pointers down the trie if possible, and removing any pointers that are made invalid. Once a pointer reaches a leaf of the trie and has matched a trace, Legion has the option to record or replay the trace, by wrapping the tasks with `tbegin` and `tend` calls.

Legion uses a scoring function to select which matched trace to replay when faced with multiple valid choices. The scoring function is based on the length of the candidate trace multiplied by a count of the number of times the trace has appeared. In calculation of the score, we impose a maximum value of the count that can be used, and exponentially decay the value of the count by how many tasks have been encountered since the trace last appeared. Finally, we increase the score slightly if a trace has already been replayed.

Our scoring function encodes heuristics about trace selection and aims to balance exploration and exploitation. We naturally prefer long traces over shorter ones, as longer traces have the potential to eliminate more runtime overhead. The capping of the appearance count allows for Legion to eventually switch from a trace that appeared early during program execution to a better trace that appears later in the execution. Next, decaying the appearance count ensures that a seemingly promising trace that occurs infrequently, does not eventually hit a threshold, and disrupts a steady state. Finally, since recording new traces has a cost, when faced with traces of a similar score, we bias Legion towards a trace it has already replayed.

5.3.4 Achieving Responsiveness and Quality

Legion's trace finder accumulates tasks into a buffer and mines the buffer for traces using Algorithm 3. An important question is what should the size of that buffer be? The size of this buffer trades off between responsiveness of Legion's trace identification and the quality of traces Legion is able to find. With a small buffer, Legion can identify traces early but will not be able to identify traces in programs with large loops. Meanwhile, a large buffer allows Legion to identify long traces in complex applications but introduces significant startup delay in smaller applications.

We did not want end users (especially non-expert users of high-level Legate libraries) to be required to continually adjust the buffer size parameter as their application changes. As such, some strategy to adapt the buffer size along this tradeoff space is necessary. We found that a strategy that attempts to dynamically resize the buffer based on what traces to find is unsatisfactory, as the system is unable to differentiate between an application currently not repeating operations versus an application repeating a sequence of operations larger than the buffer size. Instead, we propose a strategy that selects a large fixed buffer size upfront, and then samples smaller pieces of the buffer in a principled manner to be responsive to the occurrence of short traces.

Legion samples from the buffer guided by the *ruler function* sequence [134], which provides a practically useful sampling strategy with provable guarantees. The ruler function counts the number of times a number can be evenly divided by two. Applying it to the sequence $1, 2, 3, 4, \ldots$ yields the sequence $0, 1, 0, 2, \ldots$. Raising the sequence to the power two yields $1, 2, 1, 4, \ldots$, which we can interpret as subsets of the buffer to analyze. For example, with a buffer size of four, as tasks arrive Legion would first analyze the first task, then the first two tasks, then the third task, and finally all four tasks. A visualization of this sampling policy is in Figure 5.5. This sampling policy lets Legion quickly react to changes in the application by analyzing recent pieces of the buffer while allowing larger traces to be found by infrequently analyzing longer components of the buffer. For example, sampling the full buffer in Figure 5.5 is required to find a trace that repeats in positions H2-H4 and H5-H7. In practice, we use the exponentiated ruler function as the multiples of a larger constant (such

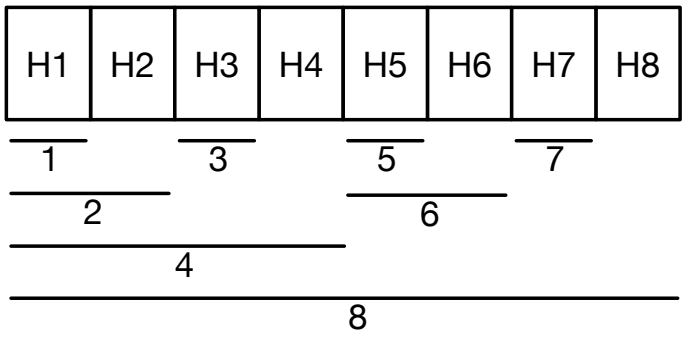


Figure 5.5: Visualization of Legion's buffer sampling strategy on a buffer of size 8. After processing the i'th task, Legion mines the buffer slice labeled i.

as 250) to sample the buffer with. Finally, given that our algorithm in Section 5.3 runs in $O(n \log(n))$, we show that our sampling strategy increases the total runtime complexity of processing the buffer by only an extra log factor, yielding a total of $O(n \log^2(n))$. This technique enables all of the applications we evaluate (Section 5.6) to be run with the same buffer size configuration parameter.

5.4 Automatic Tracing Implementation Concerns

We now discuss important aspects of a realistic implementation of automatic tracing in Legion. In particular, we discuss the specifics of implementing automatic tracing in a distributed context, a conscious decision not to perform speculation when replaying traces, and the necessary extensions Legion's underlying tracing engine.

5.4.1 Distributing the Analysis

The automatic tracing analysis as presented in Section 5.3 is sequential, processing tasks as they are issued by the application or the higher-level Legate runtime system. In a distributed setting, automatic tracing leverages the interaction with Legion's *dynamic control replication* [23] to maintain acting as a sequential analysis, except for one component, which we discuss next. With control replication, the application executes on each node and Legion shards the dependence analysis and execution across nodes. The main restriction of control replication is that the application must

issue the same sequence of tasks on every node. Our implementation of automatic tracing is a layer between the application and the rest of the Legion runtime system, meaning that standard calls into the Legion runtime are intercepted, and the automatic tracing infrastructure forwards a (possibly different) set of calls into the rest of Legion. Therefore, the automatic tracing analysis inherits the control replication requirements of the application. In particular, each node must agree on which traces to replay and when during program execution to record and replay the traces.

The only source of non-determinism in the automatic tracing analysis that could cause control divergence between nodes is the asynchronous processing of token buffers described in Section 5.3.2. An instance of Legion (and the automatic tracing analysis) exists on each node of the target machine, and each instance maintains a local history buffer of tasks to run asynchronous analyses on. Each issued asynchronous analysis may complete earlier on one node than another, resulting in that node replaying a trace before another node has identified that trace as a candidate. However, making the analysis synchronous would result in stalling the application until analyses complete. We resolve this tension by having each node agree on a count of processed operations to issue before ingesting the results of an asynchronous analysis. If any node had to wait on an asynchronous analysis to complete, all nodes increase their count of operations to wait on for the next analysis. This strategy reaches a steady state where analysis results are ingested in a deterministic manner without stalling the application.

5.4.2 (The Lack of) Speculation

Speculation is a common technique in computer architecture to efficiently execute programs with data-dependent control flow. As automatic tracing in Legion has similarities to speculative components in architecture like trace caches [108], a natural design decision was if Legion should speculate on whether traces would be issued by the application. Concretely, if Legion decided to replay the trace $T_1T_2T_3T_1T_2T_3$ and has seen the tasks $T_1T_2T_3$ so far, should it wait until the second $T_1T_2T_3$ has been issued, or speculatively issue the trace $T_1T_2T_3T_1T_2T_3$ and roll back if the next three issued

tasks differ from $T_1T_2T_3$? Our implementation of automatic tracing in Legion does not speculate and waits for the entirety of a trace to arrive before issuing the trace to the rest of the runtime system's analysis. The relative costs of different operations within the Legion runtime system made the potential upside of speculation not worth the implementation complexity.

Legion employs a pipelined architecture where a task flows through three stages: 1) the application phase, where the task is launched (into the automatic tracing analysis), 2) the dependence analysis phase, where the task is analyzed or replayed as part of a trace, and 3) the execution phase, where the task is executed. Depending on the cost ratio of the application and analysis phases, speculation may be beneficial as Legion waits for an entire trace to pass through the application phase. Legion's analysis phase is an order of magnitude more expensive than the application phase, letting the application phase run far ahead of the analysis phase. Thus, waiting for an entire trace to be issued by the application rarely stalls the pipeline and gets exposed in the overall runtime. Therefore, we concluded that designing a trace prediction algorithm and implementing a backup-rollback-recover scheme on speculation failures was not worth the complexity.

5.4.3 Non-Idempotent Traces

An important improvement to Legion that was needed for automatic tracing was support for *non-idempotent traces*; previously, Legion only supported *idempotent* traces. A trace is idempotent when its *preconditions* imply its *postconditions*. The preconditions of a trace capture the state of the dependence analysis when the trace was recorded, and the postconditions capture the state of the dependence analysis after the trace is executed. Intuitively, a trace may only be replayed if its preconditions are satisfied, i.e. at the start of the trace the dependence analysis is in the same state as when the trace was recorded.[3]

The impact of idempotency is that for back-to-back replays of an idempotent trace, the trace's preconditions are known to hold and do not need to be verified. For

[3]We refer readers to Lee et al. [87] for more information about trace conditions and idempotency.

several technical reasons such as implementation complexity and the costs of checking trace preconditions, Legion only supported idempotent traces before our work. We found that it was necessary to relax this restriction and support non-idempotent traces. A consequence of modeling trace identification as a string analysis problem is that the conversion of task sequences into strings loses information about the pre- and post-conditions of a trace. As such, it is difficult to bias the string algorithms towards sub-strings that have semantic properties such as idempotency. Supporting non-idempotent traces involves recording multiple instances of the trace for each set of preconditions it may have (as the postconditions do not imply the preconditions), selecting the instance that is applicable in the dependence analysis state when replay is requested, and eagerly applying the postconditions to the dependence analysis state after replay.

5.5 Alternatives to Automatic Tracing

Having described automatic tracing and some important implementation details, an astute reader may wonder if alternative approaches may exist. I discuss in this section some alternatives and common suggestions that are raised after presenting the sample program in Section 5.1.1.

First, the memoization algorithm for reusing compiled fused tasks presented in Section 4.3.2 based on De-Bruijn index renaming is not applicable to *detect* repeated sequences of tasks. The process of variable renaming to canonicalize a sequence of tasks enables detection of isomorphic sequences only once a candidate sequence has been extracted; the variables are renamed from the "starting point" of the beginning of the sequence. A matching algorithm such as the one described is required to first identify where these "starting points" may be, before a renaming algorithm could be used to detect isomorphisms.

Various point solutions can be proposed to handle the exact issue described in Section 5.1.1, where a trace is requested to be replayed with the same tasks but different store arguments. For example, alternative implementations may stop raising a runtime error for such replays, or record multiple copies of a trace and replay the

correct one based on the store arguments. Aside from the optimizations that a runtime can employ by starting work and analysis at the point that a `begin_trace` operation is received, these solutions can result in unpredictable performance (whether or not a trace is replayed at all) and do not address other potential differences between traces that abstraction layers may introduce.

The most compelling of these point solutions is to enable Legion to replay *isomorphic* traces. Isomorphic traces would be two traces with the same sequence of tasks but store arguments that are the same up to a consistent renaming, i.e. the traces have the same pattern of store usage. Such a capability would naturally handle the issue described in Section 5.1.1. In addition to the same argument made previously (that this addresses a point issue), we believe that there are serious implementation challenges to supporting isomorphic trace replay. There are at least two challenges we encountered when developing a plan to implement isomorphic trace replay. The first is that while the dependencies and copy patterns within a trace can be reproduced in an isomorphic trace, the data movement and synchronization required at the boundaries of the trace are underspecified under the isomorphism. We do not believe that there is enough information (at least in the current structure of Legion tracing) to identify the proper mapping from data in the trace under the isomorphism to the data outside the trace. The second challenge is similar, where the isomorphism does not provide enough information about how the trace side-effects computations outside of the trace. It would be unsafe for the isomorphism to be applied within the trace and modify data within the trace that is referenced from outside the trace and cannot be modified.

We believe that automatic tracing is a more general solution to the problem of applying tracing to applications developed from the composition of multiple modules. It lifts the burden of reasoning about tracing from the programmer to the runtime system and naturally supports how layers of abstraction may perturb the concrete task stream issued to Legion.

5.6 Evaluation

5.6.1 Experimental Setup

We evaluate our implementation of automatic tracing on the largest and most complex Legion applications written to date, including production scientific simulations (some Legion-only and others in Legate) and a distributed deep learning framework. We perform a mixture of weak-scaling (Section 5.6.2) and strong-scaling (Section 5.6.3) experiments and show that our implementation of automatic tracing is able to match the performance of manually traced code when trace annotations already exist, and that Legion can identify traces in programs that were previously not traced. We then evaluate the overhead that automatic tracing imposes on Legion applications (Section 5.6.4) and visualize Legion's search process (Section 5.6.5).

We evaluated our implementation of automatic tracing on the Eos and Perlmutter supercomputers. Each node of Eos is an NVIDIA DGX H100, containing 8 H100 GPUs with 80 GB of memory and a 112 core Intel Xeon Platinum. Each node of Perlmutter contains 4 NVIDIA A100 GPUs with 40 GB of memory and a 64 core AMD EPYC 7763. Nodes of Eos are connected with an Infiniband interconnect, while Perlmutter uses a Slingshot interconnect. We compile Legion on Eos with the UCX networking module, and use the GASNet-EX [31] networking module on Perlmutter. We do not execute each application on both Perlmutter and Eos due to differences between the local environments on each machine. In our experiments, we evaluate the relative performance differences between traced and untraced programs, and comparisons between machines are not significant.

5.6.2 Weak Scaling

In this section, we discuss weak scaling results of applications using automatic tracing. In a weak scaling study, we increase the problem size as the size of the target machine grows to keep the problem size per processor constant. For each application, we perform a sweep over different sizes of the problem to vary the task granularity, thus affecting the impact of runtime overhead. These different problem sizes are denoted in

the graph by the "-s", "-m" and "-l" label suffixes which stand for small, medium and large. At smaller problem sizes, more runtime overhead can be exposed, while larger problem sizes make it easier to hide runtime overhead. In each weak-scaling plot, we report the steady-state throughput of each configuration and problem size after a number of warmup iterations (discussed in Section 5.6.4). We report throughput in iterations per second achieved by each configuration, so within a particular problem size, higher is better; across problem sizes, the smaller problem sizes will achieve a higher iterations per second than the larger problem sizes.

S3D

S3D [123] is a production combustion chemistry simulation code that has been developed over the course of many years by different scientists and engineers. The Legion port of S3D implements the right-hand-side function of the Runge-Kutta scheme, and interoperates with the legacy Fortran+MPI driver of the simulation. The integration between Legion and the legacy Fortran+MPI code leads to various constraints that the manual trace annotations interact with. For example, during the first 10 iterations, a hand-off between Legion and Fortran+MPI must occur every iteration, while after the first 10 iterations a hand-off is required only every 10 iterations. While not unmanageable, these interactions have led to relatively complicated logic to manually trace the main loop. We scale S3D on Perlmutter, and compare the performance of automatic tracing to manually traced and untraced versions of S3D. The results are shown in Figure 5.6. Even on a single node, tracing has a noticeable performance impact on the smaller problem sizes and affects the scalability of S3D. Legion with automatic tracing achieves within 0.92x–1.03x of the performance of the manually traced version, and between 0.98x–1.82x speedups over the untraced version. Manual annotations can slightly outperform the automatically collected traces by leveraging application knowledge to select traces that have lower replay overhead.

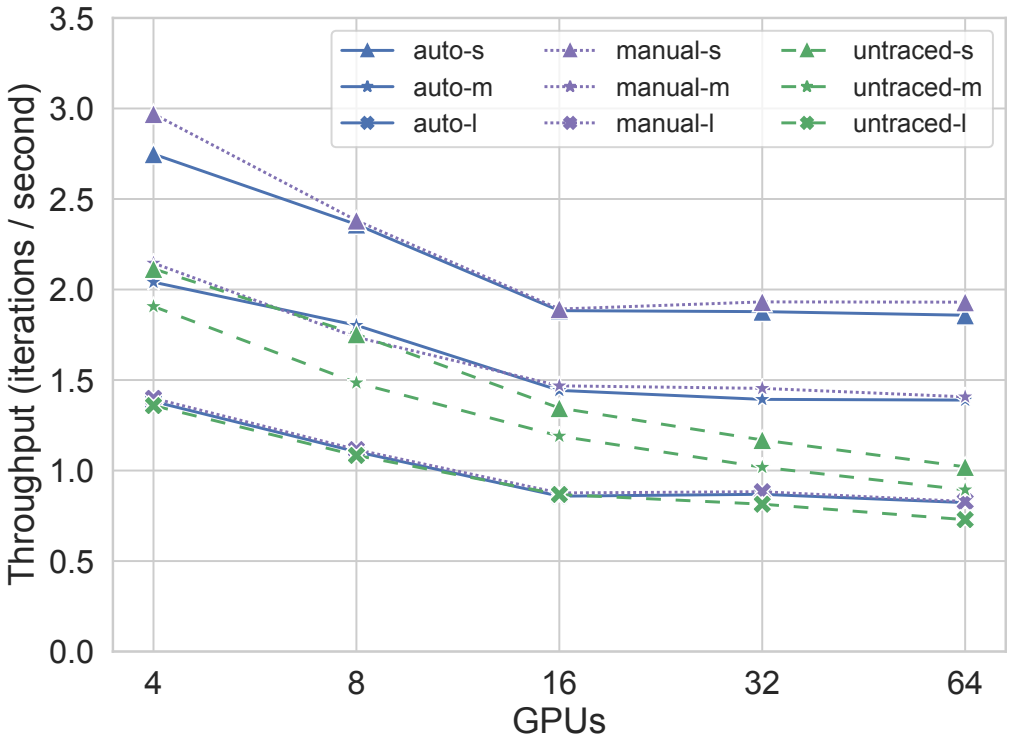


Figure 5.6: Weak scaling of S3D on Perlmutter. Legion with automatic tracing ("auto") performs competitively with hand-annotated traces.

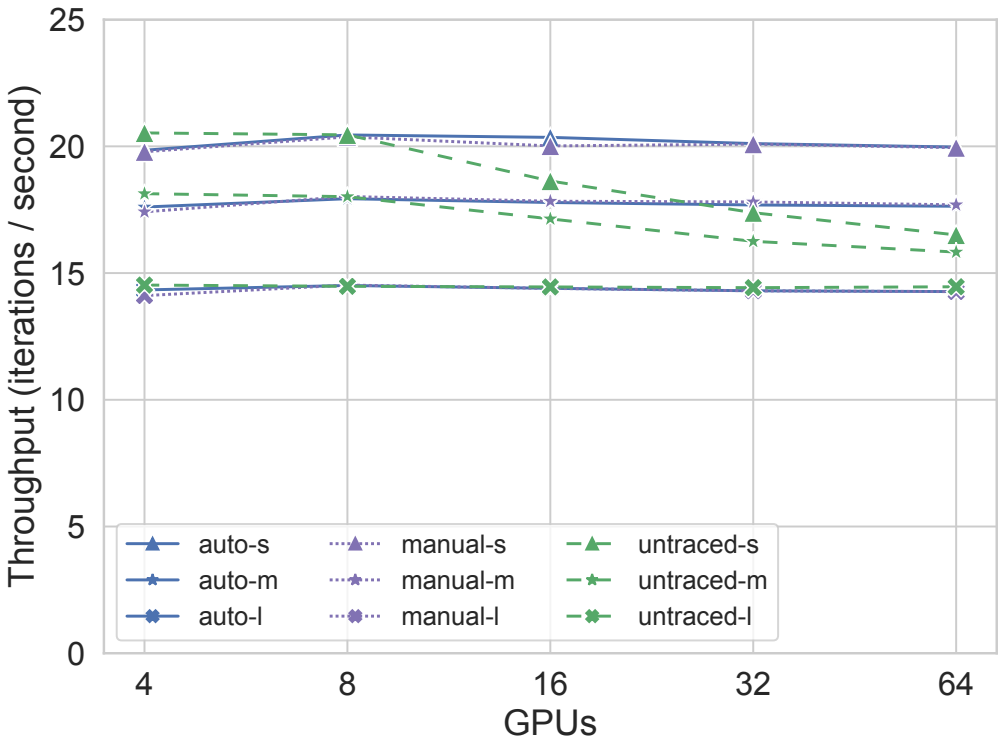


Figure 5.7: Weak scaling of HTR on Perlmutter. Legion with automatic tracing ("auto") performs competitively with hand-annotated traces.

HTR

HTR [55] is a production hypersonic aerothermodynamics application. HTR performs multi-physics simulations of hypersonic flows at high enthalpies and Mach numbers, such as for simulations of the reentry of spacecraft into the atmosphere. Like S3D, we evaluate the performance of Legion with automatic tracing on HTR on Perlmutter, and compare it against a manually traced version and an untraced version. While HTR without tracing performs competitively to the traced version at small GPU counts, Figure 5.7 shows that tracing is necessary for performance at scale. Legion with automatic tracing achieves within 0.99x–1.01x of the performance of the manually traced version, and between 0.96x–1.21x speedups over the untraced version.

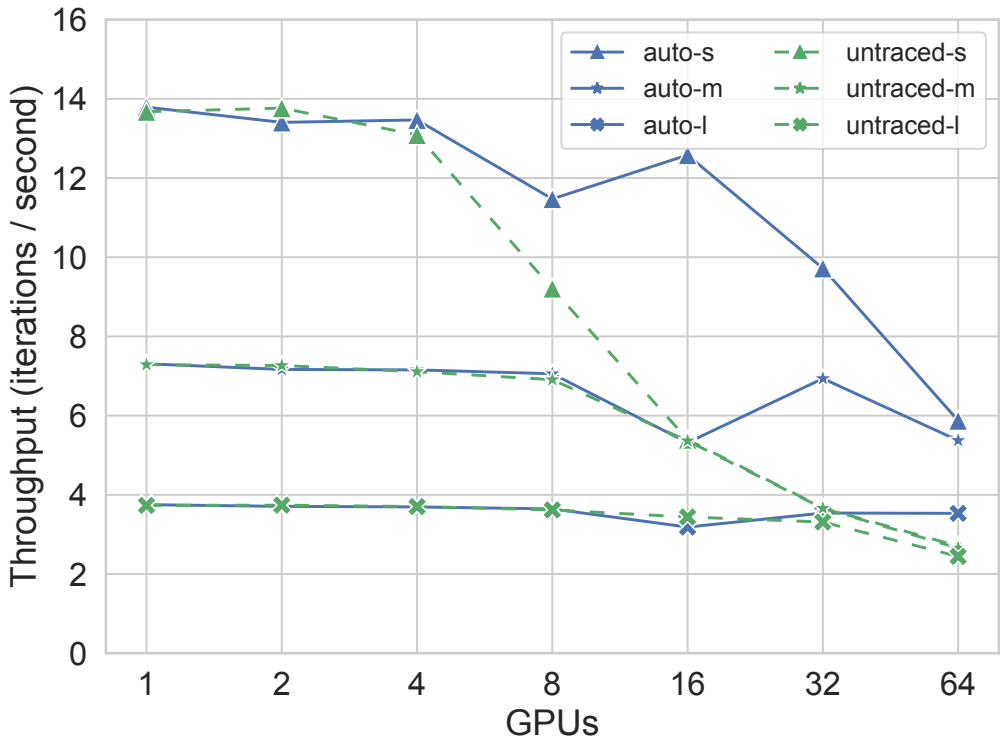


Figure 5.8: Weak scaling of CFD on Eos, where Legion with automatic tracing ("auto") outperforms the untraced version.

CFD

CFD is a cuPyNumeric application that solves the Navier-Stokes equations for 2D channel flow [18]. Unlike S3D and HTR, there is not a manually traced version of CFD, due to the difficulties around composition discussed in Section 5.1.1. Developing a manually traced implementation of CFD would require either rewriting the application to remove any dynamic region allocation, or manual examination of allocator logs to find the number of iterations in the steady state. As a result, we compare CFD with automatic tracing to the standard untraced version on different problem sizes, which is the performance that cuPyNumeric users are able to achieve today.

Figure 5.8 shows weak scaling results for CFD on Eos. These results are similar to HTR, where leveraging tracing is necessary for performance at scale. On the smallest

problem size, even though the tracing removes a large amount of runtime overhead, the tasks are too small to hide the communication latency at larger scales, leading to the observed fall off in performance. On larger problems, CFD with automatic tracing is able to maintain high performance while the untraced version falls off, yielding between 0.92x–2.64x speedups.

TorchSWE

TorchSWE is a cuPyNumeric port of the MPI-based TorchSWE [44] shallow-water equation solver, and is the largest cuPyNumeric application developed so far. Similarly to CFD, there is no manually traced version to compare to. However, unlike CFD, performing a rewrite of TorchSWE to enable manual tracing would be difficult, as TorchSWE contains an order of magnitude more lines of code. Weak scaling results for TorchSWE on Eos are shown in Figure 5.9, which show that TorchSWE is significantly impacted by Legion runtime overhead without tracing.

These results demonstrate that there does not exist a problem size for TorchSWE on Eos that can hide Legion's runtime overhead without tracing. Even the large problem size, which nearly reaches the GPU's memory capacity, exposes Legion runtime overhead at 8 GPUs. The reason for this is that TorchSWE maintains a large number of fields for each simulated point, and issues different array operations on each field. The amount of data needed for each element in the simulation does not allow the task granularity to be easily increased to the untraced Legion minimum of $\tilde{1}$ms per task, as each new element added increases the memory footprint more than it increases the average task granularity. For such applications, leveraging tracing is a requirement, and automatic tracing enables complex applications like TorchSWE to do so automatically. TorchSWE itself contains enough task parallelism to hide communication latencies, but needs tracing to first lower runtime overhead. With automatic tracing, we are able to achieve between 0.91x–2.82x speedup on TorchSWE, achieving nearly perfect scalability on 64 GPUs.

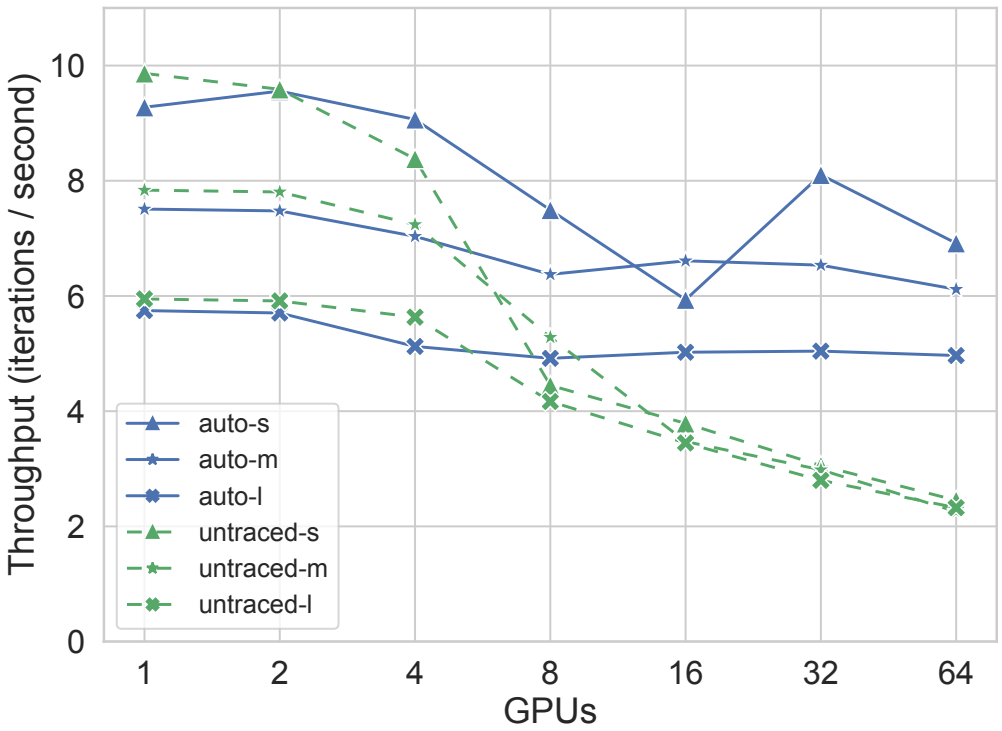


Figure 5.9: Weak scaling of TorchSWE on Eos, where Legion with automatic tracing ("auto") outperforms the untraced version.

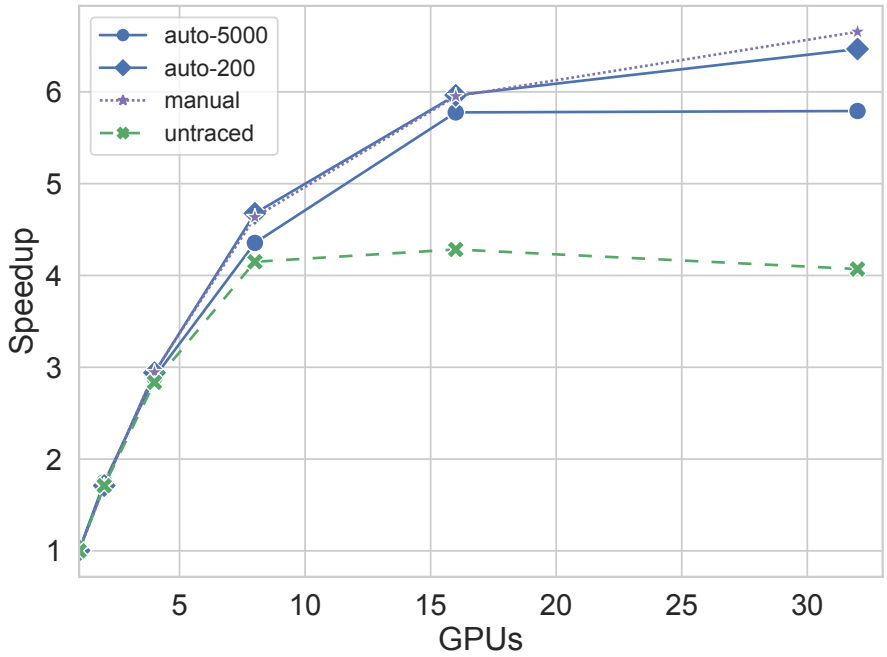


Figure 5.10: Strong scaling of FlexFlow on Eos.

5.6.3 Strong-Scaling

We now move from scientific simulation codes to distributed deep neural network training with FlexFlow [74, 125]. FlexFlow is a deep neural network framework that searches for hybrid parallelization strategies for different layers of the network. We perform a strong-scaling experiment with FlexFlow on Eos to train the largest (`pilot1`) network from the CANDLE [82] initiative[4]. A strong-scaling study fixes the problem size on a single processor, and increases the number of processors while keeping total problem size constant. To strong scale the training, we fix the batch size for a single GPU, and then increase the number of GPUs available.

We compare the performance of FlexFlow with manual trace annotations, two configurations of Legion (discussed next), and no tracing. As seen in Figure 5.10, as FlexFlow scales up, the tasks become smaller and begin to expose Legion runtime

[4]Due to engineering limitations in FlexFlow at the time of experiment collection, the network was parallelized only with data parallelism.

overhead without tracing, leading to slowdowns when scaling up. The two configurations of Legion differ in the maximum trace length to be replayed (Legion's history buffer is the same, but recorded traces are broken into pieces of a given maximum size). The first (auto-5000) is the standard configuration with no maximum, as used in all other experiments, and the second (auto-200) has a maximum length of 200 tasks, which is similar to the length of the manually annotated trace. As FlexFlow strong scales, the cost of Legion issuing the trace replay starts to become exposed as the execution time of the trace decreases, leading to shorter traces exposing less latency, and thus performing better[5]. On 32 GPUs, the configuration of Legion with a maximum trace length of 200 achieves between 0.97x the performance of the manually traced FlexFlow, and achieves a 1.5x speedup over the untraced FlexFlow.

5.6.4 Overheads of Automatic Tracing

We now discuss the overheads that automatic tracing imposes over standard execution with Legion. While we inherit the overheads of Legion's existing tracing infrastructure [87] (the cost of memoizing traces), automatic tracing imposes two new sources of overhead to measure: 1) the overhead on task launches and 2) the time taken until a steady state is reached.

As discussed in Section 5.3, the automatic tracing analysis intercepts the application's task launches and performs some analysis work before forwarding the task launches to Legion. This analysis work includes launching asynchronous token buffer processing jobs and manipulating traversals of the trie data structures used for online trace identification. To quantify this overhead, we ran a two node experiment on Perlmutter and measured the time it took to launch (not analyze or execute) Legion tasks with and without automatic tracing enabled. We ran a two node experiment to ensure that the coordination logic discussed in Section 5.4.1 was included in timing. We found that task launching took on average 7μs without automatic tracing, and on average 12μs with automatic tracing. While automatic tracing increases the task launch overhead, this overhead is still significantly lower than the amount of time it

[5]The Legion team is aware of this shortcoming and plans to address it in the future. Part of the work to reduce this overhead is discussed in Chapter 6.

Application	Iterations Until Steady State
S3D	50
HTR	50
CFD	300
TorchSWE	300
FlexFlow	30

Table 5.1: Warmup iterations before Legion with automatic tracing reaches a replaying steady state.

takes to replay a task as part of a trace, which is 100μs. As such, the task launching cost of automatic tracing can still be effectively hidden by the asynchronous runtime architecture. The asynchronous analysis jobs that Legion launches to process task histories do not affect the critical path, and utilize Legion's background worker threads. While in theory these jobs could compete for the resources necessary for Legion's dependence analysis, we have not yet encountered an application where they caused a detriment in performance.

To measure the time taken until Legion with automatic tracing reaches a steady state of replaying traces on our iterative applications, we report the number of iterations until a steady state is reached. Table 5.1 contains the iteration counts needed for each application in Section 5.6.2 and Section 5.6.3, which range from 30 to 300. These simulation and machine learning workloads would be run in production for a significantly larger number of iterations, so speedup in the steady state corresponds closely to end-to-end speedup. We note that the cuPyNumeric applications have a larger number of required warmup iterations due to the dynamic behavior discussed in Section 5.1.1, where a single application-level iteration of the program does not necessarily correspond to a repeated sequence of tasks.

In terms of resource utilization, automatic tracing requires a modest amount of CPU memory to store the history buffer of tasks for analysis. Legion runs the asynchronous string analysis (Section 5.3.2) on the existing pool of background worker threads. We have not found these resource requirements to impact application performance or memory utilization.

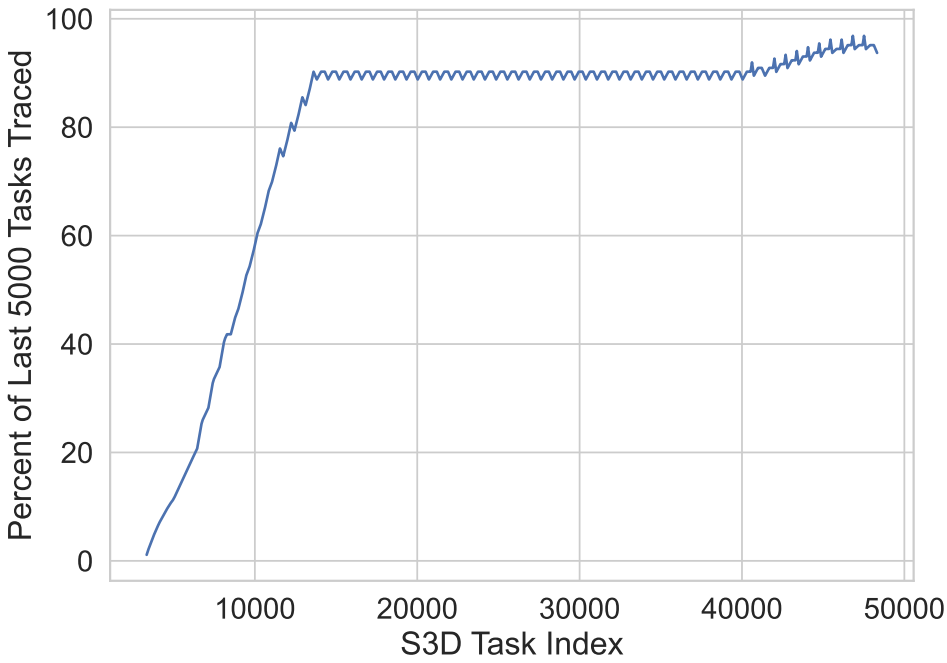


Figure 5.11: Visualization of Legion finding traces in S3D.

5.6.5 Trace Search

To give intuition about the search process that the automatic tracing analysis performs, we constructed a visualization of the amount of runtime overhead that Legion is removing over time. Figure 5.11 is a visualization of S3D over time (for 70 iterations), where for each task launched by S3D, we display how many of the previous 5000 tasks were traced. For iterative computations, this procedure yields the expected result, where Legion spends time during program startup discovering new traces, and then settles into a steady state. The amount of traced operations increases slightly by the end of the execution, as Legion finds a better set of traces that lowers the number of untraced operations.

5.7 Conclusion

This chapter focused on automatically amortizing the cost of the dynamic dependence analysis performed by Legion, which is critical for correctly and efficiently composing Legate programs. This analysis can be amortized through tracing, which memoizes and replays the analysis for repeated program fragments. Before our work, tracing in Legion required manual annotation, and was difficult (or almost impossible) to use tracing in programs where composition and abstraction obscure the concrete task stream issued to Legion. We argue to architect Legion as a just-in-time compiler, automatically identifying hot sequences of tasks as traces, and implicitly switching to memoized dependence analysis results without programmer intervention. This architecture enables iterative Legate programs that hide task launches deep within library implementations to be traced, transparently accelerating them and enabling efficient scaling of problems constrained by dependence analysis costs.

Chapter 6

Controlling Overheads (Task-Based Execution)

The previous chapter focused on the problem of controlling the overheads that dynamic analysis within Legate imposes on the overall runtime of Legate programs. Specifically, Chapter 5 tackled automatically tracing Legate programs to control the overheads of Legion's dependence analysis, which is an expensive dynamic analysis that is critical for composition. This chapter focuses on additional overheads below Legion that impact absolute performance as well as strong-scaling. We focus now on overheads within the Realm [122] runtime system, which affect the absolute performance and scalability of explicitly-parallel task-based programming systems when compared against bare-metal systems like MPI. This section of the thesis takes a detour from the Legate runtime itself and confronts these overheads by diving into a connection between task-based programming models and *actor-based* programming models, a different family of programming models for distributed machines that historically has higher peak performance than task-based programming models. This connection is then exploited to develop a compilation strategy that reduces the overheads of task-based programming models by converting task-based programs into actor-based programs. The compilation strategy allows task-based programs to execute with overheads comparable to efficient actor-based systems, allowing task-based

programs (like the Legate ecosystem) to have higher absolute efficiency and dramatically improved strong-scaling.

Prior Work Declaration. This chapter is based on material in the publication "On the Duality of Task and Actor Programming Models" [140]. This chapter builds on the Realm runtime system, which is prior work and discussed in Sean Treichler's thesis [121].

6.1 Overview

Actor-based programming models are the basis for some of the first systems for distributed memory machines [59, 78, 8]. Actor models send and receive explicit messages between *actors* to perform data movement and synchronization. Due to the simplicity of the message passing interface in most actor models, the underlying systems are embodied by thoroughly optimized implementations to minimize overheads associated with sending and receiving messages. While the simplicity of actor models ensures low-overhead implementations, it often incurs a latent cost: as programs become larger and more complex, the burden of maintaining their correctness tends to scale super-linearly with the size of the actor program. Sophisticated programs accumulate interacting features that actors must support with a burgeoning set of asynchronous messages that may arrive in a growing set of permutations. Consequently, actor models have been subject to the criticism that they are error-prone and result in programs that are difficult to maintain and evolve [91].

In contrast, task-based models strive to deliver higher productivity in response to the increasing complexity of both modern hardware and software. Programs are organized as (or in Legate's case, eventually lowered to) directed acyclic graphs (DAGs), often constructed dynamically, of short-lived computations called *tasks*. Explicitly-parallel task-based models require clients to directly construct the DAG by specifying data movement and dependencies between tasks, while implicitly-parallel models infer the DAG from the data usage of tasks. Regardless of the DAG construction mode, task-based programs are simpler to modify with only local reasoning, making it easier

to compose modules together and maintain software over long spans of time [91].

Due to the generality of the DAG execution model, task-based systems are challenging to fully optimize and manifest overheads associated with executing the DAG. If the task granularity is sufficiently large, these overheads have a negligible performance impact. However, when strong-scaling or exploiting fine-grained parallelism, the overheads can eventually inhibit performance as they come to dominate the runtime of the program. Modern accelerators, with growing compute power, are shrinking the execution times for tasks with each new generation, thereby placing increasing pressure on task-based systems to lower overheads to maintain scalability.

The complementary nature of the trade-offs associated with tasks and actors suggests a deeper relationship between the two classes of programming models. Inspired by the classic duality between message-based and procedure-based operating systems [85], we make the observation that actor and task programming models are also duals of each other. Actor-based programs are characterized by long-running actors that communicate through messages, similar to processes in message-based operating systems. In contrast, task-based programs are characterized by a large, changing number of short-lived tasks that communicate through shared data, similar to small procedures modifying shared data in procedure-based operating systems. Importantly, we further claim that this duality extends beyond functional equivalence, and is a performance duality where programs written in one style can be translated to the other and achieve comparable performance.

We illuminate the sources of overheads encountered in task-based models through different translation strategies between task-based and actor-based models. We then leverage these insights to develop compilation techniques for task-based systems that transform demarcated subgraphs of the complete program DAG into actor-based programs. The compilation strategy results in surprisingly simple actors that efficiently execute the target subgraph, greatly reducing overheads for important and repeatedly executed components of the complete program DAG. Iterative applications, such as those in domains ranging from deep learning to scientific computing, are amenable to our graph compilation techniques as they perform repetitive computations. As depicted in Figure 6.1, we exploit the duality in the task-to-actor direction to bridge

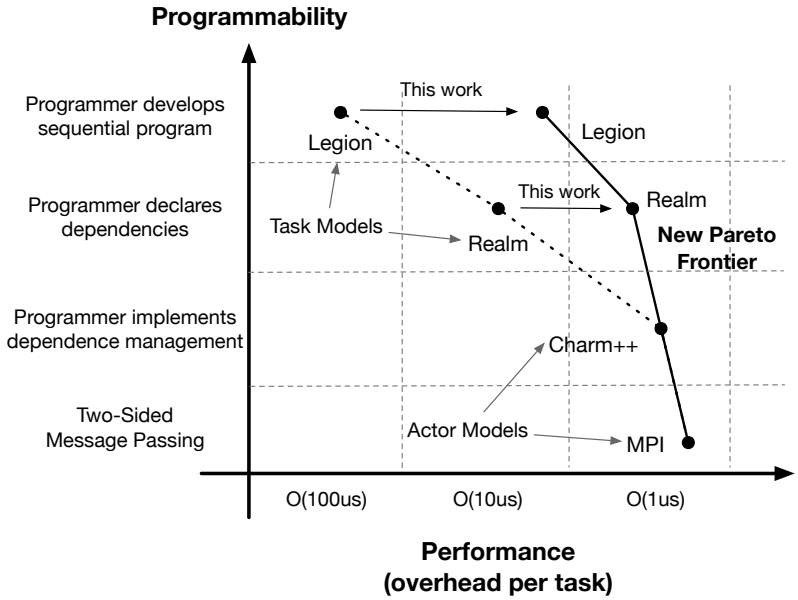


Figure 6.1: We define a new Pareto frontier (top-right is best) for distributed programming along the performance-programmability tradeoff. See Section 6.5 for performance details.

the gap in performance while preserving the programmability characteristics.

6.2 Background

We now provide specific background about actor-based and task-based programming models. While the rest of this thesis has discussed the task-based, user-facing Legate programming model and intermediate representations, this section focuses on, and provides historical context around general-purpose, explicitly-parallel task-based programming models. Such a model describes Realm, the task-based system at the bottom of the Legate runtime stack.

6.2.1 Actor-Based Programming Models

Actor-based models are characterized by long-lived, stateful objects called *actors* that maintain arbitrary local state and communicate through asynchronous messages. In modern actor-based systems, actors are often associated with machine resources (e.g.,

a GPU or a CPU core), and applications are comprised of actors performing local computations and notifying other actors to start follow-up work.

Actor-based programming models are well-studied and have been extensively formalized in prior work [3, 64]. For simplicity and the focus of this work, we consider a simple actor-based runtime that might be embedded within a standard host language, as shown in Figure 6.2. This runtime is a simple model of systems like Charm++ [76, 78] or Ray [92]. The `Actor` object is extended (in potentially multiple ways) by the application to contain arbitrary private state and an implementation of the `handle_message` function. Actors are registered with the runtime to a concrete resource. For simplicity, we assume that there is a fixed set of actors registered at program initialization time; in practice however, most actor systems allow for the dynamic creation of new actors [76, 78, 92]. The main primitive offered by the runtime is the ability to send a target actor a message, which (remotely) invokes the target actor's `handle_message`. Applications in actor-based programming models are often structured as state machines (discussed more in Section 6.3.3) that accept messages until an expected set is received to execute some application computation. These state machines only modify the private state of the actor handling each message; all inter-actor communication and coordination occurs through message passing.

While not a common classification, we also consider two-sided messaging systems like MPI to be within the actor-based programming model family, as they share the properties of having long-running processes that react to incoming messages. In MPI, each message send must be paired with a message receive operation on the destination side. Two-sided messaging can be simulated with the actor model we present by buffering arriving messages until they are handled by the corresponding receive operations. Unsurprisingly, this is actually how non-blocking message sends and receives in MPI (i.e., `MPI_Isend` and `MPI_Irecv`) are implemented, with the associated tags serving an analogous purpose to the kind of message handler to be invoked in an actor model.

```
class Processor;
class ActorRT {
  void send_message(int aid, int mid, void* args, int len);
  void register_actor(Actor* a, int aid, Processor target);
};
// Actor is extended by the application.
class Actor {
  // Actors maintain arbitrary, but private state.
  void handle_message(
    int mid, ActorRT* rt, void* args) {
      // Perform arbitrary computation, such as
      // sending more messages to other actors.
    }
};
```

Figure 6.2: Actor-based Runtime System

```
class Event, Allocation, Processor, Memory;
using Events = vector<Event>;
class TaskRT {
  void register_task(int tid, void (*task) (TaskRT*, void*));
  Event launch(Processor p, int tid, void* args, Events pre);
  pair<Event, Allocation> alloc(Memory m, int s, Events pre);
  Event copy(Allocation src, Allocation dst, Events pre);
};
void task(TaskRT*, void* args) {
  // Stateless task body that may unpack data
  // from args and perform computation.
}
```

Figure 6.3: Task-based Runtime System

6.2.2 Task-Based Programming Models

Historically, task-based programming models emerged after actor-based programming systems, aiming to provide more composable and more accelerator-friendly abstractions for parallel computing. The core concept in task-based programming models is a *task*, which is a stateless, user-defined function. Tasks may launch other tasks, and are issued onto a target processor to run asynchronously from the launching task. Task-based applications express their computation as a graph of tasks that operate over shared data structures. These task graphs can be constructed in an offline or static manner [46], or in an online, dynamic fashion [122, 25, 10, 92, 34].

Figure 6.3 presents a simplified interface for Realm [122], which sits at the bottom of the Legate runtime stack, and is an explicitly-parallel task-based system that supports dynamic task graph construction. All operations in the task-based model return an *event* that represents the asynchronous completion of the operation[1]. Applications

[1]Events that carry data, such as the return value of a task, are often referred to as *futures*.

may launch tasks on processors, allocate data in memories across the machine, and copy data between allocations. Unlike actor-based models, tasks may have side-effects on allocations that outlive their individual lifetimes, and tasks can share state through these side-effects. Each asynchronous operation is predicated on a set of events that must complete before the operation executes. The task-based runtime schedules operations that have all event preconditions satisfied, automatically overlapping the execution of independent operations (such as data movement and computation). The interface in Figure 6.3 can be embedded within a general purpose language, and is the target for arbitrary computation to dynamically construct a task graph by computing dependencies and issuing tasks.

6.3 Equivalence and Duality

We now describe a functional equivalence and performance duality between actor-based and task-based models. We are inspired by the work of Lauer et al. [85], which showed an equivalence and duality between message- and procedure-based operating systems. The duality arises in how applications developed in either system can share core application logic, while the differences arise only in the synchronization of when that shared application logic should execute. We reveal this duality through different reduction strategies that expose the structure underlying the two programming models. We then discuss the inherent tradeoffs made between performance and programmability in the two models.

In the presented reductions, we assume an orthogonal scheduling system exists that, for actors, selects available messages to handle, and for tasks, selects tasks with all satisfied event preconditions to run. The reductions maintain the space of possible schedules (observed message handler invocations or task execution orders) that the scheduler could realize.

6.3.1 Reducing Actors To Tasks

We reduce actors to tasks by demonstrating an actor-based system implemented

```
// Let A be an actor handling messages M1 and M2, registered to processor P.
// Create an allocation for A in a memory visible to P. For simplicity,
// ignore the returned event.
auto result = task_rt->alloc(P.memory(), sizeof(A), {});
A* stateA = result.second.get_base_pointer();
// Register tasks for each message A handles.
void task_A_M1(void* args) {
  stateA->handle_message(M1, args);
}
void task_A_M2(void* args) {
  stateA->handle_message(M2, args);
}
...
void task_A_MN(void* args) {
  stateA->handle_message(MN, args);
}
register_tasks(
  {A_M1, task_A_M1},
  {A_M2, task_A_M2},
  ...
  {A_MN, task_A_MN}
);
// actor_rt->send_message(A, A_M1, args) translates to a task launch
// with no event preconditions, which can be invoked from anywhere
// on the machine.
task_rt->launch(P, A_M1, args, {});
```

Figure 6.4: Actor to task reduction pseudocode.

with a task-based system. While tasks are stateless functions, tasks may emulate stateful actors by providing them access to persistent state. The reduction is straightforward and shown in Figure 6.4: actor objects are allocated on the resource they are bound to, and every message sent to an actor A is mapped to a task launch that is given access to A's allocation to invoke A's message handler (potentially modifying A's state). All potential orderings of message handler invocations in the actor-based program enforced by message causality are preserved by the translation, as new tasks are created only when new messages are handled. Notably, the reduction does not leverage the task-based model's event/dependence infrastructure and is relatively opaque; coordination and communication is still the responsibility of the programmer, hidden inside the existing actors' message handler implementations. However, this reduction can preserve performance by eliding the dependence infrastructure. Each task launch with no preconditions can be implemented efficiently with a single message, similar to the execution of the actor program without the reduction. While simple, this reduction also models the core of systems like Ray [92], which provide actors and tasks in the same language.

6.3.2 Reducing Tasks to Actors

We now reduce task-based models onto actor-based models, which starts to expose the structure of the programming model design space. We discuss two fundamentally different reduction strategies, which encapsulate different points in a tradeoff between performance and programmability.

The first strategy is to construct a collection of actors that form a *runtime system* to execute an *arbitrary* DAG that is constructed *dynamically* and *incrementally* as tasks are issued into the system one at a time. A simple set of actors structured like a runtime system are described in Figure 6.5, which contains a *scheduler* actor that manages pending tasks and event dependencies along with a set of *worker* actors that execute ready tasks on different processors. This reduction strategy is a simplified model of how task-based systems are implemented today, as independent processes communicating through active messages or remote procedure calls.

This reduction establishes functional equivalence between actor-based and task-based models, but also reveals where the performance differences between the two models arise. Task-based models allow for the declarative specification of dependencies between computations, but this abstraction comes at the cost of a generic runtime system. In contrast, high-performance actor-based applications avoid generic dependence infrastructure and synchronize in an *application-specific* manner. Actors directly encode the specific communication patterns and dependence logic for a particular application, rather than through an intermediate layer.

6.3.3 Specialization in Actor Models

To gain intuition for the specialization in efficient actor programs, we develop an example in Figure 6.6. Any parallel computation can be described as a directed acyclic graph (DAG), where the vertices represent atomic computations, and the edges represent dependencies between computations. The DAG itself may be dynamically constructed, and represents an "unrolled" version of the application. Figure 6.6a depicts a DAG with the application logic contained in the functions `f1`, `f2`, `f3` and `f4`, which are specified to be executed on the processors `P1` and `P2`. Actor-based and

```
// A simple runtime system creates one worker per processor
// and a scheduler that interfaces with the application.
class Scheduler : Actor {
  void handle_message(int mid, void* args) {
    switch (mid) {
      case LAUNCH: {
        auto [tid, p, ev, targs, preds] = unpack(args);
        register_pending_task(ev, p, preds, {tid, targs});
        send_message(this, SCHEDULE, {});
        break;
      };
      case TASK_DONE: {
        notify_waiting_tasks(unpack_event(args));
        send_message(this, SCHEDULE, {});
        break;
      };
      case SCHEDULE: {
        for (auto [ev, tid, p, args] : get_ready_tasks()) {
          send_message(get_worker(p), EXEC, {ev, tid, args});
        }
        break;
      };
    }
  }
};

// A worker actor is bound to an individual processor and handles executing
// tasks sent to that processor.
class Worker : Actor {
  void handle_message(int mid, void* args) {
    auto [tid, ev, targs] = unpack(args);
    execute_task_body(tid, task_args);
    send(SCHED_ID, TASK_DONE, {ev});
  }
};

// Tasks are launched by sending a message to the scheduler.
Event launch(Processor p, int tid, void* args, Events pre) {
  Event ev = generate_fresh_event();
  send_message(SCHED_ID, LAUNCH, {tid, p, ev, args, pre});
  return ev;
}
```

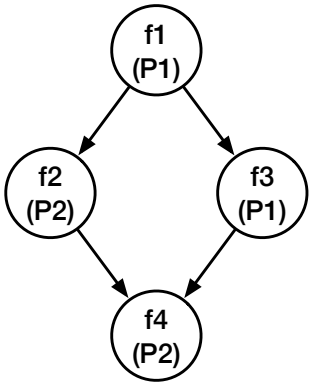
Figure 6.5: Runtime reduction pseudocode for tasks to actors.

task-based implementations of this DAG are shown in Figure 6.6c and Figure 6.6b respectively. The task-based implementation wraps the application logic in tasks and directly translates the DAG into tasks and events. As the reduction in Figure 6.5 demonstrates, an underlying runtime system uses additional messages and synchronization to submit tasks and coordinate execution based on edges in the DAG. In contrast, actor-based programs are frequently structured as *state machines* that directly communicate to negotiate dependencies, instead of through an intermediate runtime layer. These state machines can be structured to send only a single message per edge in the DAG. Specializing the actors to a specific DAG eliminates overheads that general task-based systems incur such as extra coordination messages, dynamic dependence management, and task submission costs.

In the general case, a task-based system must be implemented in the generic flavor discussed in Section 6.3.2, as the desired DAG is expressed in a dynamic and incremental manner: tasks may be spawned at any time from any processor, and may depend on an event produced anywhere in the distributed system. However, if the unit of work submission is a DAG instead of a single task, there is an opportunity to perform the same specializations that users of actor-based models employ, at least within that DAG. In particular, a *fixed* DAG G of tasks can be compiled into a set of actors that exploit the structure of G to directly communicate and eschew the standard dependence infrastructure of the task-based runtime system, sending exactly one cross-processor message per edge in G. We present an algorithm to perform this specialization in Section 6.4.2, and demonstrate that an implementation in the Realm runtime system lowers overheads by up to a factor of five.

6.3.4 Duality and Tradeoffs

Actor-based and task-based models serve as different vehicles for programmers to materialize a DAG into a concrete implementation. The atomic computations represented by the nodes of the DAG may be abstracted from the implementation substrate, as done in Figure 6.6, and be used in either a task-based or actor-based implementation. The remaining difference between programs in the two models is the



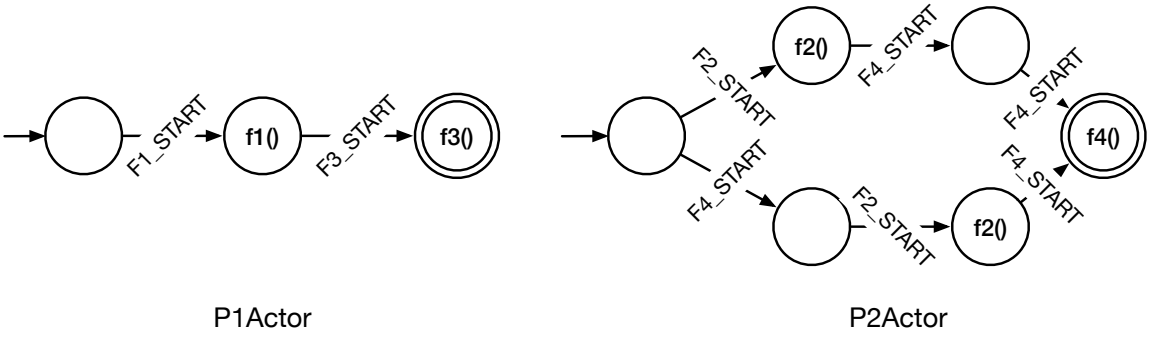
(a) Example program DAG.

```
register_tasks(
  {T_F1, f1}, {T_F2, f2},
  {T_F3, f3}, {T_F4, f4}
);
Event e1 = launch(P1, T_F1);
Event e2 = launch(P2, T_F2, e1);
Event e3 = launch(P1, T_F3, e1);
Event e4 = launch(P2, T_F4, {e2,e3});
```

(b) Task-based implementation of Figure 6.6a.

```
class P1Actor : Actor {
  void handle_message(
    int id, void* args
  ) {
    if (id == F1_START) {
      f1();
      send_message(P2_ID, F2_START);
      // Use a message send to make
      // the F3 start state explicit.
      send_message(P1_ID, F3_START);
    } else if (id == F3_START) {
      f3();
      send_message(P2_ID, F4_START);
    }
  }
};
class P2Actor : Actor {
  // Maintain a notification count
  // for the F4 start state.
  int count = 2;
  void handle_message(
    int id, void* args
  ) {
    if (id == F2_START) {
      f2();
      send_message(P2_ID, F4_START);
    } else if (id == F4_START) {
      if (atomicSub(&count, 1) == 0)
        f4();
    }
  }
};
```

(c) Actor-based implementation of Figure 6.6a.



(d) Visualization of actor state machines.

Figure 6.6: Example computation developed in actor-based and task-based programming models.

implementation of dependence management that coordinates the shared application logic; that responsibility is either the programmer's (actors) or the runtime system's (tasks). This glue code (state machine construction or declarative task-graph construction) is separate from the shared application code, and we have shown in the previous sections how either construction may be translated into the other.

While Lauer et al. [85] conclude that the performance of message-based and procedure-based operating systems is dependent on the underlying hardware, we believe that the choice of actors versus tasks is a tradeoff between performance and productivity. Actor-based models have been historically viewed and experimentally shown [116] to exhibit significantly lower overheads than task-based models, due to the specialization discussed in Section 6.3.3. We argue that this performance comes at a cost of higher burden on the programmer to manage the dependencies between computations and to exploit available parallelism.

The implementations of dependence state machines in actor-based programs are lower-level and programmer-managed. As seen in Figure 6.6d, actor state machines must explicitly manage communication and ensure all dependencies are correctly met, all while simultaneously exploiting as much parallelism as possible. When the application DAG is modified to include a new vertex or edge, or needs to be ported to a new machine with a different memory hierarchy and set of processors, the state machines of the actors may require significant modifications. These alterations may include new messages and communication patterns, changes to the expected number of incoming messages for a state, or new states to handle the interaction between new vertices and all existing vertices that may potentially execute in parallel. Targeting accelerators like GPUs incurs additional complexity to actor state machines. A common paradigm in actor models [78] is to treat the completion of asynchronous accelerator computations as additional messages sent to an actor, increasing the number of states and messages to reason about. Higher-level languages have been developed within the actor programming community, like Dagger [77], to simplify the development of actor state machines, but the fundamental challenges of managing large state machines remain.

In contrast, task-based models specify dependencies in a declarative style and

leave satisfying those dependencies while maximizing parallelism to the runtime system. This declarative nature means that program modifications like adding new dependencies, new tasks, or changing where tasks run are considerably easier. Additionally, the declarative specification naturally incorporates asynchronous accelerators — the runtime is responsible for ensuring that tasks dependent on asynchronous work start only when the work completes. Higher-level, implicitly-parallel task-based systems [25, 10] further simplify programming by automatically inferring parallelism, not even requiring the user to describe dependencies [91]. As discussed, task-based systems have historically traded this ease of programming for overheads that come from implementing the abstractions they provide. For example, the task-based Realm system has up to 7 times the overhead of an efficient actor system like Charm++ (Section 6.5). In this work, we make significant progress towards collapsing this trade-off space for applications that are able to present repeatedly executed subgraphs to the task-based runtime system, such as those present in iterative computations that are found in domains spanning the gamut from deep learning to scientific computing. For such applications, we are able to provide the programmability properties of task-based programming models while delivering overheads approaching efficient actor-based programming models.

6.4 Compiling Task Graphs to Actors

We now leverage the duality between actors and tasks to develop a compilation strategy that reduces the generality overheads incurred by explicitly-parallel task-based models like Realm. Our approach lowers task graphs to a set of specialized actors that avoid dynamic dependence management and extraneous communication. We then show in Section 6.4.4 how to leverage this intermediate representation from an implicitly-parallel task-based system like Legion.

6.4.1 Interface

Applications define a task graph G with a set of vertices V and edges E. Each vertex in V is an operation, such as a task or copy, and edges describe dependencies between operations. Vertices may also be *external* pre- or post-conditions, representing dependencies that either come from outside of the graph, or must be notified when operations within the graph complete. Applications dynamically construct graphs and register them with the runtime; afterwards, the entire graph may be launched as a single operation. The target graphs for compilation are subgraphs of the complete program graph that are performance critical and repeatedly executed. The compiled subgraphs are then dynamically issued by the application and stitched into the larger program graph through the external pre- and post-conditions. Control flow structures such as loops and conditionals are the purview of the application dynamically building the larger program graph, rather than constructs within the compiled subgraph.

6.4.2 Compilation

The goal of compilation is to specialize the runtime system itself to the input task graph G, resulting in a set of actors specialized to G [60]. These specialized actors avoid expensive synchronization structures (like events) and communicate with the minimum number of cross-actor messages. Constructing specialized actors involves building a state machine for each actor that executes vertices in G and transitions upon receiving messages from other actors. To maximize performance, the state machine must exploit all potential parallelism in G by executing vertices as soon as their dependencies are satisfied. Topologically unordered vertices in G may execute in any order, and the completion messages for these vertices may also arrive in any order. The state machine handling all of these potential orderings to execute vertices in G as soon as possible would contain a state for each prefix of every topological ordering of G, as any one ordering may result in different vertices being ready at any given point in time. An example task graph and state machine handling all valid topological execution orderings is shown in Figure 6.7. Explicitly enumerating and generating code for such a state machine in the syntactic style of Figure 6.6d would

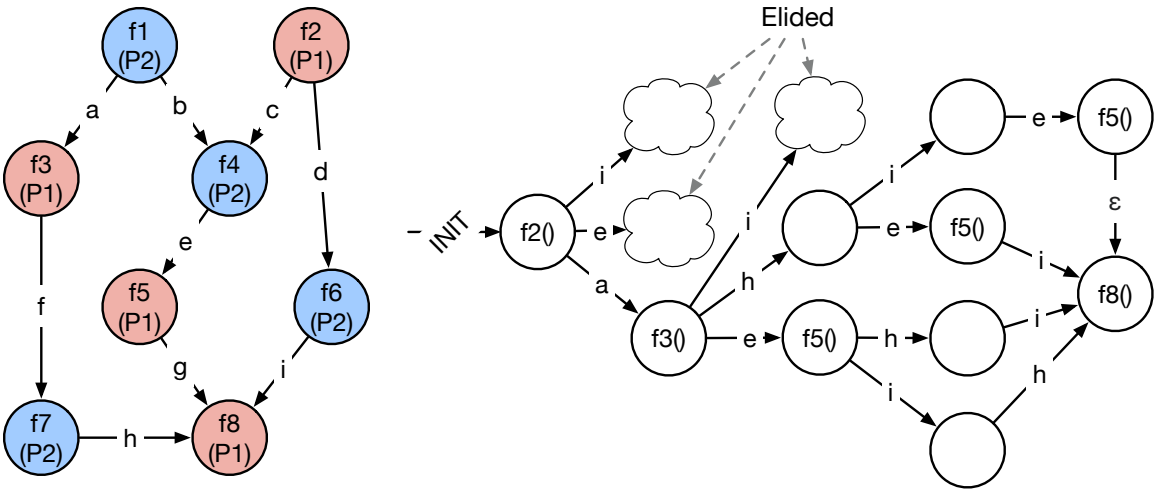


Figure 6.7: Example task graph and a parallel message handling state machine.

be infeasible due to the factorially large number of states[2]. Instead, we separate the process into two phases, described in Algorithm 4. A compilation phase first pre-processes G and constructs a set of actors. Then, an interpretation phase executes the graph, where each actor interprets a series of commands to execute operations. The factorially large state machine is encoded implicitly through different dynamic configurations of each actor's interpreter data structures.

The compilation step collects all processors and data movement channels used by vertices of G. For each resource r, we take the subgraph of G that runs on r (referred to as G_r) and pre-process G_r into an indexing structure that enables constant-time lookup of edges. Then, a set of counters is prepared for each vertex in G_r that maintains the number of pending incoming dependencies for that vertex. The combination of counter values for each vertex in G_r corresponds to a logical state in an actor state machine running on r. Finally, the compilation step creates a worker actor for executing the corresponding subgraph associated with resource r.

Each worker actor is structured as an interpreter that processes incoming messages

[2]Managing this large state space is difficult for humans too, who often trade off exploiting available parallelism to simplify the encoded state machine and ensure correctness, ultimately leaving performance on the table.

Algorithm 4: Task Graph Compilation Algorithm

```
1  Compile (G)
2      resources ← all processors and memory channels used in G
3      foreach r ∈ resources do
4          V_w ← nodes of G running on r
           /* An edge list pre-processed for O(1) lookup of edges.  Maintains an
              atomic counter for each node in V_w's incoming edges.  */
5          E_w ← in and out frontier of V_w in G
6          RegisterActor(Worker(r, (V_w, E_w)), r)
7  class Worker (r, (V, E))
8      HandleMessage (id, data)
9          switch id do
10             case INIT do
11                 E ← reset(E)
                   /* Start all ready to execute work.  */
12                 foreach v ∈ V | ∄ (_, v) ∈ E do
13                     SendMessage(this, EXECUTE_OP, v)
14             case COMPLETED_EDGE do
15                 (src, dst) ← unpack(data)
                   /* Concretely computed by an atomic decrement to an indexing
                      structure.  */
16                 E ← E − (src, dst)
17                 if (_, dst) ∉ E then
18                     SendMessage(this, EXECUTE_OP, dst)
19             case EXECUTE_OP do
20                 op ← unpack(data)
                   /* Execute op on this actor's resources.  */
21                 Execute(op, r)
22                 foreach (op, dst) ∈ E do
23                     SendMessage(owner(dst), COMPLETED_EDGE, (op, dst))
24 Execute (G)
25     foreach w ∈ RegisteredWorkers(G) do
26         SendMessage(w, INIT)
```

and updates its state. Upon receiving a message to execute an operation, the worker executes the corresponding operation and then sends a message to the actors responsible for running each dependent operation in G. When receiving a message that an incoming edge has completed, the worker decrements the corresponding counter and potentially enqueues the target operation for execution. Graph execution is initiated by sending each worker actor an initialization message, which upon receiving, each worker enqueues all operations with no predecessors.

The worker actors themselves contain the minimal functionality for correctness to ensure low-overhead task graph execution. Because G is known and fixed, the signaling of dependencies can be done without any intermediate structures or extra messages, such as those described in Section 6.3.2. Furthermore, the actors avoid complex concurrent data structures, and instead coordinate dependencies within an actor using lock-free atomic decrements for each edge in G. We show in Section 6.5 that this compilation strategy improves overheads in the Realm runtime system by 1.7-5.3x.

6.4.3 Optimizations for Accelerators

There is a natural tension between task-based programming models and programming models for accelerators such as GPUs. Programming models for accelerators prefer client applications to run far ahead, creating significant quantities of *asynchronous operations* (e.g. GPU kernels or data movement operations) to hide the latency of work creation and push dependence tracking down onto the accelerator, only synchronizing with the device sparingly. Alternatively, in a task-based program, most tasks targeting an accelerator will generate one or a few asynchronous operations as part of their execution, and, in an unoptimized implementation, will then need to synchronize with the device at the end of each task to ensure all their asynchronous operations are reflected in the completion of the task for any downstream dependencies. This approach incurs significant overheads due to both over-synchronization of the device and the exposed latency of work creation.

In some task-based systems, such as Realm, over-synchronization of the device is

addressed by associating a GPU stream with each task running on an accelerator. Tasks are required to accumulate any asynchronous operations that they create as part of their execution into the associated stream. Each task can then finish without needing to explicitly synchronize with the device and Realm ensures that downstream dependencies of the task will not start until all asynchronous operations associated with the task's stream have completed. While this solves the problem of device over-synchronization, it does not address the issue of exposed latency of asynchronous work creation when tasks start running because downstream tasks cannot begin executing and launching their asynchronous operations until after the asynchronous operations from their predecessor tasks have finished executing and drained off the device, potentially leaving the accelerator idle.

For the general case of dynamically created task programs, it is difficult to further hide the latencies of asynchronous work creation because it is challenging to optimize the graph as it is being both created and executed simultaneously. However, when compiling an entire task graph, we can perform an additional optimization to further hide the launch latency of asynchronous operations inside of tasks by recognizing scenarios when we can start tasks that are only going to launch asynchronous operations on the accelerator early and push dependence tracking down onto the device. To perform this optimization, we recognize scenarios when two tasks with a dependence are both going to run on an accelerator managed by the same hardware driver (e.g. the CUDA driver). We then check if the client has marked the downstream task as being *control independent*, meaning that the task's logic for launching asynchronous operations is independent from the data being passed to the task. This permits the task to start before its dependencies are ready as long as the dependencies for the task are incorporated into the stream associated with the task, so any asynchronous operations the task creates will not access the data passed to the task prematurely.

We specialize the target task graph G for asynchronous accelerators by decoupling the host-side of operations that launch asynchronous work from the device-side asynchronous work itself, as visualized in Figure 6.8. This specialization is performed as part of the compilation process described in Section 6.4.2. For every node n within

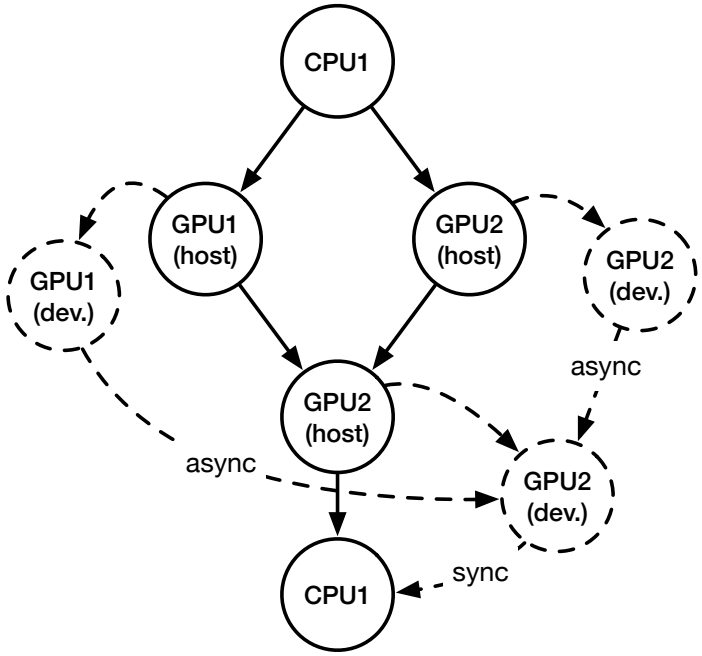


Figure 6.8: Transformation for accelerators. Dashed nodes and edges are added.

G that launches accelerator work we add a new *async* node n_a representing the asynchronous work. Then, for every outgoing edge $(n, d) \in G$, if d has a corresponding async node d_a, we add an async edge (n_a, d_a). If d does not have a corresponding async node, then we add a *sync* edge (n_a, d). When executing G using Algorithm 4, every asynchronous node records state representing its completion, such as a CUDA event, on their task's stream upon completion of the task. Async nodes incorporate prior async edges by merging CUDA events from their predecessors into their task's stream to serve as preconditions. Sync edges are lowered by sending a message to the destination actor when the source asynchronous operation completes. This transformation enables significant run-ahead and offloads synchronization to hardware-supported mechanisms whenever possible. While presented in the context of accelerators, this procedure could be used wherever the computation in a task is decoupled into separate stages, such as when performing asynchronous I/O.

6.4.4 Lowering Implicitly-Parallel Models

While explicitly-parallel task-based models are useful for certain applications, they are especially useful as a lowering target for implicitly-parallel task-based models like Legion and Legate. As discussed in Chapter 5, these implicitly-parallel models perform a dynamic dependence analysis to extract parallelism from a sequentially expressed application. These models amortize the cost of the dependence analysis by memoizing and replaying the results of the dependence analysis on traces (repeated sequences of tasks), which may either be annotated by the programmer or discovered automatically (Chapter 5). This compilation process is integrated into Legion's tracing module by lowering the traced, explicitly-parallel task-based program directly to a compiled Realm task graph. This further reduces the overhead of trace replay by eliminating the cost of dynamic task graph reconstruction. We show in Section 6.5 that combining tracing with task graph compilation yields significant improvements in strong-scaling performance.

6.4.5 Compilation at Scale

There are two primary ways to compile task graphs for programs that execute on multiple nodes, presenting different trade-offs for clients. The first way is to treat the entire program as a single graph that spans all the nodes of the machine. This approach permits the compilation framework to observe the entire graph and perform optimizations that potentially span multiple nodes, thereby ensuring that the graph is fully optimized. The downside to this approach is that the size of the task graph grows with the scale of the machine being targeted, since all the tasks for each processor are explicitly represented. Since some of the optimizations performed are super-linear in their complexity, the cost of compilation can become prohibitive. Even representing the global task graph can require too much memory and become infeasible at large scales.

The alternative way to compile programs for multiple nodes is to *shard* the program so that each node produces its own task graph, with all the graphs cooperating

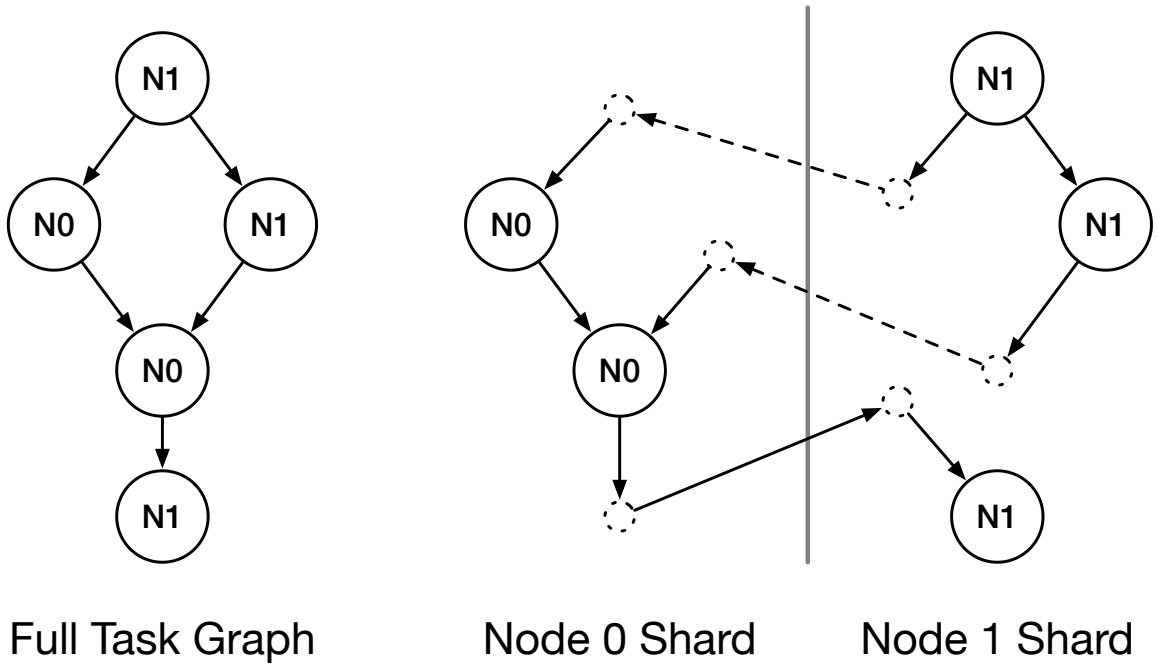


Figure 6.9: Example of task graph sharded onto two nodes.

together to execute the program. We leverage the external pre-condition and post-condition vertices of a graph to coordinate the inter-node dependencies, while intra-node dependencies are expressed through direct edges, as visualized in Figure 6.9. This sharded approach is a tradeoff between compilation time and execution performance. Sharding the global task graph into a compiled graph per node relinquishes some performance, as optimizations can only be performed locally in each shard graph and not across shard graphs. In practice, we found that this performance loss is minimal because, while intra-node dependencies can often be satisfied by hardware supported, light-weight communication and synchronization mechanisms (see Section 6.4.3), satisfying inter-node dependencies still fundamentally requires a more expensive network message. Furthermore, by creating a graph on each node, compilation is parallelized and the size of each node's graph is proportional to the amount of computation being performed on that node, ensuring that each node's graph fits into memory. For all our multi-node experiments in Section 6.5, we opted for sharded graph compilation. This includes both Realm-only programs for which we manually construct the sharded task graphs, as well as Legion programs that leverage support

for control replication [23] to implicitly construct sharded traces that are lowered onto sharded task graphs. Creating sharded task graphs is the best way we have found to ensure that our approach scales to large numbers of nodes.

6.5 Evaluation

Overview

We evaluate our work on micro-benchmarks and end-to-end applications implemented within the Legion [25] and Realm [122] runtime systems. Using the Task Bench [116] framework, we first show that our work significantly lowers the overheads imposed by both explicitly-parallel and implicitly-parallel task-based models, and enables explicitly-parallel models to support fine-grained computations with competitive performance to low-level actor frameworks like MPI [59] and Charm++ [78]. We then show that our work improves strong-scaling of end-to-end applications developed in implicitly-parallel task-based systems.

This work is currently only integrated into Legion and Realm, and has not yet been explored in conjunction with the full Legate runtime system. Aside from the engineering work involved in the integration, the current production implementation of Legate has overheads in its hot-paths for task creation and launching, along with a bottleneck caused by being driven by the Python interpreter. As a result, the improvements delivered by the techniques in this chapter may not yet yield end-to-end performance gains. We believe that in the future, Legate will be able to leverage these improvements, and I am currently working on a production implementation of the subgraph module to be landed into Realm.

Experimental Setup.

We ran all experiments on the NVIDIA Eos supercomputer. Each node of Eos is an NVIDIA DGX H100, containing 8 H100 GPUs with 80 GB of memory and a 112 core Intel Xeon Platinum. Nodes of Eos are connected with an Infiniband interconnect. We compile Legion and Realm with the GASNet-EX [31] networking module. We

compare against Open MPI 4.1.7, Charm++ 6.9.0, StarPU 1.4.7, Ray 2.47.1, and run all applications with CUDA 12.4.1.

6.5.1 Measuring Overheads With Task Bench

Task Bench [116] is a framework for comparing runtime system overheads. Task Bench defines a graph of tasks (atomic, coarse grained work items) in a generic interface to be implemented by each benchmark system. The task graph is a two-dimensional grid with width corresponding to available parallelism, height corresponding to number of timesteps to execute, and dependencies that relate tasks from timestep $t-1$ to timestep t, as shown in Figure 6.10. A Task Bench experiment fixes a graph structure, and varies only the amount of computation performed at each vertex. Figure 6.11 shows the FLOPs achieved by various systems on a single node (8 GPUs) running on a Task Bench graph with a stencil dependence structure (shown in Figure 6.10) and width of 8. Each Task Bench task launches a CUDA kernel that performs floating point operations in a loop with the number of iterations controlled by the x-axis of the graph. At high iteration counts (long task run time), most systems achieve close to peak (non-tensor-core) FLOPS available on a DGX H100 (272 TFLOPS), but as the number of iterations decreases, overheads decrease the total FLOPS each system achieves.

To evaluate overhead, Task Bench uses these performance curves to define a metric called *minimum effective task granularity* (METG). The METG(50) is the smallest task granularity where a system achieves at least 50% of the peak FLOPS, quantifying the task granularity at which overheads dominate the execution time. METG is superior to several common alternative metrics for measuring runtime system efficiency: weak-scaling hides arbitrary overhead if the problem sizes are too large, strong-scaling does not separate changing application costs (e.g. increased communication) from runtime costs, and tasks-per-second fails to consider the amount of useful application work performed. We refer readers to Slaughter et al. [116] for a complete discussion of the METG metric. For non-task-based systems, METG encapsulates the amount of application work required to offset operations like messaging

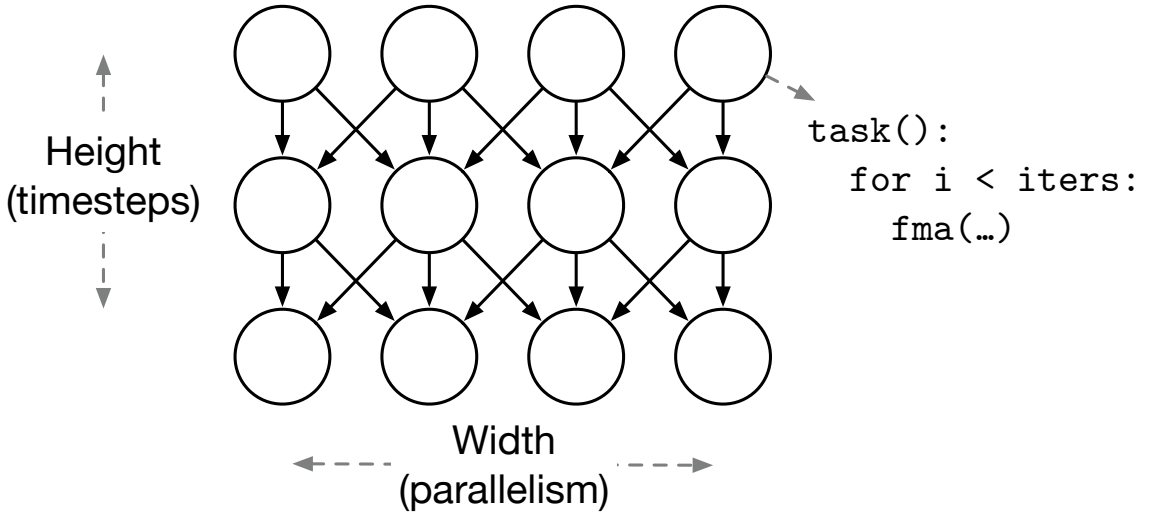


Figure 6.10: Task Bench benchmark structure.

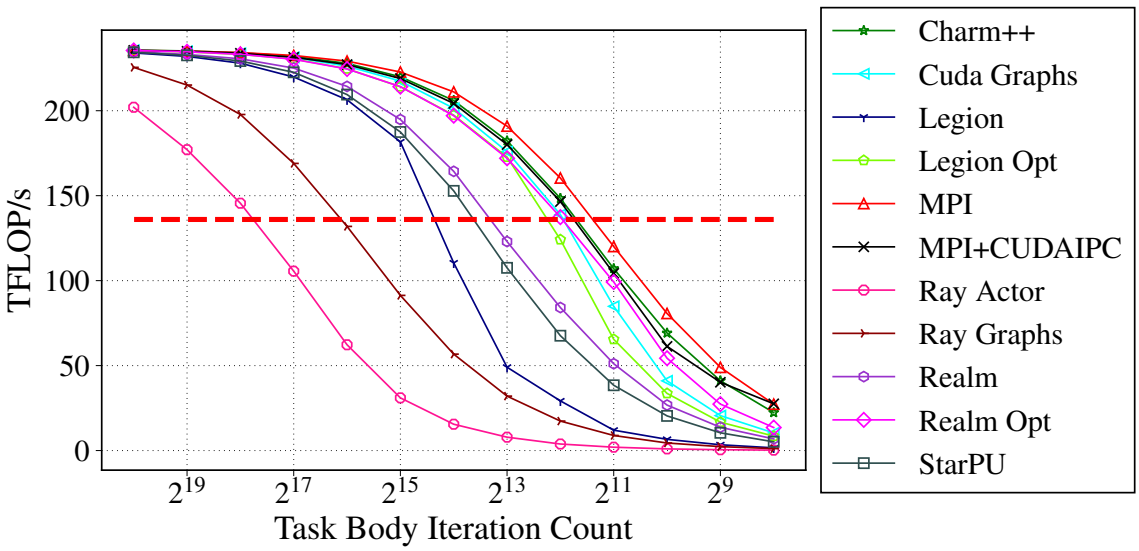


Figure 6.11: FLOPs achieved on 1 node by each system on a stencil Task Bench graph with width 8.

and synchronization.

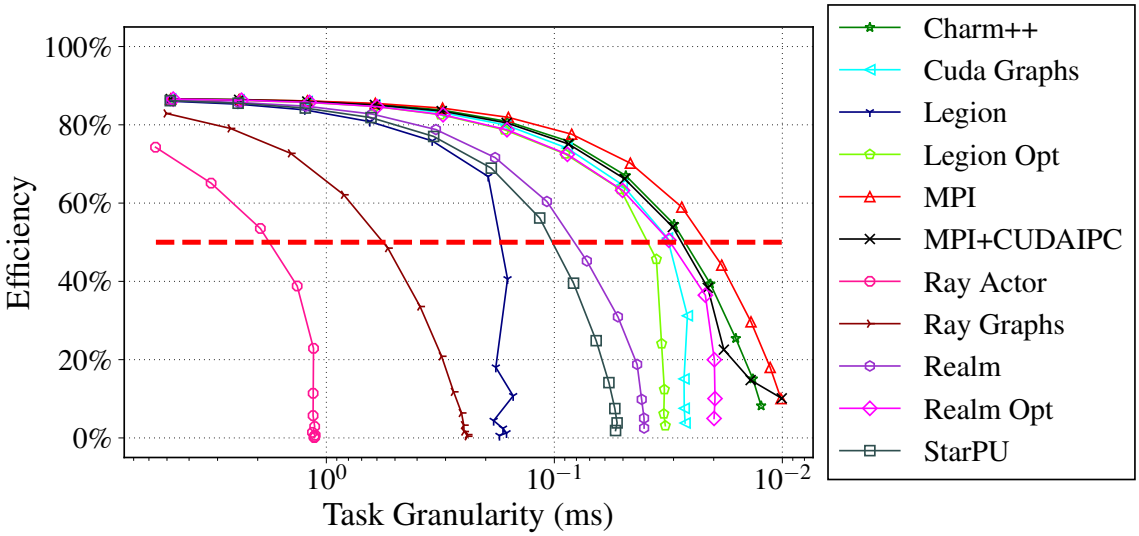
We optimize Legion and Realm with the techniques in this chapter and refer to the optimized implementations as Legion Opt and Realm Opt. Our implementations of Legion Opt and Realm Opt use the compilation algorithms in Section 6.4.2 and Section 6.4.3 to bypass Realm's existing execution path and target actor threads that communicate directly with active messages and atomic operations. We compare Legion Opt and Realm Opt against the well-known HPC actor systems Charm++ [78] and MPI [59], the implicitly-parallel tasking system StarPU [10], and the popular task and actor system from the ML community, Ray [92]. On a single node, we also compare against CUDA Graphs and an MPI implementation that uses CUDA IPC Events to lower GPU synchronization overheads within a single node. When possible, implementations for each system were taken as-is from the previously published Task Bench implementations, or adapted to utilize GPUs. We do not compare against serverless or cloud systems such as Durable Functions [38, 37]. These systems make different design decisions for features like fault-resilience, process isolation and elasticity, and consequently incur several orders of magnitude higher overheads than any system measured in our experiments. LegionOpt, RealmOpt and the selected comparison systems do not consider these axes, and can thus operate at significantly lower overheads. Ray supports some of these features (fault-resilience and elasticity), and the ramifications of doing so can be seen in the overhead results.

Single-Node Experiments

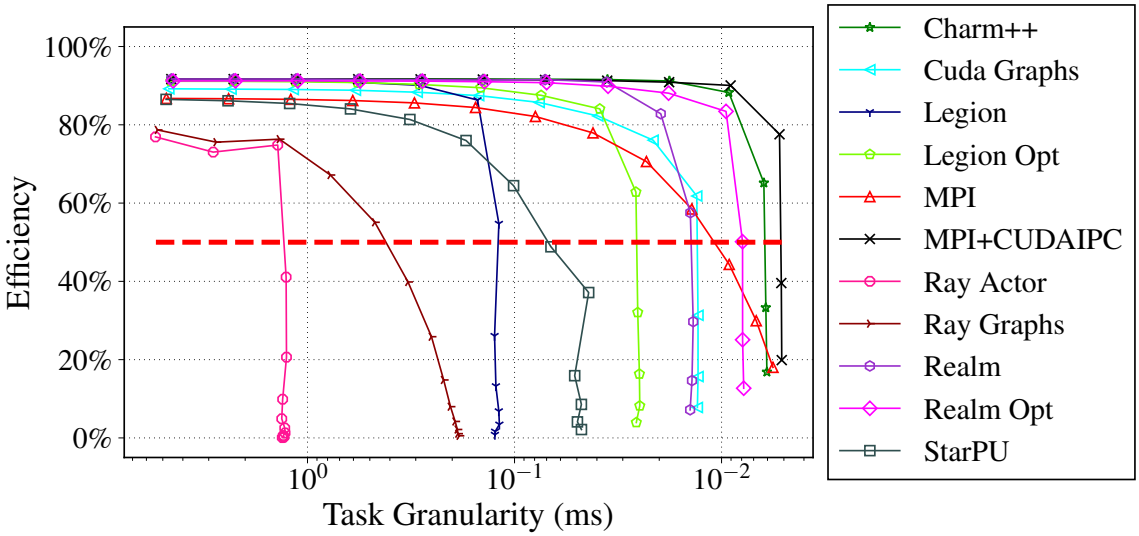
Figure 6.12a presents curves that plot task granularity against the efficiency achieved by various systems on a single node with a stencil task graph of width 8. These curves are derived from Figure 6.11, where the x-axis now corresponds to the runtime in milliseconds of each Task Bench task, instead of the number of kernel iterations. We see three distinct groups of systems, in order of increasing METG: actor and compiled task-based systems, standard task-based systems, and Ray. This configuration's METG is mostly determined by how fast each system can issue CUDA kernels. The actor and compiled task-based systems achieve METG(50)s between 22us-39us (MPI and Legion Opt, respectively). These systems are also competitive with CUDA

Graphs; since we expect CUDA graphs to be an efficient way of launching kernels, this demonstrates we are achieving high absolute efficiency. CUDA Graphs performed worse than we expected, as it could bypass kernel submission overheads by operating within the CUDA driver. The standard task-based systems (Legion, Realm and StarPU) accumulate overheads from numerous sources, achieving METG(50)s between 83us-173us. The final system is Ray, where we report results using Ray's actors as well as Ray's graph compiler [11]. We initially developed an implementation purely using Ray's tasking interface, but found the performance to be too poor to easily visualize with the other systems; each Ray task created a new process, causing re-initialization of structures like the CUDA runtime, requiring task granularity of at least hundreds of milliseconds. Ray Actors achieve a METG(50) of 1.8ms, which is improved by compilation to 570us, both larger than HPC tasking systems. These overheads are likely due to Ray supporting features such as resilience and elasticity.

Figure 6.12b contains single-node results with a graph of width 32, exposing 4-way task-parallelism on each GPU and offering an opportunity to hide latency. With task parallelism, the METG(50) of HPC systems falls into the single microseconds, where MPI+CUDAIPC, Charm++ and Realm Opt achieve METG(50)s of 5.1us, 6.1us and 7.9us respectively. Other systems fall off earlier for different reasons. The vanilla MPI implementation is written in a bulk-synchronous style, unable to perform fine-grained interleaving of CUDA kernels that the MPI+CUDAIPC implementation can. Legion Opt achieves a METG(50) of 25us (a 4.6x improvement over Legion) despite Realm opt achieving a METG(50) of 7.9us. This overhead originates from within task execution itself: for correctness reasons, Legion tasks always query the runtime system for pointers to the task's data in case different mapping decisions for a task's data have been made from one iteration to the next. These queries took roughly 20us in aggregate to complete, placing a floor on the smallest tasks that Legion can execute. In contrast, the Task Bench implementations in lower level systems like Realm (or Charm++ and MPI) preallocate all necessary data, and pass direct pointers into task invocations to avoid overheads.



(a) 1 node (8 GPUs), width 8



(b) 1 node (8 GPUs), width 32

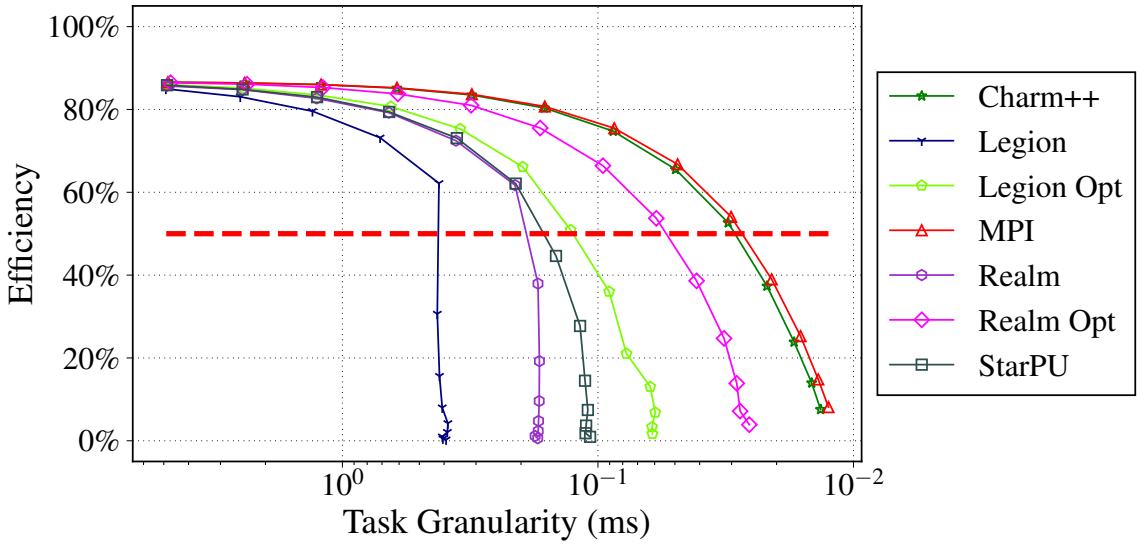
Figure 6.12: Task Bench METG curves of different systems on stencil task graphs on 1 node.

Multi-Node Experiments

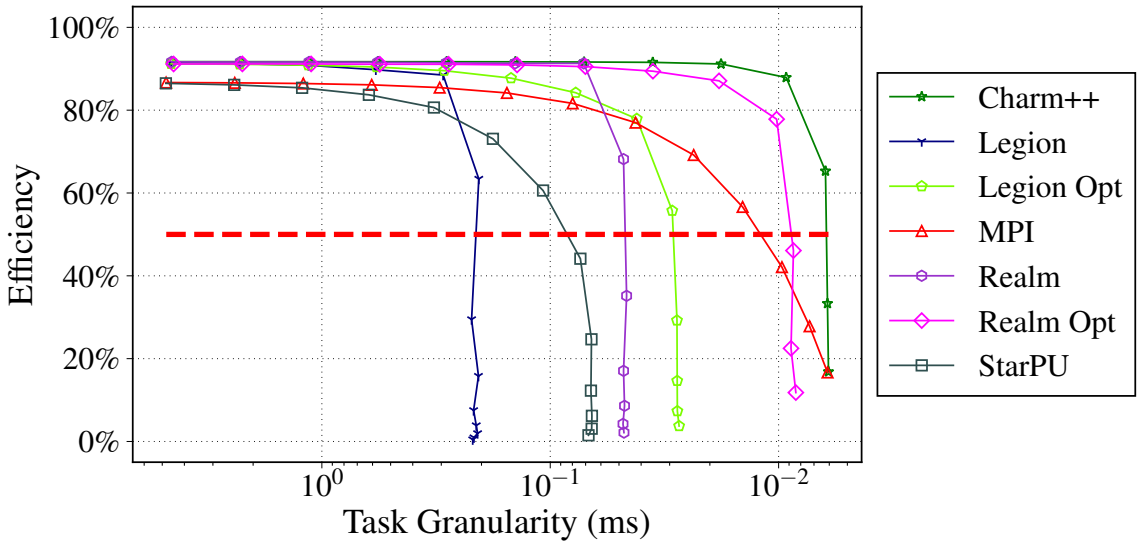
We now move to multiple nodes, where Figure 6.13a contains results on 4 nodes using a 32-wide graph (1 task per GPU). Like Figure 6.12a, this configuration is latency constrained, without parallelism to hide communication costs. Charm++ and MPI perform the best, achieving METG(50)s of 29us and 27us respectively. Realm Opt achieves a METG(50) of 54us, 3.5x better than standard Realm. The performance difference between Realm Opt and Charm++/MPI arises from the tradeoff discussed in Section 6.4.5: the Realm Task Bench implementation is developed in a sharded style for scalability, where each node constructs a local graph and connects to other nodes with Realm's standard dependence infrastructure. While still making a single network round-trip, the dependence infrastructure has higher overhead than a Charm++/MPI message handler. Legion Opt achieves an METG(50) of 125us, 2.5x better than standard Legion. While Ray is a distributed runtime system, we did not evaluate it due to its lack of competitiveness on a single node.

The final Task Bench experiment is in Figure 6.13b, a 4-node configuration with 4-way task parallelism on each GPU. The results are similar to Figure 6.12b, where Charm++ achieves a METG(50) of 6.1us and Realm Opt achieves a METG(50) of 8.8us (a 5.2x improvement over standard Realm); the bulk-synchronous MPI implementation degrades in performance earlier. With some task-parallelism to exploit, Realm Opt is able to close the performance difference experienced in Figure 6.13a. Legion Opt supports a similar METG(50) as on a single node (28us), running into the same intra-task overheads, but achieving a 7.1x improvement over standard Legion.

Our experiments show that our work dramatically lowers the overheads imposed by both explicitly-parallel and implicitly-parallel task-based systems, improving the METG(50) of both Legion and Realm by between 3.3x-7.1x and 1.77x-5.3x respectively. By transforming critical subsets of task-based programs into actors, we recover a significant amount of the performance difference with native actor programs, and deliver competitive runtime system overheads not previously achieved by existing task-based systems.



(a) 4 nodes (32 GPUs), width 32



(b) 4 nodes (32 GPUs), width 128

Figure 6.13: Task Bench METG curves of different systems on stencil task graphs on 4 nodes.

6.5.2 Strong Scaling Implicit Parallelism

We now demonstrate that these reduced overheads yield end-to-end strong-scaling improvements for applications developed in implicitly-parallel task-based systems. In each application, Legion is already performing tracing to eliminate dynamic dependence analysis overheads — the remaining difference is the efficiency of the traced task graph's execution. Each benchmark application has been heavily optimized separately from this work, and many have appeared in existing publications [115, 23, 87]. We also note that while strong-scaling studies tend to present very large problem sizes (often not even fitting into a single GPU memory), our experiments instead start scaling at a modest problem size, filling roughly half the memory of just one H100 GPU. The time spent in graph compilation was negligible, less than 50ms on every node.

Stencil

The smallest application is a stencil benchmark from the Parallel Research Kernels [126], with results in Figure 6.14a. The benchmark performs a radius-2 star-shaped stencil over a two-dimensional grid. The benchmark has no task parallelism, and is thus sensitive to any latencies when launching kernels or performing the halo exchange at grid boundaries. As a result, Legion Opt is able to continue improving performance after 16 GPUs while standard Legion falls over, resulting in an improvement of 3.4x at 32 GPUs.

MiniAero

MiniAero is a 3D unstructured mesh proxy application from the Mantevo suite [47] that implements an explicit solver for the compressible Navier-Stokes equations, with results in Figure 6.14b. Unlike Stencil, MiniAero has many opportunities to exploit task parallelism, benefiting from lower overheads to enable more utilization of the GPUs. Similarly to the performance of Stencil, standard Legion falls off after 16 GPUs, while Legion Opt continues scaling to 32 GPUs, ultimately achieving a 4.7x improvement. We attempted to compare against the reference implementation, but found that it depended on unsupported versions of Kokkos, and did not run when

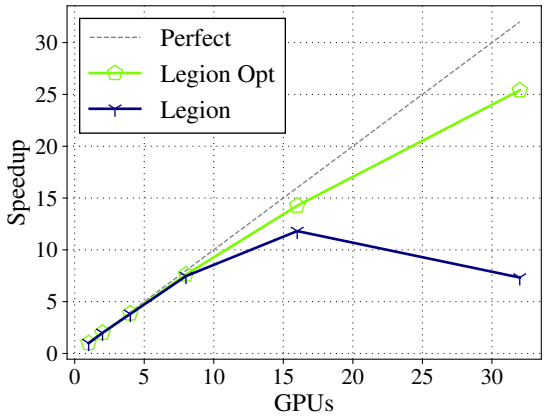
ported to a newer version.

PENNANT

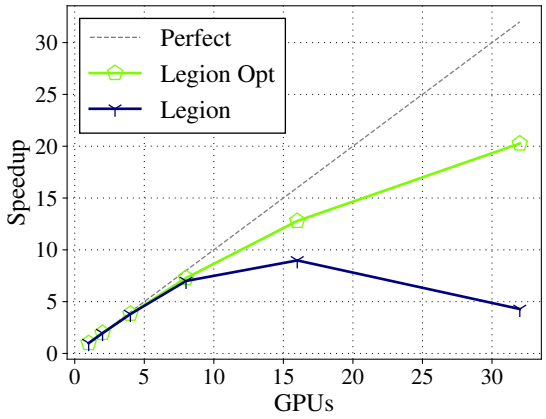
PENNANT is a 2D unstructured mesh proxy application simulating Lagrangian hydrodynamics [58], with results in Figure 6.14c. The PENNANT main loop additionally contains some all-reduce operations that cannot be hidden by parallel work, and thus are exposed and affect the total speedup achievable. As part of graph compilation, we pre-plan and optimize copies present in the task graph, lowering the latency of small copy operations, which improves the all-reduce performance. The standard Legion implementation falls over at 8 GPUs, while Legion Opt continually delivers speedup up to 32 GPUs, achieving a 5.0x improvement. The reference PENNANT GPU implementation only runs on a single GPU, and employs optimizations that make it not directly comparable to a multi-GPU implementation.

Conjugate Gradient

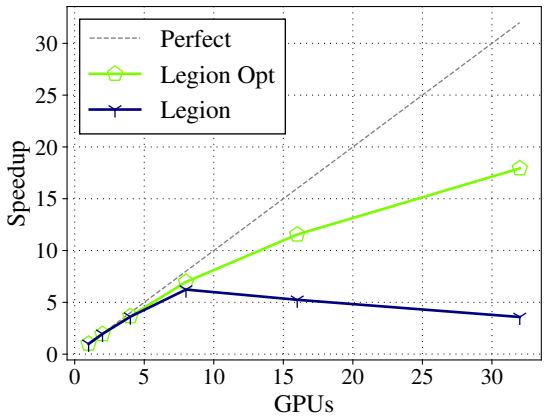
Our final benchmark is a conjugate-gradient (CG) solver for a 1-D Poisson problem, with scaling shown in Figure 6.14d. We additionally compare against a baseline implementation developed using PETSc [13], an industry-standard sparse linear algebra library developed with MPI. Legion Opt, standard Legion and PETSc all strong scale well, with PETSc doing the best, and Legion Opt improving upon the scalability of Legion. Legion Opt achieves 88% of PETSc's performance at 32 GPUs, while standard Legion only achieves 67%. The conjugate gradient application has very little task-parallelism, similar to Figure 6.13a. In the 4-node Task Bench configuration without task-parallelism, Legion Opt's METG(50) was 125us, which the average task duration of CG at 32 GPUs approaches, corresponding to the beginnings of the performance difference against PETSc.



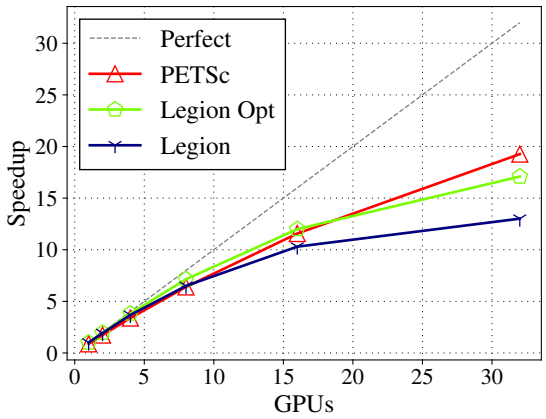
(a) 2D Stencil



(b) MiniAero



(c) PENNANT



(d) Conjugate Gradient

Figure 6.14: Strong scaling performance of end-to-end Legion applications (higher is better).

6.6 Conclusion

This chapter is the final component describing the contributions of this thesis to the Legate runtime system. It discusses optimizations within the Realm runtime system to execute task-based computations at overheads comparable with the best "bare-metal" systems that are used by experts who want complete control over execution. This chapter argues that these optimizations are not ad-hoc, but can be explained by understanding the relationship of task-based models to actor-based models, which the "bare-metal" systems can often be characterized as. I argue that this connection is a duality, where programs in task-based programming models and actor-based programming models can be translated back and forth, and the optimization proposed is just one efficient translation strategy. By showing that task-based models can execute computations with overheads comparable to the best actor-based models, we show a path for Legate programs to perform as efficiently as the best hand-tuned applications on both large-scale and small-scale problems.

Chapter 7

Implementing a Legate Library

The previous chapters of this thesis described the architecture of the overall Legate runtime system, and described several key program representations and optimization strategies within Legate to enable the composition of independent modules with high performance. In this chapter, we move from the internals of Legate to focus on the design and implementation of Legate Sparse [141], a prototypical Legate library. We describe how the architecture of Legate allows for the implementation of Legate Sparse to be independent of the context in which Legate Sparse is used; even individual operations within Legate Sparse are developed without consideration for execution strategies in other parts of the library. We do not include direct evaluation results for Legate Sparse in this chapter, as we used Legate Sparse to develop benchmark applications that were evaluated and discussed in Chapters 3 and 4.

7.1 SciPy Sparse

Legate Sparse attempts to provide a distributed, drop-in replacement for the SciPy Sparse library. SciPy Sparse [128, 12] is a sub-module of the SciPy Python library that provides a high-level API for linear algebra operations over different types of sparse matrices. SciPy Sparse supports several common sparse matrix formats, including the CSR (compressed sparse rows), CSC (compressed sparse columns), DIA (diagonal)

and COO (coordinate) formats, and supports format conversions and data reorganization operations between these formats. On these sparse matrices, SciPy Sparse supports a variety of basic mathematical operations, such as matrix-vector products, matrix-matrix products and diagonal computations, as well as higher-level linear algebra operations like iterative solves and eigenvalue computations. SciPy Sparse is directly composable with NumPy, as many operations within the API natively accept and return NumPy arrays. The standard implementation of SciPy implements the API with a combination of calling out to C operations and utilizing existing NumPy routines. Finally, the SciPy Sparse API has no notions of execution or distribution strategies, meaning that all parallelism performed by Legate Sparse must be implicit.

7.2 Sparse Data Representation

The standard single-node representations of common sparse matrix formats store metadata about the indices of non-zero matrix entries and their values in packs of arrays. For example, the COO (coordinate) format stores three arrays, where the first two arrays store the row coordinate and column coordinate of each non-zero entry of the matrix, and the last array stores the value. The CSR (compressed sparse rows) format further compresses the COO format by implicitly representing the rows that contain non-zero entries: it maintains an array (often called `pos` or `indptr`) where the column coordinates and values for row i are stored within range $[\texttt{pos}[i], \texttt{pos}[i+1])$ of an array called `crd`. The CSC format is similar to CSR, but compresses the columns instead of the rows.

We use Legate stores to extend these single-node representations into distributed sparse matrix data structures by mapping each of the arrays used to represent sparse matrices to stores. For instance, the row, column and value arrays in the COO format are represented directly as stores in Legate Sparse. Formats such as CSR and CSC are represented in a similar manner, but store the range of coordinates for a row or column i in a tuple at `pos[i]`, as depicted in Figure 7.1, which allow for tighter interaction with Legate's partitioning constraints (discussed more in Section 7.2.1).

Our decision to represent sparse matrices as a set of stores instead of a collection

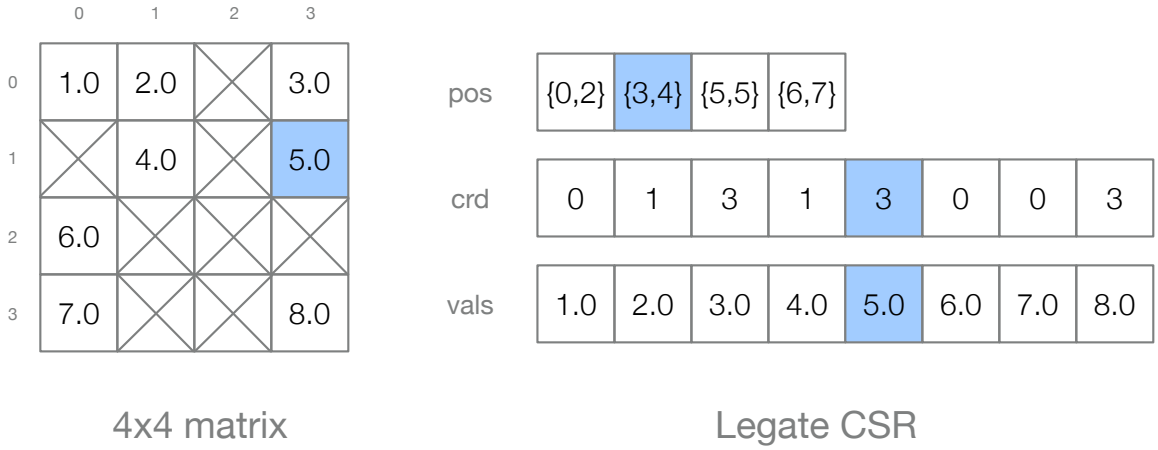


Figure 7.1: Legate Sparse's CSR sparse matrix encoding.

of local sparse matrices per rank (as used by PETSc and Trilinos) has both benefits and downsides. Using a set of stores aligns with the Legate programming model, and careful choices of partitioning enable the description of non-trivial communication patterns. Additionally, this choice allows for inter-operation with other Legate libraries: since sparse matrices are constructed from stores, users of Legate Sparse can directly construct sparse matrices out of cuPyNumeric arrays, or extract and operate on the arrays that back a sparse matrix. A downside of this decision is that the partitioned pieces of the global sparse matrix passed to individual tasks are not valid sparse matrices from the perspective of external libraries like cuSPARSE. As a result, we pay a small performance penalty when reshaping these local pieces into formats accepted by these libraries when we use them. Previously discussed experiments with Legate Sparse (Chapters 3 and 4) show that this choice of sparse matrix representation has low overhead while allowing for direct use with Legate and close alignment with SciPy Sparse's programming model.

7.2.1 Partitioning Constraint Interaction

The small variation from the standard representation of sparse matrices allows us to directly employ Legate's image constraints to relate partitions of the `pos` and `crd` stores with one another. We also use images to relate partitions of the `crd` store with referenced indices in dense vectors and matrices. For example, consider a distributed

SpMV ($y = A \cdot x$), where A is stored as CSR. Performing an SpMV requires accessing the locations in x corresponding to the non-zero coordinates stored in A's `crd` region. We use an image from the partition of A's `crd` region to compute the referenced locations of x. Images allow for the co-partitioning of the regions used to define sparse data structures, and to implement the communication usually described by MPI-like scatter/gather operations in a higher-level manner.

A concrete example of a task implementation using the Legate constraint interface is shown in Figure 7.2, which implements a row-based distributed sparse matrix-vector multiplication (SpMV). Upon execution of the task launching code in Figure 7.2, the `task` object contains the following partitioning constraints: `equals(y, pos)`, `image(pos, crd)`, `image(pos, vals)`, and `image(crd, x)`. The constraints require `y` and `pos` to be partitioned in the same way, as each element of `y` and `pos` correspond to a row of the matrix `A`. Then, an image relates `pos` with `crd` and `vals`, as the range of coordinates and values stored for each row is represented by `pos`. Finally, the coordinates of `x` that each row of `A` will gather from are stored in `crd`; this means another image is required to link `crd` and `x`.

When Legate solves these constraints, it realizes that the choices of partitions for `y` and `pos` are independent, while the partitions for `crd`, `vals` and `x` are dependent on choices for partitions for other regions. Then, Legate examines the existing partitions for `y` and `pos` and selects the existing partitions if they are aligned. Otherwise, it selects an existing partition that keeps the sparse matrix in place. Once these initial partitions have been selected, Legate invokes Legion's image operation to construct partitions of `crd`, `vals` and `x` to satisfy the remaining constraints.

This implementation of SpMV in Legate Sparse has several attractive properties. First, it does not require a specific data partitioning stored within Legate Sparse; an arbitrary partitioning of `A` may be used, perhaps created by a graph partitioning framework before the invocation of SpMV, and the implementation will adapt to the existing distributed layout. Next, the implementation does not contain explicit communication that assumes distributed data layouts for the input or output vectors, allowing for data movement required at the library boundary with cuPyNumeric to be implicitly discovered by Legate. Legate can discover the required communication,

```
def spmv(self, A, x):
  # Compute y = A @ x and return y.
  y = cupynumeric.zeros(A.shape[0])
  task = ctx.create_task(ROW_SPLIT_SPMV)
  # Add all regions to the task.
  task.add_output(y)
  task.add_input(A.pos, A.crd, A.vals, x)
  # Describe partitioning constraints.
  task.add_alignment_constraint(y, A.pos)
  task.add_image_constraint(A.pos, [A.crd, A.vals])
  task.add_image_constraint(A.crd, x)
  task.execute()
  return y
```

Figure 7.2: Python implementation of a row-based distributed CSR SpMV (adapted from DISTAL [136] generated code).

and dispatch to the correct APIs for inter-node and intra-node data movement, along with discovering the communication partners required for the sparse communication of `x`.

7.3 Library Kernel Implementation

Having described the abstractions with which distributed operations are defined in Legate Sparse, we now discuss our process for implementing the SciPy Sparse API. Our prototype supports the COO, CSR, CSC and DIA sparse matrix formats, and of the estimated 492 functions in SciPy Sparse, our prototype implements 176 (35%) functions; 14 were implemented by using the DISTAL compiler, 156 were ported from existing SciPy or CuPy implementations, and 6 had to be handwritten. In this section, we discuss these three cases, as well as the portions of the API that we have not yet implemented.

Since the original development of Legate Sparse in 2022-2023, it has transitioned into an engineering-maintained project at NVIDIA. Subsets of the implementation have been adapted to support requested optimizations, removed to control the exposed library surface area, or new components have been added based on customer feedback. We discuss here the status and development strategy of the original prototype of Legate Sparse.

7.3.1 Generating Kernels with DISTAL

We used the DISTAL [136, 137] compiler (work done in PhD that is not part of this thesis) to generate implementations for components of the SciPy Sparse API that perform tensor algebraic computations. These functions are performance critical (such as SpMV or SpMM), and require custom code tailored to the specific operation, sparse matrix formats and target hardware. This custom code is tedious and difficult to write; despite DISTAL being used to generate implementations of only 14 functions in the SciPy Sparse API, the generated code accounts for 46% of the total C++ and CUDA in Legate Sparse (2854/6135 LOC) and 12% of the total Python in Legate Sparse (697/5748 LOC). By generating this performance sensitive code, we enhance the maintainability of Legate Sparse, and allowed developer time to be spent elsewhere when optimizing the library. We first give an overview of DISTAL, and then describe how it was used to generate code for Legate Sparse.

DISTAL compiles a tensor algebra domain specific language (DSL) into C++ code that directly targets the Legion runtime. DISTAL allows for the separate specification of 1) the desired tensor computation, 2) the sparse data format of each operand, 3) the distributed algorithm to use, and 4) the data distribution of the operands. This flexibility allows for the high level description of many kernels of interest within SciPy Sparse. Since Legate's constraint solver already considers the existing data distributions of regions, we only use the first 3 input languages of DISTAL. DISTAL generates code directly targeting the Legion API, so we perform slight manual modifications to the generated code to target our higher-level abstractions; these changes could be automated (in the modern day with an LLM coding assistant), but we have not found the manual work to be burdensome for developing our prototype.

DISTAL code to generate a distributed and multi-threaded CPU SpMV is found in Figure 7.3, the generated C++ task body is found in Figure 7.4, and the constraint based task launching code in Figure 7.2 is the result of adapting DISTAL-generated C++ task launching code. The DISTAL C++ code declares some runtime parameters, initializes the tensor operands, describes the desired computation, and then schedules the computation for the target machine. The algorithm specified by the scheduling language distributes the rows of the matrix across all processors, and then

```
// Runtime parameters: input sizes and processors.
Param n, m, procs;
// Define the tensor operators.
Tensor<double> y({n}, {Dense}), x({m}, {Dense});
Tensor<double> A({n, m}, {Dense, Compressed});
// Describe the desired computation.
IndexVar i, j, io, ii;
y(i) = A(i, j) * x(j);
// Schedule the computation.
DISTAL::compile(y.schedule()
                  .divide(i, io, ii, procs)
                  .distribute(io)
                  .communicate(io, {y, A, x})
                  .parallelize(ii, CPUThread));
```

Figure 7.3: Distributed, multi-threaded CSR SpMV in DISTAL.

```
void CSRSpMVTask(vector<Store> stores) {
  auto y = stores[0];
  auto pos = stores[1];
  auto crd = stores[2];
  auto vals = stores[3];
  auto x = stores[4];
  #pragma omp parallel for
  for (int i = y.bounds.lo; i <= y.bounds.hi; i++) {
    auto val = 0.0;
    for (int jA = pos[i].lo; jA <= pos[i].hi; jA++) {
      val += vals[jA] * x[crd[jA]];
    }
    y[i] = val;
  }
}
```

Figure 7.4: DISTAL-generated C++ task for row-based, multi-threaded CSR SpMV, with minor modifications.

parallelizes execution across the rows between CPU threads. To achieve peak performance on GPUs, we hand-modified the DISTAL-generated CUDA code to make calls into cuSPARSE when applicable. In our experience, this aspect was the most error prone step in developing the sparse linear algebra kernel implementations and could be made easier in the future with better compiler support for external library interaction, such as in the Mosaic system [15].

7.3.2 Porting SciPy and CuPy Implementations

The largest subset (156/176 functions) of our implementation of SciPy Sparse was done by porting existing implementations of the API in SciPy and CuPy. While developing Legate Sparse, we found that many functions in SciPy Sparse were implemented using parallel NumPy operations and previously defined SciPy Sparse kernels.

By leveraging the composability that the Legate ecosystem offers, we were able to bootstrap our library with itself and cuPyNumeric to obtain distributed and accelerated implementations of these functions without any distributed programming.

The classes of functions that we were able to directly port varied in complexity. The simplest of these functions were non-zero preserving, element-wise, unary operations on sparse matrices that are implemented by using the corresponding NumPy operation on the array storing the values of the sparse matrix. Some more complicated ported functions include computing sums across different axes of sparse matrices, and format conversions between sparse matrix formats. The most complex operations that we directly ported to Legate Sparse were higher-level operations such as solves and integrations. We ported several iterative linear solvers (CG, CGS, BiCG, BiCGSTAB, GMRES), Runge-Kutta integration and eigensolvers from SciPy and CuPy implementations to distributed implementations using Legate Sparse and cuPyNumeric.

7.3.3 Hand-Written Implementations

The final group of functions in Legate Sparse were those that required completely hand-written implementations. These functions include sorts and auxiliary operations that are implemented within SciPy with calls to C/C++ or Python loops that directly index into NumPy arrays. For these operations, we developed distributed and accelerated implementations using Legate's constraint-based parallelism framework paired with C/C++ and CUDA code for tasks adapted from the SciPy implementations.

7.3.4 Unimplemented Components

Having covered how we implemented components of SciPy Sparse, we now discuss the remaining portions of the API and the path forward to implementing them. Out of the 316 remaining functions in SciPy Sparse, 116 are defined on sequential matrix formats (list-of-lists and dictionary-of-keys) used for matrix assembly in shared memory, which we do not plan to support. 72 of the remaining 200 functions are defined on the BSR (block sparse rows) sparse matrix format, which we plan to support, and are able

to use DISTAL to generate kernels for. This leaves 128 functions in SciPy Sparse defined on sparse matrix formats that we support in Legate Sparse (CSR, CSC, DIA, COO). Of these functions, we believe there is a path forward to a nearly complete implementation: 8 are possible to generate with DISTAL, 44 are possible to port from SciPy, 60 require a combination of porting and hand-writing, and 14 are specific to SciPy's implementation. The functions that require hand-writing cover different components of the API, including sparse matrix reshaping operators, operators that slice and update pieces of sparse matrices, and functions that call to external libraries like SuperLU.

7.4 Conclusion

This chapter discusses the implementation of Legate Sparse, and shows what a realistic library that leverages the Legate programming model and runtime system actually looks like. Legate's abstractions enabled the development of Legate Sparse to focus on domain-specific concerns, such as actually building individual sparse matrix operations, instead of handling interactions with external libraries and unrelated components within the library. The composability offered by Legate allowed Legate Sparse to quickly bootstrap itself with cuPyNumeric to get large swaths of functionality almost "for free" once a core subset of the library had been implemented. The design of the user-facing Legate programming model enables Legate library developers to be logically isolated from external factors, deferring the responsibility of efficient and correct composition to the Legate runtime.

Chapter 8

Related Work

In this thesis, I described the Legate programming model and multi-layered runtime system. The Legate programming model, runtime system, and analysis pipeline are co-designed to enable applications built from independent distributed and accelerated libraries to achieve high performance. I now discuss the larger body of work that Legate lives within, discussing related programming models and program analysis techniques for high performance parallel programming, as well as other work that aims to enable composable parallel programming.

8.1 Legion and Realm

This thesis focused on the Legate programming system, which contained layers above the Legion and Realm runtime systems, as well as extensions to the Legion and Realm runtime systems that enable the development of independent composable libraries. There is a long history of work building program analysis and optimization techniques that enable high performance within the Legion and Realm runtime systems, as well as research directions that have impacted the design of the Legate runtime system.

Legion's implicitly-parallel programming model, which separates the logical description of the computation from the mechanism of how that computation is realized, has been refined over many years. The original Legion programming model proposed first-class regions, partitions of those regions, and tasks that operated over regions

and the sub-regions derived from partitions [25]. This model was then extended to include *fields*, where logical regions could contain compound data types with multiple logical components [26]. The introduction of first-class fields allowed for more efficient runtime analysis and finer-grained control over data movement and physical data layout. Partitioning in Legion was originally done by directly constructing the sets of points that describe the data in each element of the partition. The dependent partitioning framework [124] introduced a high-level language to describe partitions compactly and construct them efficiently. The final major extension to the Legion programming model was the introduction of *index launches*, which issued a group of parallel tasks as a single operation. Index launches captured common paradigms in distributed programming, and their efficient representation allowed for Legion to analyze them significantly more efficiently than as individual tasks.

Legion's dependence analysis extracts parallelism and inserts the necessary communication and dependencies required to maintain a sequential semantics of the input implicitly-parallel program. The techniques to perform this dependence analysis in an efficient and scalable manner are critical components of Legion and enable the Legion ecosystem above Legion to exist. The dependence analysis infrastructure in the Legion runtime system bears a strong resemblance to the hardware implementation of an out-of-order processor [25, 20]. Analysis for each task is broken up into multiple stages that are executed in a pipelined manner, allowing for the overlap of the different stages and the hiding of end-to-end analysis latency. A variety of algorithms have been employed within the history of Legion to actually discover dependencies between tasks based on the data each task uses [24]. These algorithms were found to have a surprising connection to the visibility problem in computer graphics: identifying what version of data should be visible to a task can be reduced to the problem of identifying what objects in a 3D scene are visible from the camera when rendering the scene as a 2D image. As discussed in Chapter 5, dynamic tracing is a memoization technique used within Legion to eliminate the cost of this dependence analysis for iterative programs.

Regent is a higher-level programming language that embeds the Legion programming model into a language as opposed to being a C++ library [114]. Regent is

able to statically check many of the requirements and invariants required of the Legion programming model and performs many optimizations that programmers directly targeting Legion have to do explicitly. Regent combines this analysis with Terra-based meta-programming [54] to enable a productive environment to develop high-performance task-based programs. A static version of Legate's constraint-based partitioning analysis was originally developed within Regent [86]. Regent, like Legate, aims to make it easier to program high-performance distributed machines. However, Regent focuses on automating components of using the Legion programming model, while Legate instead focuses on techniques to enable the composition of independent distributed software. Components of this thesis that improve parts of Legion and Realm (Chapters 5 and 6) can directly benefit existing Regent applications.

A key innovation in the Legion ecosystem is *control replication*, which is a program analysis that enables the scalability of implicitly-parallel programming models to large node counts. In standard task-based parallel programming, a single leader node launches tasks to create work for the entire distributed machine. As the scale of the target machine increases, the cost of creating and analyzing these tasks on a single node results in poor scalability, as the single leader node falls behind. Control replication distributes the creation of work itself and the dependence analysis required across all nodes in the target machine, enabling scalability comparable to hand-tuned MPI programs. Control replication was first implemented as a static analysis within Regent [115], before being extended to a dynamic program analysis automatically performed by Legion [23]. Legate relies on dynamic control replication to automatically scale Python programs to large node counts.

Realm [122] is an explicitly-parallel runtime system targeted by Legion. The results of Legion's dependence analysis are explicitly-parallel task graphs; Realm executes these task graphs on the target machine, leveraging the available hardware and network interfaces. The key innovation of Realm was *composable asynchrony*, which made every operation exposed by the Realm explicitly asynchronous. Invocations of asynchronous operations return *event* structures which are chained as preconditions into future operation launches. Realm's focus on asynchrony enables portability across and resilience against the highly variable latencies of communication across

modern hierarchical networks. Realm's implementation is designed to be extensible and portable across a wide variety of hardware; for more discussion on the concrete design elements, I defer interested readers to Sean Treichler's thesis [121].

8.2 Python at Scale

Python is currently the most popular language for scientists due to its simplicity and wide ecosystem of high-level libraries. Developing techniques to accelerate and scale these Python codes is an effective way to enable non-expert users to achieve high performance at scale.

8.2.1 Scalable Implementations of Python Libraries

A variety of systems have taken a similar approach to Legate by providing distributed drop-in replacements of popular Python libraries. I will discuss a few of these systems, but the main difference between them and our work on Legate is that these systems offer implementations of individual Python libraries. These implementations are bespoke and are not built with composability with external libraries as a first-class principle. As such, users of these individual systems can be productive and achieve high performance, but cannot compose these systems with other libraries and continue to achieve high performance. I view these systems as effective point solutions, but they do not offer a path towards building a larger ecosystem of high-performance Python libraries that mirrors what we have in the sequential world.

Arkouda [90] and Num-S [57] are distributed implementations of NumPy, similar to cuPyNumeric. Arkouda is implemented with the Chapel [40] programming system, and mostly has demonstrated results on clusters of CPUs. Similarly, Num-S is a distributed NumPy implementation based on the Ray runtime system (discussed in Section 8.4) that has also demonstrated CPU-only results. There are a few capabilities that set cuPyNumeric and Legate apart from these systems. First, efficiently supporting distributed GPU execution is a significant technical lift over just supporting clusters of CPUs. Second, Legate's data model allows for rich partitions and

references to arbitrary subsets of stores, which allows cuPyNumeric to support slicing and mutation of NumPy arrays (and their slices). I am not aware of other systems with similar capabilities.

Modin [101] is a distributed implementation of Pandas [132] (using Ray [92]) that scales dataframe workloads across multi-core CPUs and clusters of CPUs. I believe recent extensions of Modin also support GPU execution. Modin is more fully featured than the prototype work on Legate DataFrame [105], but the core difference remains that Modin is a stand-alone project. Composing Modin code with other distributed implementations of libraries in the Python ecosystem is unlikely to achieve high performance.

8.2.2 Machine Learning Systems

Jax [35], PyTorch [100, 6], and TensorFlow [2] are machine-learning systems that compile NumPy-like descriptions of neural networks to perform optimizations And transformations like fusion, distribution, and automatic differentiation. These machine learning systems are enormously popular and impactful, and have enabled machine learning researchers with little background in high-performance programming to efficiently train and serve neural networks. These systems heavily leverage domain-specific knowledge about the properties of machine learning programs to drive their optimization and analysis. This domain-specific knowledge includes assumptions that the target programs have a small, fixed set of core operations, regular and statically-known communication patterns and no aliasing or mutation. As a result, optimizations employed by these machine learning systems can perform transformations far beyond what more general systems like Legate can do. In contrast, Legate can support a much wider class of programs than these machine learning systems are capable of expressing and optimizing. Legate supports programs with dynamic dependencies, dynamic and data-dependent control flow, and large amounts of mutation and aliasing, such as found in the scientific simulation and data analytics discussed earlier in this thesis. I consider Legate as a different point than these machine learning

systems in the design space of programming systems for distributed and heterogeneous machines, where Legate relinquishes some potential optimizations to support the composition of more general-purpose programs.

8.2.3 DaCe

A system that is similar to Legate in both vision and capability is DaCe [27]. DaCe accepts Python and NumPy programs and compiles them to high-performance multi-core CPU and GPU implementations through an intermediate representation called Stateful DataFlow MultiGraphs. DaCe supports automatic optimization of these graphs, as well as an interactive user interface to apply transformations to a visual representation of the graph. One of the core differences between DaCe and Legate is that DaCe is explicitly-parallel. DaCe programs include manual communication and synchronization using communication libraries like MPI, and the DaCe compiler mostly performs transformations on program fragments that execute between communication operations. However, some MPI support is built into the language, allowing for some kinds of transformations over distributed operations. These transformations over explicitly-parallel programs with communication require fundamentally different analysis algorithms than what are used in Legate, which instead comes at the problem with a different programming model.

8.3 Shared-Memory Composition

The study of composable parallel programming dates back to the 1970's with Jack Dennis's work on dataflow programming and parallel processor design to support modular parallel programs [53, 52]. Since then, advancements in multi-core processor architecture and scheduling technology (such as Cilk [30] and PThreads [93]) allow for modern software to correctly orchestrate parallel and concurrent modules within a single address space. A striking difference in modern computer architecture from the architectures studied by Dennis is the relative speeds of data movement and compute,

where executing computation on modern computer architecture is orders of magnitude cheaper and more efficient than moving data between memories. While modern shared-memory software can *correctly* orchestrate modular parallel components, significant efficiency can be lost at module boundaries, particularly for data-intensive parallel computations. The Weld [98] system leveraged a lazy execution strategy and a data-parallel intermediate representation to fuse NumPy, Pandas, and other common Python libraries into kernels that minimized data movement from main memory. Chapter 4 targets a similar goal for distributed programs. Split Annotations [99] was developed after Weld, attempting to solve a similar problem without requiring compiler support. Split Annotations marked library functions with partitioning annotations that described what components of the input data were read by a library function and which components of the output were produced; then, a runtime system reordered sub-components of library computations to operate on loaded data within caches instead of execution orderings that repeatedly read from main memory. Fern [16] built upon the ideas in Weld and Split Annotations to support a wider variety of partitioning structures, enabling a wider class of programs to benefit from this style of approach.

A large body of work leverages domain-knowledge to perform fusion-like optimizations that reduce the data movement in composed programs or to even reduce the asymptotic complexity of aggregate operations. Deforestation approaches aim to remove temporary lists and trees in functional programs [130]. Fusion in collection-oriented languages combines operations like map and reduce into single passes over data structures [133, 42, 36, 65, 63]. Various compilers have been developed to generate fused code for operations over dense [103, 43, 127] and sparse tensors [80, 28, 81]. Machine learning frameworks perform operator fusion within neural networks [73, 94, 109, 35, 88]. Polyhedral compiler analyses [4] perform powerful fusion and tiling of programs representable as affine loop nests; some polyhedral analyses are able to schedule communication between processes in distributed-memory environments [5].

While I stated previously that improvements in scheduling have enabled the correct composition of concurrent and parallel programs, recent work has investigated

the performance aspects of composing parallel programs that spawn and schedule threads internally. The USF [107] framework proposes a user-level process scheduling interface to enable applications that interact with multiple threading runtime systems to control the scheduling policies without requiring special permissions. This proposed control over scheduling allows for workloads with oversubscription as a result of module-local thread creation to achieve higher performance than with standard schedulers.

8.4 Task-based Programming Models

Task-based programming models have rapidly grown in popularity to target modern distributed and heterogeneous machines. A variety of task-based systems have been developed by the high-performance computing, cloud computing, and machine learning communities. Each community designs these systems with different priorities and driving applications. I will discuss some systems from each of these communities in-turn.

8.4.1 High Performance Computing

In the HPC community, several task-based programming systems were developed concurrently with Legion and Realm. The most similar to Legion is a system called StarPU [10], which exposes a region-like data model and organizes computations as tasks operating over this data. Early versions of StarPU required users to explicitly annotate dependencies between tasks, while later versions added an automatic dependence analysis similar to Legion. StarPU's analysis is weaker than Legion's, not supporting as flexible of a data model and requiring users to manually control replicate their programs to scale efficiently to multiple nodes. StarPU tries to hide mapping from the programmer, instead believing the runtime system should be responsible for mapping tasks to processors and data to memories. The result of this philosophy is that extensive research in the StarPU community has been focused towards developing scheduling algorithms to automate the mapping problem. CUDASTF [9] is a

successor to StarPU that adds native multi-GPU task-based programming support to the CUDA programming model. CUDASTF supports a similar data model to StarPU and allows for minimal code changes to leverage multiple GPUs on a single node in CUDA. PARSEC [34] is another task-based system from the HPC community that expresses task graphs in an algebraic language rather than the dynamic task streams used in Legion and StarPU. The structure required by this algebraic language means that the ParSEC community tends to focus on applications in dense linear algebra. HPX [75] is a task-based system built around an "Active Global Address Space" (AGAS) model, where distributed memory is globally addressable, and physical pages are migrated around the machine to increase locality. HPX combines the AGAS model with lightweight tasks to launch work on remote nodes and overlap the costs of remote data access and page migration with independent compute.

8.4.2 Cloud Computing

I discuss two classes of task-based systems found in the cloud computing community. The first is Spark [143] (and its underlying runtime system), a data analytics framework built to tackle the challenges of writing efficient data analytics computations on commodity clusters. Spark was built to replace MapReduce [50] by introducing the concept of *Resilient Distributed Datasets* (RDDs) that enable both fault tolerance and flexible computation on distributed sets of records. Spark efficiently supports coarse-grained parallel operations like maps, reductions and joins on RDDs, and checkpoints intermediate results so that data is not lost upon node failures. Higher-level query processing engines have been developed on top of these coarse-grained parallel operations in Spark [7]. Spark does not support workloads that perform fine-grained reads and writes to distributed data structures and only efficiently supports the dependence structures that arise from these coarse-grained parallel operations.

The second class of task-based programming systems in the cloud community are used in serverless workloads, either as an execution engine or programming model for *function-as-a-service* (FaaS) operations. Examples of recent models in this class from the research community include Durable Functions [38], Netherite [37] and Fix [51].

Unlike task-based models in the high-performance computing community, FaaS task-based models from the cloud computing community focus on resiliency to failures (as FaaS workloads are regularly pre-empted) and interaction with durable external state. These models have begun to include data models to assist the programmer in managing distributed data, though these models focus on bulk objects stored in external durable storage (such as Amazon's S3 buckets), rather than fine-grained subsets as in task-based models. FaaS models also focus on security guarantees, isolating function invocations on the same physical server from influencing or accessing the data of other invocations.

Dask [49] is a Python-based tasking system used to provide distributed backends to various popular Python libraries. Dask sits in many places, as a popular technology for Python-at-scale for a variety of workloads; I choose to discuss it in this section, but it could be grouped into Section 8.2 or Section 8.4.3. Dask supports a simple user-facing programming model, allowing for the launching of tasks that return data-carrying futures. Dask supports features like resilience against node failures, cluster elasticity, and automatic spilling of larger-than-memory workloads to disk. However, its simple data model and per-task overheads impede it from achieving the same peak performance as Legate on multi-GPU, multi-node machines.

8.4.3 Machine Learning

A variety of task-based systems have gained popularity for the execution and management of large-scale machine learning workloads. I will discuss two of these systems: Ray [92] and Pathways [19]. Ray is a hybrid task and actor (discussed more in Section 8.5) programming model originally developed for reinforcement learning (RL) applications. Ray started out as a Spark library for distributed RL computations, but the authors moved away from Spark due to the inflexibility and overheads of the programming model. Ray has a simpler data model than many of the task-based systems from the high-performance computing community, supporting data-carrying futures which are stored in a centralized object store. Ray tasks can accept futures as arguments, and the Ray runtime coordinates dependencies implied by these futures

and handles the movement of future data to where the data is needed. Similarly to cloud-computing systems, Ray's design supports elasticity (adding and removing nodes from a Ray cluster) and resilience to node failure, making it a common choice for deploying distributed machine-learning applications in the cloud. Ray's immutable, future-based data model with a centralized object store has made it difficult to deliver high-performance data movement between GPUs, leading users to manage communication through collective libraries like NCCL. Ray's runtime system imposes considerable overheads compared to task-based systems in the high-performance computing community, and recent work in the Ray community has invested in a graph compilation approach to reduce overheads [11].

Pathways is a task-based system developed at Google to support distributed JAX [35] workloads on Google's TPU clusters. Pathways employs a similar execution strategy as Legion and Realm, where the control plane (which launches parallel computations) is separated from the execution plane (where tasks are running and data movement happens) to enable decoupling and overlap of the two stages. Pathways also natively supports *gang-scheduling* of parallel task groups, which is necessary to correctly execute task groups that communicate with each other through synchronized collectives.

8.5 Actor-based Programming Models

Actor programming models were originally proposed by Irene Grief [64] and since then have found a variety of uses in modeling concurrency in a variety of domains, such as multi-core computing [41, 71, 3], distributed and cloud computing [92, 39, 8], and high-performance computing [76, 78]. The widespread influence of actors on many different communities has resulted in the term actors implying different features and capabilities in each of these communities. In this section, I'll focus on actor-based programming models in the context of targeting high-performance distributed and accelerated machines.

The canonical actor programming model in the high-performance computing community is Charm++ [78]. Charm++ programs are structured as a set of actors called

chares that send explicit messages to other chares to move data and communicate. Charm++ efficiently supports fine-grained messaging, which encourages applications to over-decompose problems; over-decomposition from the application side has led the Charm++ research community to focus on automated load-balancing algorithms for chare migration [110]. As discussed in Chapter 6, efficient programs in actor-based programming models like Charm++ are structured as state machines that wait for messages from chares before performing local computations. While efficient, these state machines can become difficult to maintain as programs scale in size, as they can require global knowledge of the entire program to maintain and additional program logic can result in non-local modifications to the state machines. In contrast, task-based abstractions that abstract over parallelism-related concepts like dependencies, data partitioning and communication enable large Legate programs to be defined independently and compose with high performance. Language support within the Charm++ community has focused on simplifying the construction and maintenance of these state machines [77].

Given the dichotomy between actor- and task-based models discussed in Chapter 6, researchers have introduced programming models that attempt to provide support for both actors and tasks within the same system [92, 72, 38]. This tension has been best evinced within Ray [92], which was initially conceived as a task-based programming model. The overheads from the implementation of tasking were large enough that actors were introduced to lower overheads of certain computational patterns. The actor model allowed Ray to lower overheads, but exposed end-users to the difficulties of state management. We show in Chapter 6 how these overheads may be avoided while retaining the productivity benefits of task-based programming models.

Our work in Chapter 6 was inspired by the duality between message- and procedure-based operating systems presented by Lauer et al. [85]. Lauer's work spurred further debate about whether threading-based or event-driven architectures were the right choice for productivity and performance [96, 129, 67]. Other work explored similar dualities in other domains, such as in fault-tolerance [112].

8.6 Just-in-Time Compilation

Just-In-Time (JIT) [70, 61, 97] compilers are a large and independent area of engineering and research; I focus on related components of the JIT compilation literature relevant to Legion's automatic tracing sub-system. JIT compilers for dynamic languages have a tiered execution system, where the target language is first translated to bytecode, which is executed by an interpreter. Frequently executed program fragments are then compiled into native instructions for significantly faster execution. Legion employs a similar architecture when performing automatic tracing where a task-based runtime system's dynamic analysis acts as the slow but general interpreter, and uses a tracing engine as the fast but specialized compiler.

Tracing-based JIT compilers such as TraceMonkey [62] record sequences of instructions executed at runtime and generate optimized code for those sequences. Method-based JIT compilers identify frequently invoked functions in the target program and compile type-specialized versions of those functions. JIT compilers identify the desired instruction sequences or methods to compile by relying on code landmarks like function definitions and basic block addresses to maintain counters of frequently executed program fragments. Since Legion's automatic tracing analysis views an unrolled stream of tasks, it must employ novel techniques for identification of traceable program fragments.

JIT compilers also perform dynamic analysis to recover data structures like call-graphs from the target program. Sampling-based methods [145] have been developed to balance runtime cost of profiling each function call with the accuracy of the sampled data structure. Discovering traces in our work requires long contiguous sequences of issued tasks to be analyzed together, as a trace must repeat several times to be considered by Legion. Breaking up these sequences with independent and non-contiguous samples can lead to a loss of precision when discovering traces. Instead, Legion's approach to automatic tracing employs an always-on approach where all tasks are analyzed, and uses a sub-sampling method on the set of collected tasks to manage the trade off between responsiveness of the trace analysis and length of the discovered traces. An always-on approach is cheaper to use in the task-based runtime

system context than within a standard JIT compiler as tasks are relatively coarse when compared to bytecode instructions.

Chapter 9

Conclusion

In this thesis, I discussed the challenge of writing composable software that targets modern high-performance distributed and accelerated machines. I presented many ways in which modern high-performance software does not compose either correctly or efficiently. This thesis then introduced the Legate programming model and runtime system, which provides the user-facing abstractions and program analysis techniques to enable high-performance composable software to be built. We showed that with Legate, end users can build high-performance software from high-level Python programs that assemble multiple independent libraries with no distribution or parallelism anywhere in the code. The contributions of this thesis span the full Legate stack, from front-end program representation and analysis down to techniques to squeeze out the last bit of performance, removing overheads imposed by the entire system. Concretely, this thesis discussed the architecture of the Legate runtime system and concurrently introduced the key ideas that enable the efficient composition of programs across different axes:

- Chapter 3 discusses components of Legate that enable independent modules to efficiently share distributed data, including the Legate front-end program representation (with constraint-based partitioning) and integration with Legion to map logical data onto shared physical allocations.

- Chapter 4 discusses the Legate middle-end, which includes program representation and analysis to enable the fusion of computations across different modules.
- Chapter 5 discusses just-in-time compilation techniques that automatically amortize the expensive but critical dynamic analysis performed by Legate.
- Chapter 6 discusses compilation techniques at the lowest level of the Legate runtime system to execute parallel computations with overheads comparable to low-level systems like MPI.

Put together, the Legate framework provides an infrastructure to build parallel computations from individually optimized components and execute the composed code with performance competitive to the best hand-tuned implementations.

9.1 Future Work

There are many interesting avenues that future work building on this thesis may explore. I now detail several of these directions.

Aggressive Composability Analyses.

Legate's program representations and analyses form a foundation upon which more aggressive optimizations for composition can be explored and built. I believe there are opportunities across the runtime for further optimization, both at the high-level front-end layer and in the low-level execution engines. In particular, more expensive and global optimizations can be performed for repeated traces of Legate programs. The partitioning constraint resolution process in Chapter 3 makes local and fast decisions about constraint resolution, but more interesting decisions could be made by an algorithm that resolves constraints across an entire trace — partitioning choices that are locally suboptimal but globally optimal can be explored. Chapter 6 discussed compiling the control and dependence management of task-based models into actor-based models to reduce the overheads of coordination. I believe that even further optimization can be performed in this compilation process, particularly around

data movement. Specialized kernels can be generated for data movement operations within the Realm runtime, or communication patterns in a computation graph can be converted into efficient collective operations. Either of these directions leverage the high-level representations of Legate programs to perform optimizations that experts make when building a monolithic software artifact out of separate logical pieces.

A Plug-and-Play Fusion Ecosystem.

As a central analysis component for different modules in a target program, Legate can act as a system that coordinates specialized program analyses from each of these different modules. The fusion approach described in Chapter 4 leverages a small set of optimizations hand-selected to effectively optimize the class of computations found in cuPyNumeric. I believe that Legate should leverage the capabilities of the MLIR [84] ecosystem to allow for multiple domain-specific representations to exist in task implementations, and then have libraries register their own desired optimizations with the central runtime system. There are several examples where an approach like this may have immediate benefit. Consider cuPyNumeric leveraging the MLIR `linalg` dialect for dense linear algebra computations, and Legate Sparse leveraging MLIR's `sparse` dialect and analysis passes for sparse linear algebra. These libraries could register these analyses with Legate, and discover fused dense and sparse linear algebra operations at runtime when composed, with neither library knowing the full extent of the program. Alternatively, cuPyNumeric with the MLIR `affine` dialect could interact with a specialized dialect for stencil operations [66] and extract optimized stencil implementations from high-level NumPy programs. Legate could even be used as an engine to create unified program representations before operations are lowered to tasks, enabling domain-specific optimizations often found in machine-learning libraries. Applying all these different optimizations from different sources will likely require tackling the phase-ordering problem, and perhaps equality saturation [135] will be a core technology in such a system.

9.2 Exciting New Hardware.

Separately from Legate, distributed and accelerated computing platforms are undergoing rapid change due to the insatiable desire for high performance from the machine learning community and the end of Moore's law. Individual processors like GPUs are becoming increasingly specialized, introducing fixed-function units for critical computations that come with a variety of user-facing programming challenges. Modern networks are increasing their bandwidth, as well as supporting lower-latency communication methods and pushing collective communication patterns into hardware-supported protocols directly in the network switches. Finally, accelerated systems are becoming more vertically integrated, where a single processor is increasingly a distributed system, and networking infrastructure is more tightly coupled to the computational elements. Put together, these machines are becoming increasingly powerful, but increasingly difficult to program, and the gap between the performance of a mediocre program and a well-tuned program is immense. Developing programming systems to control the complexity of emerging hardware is an exciting problem, and perhaps leveraging ideas from the Legate ecosystem is a way forward for these kinds of specialized machines. I have done some initial work in this area, leveraging ideas from task-based parallel programming to automate the concurrency and asynchrony required in the kernel development process of modern accelerators [139] and to automatically find good pipeline schedules for certain classes of kernels [117].

Bibliography

[1] PETSc VecAXPBYPCZ manual page. `https://petsc.org/main/manualpages/Vec/VecAXPBYPCZ/`. Accessed: April 6, 2026.

[2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, 12 1986.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[5] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, page 126–138, New York, NY, USA, 1993. Association for Computing Machinery.

[6] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski,

Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 929–947, New York, NY, USA, 2024. Association for Computing Machinery.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[8] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[9] Cédric Augonnet, Andrei Alexandrescu, Albert Sidelnik, and Michael Garland. Cudastf: Bridging the gap between cuda and task parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '24. IEEE Press, 2024.

[10] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[11] Ray Authors. Ray compiled graph documentation. Technical report, AnyScale, 2024.

[12] SciPy Authors. `scipy.sparse` documentation. `https://docs.scipy.org/doc/scipy/reference/sparse.html`, 2022.

[13] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. PETSc Web page. `https://petsc.org/`, 2022.

[14] Kihiro Bando, Steven Brill, Elliott Slaughter, Michael Sekachev, Alex Aiken, and Matthias Ihme. *Development of a discontinuous Galerkin solver using Legion for heterogeneous high-performance computing architectures.* 2021.

[15] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. Mosaic: An interoperable compiler for tensor algebra. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[16] Manya Bansal, Dillon Sharlet, Jonathan Ragan-Kelley, and Saman Amarasinghe. Lightweight and locality-aware composition of black-box subroutines. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[17] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. *SIGOPS Oper. Syst. Rev.*, 40(5):394–403, oct 2006.

[18] Lorena Barba and Gilbert Forsyth. Cfd python: the 12 steps to navier-stokes equations. *Journal of Open Source Education*, 2(16):21, 2019.

[19] Paul Barham, Aakanksha Chowdhery, Jeff Dean, Sanjay Ghemawat, Steven Hand, Dan Hurt, Michael Isard, Hyeontaek Lim, Ruoming Pang, Sudip Roy, Brennan Saeta, Parker Schuh, Ryan Sepassi, Laurent El Shafey, Chandramohan A. Thekkath, and Yonghui Wu. Pathways: Asynchronous distributed dataflow for ml, 2022.

[20] Michael Bauer. *Programming Distributed Heterogeneous Architectures with Logical Regions*. PhD thesis, Stanford University, 2014.

[21] Michael Bauer and Michael Garland. Legate numpy: accelerated and distributed array computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. Scaling implicit parallelism via dynamic control replication. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 105–118, New York, NY, USA, 2021. Association for Computing Machinery.

[23] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. Scaling implicit parallelism via dynamic control replication. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 105–118, New York, NY, USA, 2021. Association for Computing Machinery.

[24] Michael Bauer, Elliott Slaughter, Sean Treichler, Wonchan Lee, Michael Garland, and Alex Aiken. Visibility algorithms for dynamic dependence analysis and distributed coherence. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 218–231, New York, NY, USA, 2023. Association for Computing Machinery.

[25] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.

[26] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Structure Slicing: Extending Logical Regions with Fields. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 845–856, New Orleans, LA, USA, November 2014. IEEE.

[27] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N. Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. Compiler support for sparse tensor computations in mlir. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(4):1–25, 2022.

[29] Laura Susan Blackford, J. Choi, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, K. Stanley, J. Dongarra, S. Hammarling, G. Henry, and D. Walker. Scalapack: a portable linear algebra library for distributed memory computers - design issues and performance. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, page 5–es, USA, 1996. IEEE Computer Society.

[30] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on*

Principles and Practice of Parallel Programming, PPOPP '95, page 207–216, New York, NY, USA, 1995. Association for Computing Machinery.

[31] Dan Bonachea and Paul H. Hargrove. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18)*, volume 11882 of *Lecture Notes in Computer Science*. Springer Int'l Publishing, October 2018. https://doi.org/10.25344/S4QP4W.

[32] Uday Bondhugula. High performance code generation in MLIR: an early case study with GEMM. *CoRR*, abs/2003.00532, 2020.

[33] Uday Kumar Reddy Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model*. PhD thesis, Ohio State University, USA, 2008. AAI3325799.

[34] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Dague: A generic distributed dag engine for high performance computing. In *2011 IEEE Int'l Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1151–1158, 2011.

[35] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[36] Kevin J. Brown, HyoukJoong Lee, Tiark Romp, Arvind K. Sujeeth, Christopher De Sa, Christopher Aberger, and Kunle Olukotun. Have abstraction and eat performance, too: Optimized heterogeneous computing with parallel patterns. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 194–205, 2016.

[37] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S. Meiklejohn, and Xiangfeng

Zhu. Netherite: efficient execution of serverless workflows. *Proc. VLDB Endow.*, 15(8):1591–1604, April 2022.

[38] Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. Durable functions: semantics for stateful serverless. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021.

[39] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, New York, NY, USA, 2011. Association for Computing Machinery.

[40] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[41] Dominik Charousset, Thomas C. Schmidt, Raphael Hiesgen, and Matthias Wählisch. Native actors: a scalable software platform for distributed, heterogeneous environments. In *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE! 2013, page 87–96, New York, NY, USA, 2013. Association for Computing Machinery.

[42] Siddhartha Chatterjee, Guy E. Blelloch, and Allan L. Fisher. Size and access inference for data-parallel programs. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, page 130–144, New York, NY, USA, 1991. Association for Computing Machinery.

[43] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[44] Pi-Yueh Chuang. Torchswe: Gpu shallow-water equation solver. https://github.com/piyueh/TorchSWE, 2021.

[45] NVIDIA Corporation. Welcome to Legate.STL — NVIDIA legate.core — docs.nvidia.com. `https://docs.nvidia.com/legate/24.06/legate.stl/source/legate-stl.html`. [Accessed 04-03-2026].

[46] M. Cosnard, E. Jeannot, and T. Yang. Slc: Symbolic scheduling for executing parameterized task graphs on multiprocessors. In *Proceedings of the 1999 International Conference on Parallel Processing*, pages 413–421, 1999.

[47] Paul Crozier, Heidi Thornquist, Robert Numrich, Alan Williams, H. Edwards, Eric Keiter, Mahesh Rajan, James Willenbring, Douglas Doerfler, and Michael Heroux. Improving performance via mini-applications. 01 2009.

[48] Modan K. Das and Ho-Kwok Dai. A survey of dna motif finding algorithms. *BMC Bioinformatics*, 8(7):S21, Nov 2007.

[49] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016.

[50] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, January 2010.

[51] Yuhan Deng, Akshay Srivatsan, Sebastian Ingino, Francis Chua, Yasmine Mitchell, Matthew Vilaysack, and Keith Winstein. Fix: Externalizing network I/O in serverless computing, 2025.

[52] Jack B. Dennis. *Modular, asynchronous control structures for a high performance processor*, page 55–80. Association for Computing Machinery, New York, NY, USA, 1970.

[53] J.B. Dennis. A parallel program execution model supporting modular software construction. In *Proceedings. Third Working Conference on Massively Parallel Programming Models (Cat. No.97TB100228)*, pages 50–60, 1997.

[54] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. Terra: a multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and*

Implementation, PLDI '13, page 105–116, New York, NY, USA, 2013. Association for Computing Machinery.

[55] Mario Di Renzo, Lin Fu, and Javier Urzay. Htr solver: An open-source exascale-oriented task-based multi-gpu high-order code for hypersonic aerothermodynamics. *Computer Physics Communications*, 255:107262, 2020.

[56] S. Ebadi, A. Keesling, M. Cain, T. T. Wang, H. Levine, D. Bluvstein, G. Semeghini, A. Omran, J.-G. Liu, R. Samajdar, X.-Z. Luo, B. Nash, X. Gao, B. Barak, E. Farhi, S. Sachdev, N. Gemelke, L. Zhou, S. Choi, H. Pichler, S.-T. Wang, M. Greiner, V. Vuletić, and M. D. Lukin. Quantum optimization of maximum independent set using rydberg atom arrays. *Science*, 376(6598):1209–1215, 2022.

[57] Melih Elibol, Vinamra Benara, Samyu Yagati, Lianmin Zheng, Alvin Cheung, Michael I. Jordan, and Ion Stoica. Nums: Scalable array programming for the cloud, 2022.

[58] Charles R. Ferenbaugh. Pennant: an unstructured mesh mini-app for advanced architecture research. *Concurr. Comput. : Pract. Exper.*, 27(17):4555–4572, December 2015.

[59] Message P Forum. Mpi: A message-passing interface standard. Technical report, USA, 1994.

[60] Yoshihiko Futamura. Partial computation of programs. In Eiichi Goto, Koichi Furukawa, Reiji Nakajima, Ikuo Nakata, and Akinori Yonezawa, editors, *RIMS Symposia on Software Science and Engineering*, pages 1–35, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg.

[61] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 30th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, PLDI '09, page 465–478, New York, NY, USA, 2009. Association for Computing Machinery.

[62] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

[63] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, page 223–232, New York, NY, USA, 1993. Association for Computing Machinery.

[64] I. Grief and Irene Greif. Semantics of communicating parallel processes. Technical report, Massachusetts Institute of Technology, USA, 1975.

[65] Torsten Grust. *Monad Comprehensions: A Versatile Representation for Queries*, pages 288–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[66] Tobias Gysi, Christoph Mueller, Oleksandr Zinenko, Stephan Andreas Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. Domain-specific multi-level rewriting for hpc: A case study with mlir. *ACM Transactions on Architecture and Code Optimization*, 4:51:1–51:23, 2021.

[67] Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proceedings of the 9th International Conference on Coordination Models and Languages*, COORDINATION'07, page 171–190, Berlin, Heidelberg, 2007. Springer-Verlag.

[68] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian

Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[69] Michael Allen Heroux and Jack. Dongarra. Toward a new metric for ranking high performance computing systems. 6 2013.

[70] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, 18(4):355–400, jul 1996.

[71] Christopher R. Houck and Gul Agha. Hal: A high-level actor language and its distributed implementation. In Kang G. Shin, editor, *ICPP (2)*, pages 158–165, 1992.

[72] Shams M. Imam and Vivek Sarkar. Integrating task parallelism with actors. *SIGPLAN Not.*, 47(10):753–772, October 2012.

[73] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 47–62, New York, NY, USA, 2019. Association for Computing Machinery.

[74] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. *CoRR*, abs/1807.05358, 2018.

[75] Hartmut Kaiser, Patrick Diehl, Adrian S. Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R. Brandt, Nikunj Gupta, Thomas Heller, Kevin Huck, Zahra Khatami, Alireza Kheirkhahan, Auriane Reverdell, Shahrzad Shirzad, Mikael Simberg, Bibek Wagle, Weile Wei, and Tianyi Zhang. Hpx - the c++ standard library for parallelism and concurrency. *Journal of Open Source Software*, 5(53):2352, 2020.

[76] L. Kal'e, W. Fenton, B. Ramkumar, Vikram Saletore, and A. Sinha. Supporting machine independent parallel programming on diverse architectures. 10 1995.

[77] Laxmikant V. Kalé and Milind A. Bhandarkar. Structured dagger: A coordination language for message-driven programming. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume I*, Euro-Par '96, page 646–653, Berlin, Heidelberg, 1996. Springer-Verlag.

[78] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: a portable concurrent object oriented system based on c++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, page 91–108, New York, NY, USA, 1993. Association for Computing Machinery.

[79] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir, editor, *Combinatorial Pattern Matching*, pages 181–192, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[80] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[81] Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A language for physical simulation. *ACM Trans. Graph.*, 35(2), March 2016.

[82] Argonne National Laboratory. CANDLE — Exascale Deep Learning and Simulation Enabled Precision Medicine for Cancer — wordpress.cels.anl.gov. `https://wordpress.cels.anl.gov/candle/`. [Accessed 06-05-2024].

[83] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[84] Chris Lattner, Jacques A. Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR*, abs/2002.11054, 2020.

[85] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, April 1979.

[86] Wonchan Lee, Manolis Papadakis, Elliott Slaughter, and Alex Aiken. A constraint-based approach to automatic data partitioning for distributed memory execution. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[87] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. Dynamic tracing: memoization of task graphs for dynamic task-based runtimes. In *Proceedings of the Int'l Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press, 2018.

[88] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. Automatic horizontal fusion for gpu kernels. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, page 14–27. IEEE Press, 2022.

[89] Jongseok Lim, Han-gyeol Lee, and Jaewook Ahn. Review of cold rydberg atoms and their applications. *Journal of the Korean Physical Society*, 63(4):867–876, 2013.

[90] Michael Merrill, William Reus, and Timothy Neumann. Arkouda: interactive data exploration backed by chapel. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop*, CHIUW 2019, page 28, New York, NY, USA, 2019. Association for Computing Machinery.

[91] Seema Mirchandaney, Alex Aiken, and Elliott Slaughter. Speaking pygion: Experiences writing an exascale single particle imaging code. In *Asynchronous Many-Task Systems and Applications: Second International Workshop, WAMTA 2024, Knoxville, TN, USA, February 14–16, 2024, Proceedings*, page 1–8, Berlin, Heidelberg, 2024. Springer-Verlag.

[92] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017.

[93] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., USA, 1996.

[94] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 883–898, New York, NY, USA, 2021. Association for Computing Machinery.

[95] NVIDIA. legate-io. Accessed 2026-03-05.

[96] John Ousterhout. Why threads are a bad idea (for most purposes), 1996.

[97] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, page 1, USA, 2001. USENIX Association.

[98] Shoumik Palkar, James Thomas, Deepak Narayanan, Anil Shanbhag, Rahul Palamuttam, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. Weld: Rethinking the interface between data-intensive applications. *CoRR*, abs/1709.06416, 2017.

[99] Shoumik Palkar and Matei Zaharia. Optimizing data-intensive computations in existing libraries with split annotations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 291–305, New York, NY, USA, 2019. Association for Computing Machinery.

[100] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: an imperative style, high-performance deep learning library*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[101] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. Towards scalable dataframe systems. *Proc. VLDB Endow.*, 13(12):2033–2046, July 2020.

[102] QuEraComputing. Bloqade.jl: Package for the quantum computation and quantum simulation based on the neutral-atom architecture., 2023.

[103] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, jun 2013.

[104] RAPIDS. legate-boost, 2023. Accessed 2026-03-04.

[105] RAPIDS. legate-dataframe, 2024. Accessed 2026-03-04.

[106] RAPIDS. legate-raft, 2024. Accessed 2026-03-04.

[107] Aleix Roca and Vicenç Beltran. Rethinking thread scheduling under oversubscription: A user-space framework for coordinating multi-runtime and multi-process workloads. In *Proceedings of the 31st ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '26, page 53–67, New York, NY, USA, 2026. Association for Computing Machinery.

[108] E. Rotenberg, S. Bennett, and J.E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th Annual IEEE/ACM Int'l Symposium on Microarchitecture. MICRO 29*, pages 24–34, 1996.

[109] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[110] Vikram A. Saletore. A distributed and adaptive dynamic load balancing scheme for parallel processing of medium-grain tasks. In *Proceedings of the Fifth Distributed Memory Computing Conference (5th DMCC'90)*, volume II, Architecture Software Tools, and Other General Issues, pages 994–999, Charleston, SC, April 1990. IEEE.

[111] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD Int'l Conference on Management of Data*, SIGMOD '03, page 76–85, New York, NY, USA, 2003. Association for Computing Machinery.

[112] Santosh K. Shrivastava, Luigi V. Mancini, and Brian Randell. The duality of fault-tolerant system structures. *Software: Practice and Experience*, 23(7):773–798, 1993.

[113] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. Loop rerolling for hardware decompilation. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[114] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: a high-productivity programming language for hpc with logical regions.

In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[115] Elliott Slaughter, Wonchan Lee, Sean Treichler, Wen Zhang, Michael Bauer, Galen Shipman, Patrick McCormick, and Alex Aiken. Control replication: compiling implicit parallelism to efficient spmd with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[116] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Lee, Sean Treichler, Patrick McCormick, and Alex Aiken. Task bench: a parameterized benchmark for evaluating parallel runtime performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020.

[117] Rupanshu Soi, Rohan Yadav, Fredrik Kjolstad, Alex Aiken, Maryam Mehri Dehnavi, Michael Garland, and Michael Bauer. Optimal software pipelining and warp specialization for tensor core gpus, 2025.

[118] James A. Storer and Thomas G. Szymanski. Data compression via textual substitution. *J. ACM*, 29(4):928–951, oct 1982.

[119] Jens Stoye and Dan Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270(1):843–856, 2002.

[120] The Jupyter Book Community. Ecosystem — jupyter meets the earth. `https://jupyterearth.org/jupyter-resources/introduction/ecosystem.html`, 2022. Accessed: 2026-04-01.

[121] Sean Treichler. *Realm: Performance Portability through Composable Asynchrony*. Ph.d. dissertation, Stanford University, 2016.

[122] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: an event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, page 263–276, New York, NY, USA, 2014. Association for Computing Machinery.

[123] Sean Treichler, Michael Bauer, Ankit Bhagatwala, Giulio Borghesi, Ramanan Sankaran, Hemanth Kolla, Patrick Mccormick, Elliott Slaughter, Wonchan Lee, Alex Aiken, and Jacqueline H. Chen. S3d-legion: An exascale software for direct numerical simulation of turbulent combustion with complex multicomponent chemistry. 11 2017.

[124] Sean Treichler, Michael Bauer, Rahul Sharma, Elliott Slaughter, and Alex Aiken. Dependent partitioning. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, page 344–358, New York, NY, USA, 2016. Association for Computing Machinery.

[125] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, Carlsbad, CA, July 2022. USENIX Association.

[126] Rob F. Van der Wijngaart and Timothy G. Mattson. The parallel research kernels. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2014.

[127] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[128] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[129] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, page 4, USA, 2003. USENIX Association.

[130] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[131] Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[132] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[133] Sam Westrick, Mike Rainey, Daniel Anderson, and Guy E. Blelloch. Parallel block-delayed sequences. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, page 61–75, New York, NY, USA, 2022. Association for Computing Machinery.

[134] Wikipedia. Ruler function — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Ruler%20function&oldid=1193825609`, 2024. [Online; accessed 02-May-2024].

[135] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[136] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. Distal: the distributed tensor algebra compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 286–300, New York, NY, USA, 2022. Association for Computing Machinery.

[137] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. Spdistal: Compiling distributed sparse tensor computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[138] Rohan Yadav, Michael Bauer, David Broman, Michael Garland, Alex Aiken, and Fredrik Kjolstad. Automatic tracing in task-based runtime systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 84–99, New York, NY, USA, 2025. Association for Computing Machinery.

[139] Rohan Yadav, Michael Garland, Alex Aiken, and Michael Bauer. Task-based tensor computations on modern gpus. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025.

[140] Rohan Yadav, Joseph Guman, Sean Treichler, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. On the duality of task and actor programming models, 2025.

[141] Rohan Yadav, Wonchan Lee, Melih Elibol, Manolis Papadakis, Taylor Lee-Patti, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. Legate sparse: Distributed sparse computing in python. In *Proceedings of the*

International Conference for High Performance Computing, Networking, Storage and Analysis, SC '23, New York, NY, USA, 2023. Association for Computing Machinery.

[142] Rohan Yadav, Shiv Sundram, Wonchan Lee, Michael Garland, Michael Bauer, Alex Aiken, and Fredrik Kjolstad. Composing distributed computations through task and kernel fusion. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS '25, page 182–197, New York, NY, USA, 2025. Association for Computing Machinery.

[143] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, page 10, USA, 2010. USENIX Association.

[144] David Kai Zhang, Rohan Yadav, Alex Aiken, Fredrik Kjolstad, and Sean Treichler. Kdrsolvers: Scalable, flexible, task-oriented krylov solvers. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC Workshops '25, page 1351–1365, New York, NY, USA, 2025. Association for Computing Machinery.

[145] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, page 263–271, New York, NY, USA, 2006. Association for Computing Machinery.

[146] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[147] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.